Библиотека инженера

Петров И.В.

Программируемые контроллеры

Стандартные языки и приемы прикладного проектирования

ISBN 5-98003-079-4

- Стандарт МЭК 61131
- Инструменты программирования
- Языки МЭК
- Стандартные компоненты
- Примеры программирования

пусть эта книга принесет вам удачу

Серия «Библиотека инженера»

И. В. Петров

Программируемые контроллеры. Стандартные языки и приемы прикладного проектирования

Под ред. проф. В. П. Дьяконова

Москва
СОЛОН-Пресс
2004

УДК 681.5
ББК 32.96
П30

Петров И. В.
П30 Программируемые контроллеры. Стандартные языки и приемы прикладного проектирования / Под ред. проф. В. П. Дьяконова. — М.: СОЛОН-Пресс, 2004. — 256 с.: ил. — (Серия «Библиотека инженера»)

ISBN 5-98003-079-4

Описана практика применения промышленных программируемых контроллеров, широко применяющихся для автоматизации производства. Излагаются языки программирования на основе действующего стандарта МЭК 61131-3 и многочисленные примеры подготовки программ для промышленных программируемых контроллеров.

Для специалистов по автоматизации производственных процессов и производственного оборудования, а также для студентов и преподавателей высших технических заведений.

ISBN 5-98003-079-4

Предисловие научного редактора

Более тридцати лет прошло со знаменательного события — появления первого микропроцессора корпорации Intel — 4004. Частота его работы была в тысячи раз меньше, чем частота электронных устройств на уже созданных тогда в СССР мощных СВЧ-полевых транзисторах. Однако микропроцессор обладал новым удивительным в те годы качеством — он мог выполнять самые разнообразные функции, изменяемые программно. И это обеспечило невиданные темпы развития микропроцессорной техники.

Сейчас, когда череда процессоров для *персональных компьютеров* (ПК) только одной корпорации Intel насчитывает ряд поколений (от ушедшего в историю 8086 до новейших Pentium 4, Pentium M (Centrino) и Inatium) и сотни конкретных типов, у многих создалось впечатление, что микропроцессоры это прежде всего *принадлежность персональных компьютеров*. И впрямь, взлету роли и значимости ПК трудно что-то противопоставить. А данные о частотах работы современных процессоров в единицы ГГц (в перспективе ТГц), о числе транзисторов в десятки миллионов на кристалле и о ничтожной (уже до долей ватта) потребляемой мощности этих сложнейших устройств даже специалисты воспринимают как фантастику наших дней.

И тем не менее не стоит забывать, что изначально микропроцессоры были созданы не столько для работы в ПК, сколько в качестве программно-управляемых устройств для автоматизации промышленности и бытовой техники. На их основе были созданы *программируемые логические контроллеры* (ПЛК) — устройства, автоматизирующие работу как отдельных аппаратов, например станков с программным управлением или стиральных машин и микроволновых печей, так и огромных производственных комплексов. Сегодня в своей квартире мы, увы, не всегда встретим персональный компьютер, но, даже не подозревая об этом, пользуемся работой доброго десятка программируемых микроконтроллеров. Таким образом, программируемые логические контроллеры — даже более распространенные устройства, чем ПК, количество которых во всем мире недавно превысило магическую цифру в 1 миллиард.

По микропроцессорам и ПК у нас написаны и изданы сотни книг. А вот литературы по программируемым логическим контроллерам сейчас практически нет. Отчасти это обусловлено об-

щим крайне неблагоприятным для нашей науки и техники положением, возникшим после распада СССР и ввода ряда явно непродуманных и неприемлемых для нашего менталитета и состояния промышленности рыночных реформ. В результате навязанной нам экономической политики произошел распад занятых автоматизацией промышленности коллективов ученых и технических специалистов.

К счастью, наука и техника принадлежат к отраслям деятельности человечества, которые, испытывая отдельные спады в той или иной конкретной стране, в целом непрерывно развиваются и совершенствуются. Сложившееся у нас положение имеет и свои положительные стороны. Прежде всего надо отметить интеграцию нашей промышленности с мировой промышленностью, без чего создание конкурентно-способных изделий и товаров невозможно. Пал «железный занавес» как в общении специалистов разных стран, так и в обмене конструктивными идеями и разработками. В итоге на наш рынок стали поступать новейшие микроэлектронные изделия и программные продукты ведущих западных корпораций и фирм. Ширится их применение в наших разработках и товарах как внутреннего потребления, так и идущих на экспорт.

Но, пожалуй, самое главное состоит в том, что созданный в годы СССР научный, технический и промышленный потенциал в наше время вновь стал расти, причем на качественно новом уровне — интеграции с мировым промышленным потенциалом. К счастью, далеко не все наши специалисты, окончившие технические вузы, ринулись в торговлю, коммерцию и юриспруденцию или в поисках лучшей доли эмигрировали на Запад. Многие остались верными выбранному пути и, несмотря на его временные трудности, влились в ряды разработчиков новейшей аппаратуры и технических средств автоматизации производства. Разумеется, теперь уже использующей мировые достижения в науке и технике.

К таким специалистам относится и автор этой книги — Игорь Петров. Это, безусловно, человек новой формации, понимающий нынешние рыночные реалии и вовсе не пасующий перед ними. Будучи моим студентом и аспирантом, Игорь Петров выбрал путь как по своему образованию (промышленная электроника), так и по призванию. Он является техническим директором компании «Пролог», успешно разрабатывающей и внедряющей современные программируемые логические контроллеры в наше производство. И вот теперь он решился на подготовку этой книги, которая

обобщает многолетний опыт его практической работы в области программирования и применения программируемых логических контроллеров. В области, безусловно, актуальной, интересной и для нас новой.

Автор книги опытный программист. И ему удалось описать все важнейшие аспекты программирования таких массовых устройств, как программируемые контроллеры. Приятно отметить, что это описание сделано хорошим и ясным языком, вполне понятным не только опытным специалистам, но и студентам технических вузов и университетов. Книга хорошо иллюстрирована. Она компактна, но содержит практически весь необходимый материал по программированию ПЛК с рядом интересных примеров. Уважительное отношение автора к стандартам также является ее достоинством, как и отражение новейшего программного инструментария.

Все это позволяет надеяться на то, что данная книга найдет своего читателя в лице специалистов в области автоматизации промышленности, а также преподавателей и студентов технических вузов соответствующего книге профиля.

Дьяконов В. П.,
доктор технических наук,
профессор

Введение

Книга описывает программирование систем управления промышленной автоматикой, построенных на базе программируемых логических контроллеров (ПЛК). Основное внимание уделяется технологии создания программного обеспечения для систем, построенных на базе ПЛК, и практическому программированию на языках стандарта МЭК 61131-3.

К сожалению, период широкого распространения стандарта МЭК 61131-3 пришелся в России на годы перестройки. Отсутствие спроса промышленности на средства автоматизации производства привело к распаду большинства коллективов, занятых применением ПЛК, что, естественно, отразилось и на уровне подготовки специалистов. В настоящее же время наблюдается существенный рост потребности в современных инструментах производства и автоматики. Количество продаж ПЛК в Европе за 2002 год практически не изменилось, по данным же российских фирм, спрос за год увеличился в несколько раз. Широкое распространение и доступность персональных компьютеров привели к появлению большого числа специалистов, профессионально владеющих компьютерными технологиями. Поэтому не удивительно, что сегодня персональные компьютеры (ПК) массово применяют на всех уровнях промышленной автоматизации, включая классические контроллерные задачи.

Между тем во многих случаях применение промышленных ПК не оправдано экономически и технически сложно. Даже там, где задача на ПЛК решается «в одно действие» и на два порядка дешевле, нередко применяют дорогостоящие промышленные ПК, операционные системы реального времени и заказное программное обеспечение. Единственной причиной такого подхода является наличие подготовленных специалистов. Однобокости решений немало сопутствует и почти полное отсутствие современной литературы по применению ПЛК на русском языке. При знакомстве с доступными источниками 10 летней давности [7] создается впечатление, что применение ПЛК связано с примитивным и неуклюжим программированием при помощи специализированных пультов, что выглядит чрезвычайно архаично.

В последнее десятилетие появился целый класс инструментов визуального прикладного проектирования для ПЛК. Выражаясь языком специалистов звуковоспроизводящей техники, это инст-

рументы класса «High end». Мало того, понятие «программирование» для контроллеров все более вытесняется словом «проектирование». И действительно, процесс перетаскивания мышью графических объектов называть программированием сложно.

Данная книга целиком посвящена **проектированию систем на базе ПЛК с помощью инструментов, ориентированных на языки стандарта МЭК 61131-3**. Приведенный материал носит достаточно универсальный характер. Мы намеренно не будем подробно описывать конкретные ПЛК и фирменные инструменты их программирования. При подготовке книги использовался комплекс CoDeSys фирмы 3S (Smart Software Solutions). Это означает, что все примеры были реализованы в этой системе и тщательно проверены. Но с не меньшим успехом вы можете использовать другие инструменты, кратко представленные в третьей главе книги. Все они созданы известными мировыми лидерами рынка инструментов программирования ПЛК. Там, где в тексте книги необходимо обратить внимание на особенности конкретной реализации, это указано в явном виде.

Для практического применения любого универсального комплекса программирования МЭК 61131-3 с конкретным контроллером необходима адаптация системного программного обеспечения ПЛК и определенная настройка комплекса. Эта работа требует высокой квалификации и не дешева. Но это проблема изготовителей и поставщиков контроллеров. Потребитель ПЛК всегда работает с настроенным инструментом и не несет затрат по адаптации. Для знакомства с технологией ПЛК и обучения их программированию достаточно иметь бесплатную демонстрационную версию комплекса. Книга намеренно не включает CD с подобными демо-версиями, поскольку благодаря Интернету (см. ссылки в конце книги) получить необходимый пакет из первых рук не представляет сложности. Кроме того, вам не придется платить лишние деньги за диск с наверняка уже устаревшими программами.

Предполагается, что читатель, как минимум, имеет представление о работе с персональным компьютером и слова «бит», «байт», «шестнадцатеричная система счисления» не вызывают тяжелых воспоминаний. Глубокое знание математики, компьютеров, сетей, операционных систем, теории систем автоматического управления и идеологии построения ПЛК не является необходимым для освоения материала данной книги. В серьезной практической работе все эти вопросы, безусловно, возникнут. В конце книги приведен список литературы [1—38] и Интернет-источников, которые при необходимости подскажут путь поиска решения.

Многие книги и учебники по программированию содержат обширные вводные разделы. В них поверхностно излагается весь материал, далее изложение повторяется уже более подробно, иногда даже несколько раз с различных точек зрения. Это не удивительно и объясняется значительной взаимосвязью всех материалов. В данной книге вся информация сразу излагается со всеми подробностями, иногда даже с краткими ссылками вперед на еще не описанный материал. Это упрощает поиск информации, поскольку все, что касается одной проблемы, компактно сосредоточено в одном разделе книги. Такой подход более характерен для справочников и ориентирован на экономию времени подготовленных читателей, нацеленных на практическую работу и имеющих навык чтения технической документации.

Если вам необходимо постепенное погружение в материал, используйте «двухпроходное» чтение. Сначала прочитайте всю книгу «по диагонали», пропуская то, что покажется непонятным. Далее просмотрите разделы, посвященные описанию языков программирования, выберете наиболее вам понятный, изучите раздел подробнее и разберите один из примеров, реализованный на данном языке. На этом этапе необходимо включить компьютер и реализовать разобранный пример или еще лучше фрагмент собственной практической задачи. Подключать реальный ПЛК необязательно, достаточно встроенного эмулятора системы программирования (можно использовать демо-версию). Пока вы не заставите пример работать должным образом, дальнейшее чтение книги не имеет смысла.

Следует сразу обратить внимание на то, что для программирования ПЛК не требуется знание всех пяти языков МЭК 61131-3. Так, используя даже простейший, похожий на ассемблер язык IL (список инструкций), можно реализовать проект любой сложности. В то же время выбор языка существенно влияет на способ мышления. В результате существует много задач, красивое решение которых на одном языке получается практически без усилий, а на другом языке требует применения малопонятных «трюков» и, естественно, серьезной отладки. Овладение же приемами работы на всех языках и возможность совмещения их в одной задаче позволяют работать быстро и надежно.

Книга содержит большое число примеров реализации различных алгоритмов. Примеры преднамеренно не содержат описания законченных проектов промышленных систем. Для объяснения любого такого примера обязательно потребуется погружение в детали технологии конкретного производства, что не является це-

лью данной книги. Вероятность же того, что какой-либо пример совпадет с практической задачей читателя, крайне низка. Поэтому примеры нацелены на решение сравнительно простых и разнообразных задач, причем не только специфичных для классических применений ПЛК. Примеры призваны служить источником новых идей, а не набором шаблонных решений.

Предупреждения

Практическое применение ПЛК в цеховых условиях сопряжено с повышенной опасностью. Ошибки во внешних электрических цепях контроллеров, некорректный расчет устройств питания и силовых блоков, некачественное заземление, неправильно выполненная система аварийного отключения, отсутствие защиты механических узлов и прочие нарушения правил монтажа могут привести к тяжелым последствиям. **Монтаж ПЛК и сопряженного с ним оборудования должен выполнятся только квалифицированным персоналом,** имеющим соответствующие допуски.

Ошибки в прикладном программном обеспечении ПЛК способны приводить к потере синхронности работы механизмов, что может стать причиной их поломки или привести к травмам обслуживающего персонала. Правильно спроектированная система должна содержать элементы блокировки, исключающие такую возможность. Детальное рассмотрение техники проектирования безопасных систем автоматического управления не является предметом данной книги. Все примеры в книге построены таким образом, что их проработку можно выполнять в режиме эмуляции, без подключения внешнего оборудования.

Глава 1. Программируемые контроллеры

В этой главе объясняется, что такое программируемый контроллер, как он работает и для чего вообще он нужен. Кратко будут рассмотрены принципы построения аппаратных средств и системного программного обеспечения, основы систем реального времени и важнейшие технические характеристики контроллеров.

1.1. Определение ПЛК

Любая машина, способная автоматически выполнять некоторые операции, имеет в своем составе *управляющий контроллер* — модуль, обеспечивающий логику работы устройства. Контроллер — это мозг машины. Естественно, чем сложнее логика работы машины, тем «умнее» должен быть контроллер.

Технически контроллеры реализуются по-разному. Это может быть механическое устройство, пневматический или гидравлический автомат, релейная или электронная схема или даже компьютерная программа.

В случае, когда контроллер встроен в машину массового выпуска, стоимость его проектирования распределена на большое число изделий и мала в отношении к стоимости изготовления. В случае машин, изготавливаемых в единичных экземплярах, ситуация обратная. Стоимость проектирования контроллера доминирует по отношению к стоимости его физической реализации.

При создании машин, занятых в сфере промышленного производства, как правило, приходится иметь дело не более чем с единицами однотипных устройств. Кроме того, очень существенной здесь является возможность быстрой перенастройки оборудования на выпуск другой продукции.

Контроллеры, выполненные на основе реле или микросхем с «жесткой» логикой, невозможно научить делать другую работу без существенной переделки. Очевидно, что такой возможностью обладают только *программируемые логические контроллеры* (ПЛК). Идея создания ПЛК родилась практически сразу с появлением микропроцессора, т. е. 30 лет назад.

Физически, типичный ПЛК представляет собой блок, имеющий определенный набор выходов и входов, для подключения датчиков и исполнительных механизмов (рис. 1.1). Логика управления описывается программно на основе микрокомпьютерного ядра. Абсолютно одинаковые ПЛК могут выполнять совершенно разные функции. Причем для изменения алгоритма работы не требуется каких-либо переделок аппаратной части. Аппаратная реализация входов и выходов ПЛК ориентирована на сопряжение с унифицированными приборами и мало подвержена изменениям.

Рис. 1.1. Принцип работы ПЛК

Задачей прикладного программирования ПЛК является только реализация алгоритма управления конкретной машиной. Опрос входов и выходов контроллер осуществляет автоматически, вне зависимости от способа физического соединения. Эту работу выполняет системное программное обеспечение. В идеальном случае прикладной программист совершенно не интересуется, как подсоединены и где расположены датчики и исполнительные механизмы. Мало того, его работа не зависит от того, с каким контроллером и какой фирмы он работает. Благодаря стандартизации языков программирования прикладная программа оказывается *переносимой*. Это означает, что ее можно использовать в любом ПЛК, поддерживающем данный стандарт.

Программируемый контроллер — это программно управляемый *дискретный автомат*, имеющий некоторое множество входов, подключенных посредством датчиков к объекту управления, и множество выходов, подключенных к исполнительным устройствам. ПЛК контролирует состояния входов и вырабатывает определенные последовательности программно заданных действий, отражающихся в изменении выходов.

ПЛК предназначен для работы в режиме реального времени в условиях промышленной среды и должен быть доступен для программирования неспециалистом в области информатики [6].

Изначально ПЛК предназначались для управления последовательными логическими процессами, что и обусловило слово «логический» в названии ПЛК. Современные ПЛК помимо простых логических операций способны выполнять цифровую обработку сигналов, управление приводами, регулирование, функции операторского управления и т. д. В стандарте МЭК и очень часто в литературе для обозначения контроллеров применяется сокращение ПК (программируемый контроллер). Поскольку в России обозначение ПК устойчиво связано с персональными компьютерами, мы будем использовать сокращение ПЛК.

Конструкция ПЛК может быть самой разнообразной — от стойки, заполненной аппаратурой, до миниатюрных ПЛК, подобных показанному на рис. 1.2.

Рис. 1.2. Миниатюрный ПЛК фирмы SIEMENS, Германия

Впервые ПЛК были применены в США для автоматизации конвейерного сборочного производства в автомобильной промышленности (фирма Модикон, 1969 г.). Сегодня ПЛК работают в энергетике, в области связи, в химической промышленности, в сфере добычи, транспортировки нефти и газа, в системах обеспечения безопасности, в коммунальном хозяйстве, используются в автоматизации складов, в производстве продуктов питания и напитков, на транспорте, в строительстве и т. д. Реально сфера применения ПЛК даже шире сферы применения персональных компьютеров. Хотя слава ПЛК значительно меньше. Их работа происходит как бы «за сценой» и незаметна для большинства людей.

1.2. Входы-выходы

На заре своего появления ПЛК имели только *бинарные входы*, т. е. входы, значения сигналов на которых способны принимать только два состояния — логического нуля и логической единицы. Так, наличие тока (или напряжения) в цепи входа считается обычно логической единицей. Отсутствие тока (напряжения) означает логический 0. Датчиками, формирующими такой сигнал, являются кнопки ручного управления, концевые датчики, датчики движения, контактные термометры и многие другие.

Бинарный выход также имеет два состояния — включен и выключен. Сфера применения бинарных выходов очевидна: электромагнитные реле, силовые пускатели, электромагнитные клапаны, световые сигнализаторы и т. д.

В современных ПЛК широко используются *аналоговые* входы и выходы. Аналоговый или *непрерывный* сигнал отражает уровень напряжения или тока, соответствующий некоторой физической величине в каждый момент времени. Этот уровень может относиться к температуре, давлению, весу, положению, скорости, частоте и т. д. Словом, к любой физической величине.

Аналоговые входы контроллеров могут иметь различные параметры и возможности. Так, к их параметрам относятся: разрядность АЦП, диапазон входного сигнала, время и метод преобразования, несимметричный или дифференциальный вход, уровень шума и нелинейность, возможность автоматической калибровки, программная или аппаратная регулировка коэффициента усиления, фильтрация. Особые классы аналоговых входов представляют входы, предназначенные для подключения термометров сопротивления и термопар. Здесь требуется применение специальной аппаратной поддержки (трехточечное включение, источники образцового тока, схемы компенсации холодного спая, схемы линеаризации и т. д.).

В сфере применения ПЛК бинарные входы и выходы называют обычно *дискретными*. Хотя, конечно, это не точно. Аналоговые сигналы в ПЛК обязательно преобразуются в цифровую, т. е. заведомо дискретную форму представления. Но в технических документах ПЛК любой фирмы вы встретите именно указание количества дискретных и аналоговых входов. Поэтому и далее в книге мы сохраним устоявшуюся здесь терминологию.

Помимо «классических» дискретных и аналоговых входов-выходов многие ПЛК имеют *специализированные входы-выходы*.

Они ориентированы на работу с конкретными специфическими датчиками, требующими определенных уровней сигналов, питания и специальной обработки. Например, квадратурные шифраторы, блоки управления шаговыми двигателями, интерфейсы дисплейных модулей и т. д.

Входы-выходы ПЛК не обязательно должны быть физически сосредоточены в общем корпусе с процессорным ядром. В последние годы все большую популярность приобретают технические решения, позволяющие полностью отказаться от прокладки кабелей для аналоговых цепей. Входы-выходы выполняются в виде миниатюрных модулей, расположенных в непосредственной близости от датчиков и исполнительных механизмов. Соединение подсистемы ввода-вывода с ПЛК выполняется посредством одного общего цифрового кабеля. Например, в интерфейсе Actuators/Sensors interface применяется плоский профилированный кабель («желтый кабель») для передачи данных и питания всего по двум проводам.

1.3. Режим реального времени и ограничения на применение ПЛК

Для математических систем характеристикой качества работы является правильность найденного решения. В системах реального времени помимо правильности решения определяющую роль играет *время реакции*. Логически верное решение, полученное с задержкой более допустимой, не является приемлемым.

Принято различать системы жесткого и мягкого реального времени. В системах *жесткого реального времени* существует выраженный временной порог. При его превышении наступают необратимые катастрофические последствия. В системах *мягкого реального времени* характеристики системы ухудшаются с увеличением времени управляющей реакции. Система может работать плохо или еще хуже, но ничего катастрофического при этом не происходит.

Классический подход для задач жесткого реального времени требует построения *событийно управляемой системы*. Для каждого события в системе устанавливается четко определенное время реакции и определенный приоритет. Практическая реализация таких систем сложна и всегда требует тщательной проработки и моделирования.

Для ПЛК существенное значение имеет не только быстродействие самой системы, но и время проектирования, внедрения и возможной оперативной переналадки.

Абсолютное большинство ПЛК работают по методу *периодического опроса* входных данных (сканирования). ПЛК опрашивает входы, выполняет пользовательскую программу и устанавливает необходимые значения выходов. Специфика применения ПЛК обусловливает необходимость одновременного решения нескольких задач. Прикладная программа может быть реализована в виде множества логически независимых задач, которые должны работать одновременно.

На самом деле ПЛК имеет обычно один процессор и выполняет несколько задач псевдопараллельно, последовательными порциями. Время реакции на событие оказывается зависящим от числа одновременно обрабатываемых событий. Рассчитать минимальное и максимальное значения времени реакции, конечно, можно, но добавление новых задач или увеличение объема программы приведет к увеличению времени реакции. Такая модель более подходит для систем мягкого реального времени. Современные ПЛК имеют типовое значение времени рабочего цикла, измеряемое единицами миллисекунд и менее. Поскольку время реакции большинства исполнительных устройств значительно выше, с реальными ограничениями возможности использования ПЛК по времени приходится сталкиваться редко.

В некоторых случаях ограничением служит не время реакции на событие, а обязательность его фиксации, например работа с датчиками, формирующими импульсы малой длительности. Это ограничение преодолевается специальной конструкцией входов. Так, счетный вход позволяет фиксировать и подсчитывать импульсы с периодом во много раз меньшим времени рабочего цикла ПЛК. Специализированные интеллектуальные модули в составе ПЛК позволяют автономно отрабатывать заданные функции, например модули управления сервоприводом.

1.4. Условия работы ПЛК

К негативным факторам, определяющим промышленную среду, относятся: температура и влажность, удары и вибрация, коррозионно-активная газовая среда, минеральная и металлическая пыль, электромагнитные помехи (рис. 1.3). Перечисленные факторы, весьма характерные для производственных условий, обу-

Рис. 1.3. Настройка оборудования (РААЗ, г. Рославль, Смоленская обл.)

словливают жесткие требования, определяющие схемотехнические решения, элементную и конструктивную базу ПЛК. В процессе серийного производства ПЛК обязательным является технический прогон готовых изделий, включающий климатические, вибрационные и другие испытания.

ПЛК — это конструктивно законченное изделие, физическое исполнение которого определяется требуемой степенью защиты, начиная от контроллеров в легких пластиковых корпусах, предназначенных для монтажа в шкафу (степень защиты IP20), и до герметичных устройств в литых металлических корпусах, предназначенных для работы в особо жестких условиях.

Правильно подобранный по условиям эксплуатации контроллер нельзя повредить извне без применения экстремальных методов. Штатными для ПЛК являются такие аппаратные решения, как полная гальваническая развязка входов-выходов, защита по току и напряжению, зеркальные выходные каналы, сторожевой таймер задач и микропроцессорного ядра.

1.5. Интеграция ПЛК в систему управления предприятием

Контроллеры традиционно работают в нижнем звене *автоматизированных систем управления предприятием* (АСУ) — систем, непосредственно связанных с технологией производства (ТП). ПЛК обычно являются первым шагом при построении сис-

тем АСУ. Это объясняется тем, что необходимость автоматизации отдельного механизма или установки всегда наиболее очевидна. Она дает быстрый экономический эффект, улучшает качество производства, позволяет избежать физически тяжелой и рутинной работы. Контроллеры по определению созданы именно для такой работы.

Далеко не всегда удается создать полностью автоматическую систему. Часто «общее руководство» со стороны квалифицированного человека — диспетчера необходимо. В отличие от автоматических систем управления такие системы называют *автоматизированными*. Еще 10 — 15 лет назад диспетчерский пульт управления представлял собой табло с множеством кнопок и световых индикаторов. В настоящее время подобные пульты применяются только в очень простых случаях, когда можно обойтись несколькими кнопками и индикаторами. В более «серьезных» системах применяются ПК.

Появился целый класс программного обеспечения реализующего интерфейс *человек—машина* (MMI). Это так называемые системы сбора данных и оперативного диспетчерского управления (Supervisory Control And Data Acquision System — SCADA). Современные SCADA-системы выполняются с обязательным применением средств мультимедиа. Помимо живого отображения процесса производства, хорошие диспетчерские системы позволяют накапливать полученные данные, проводят их хранение и анализ, определяют критические ситуации и производят оповещение персонала по каналам телефонной и радиосети, позволяют создавать сценарии управления (как правило, Visual Basic), формируют данные для анализа экономических характеристик производства.

Создание систем диспетчерского управления является отдельным видом бизнеса. Разделение производства ПЛК, средств программирования и диспетчерских систем привело к появлению *стандартных протоколов обмена данными*. Наибольшую известность получила технология OPC (OLE for Process Control), базирующаяся на механизме DCOM Microsoft Windows. Механизм динамического обмена данными (DDE) применяется пока еще достаточно широко, несмотря на то что требованиям систем реального времени не удовлетворяет.

Все это «многоэтажное» объяснение призвано подчеркнуть еще одно немаловажное преимущество ПЛК — средства системной интеграции являются составной частью базового программного обеспечения современного ПЛК (рис. 1.4). Допустим, вы

написали и отладили автономный проект на контроллере при помощи системы подготовки программ CoDeSys. Как теперь нужно доработать программу, чтобы связать ПЛК с системой диспетчерского управления, базой данных или Интернет-сервером? Ответ: никак. Никакого программирования далее вообще не потребуется. В комплекс программирования ПЛК входит OPC-сервер. Он умеет получать доступ к данным ПЛК также прозрачно, как и отладчик. Достаточно обеспечить канал передачи данных ПЛК — OPC-сервер. Обычно такой канал уже существует, вы использовали его при отладке. Вся дальнейшая работа сводится к определению списка доступных переменных, правильной настройке сети, конфигурированию OPC-сервера и SCADA-системы. В целом, операция очень напоминает настройку общедоступных устройств локальной сети ПК.

Второй часто возникающей задачей является интеграция нескольких ПЛК с целью синхронизации их работы. Здесь появляются сети, обладающие рядом специфических требований. В це-

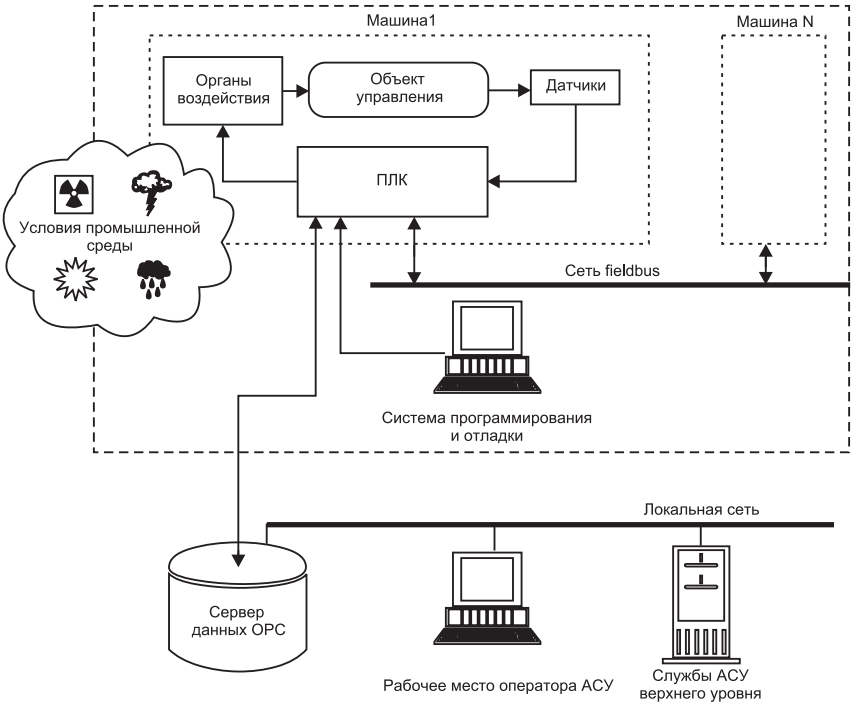

Рис. 1.4. Место ПЛК в АСУ ТП

лом это требования, аналогичные требованиям к ПЛК: режим реального времени, надежность в условиях промышленной среды, ремонтопригодность, простота программирования. Такой класс сетей получил название *промышленных сетей* (fieldbus). Существует масса фирменных реализаций и достаточно много стандартов таких сетей (Bitbus, Modbus, Profibus, CANopen, DeviceNet), позволяющих интегрировать аппаратуру различных фирм, но ни один из них нельзя признать доминирующим.

Благодаря продуктивному развитию средств сетевой интеграции появилась возможность создания *распределенных систем управления*. В 80-х гг. XX в. доминировали ПЛК с числом входов-выходов несколько сотен. В настоящее время большим спросом пользуются микроПЛК с количеством входов-выходов до 64. В распределенных системах каждый ПЛК решает локальную задачу. Задача синхронизации управления выполняется компьютерами среднего звена АСУ. Распределенные системы выигрывают по надежности, гибкости монтажа и простоте обслуживания.

1.6. Доступность программирования

Главным требованием к ПЛК всегда была и остается возможность его эксплуатации существующим техническим персоналом и возможность быстрой замены старого оборудования. Поэтому языки программирования компьютеров и встраиваемых микропроцессорных систем управления плохо подходят для программирования ПЛК. Здесь нужны более простые и наглядные языки, позволяющие излагать задачу в близких к применяемым технологиям категориях. Привлечение же к программированию специализированной фирмы неизбежно порождает зависимость, если реализация не является достаточно прозрачной. Сложный язык программирования ПЛК снижает его шансы на конкурентном рынке существенно больше, чем массогабаритные показатели.

1.7. Программный ПЛК

Программные приложения, имитирующие технологию ПЛК на компьютере (оснащенном платами ввода-вывода), получили название *программный ПЛК* (soft PLC). Программная эмуляция ПЛК удобна тем, что благодаря наличию многозадачной операционной системы можно совместить в одном месте контроллер, среду программирования и систему диспетчерского управления.

Существенный минус такого решения — большое время выхода на рабочий режим после включения питания или зависания компьютера. Особенно опасно, если перезапуск произвел «сторожевой таймер» в автоматическом режиме, в то время как состояние исполнительных механизмов не соответствует исходным позициям. Загрузка операционной системы может отнимать несколько минут, все это время система оказывается неуправляемой. Для ПЛК время «холодного» запуска измеряется миллисекундами.

Для достижения сравнимых с ПЛК технических показателей по надежности компьютер, конечно, должен быть промышленного исполнения (на базе магистралей PC/104 или VME), а не дешевый офисный «no name».

1.8. Рабочий цикл

Задачи управления требуют непрерывного циклического контроля. В любых цифровых устройствах непрерывность достигается за счет применения дискретных алгоритмов, повторяющихся через достаточно малые промежутки времени. Таким образом, вычисления в ПЛК всегда повторяются циклически. Одна итерация, включающая замер, обсчет и выработку воздействия, называется *рабочим циклом* ПЛК. Выполняемые действия зависят от значения входов контроллера, предыдущего состояния и определяются пользовательской программой.

По включению питания ПЛК выполняет самотестирование и настройку аппаратных ресурсов, очистку оперативной памяти данных (ОЗУ), контроль целостности прикладной программы пользователя. Если прикладная программа сохранена в памяти, ПЛК переходит к основной работе, которая состоит из постоянного повторения последовательности действий, входящих в *рабочий цикл*.

Рабочий цикл ПЛК состоит из нескольких фаз.

1. Начало цикла.
2. Чтение состояния входов.
3. Выполнение кода программы пользователя.
4. Запись состояния выходов.
5. Обслуживание аппаратных ресурсов ПЛК.
6. Монитор системы исполнения.
7. Контроль времени цикла.
8. Переход на начало цикла.

В самом начале цикла ПЛК производит физическое чтение входов. Считанные значения размещаются в области памяти входов. Таким образом, создается полная одномоментная зеркальная копия значений входов.

Далее выполняется код *пользовательской программы*. Пользовательская программа работает с копией значений входов и выходов, размещенной в оперативной памяти. Если прикладная программа не загружена или остановлена, то данная фаза рабочего цикла, естественно, не выполняется. Отладчик системы программирования имеет доступ к образу входов-выходов, что позволяет управлять выходами вручную и проводить исследования работы датчиков.

После выполнения пользовательского кода физические выходы ПЛК приводятся в соответствие с расчетными значениями (фаза 4).

Обслуживание аппаратных ресурсов подразумевает обеспечение работы системных таймеров, часов реального времени, оперативное самотестирование, индикацию состояния и другие аппаратно-зависимые задачи.

Монитор *системы исполнения* включает большое число функций, необходимых при отладке программы и обеспечении взаимодействия с системой программирования, сервером данных и сетью. В функции системы исполнения обычно включается: загрузка кода программы в оперативную и электрически перепрограммируемую память, управление последовательностью выполнения задач, отображение процесса выполнения программ, пошаговое выполнение, обеспечение просмотра и редактирования значений переменных, фиксация и трассировка значений переменных, контроль времени цикла и т. д.

Пользовательская программа работает только с мгновенной копией входов. Таким образом, значения входов в процессе выполнения пользовательской программы не изменяются в пределах одного рабочего цикла. Это фундаментальный принцип построения *ПЛК сканирующего типа*. Такой подход исключает неоднозначность алгоритма обработки данных в различных его ветвях. Кроме того, чтение копии значения входа из ОЗУ выполняется значительно быстрее, чем прямое чтение входа. Аппаратно чтение входа может быть связано с формированием определенных временных интервалов, передачей последовательности команд для конкретной микросхемы или даже запросом по сети.

Если заглянуть глубже, то нужно отметить, что не всегда работа по чтению входов полностью локализована в фазе чтения

входов. Например, АЦП обычно требуют определенного времени с момента запуска до считывания измеренного значения. Часть работы системное программное обеспечение контроллера выполняет по прерываниям. Грамотно реализованная система исполнения нигде и никогда не использует пустые циклы ожидания готовности аппаратуры. Для прикладного программиста все эти детали не важны. Существенно только то, что значения входов обновляются автоматически исключительно в начале каждого рабочего цикла.

Общая продолжительность рабочего цикла ПЛК называется *временем сканирования*. Время сканирования в значительной степени определяется длительностью фазы кода пользовательской программы. Время, занимаемое прочими фазами рабочего цикла, практически является величиной постоянной. Для задачи среднего объема в ПЛК с системой исполнения CoDeSys время распределится примерно так: 98% — пользовательская программа, 2% — все остальное.

Существуют задачи, в которых плавающее время цикла существенно влияет на результат, например это автоматическое регулирование. Для устранения этой проблемы в развитых ПЛК предусмотрен контроль времени цикла. Если отдельные ветви кода управляющей программы выполняются слишком быстро, в рабочий цикл добавляется искусственная задержка. Если контроль времени цикла не предусмотрен, подобные задачи приходится решать исключительно по таймерам.

1.9. Время реакции

Время реакции — это время с момента изменения состояния системы до момента выработки соответствующей реакции. Очевидно, для ПЛК время реакции зависит от распределения моментов возникновения события и начала фазы чтения входов. Если изменение значений входов произошло непосредственно перед фазой чтения входов, то время реакции будет наименьшим и равным времени сканирования (рис. 1.5). Худший случай, когда изменение значений входов происходит сразу после фазы чтения входов. Тогда время реакции будет наибольшим, равным удвоенному времени сканирования минус время одного чтения входов. Иными словами, время реакции ПЛК не превышает удвоенного времени сканирования.

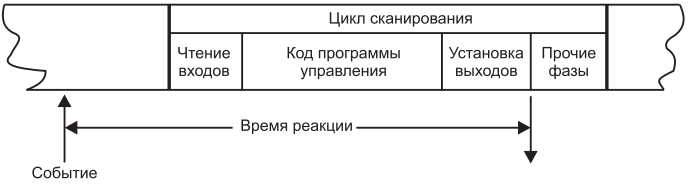

Рис. 1.5. Время реакции ПЛК

Помимо времени реакции ПЛК, существенное значение имеет время реакции датчиков и исполнительных механизмов, которое также необходимо учитывать при оценке общего времени реакции системы.

Существуют ПЛК, которые реализуют команды непосредственного доступа к аппаратуре входов и выходов, что позволяет обрабатывать и формировать отдельные сигналы с длительностью меньшей длительности рабочего цикла.

Для уменьшения времени реакции сканирующих контроллеров алгоритм программы разбивается на несколько задач с различным периодом исполнения. В наиболее развитых системах пользователь имеет возможность создавать отдельные программы, исполняемые по прерыванию, помимо кода, исполняемого в рабочем цикле. Такая техника позволяет ПЛК существенно форсировать ограничение реакции временем сканирования при небольшом количестве входов, требующих сверхскоростной реакции.

Время цикла сканирования является базовым показателем быстродействия ПЛК. При измерении времени рабочего цикла пользовательская программа должна содержать 1К логических команд. Для ПЛК, поддерживающих стандарт МЭК 61131-3, используют команды на языке IL. Иногда изготовители приводят несколько значений времени цикла, полученных при работе с переменными различной разрядности.

Ориентировочно о скорости обработки различных типов данных можно судить по тактовой частоте и разрядности центрального процессора. Хотя нет ничего удивительного в том, что восьмиразрядные ПЛК не редко оказываются быстрее 32-разрядных при выполнении битовых операций. Объясняется это тем, что в 8-разрядных микропроцессорах более распространена аппаратная поддержка работы с битами. Так, в PC-совместимых процессорах для выделения бита приходится использовать логические команды и циклический сдвиг.

1.10. Устройство ПЛК

Аппаратно ПЛК является вычислительной машиной. Поэтому архитектура его процессорного ядра практически не отличается от архитектуры компьютера. Отличия заключены в составе периферийного оборудования, отсутствуют видеоплата, средства ручного ввода и дисковая подсистема. Вместо них ПЛК имеет блоки входов и выходов.

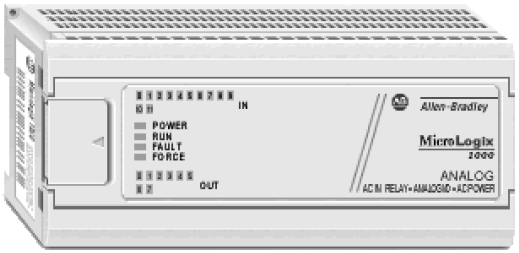

Рис. 1.6. ПЛК MicroLogix 1000 производства Allen-Bradley (Rockwell Automation), США

Конструктивно контроллеры подразделяют на моноблочные, модульные и распределенные. Моноблочные, или одноплатные, ПЛК имеют фиксированный набор входов-выходов. В модульных контроллерах (рис. 1.7) модули входов-выходов устанавливаются в разном составе и количестве в зависимости от требуемой конфигурации. Так достигается минимальная аппаратная избыточность. В распределенных системах модули или даже отдельные входы-выходы, образующие единую систему управления, могут быть разнесены на значительные расстояния.

Характерным для современных контроллеров является использование многопроцессорных решений. В этом случае модули ввода-вывода имеют собственные микропроцессоры, выполняющие необходимую предварительную обработку данных. Модуль центрального процессора имеет выделенную скоростную магистраль данных для работы с памятью и отдельную магистраль (сеть) для общения с модулями ввода-вывода.

Еще одним вариантом построения ПЛК является *мезонинная технология*. Все силовые цепи, устройства защиты контроллера выполняются на несущей плате. Процессорное ядро контроллера, включающее систему исполнения, выполнено на отдельной сменной (мезонинной) плате. В результате появляется возможность со-

Рис. 1.7. Модульная система МСТС производства ПК «Пролог», Россия

ставлять несколько комбинаций процессорного ядра и разных силовых плат без необходимости корректировки программного обеспечения. При необходимости процессор можно заменить даже в готовой системе.

На рис. 1.8 показано сверхминиатюрное ядро ПЛК Easy 215 (фирма Frenzel + Berg Electronic, Германия), выполненное в виде модуля, рассчитанного на установку в стандартную колодку микросхем, имеющих корпус DIP40. Несмотря на скромные размеры, модуль включает 16-разрядный процессор (Infineon C164), память данных, энергонезависимую память программ, встроенное ядро системы исполнения CoDeSys, 8 дискретных

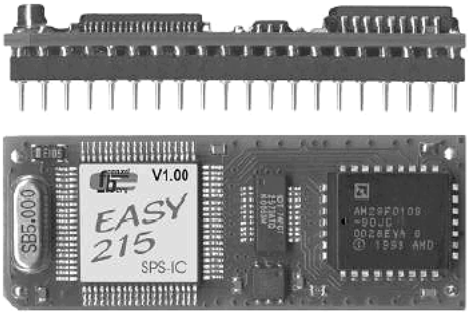

Рис. 1.8. Сверхминиатюрный ПЛК Easy215 с системой исполнения CoDeSys

входов и 8 выходов, 4 аналоговых входа, 2 парных входа квадратурных шифраторов, интерфейсы RS232, SPI и обеспечивает работу сети CANopen.

1.10.1. Системное и прикладное программное обеспечение

Системное программное обеспечение (СПО) непосредственно контролирует аппаратные средства ПЛК. СПО отвечает за тестирование и индикацию работы памяти, источника питания, модулей ввода-вывода и интерфейсов, таймеров и часов реального времени. Система исполнения кода прикладной программы является составной частью СПО. Система исполнения включает драйверы модулей ввода-вывода, загрузчик кода программ пользователя, интерпретатор команд и отладочный монитор. Код СПО расположен в ПЗУ и может быть изменен только изготовителем ПЛК.

Код прикладной программы размещается в энергонезависимой памяти, чаще всего это электрически перепрограммируемые микросхемы. Изменение кода прикладной программы выполняется пользователем ПЛК при помощи системы программирования и может быть выполнено многократно.

1.10.2. Контроль времени рабочего цикла

Правильно составленная пользовательская программа не должна содержать бесконечных циклов. В противном случае управление системе исполнения не будет передано, и, соответственно, нормальное функционирование контроллера будет нарушено. Для преодоления данной проблемы служит *контроль времени цикла*. Контроль осуществляется при поддержке аппаратно реализованного «сторожевого таймера». Если фаза пользовательского кода выполняется дольше установленного порога, то ее работа будет прервана. Таким образом, достигается предсказуемое поведение ПЛК при ошибках в программе и при «зависании» по причине аппаратных сбоев.

Обслуживание *сторожевого таймера* выполняется в рабочем цикле ПЛК. Выполнять эту операцию по прерыванию нельзя, поскольку при «зависании» процессора система прерываний достаточно часто продолжает исправно работать.

Глава 2. Стандарт МЭК 61131

В данной главе раскрыты причины, приведшие к необходимости стандартизации ПЛК, кратко описана структура МЭК 61131, обоснован выбор языков программирования контроллеров включенных в стандарт МЭК 61131-3.

2.1. Открытые системы

Привязать потребителей к своим изделиям — мечта любого производства. С этой целью секреты технологии, позволяющие получить высокие качественные показатели продукции, должны тщательно охраняться. Это классический *закрытый подход*. Для потребителя желательно иметь возможность совместно использовать изделия разных фирм. Но с точки зрения производителя это не выгодно, поскольку повышает вероятность того, что заказчик откажется от некоторых покупок в пользу конкурентов.

На самом деле это поверхностное заключение справедливо не всегда. Так, если продукция является достаточно технически сложной и имеет широкую сферу применений, то удовлетворить каждое конкретное пожелание индивидуального заказчика почти невозможно и дорого. Недовольство даже одной малосущественной для большинства деталью может привести к отказу от продукции данной фирмы вообще. При производстве же совместимой продукции (подчиненной требованиям открытого стандарта) фирма производитель может сконцентрироваться на развитии наиболее удачных своих решений. Не опасаясь потерять заказчика, производитель может отказаться от невыгодных для себя изделий или частей работы. Кроме того, благодаря совместимости появляется возможность внедрять свои передовые изделия даже в полностью захваченных областях рынка. Так начинающие коллективы получают шанс проявить себя и найти свое, пусть даже не очень большое место среди промышленных гигантов. Тем самым расширяется и сам рынок. Выгоду открытого подхода наглядно доказала фирма IBM на примере своих ПК.

В 1979 году в рамках Международной Электротехнической Комиссии (МЭК) была создана специальная группа технических экс-

пертов по проблемам ПЛК, включая аппаратные средства, монтаж, тестирование, документацию и связь.

Первый вариант стандарта был опубликован в 1982 году. Ввиду сложности получившегося документа было решено разбить его на несколько частей. В настоящее время стандарт включает следующие части.

Часть 1. Общая информация.

Часть 2. Требования к оборудованию и тестам.

Часть 3. Языки программирования.

Часть 4. Руководства пользователя.

Часть 5. Спецификация сообщений.

Часть 6. Промышленные сети.

Часть 7. Программирование с нечеткой логикой.

Часть 8. Руководящие принципы применения и реализации языков ПЛК.

Первоначально стандарт имел номер 1131, с 1997 года МЭК перешел на 5-цифровые обозначения. Теперь правильное наименование международной версии стандарта — МЭК 61131.

Далее мы сосредоточимся главным образом на языках программирования, описанных в третьей части стандарта. Для краткости, если в тексте употребляются слова «стандарт МЭК», следует понимать МЭК 61131-3. При ссылках на другие документы будет дано полное наименование.

2.2. Целесообразность выбора языков МЭК

Если посмотреть на языки стандарта МЭК с точки зрения современной информатики, то каждый их них можно подвергнуть оправданной критике (особенно SFC). Вероятно, было бы более разумным, опираясь на опыт использования наиболее популярных языков, создать один хороший универсальный язык. Эта идея не нова. Все старое программное обеспечение для контроллеров просто нужно будет переписать с нуля. В условиях конкурентного производства очень важно проводить внедрение новой техники быстро. А для этого необходимо максимально задействовать отработанные решения.

Включение в стандарт пяти языков объясняется в первую очередь историческими причинами. Разработчики стандарта столкнулись с наличием огромного количества различных вариаций похожих языков программирования ПЛК. Вошедшие в стандарт языки

созданы на основе наиболее популярных языков программирования, наиболее распространенных в мире контроллеров. Если взять любой контроллер, работающий в современном производстве, то его программу можно перенести в среду МЭК 61131-3 с минимальными затратами. Речь не идет о том, что программу можно будет использовать без какой-либо правки. Безусловно, потребуется некоторая адаптация и отладка, но несравненно меньшая, чем при создании проекта с нуля.

После принятия стандарта появилась возможность создания *аппаратно-независимых библиотек*. Это регуляторы, фильтры, управление сервоприводом, модули с нечеткой логикой и т. д. Наиболее удачные, отработанные востребованные библиотеки становятся коммерческими продуктами.

2.3. Простота программирования и доходчивое представление

Инженер, спроектировавший машину, должен иметь возможность самостоятельно написать программу управления. Никто лучше его не знает, как должна работать данная машина. Инженер, привыкший работать с электронными схемами, гораздо легче сможет выражать свои мысли в LD или FBD. Если он знаком с языками PASCAL или C, то использование языка ST не составит для него сложности.

За время развития ПЛК размер средней программы возрос более чем в 100 раз [21]. Многие решения, требовавшие раньше аппаратной поддержки, реализуются сегодня программно. Соответственно, требования к качеству программного обеспечения очень высоки. Поэтому сложную программу должны писать специалисты. Но для ответственных проектов очень важно, чтобы программа алгоритма была понятна техническому персоналу, осуществляющему настройку, сопровождение и ремонт оборудования. Они не обязаны изучать программу досконально, но понимать, что происходит, безусловно, должны.

Очень часто технологи описывают процесс примерно так: «слегка перемешать, подогреть и довести до готовности». С аппаратными средствами здесь фактически все понятно, а вот с алгоритмом управления значительно сложнее. Для более детальных обсуждений технологии необходим некий общий язык, документальный и наглядный. Диаграммы SFC справляются с этой ролью не хуже специализированных инструментов (например, UML), являясь притом действующей программой, а не просто моделью.

Страшно подумать, что придется объяснять работу сложной программы по ассемблерным или С листингам. Не исключайте ситуации, что общаться придется не на родном языке. Современные системы программирования контроллеров позволяют выполнить несколько распечаток программы с комментариями на разных языках — русском, немецком и т. д. Очевидно, это уже не маркетинговый ход разработчиков, а реальное требование современного бизнеса. Неэффективно реализованную программу можно заставить работать быстрее увеличением быстродействия процессора. Доходчивости представления достичь гораздо сложнее. Программу, в которой невозможно разобраться, придется рано или поздно выбросить.

2.4. Единые требования в подготовке специалистов

Внедрение стандарта дало фундамент для создания единой школы подготовки специалистов. Человек, прошедший обучение по программе, включающей стандарт МЭК 61131, сможет работать с ПЛК любой фирмы. В то же время, если он имел ранее опыт работы с любыми ПЛК, его навыки окажутся полезными и существенно упростят изучение новых возможностей.

Программист не ограничен применением заданных в стандарте типов данных и операций. Стандарт допускает возможность создания пользовательских типов данных и функциональных блоков. Функции и функциональные блоки великолепно реализуют инкапсуляцию деталей реализации. Созданные пользователем библиотеки абсолютно равноправны стандартным. Новые оригинальные аппаратные решения изготовителей ПЛК могут быть поддержаны собственными библиотеками. Причем при создании внешних библиотек можно использовать любые инструменты вплоть от ассемблера до C++.

Вообще стандартные компоненты МЭК для программиста, как дороги для автомашин. Количество возможных путей всегда очень ограничено. Ближе полем, но по дороге быстрее.

Глава 3. Инструменты программирования ПЛК

В этой главе рассказывается о наиболее известных комплексах программирования МЭК 61131-3, описываются общие их свойства, инструменты проектирования и отладки прикладного программного обеспечения ПЛК. Более подробно рассмотрен комплекс CoDeSys, на который будет опираться дальнейшее изложение.

3.1. Комплексы проектирования МЭК 61131-3

Контроллеры, программирование которых осуществляется со встроенного или выносного пульта, встречаются сегодня достаточно редко. Как правило, это простые специализированные ПЛК, предназначенные для управления освещением по расписанию, регулировки температуры и т. д. Все программирование таких контроллеров сводится обычно к заданию набора констант. Для программирования ПЛК универсального назначения применяются ПК. Процесс разработки и отладки программного обеспечения происходит при помощи специализированных комплексов программ, обеспечивающих комфортную среду для работы программиста.

Традиционно все ведущие изготовители программируемых ПЛК имеют собственные фирменные наработки в области инструментального программного обеспечения. Безусловно, большинство из них представляют удобные инструменты, оптимизированные под конкретную аппаратуру. Понятно, что в разработке универсальных систем программирования, приемлемых для своих ПЛК и для ПЛК конкурентов, изготовители не заинтересованы. Кроме того, это достаточно сложная задача. Системы программирования ПЛК небольших фирм в лучшем случае реализуют один из языков МЭК с некоторыми расширениями, призванными сохранить совместимость со своими же более ранними (нестандартными) системами. Крупнейшие лидеры рынка ПЛК предлагают сегодня очень мощные комплексы с поддержкой МЭК-языков, также сохраняющие преемственность и фирменные традиции («Concept» Schneider Electric, «S7» Siemens).

Открытость МЭК-стандарта — с одной стороны, и сложность реализации высококлассных комплексов программирования — с другой, привели к появлению специализированных фирм, занятых исключительно инструментами программирования ПЛК. Во Франции такие фирмы называют «*дом программирования*». Как и изделия домов мод системы программирования отличаются своим фирменным почерком, имеют свой стиль и собственных стойких поклонников. Но, к счастью, отличия комплексов сосредоточены в реализации интерфейса, в стиле графики, наборе сервисных функций, дополнительных библиотеках и в реализации системы исполнения, т. е. в том, что не касается применения стандарта.

Наибольшей известностью в мире пользуются следующие комплексы.

CoDeSys

3S Smart Software Solutions **http://www.3s-software.com**

CoDeSys это один из самых развитых функционально полных инструментов программирования МЭК 61131-3. Все дальнейшее изложение в данной книге опирается на CoDeSys. Это не означает, что приведенная информация непригодна для других систем программирования. Везде, где есть существенные отличия или тонкости реализации, это будет особо подчеркнуто. Опора на CoDeSys означает, что все примеры реализованы и протестированы именно в этой системе, если на это указано специально. Далее мы рассмотрим данный комплекс более подробно.

ISaGRAF

CJ International **http://www.isagraf.com/**

Наиболее яркая особенность ISaGRAF — это аппаратно независимый генератор TIC кода (**T**arget **I**ndependent **C**ode), благодаря чему, система исполнения интерпретирующего типа очень проста в адаптации. Какие-либо ограничения на аппаратную платформу практически отсутствуют. Помимо генерации TIC-кода, в ISaGRAF существует возможность трансляции проекта в C текст.

MULTIPROG wt

Klöpper und Wiege Software GmbH **http://www.kw-software.de/**

Исключительно продуманный, красивый и удобный инструмент с широкими возможностями моделирования и визуализации. Система исполнения базируется на собственной операцион-

ной системе реального времени ProConOS (Programmable Controller Operating System), управляющей исполнением пользовательских задач. Первая версия MULTIPROG вышла еще в начале 80-х гг. XX в. под операционную систему CP/M. В настоящее время MULTIPROG ориентирован на Windows, о чем говорит суффикс wt (**w**indows **t**echnology) в названии.

OpenPCS

Infoteam Software GmbH **http://www.infoteam.de/**

Уникальная особенность комплекса OpenPCS заключается в использовании языка IL в качестве промежуточного кода. Элементы программы, выполненные на любом МЭК-языке, можно копировать в буфер обмена Windows и вставлять в программу на другом языке с автоматическим перекодированием. Для достижения высокого быстродействия в составе комплекса присутствуют компиляторы машинного кода для ряда распространенных процессоров. Симулятор ПЛК SmartSIM позволяет проводить обучение и отладку без внешней аппаратуры.

SoftCONTROL

Softing GmbH **http://www.softing.com/**

Фирма Softing — крупный поставщик систем промышленной автоматики. По всей видимости, комплекс SoftCONTROL создавался специалистами фирмы для себя, но благодаря удачному построению перерос в универсальный инструмент. Комплекс имеет сравнительно аскетический интерфейс. Тем не менее, это не отражается на качестве реализации транслятора и отладочного инструментария. Таким образом, SoftCONTROL более напоминает выверенный годами инструментарий опытного автомеханика, чем подарочный набор. Благодаря такому подходу комплекс имеет минимальные требования как к ПК, так и к ПЛК. Язык C интегрирован в систему и может применяться в пользовательских программах равноправно МЭК-языкам.

iCon-L

ProSign (Process Design) GmbH **http://www.pro-sign.de/**

Строго говоря, iCon-L не является инструментом МЭК 1131 программирования. Этот инструмент базируется на графическом представлении функциональных блоков. Содержит элементы, позволяющие создавать последовательные (SFC) диаграммы. Уникальным свойством iCon-L является чрезвычайно развитая возмож-

ность анимации. Непосредственно в диаграмме можно выполнять визуализацию не только самого алгоритма, переменных, контроллера, но и даже управляемого объекта (см. рис. 3.8). Компактная переносимая (ANSI-C) система исполнения. Объемная библиотека блоков, включая элементы нечеткой логики. Есть возможность создавать собственные функциональные блоки на языке С.

Каждый из представленных комплексов оснащен полным набором средств быстрой разработки и отладки программ, но имеет и достаточно много фирменных «изюминок». Все комплексы имеют демонстрационные версии, содержащие много полезных примеров. Вы можете также проводить собственные опыты в программировании и проводить тестирование в режиме эмуляции. Естественно, использовать ознакомительные версии при создании коммерческих проектов нельзя.

Детальное использование экранных интерфейсов и меню команд комплексов в книге описываться не будет. Предполагается, что читатель знаком с «прелестями» Windows-интерфейса. Приемы редактирования программ и способы ввода команд относятся к индивидуальным характеристикам и достаточно подробно изложены во всех без исключения руководствах по применению и оперативных подсказках комплексов, чего, к сожалению, нельзя сказать о смысле и приемах применения самих команд. Поэтому далее мы рассмотрим более подробно инструментарий и наиболее яркие общие характеристики комплексов, позволившие им занять первые позиции на рынке программного обеспечения ПЛК.

3.2. Инструменты комплексов программирования ПЛК

Главная задача инструментов комплекса программирования ПЛК состоит в автоматизации работы разработчика прикладной системы. Он должен быть избавлен от рутинной работы и постоянного «изобретения велосипеда». Хорошо организованная среда программирования сама толкает к созданию надежного, читабельного и пригодного для повторного применения кода.

В интегрированных комплексах программирования ПЛК сложился определенный набор возможностей, позволяющий относить их к средствам быстрой разработки. Многие приемы являются общими и для систем программирования компьютеров и, вероятно, покажутся вам знакомыми. Сервисные функции систем программирования не являются требованием стандарта. Но от

полноты набора доступных программисту инструментов существенно зависит скорость и качество его работы.

3.2.1. Встроенные редакторы

Классические (с командной строкой) ассемблеры и компиляторы обрабатывают текст файла, содержащего программный модуль, и формируют объектный код. Исходный текст программы записывается в любом текстовом редакторе. Интегрированная среда предполагает наличие встроенного редактора.

3.2.2. Текстовые редакторы

Интеграция в единую среду программирования предполагает наличие у текстовых редакторов нескольких существенных свойств:

- возможность быстрого ввода стандартных текстовых элементов. Комбинации клавиш быстрого ввода, или контекстно-зависимые меню команд, предлагают мгновенную вставку в текст операторов, функций, функциональных блоков (см. рис. 3.1). Причем речь идет не только о стандартных элементах, но и о созданных программистом в текущем проекте;
- возможность быстрого автоматического дополнения ввода (CoDeSys). Например, строка: «INP1 I 3;Вход 1» по окончании ввода преобразуется в соответствии с требованиями МЭК:

  «INP1: **INT** := 3; (* Вход 1 *)»;

- автоматическое объявление переменных. Если при вводе текста программы вы используете новую переменную, система автоматически поместит необходимое описание в разделе объявлений. Тип переменной и начальное значение задаются в диалоговом окне. В этом помогают меню, весь ввод обычно выполняется мышью, без помощи клавиатуры;
- представление раздела объявлений переменных в виде текста или картотеки таблиц, разделенных и отсортированных по функциональному значению (входные переменные, локальные и т. д.);
- проверка синтаксиса и автоматическое форматирование ввода. Редактор автоматически контролирует введенный текст и выделяет цветом ключевые слова, константы и комментарии. В результате текст не только легко читается, но и оказывается синтаксически проверенным еще до трансляции;

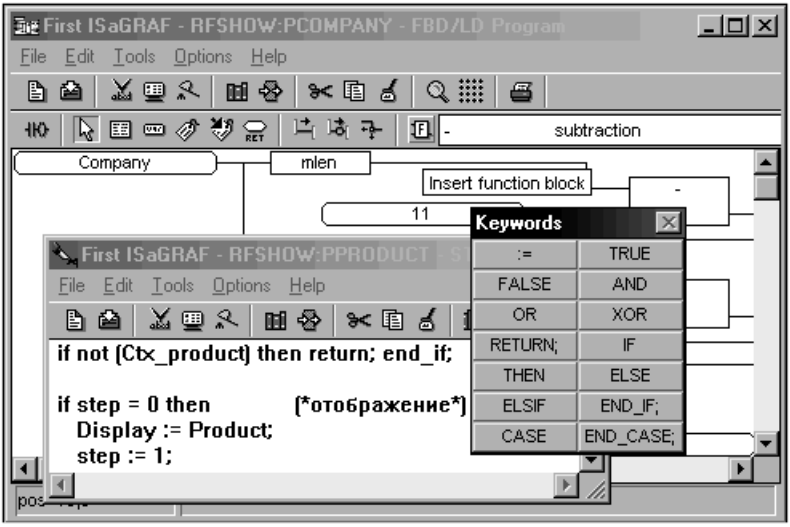

Рис. 3.1. Панель быстрого ввода ISaGRAF

- автоматическая нумерация строк — упрощает описание и сопровождение.

Эти возможности в существенной мере способствуют автоматизации процесса подготовки программ и способствуют уменьшению числа ошибок в программах.

3.2.3. Графические редакторы

Графические редакторы еще более тесно связаны с контекстом конкретных языков. Они должны обеспечивать следующие возможности:

- автоматическая трассировка соединений компонентов. Программисту вообще не приходится рисовать соединения. При вставке и удалении компонентов система автоматически проводит графические соединительные линии (см. рис. 3.2);
- автоматическая расстановка компонентов. Местоположение компонента на экране определяется автоматически с учетом порядка выполнения. Этим свойством обладают графические редакторы CoDeSys и OpenPCS. В других представленных в книге комплексах программист выбирает местоположение компонента вручную, координаты компонента сохраняются при записи проекта (см. рис. 3.3). Команда индикации по-

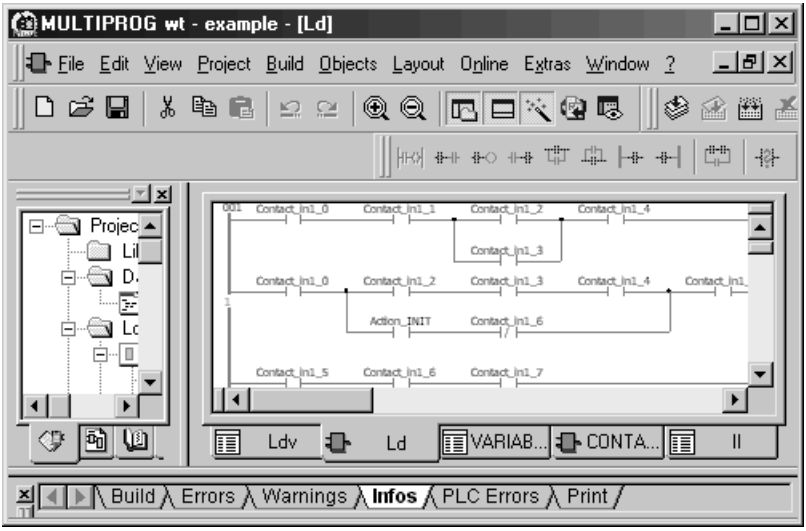

Рис. 3.2. Графический редактор релейных схем и панель ввода MULTIPROG

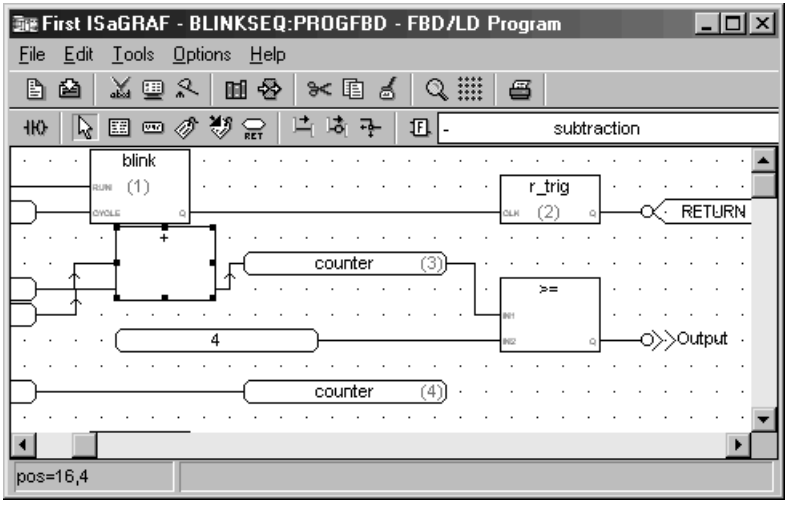

Рис. 3.3. Ручное размещение компонента (ISAGRAF)

рядка выполнения добавляет в изображение компонента порядковый номер (на рис. 3.3, числа в скобках).

- автоматическая нумерация цепей;
- копирование и перемещение выделенной графической группы компонентов с учетом их индивидуальной специфики (рис. 3.4);

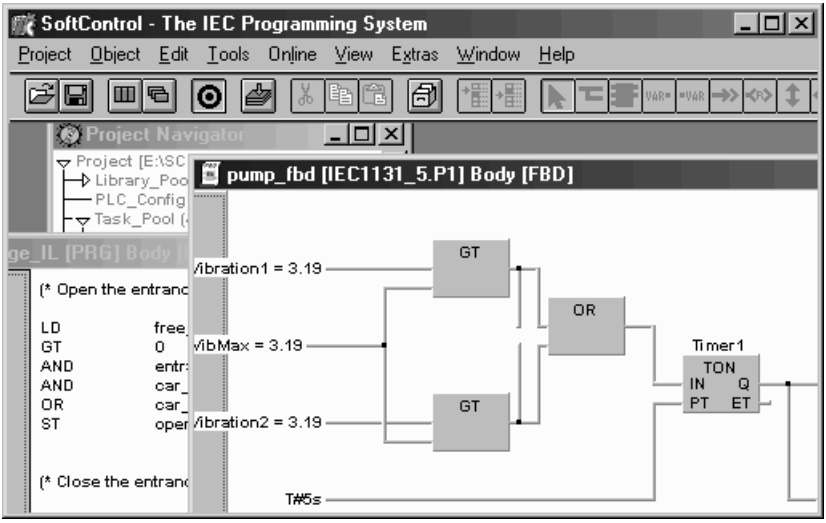

Рис. 3.4. Графический редактор функциональных блоковых диаграмм SoftCONTROL

- произвольное масштабирование изображения с целью наилучшего представления или отдельное окно общего вида. Для анализа больших разветвленных графических диаграмм удобно иметь возможность увидеть всю диаграмму или достаточно релевантную ее часть целиком (см. рис. 3.5).

В режиме исполнения встроенные редакторы отображают «ожившие» тексты и графические диаграммы (рис. 3.6). При этом:

- мгновенные значения переменных видны непосредственно в окне редактора и доступны для изменения;
- активные цепи выделены жирными линиями и цветом. Для графических диаграмм наглядно отражается последовательность выполнения.

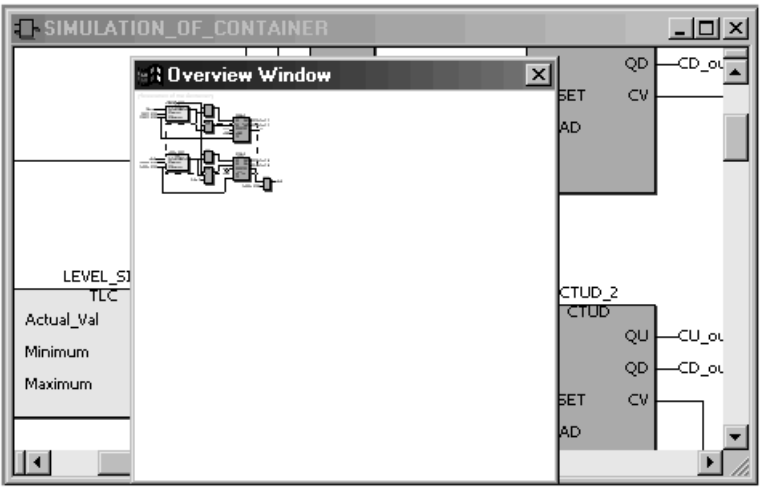

Рис. 3.5. Окна графического редактора FBD и общего обзора диаграммы (MULTIPROG)

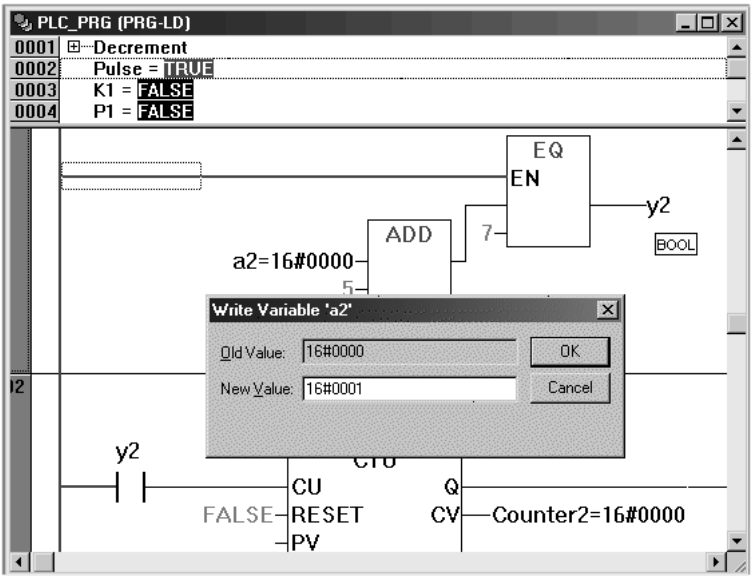

Рис. 3.6. LD-диаграмма в процессе исполнения и диалог изменения значения переменной (CoDeSys)

3.2.4. Средства отладки

Стандартный набор отладочных функций включает в себя:

- унифицированный механизм соединения с ПЛК. Работа инструментов отладки не зависит от способа соединения контроллера с отладчиком. Не имеет значения, эмулируется ли контроллер на том же самом компьютере (рис. 3.7), подключен ли через последовательный порт ПК или даже расположен в другой стране и связан через Интернет;

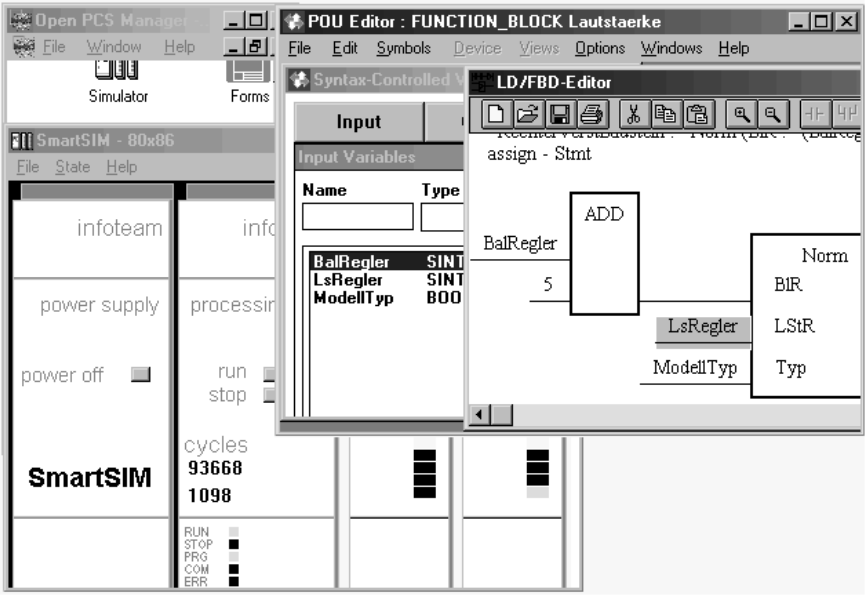

Рис. 3.7. Эмуляция ПЛК в OpenPCS

- загрузку кода управляющей программы в оперативную память и электрически перепрограммируемую память ПЛК;
- автоматический контроль версий кода. Проверка соответствия кода содержащегося в памяти ПЛК и кода полученного после текущей компиляции;
- выполнение управляющей программы в режиме реального времени;
- режим останова. Останов означает прекращение выполнения только кода управляющей программы. Все прочие фазы ра-

бочего цикла выполняются. Способность наблюдать значения входов и управлять выходами ПЛК вручную сохраняется. В этом режиме можно проводить тестирование и настройку датчиков и механизмов объекта управления;

- сброс ПЛК. Может быть несколько видов сброса. В стандарте МЭК предусмотрено два вида сброса «горячий» и «холодный». Первый включает перевод управляющей программы в исходное состояние и выполнение начальной инициализации переменных. Во втором виде сброса добавляется начальная инициализация переменных, размещенных в энергонезависимой области памяти. В CoDeSys предусмотрен еще и «заводской» сброс (original), удаляющий пользовательскую программу и восстанавливающий состояние контроллера, в котором он поступает с завода изготовителя. Кроме того, в ПЛК может произойти аппаратный сброс путем выключения питания или перезапуска микропроцессора. Система программирования должна адекватно реагировать в случае аппаратного сброса. Детальная реакция на команды сброса определяется системой исполнения. Поэтому здесь возможны некоторые отличия для разных ПЛК, даже в одной среде программирования;
- мониторинг и изменение мгновенных значений всех переменных проекта, включая входы-выходы ПЛК. Для удобства работы значения представляются в заданной пользователем системе счисления;
- фиксацию переменных, включая входы-выходы. Фиксированные переменные будут получать заданные значения в каждом рабочем цикле независимо от реального состояния ПЛК и действий управляющей программы. Данная функция позволяет имитировать элементарные внешние события в лабораторных условиях и избегать нежелательной работы исполнительных механизмов при отладке на «живом» объекте управления. Неуправляемая работа механизмов может привести к поломке и представлять опасность для окружающих людей;
- выполнение управляющей программы шагами по одному рабочему циклу. Применяется при проверке логической правильности алгоритма;
- пошаговое выполнение команд программы и задание точек останова;

- просмотр последовательности вызовов компонентов в точке останова;
- графическую трассировку переменных. Значения нужных переменных запоминаются в циклическом буфере и представляются на экране ПК в виде графиков. Запись значений можно выполнять в конце каждого рабочего цикла либо через заданные периоды времени. Трассировка запускается вручную или синхронизируется с заданным изменением значения определенной (триггерной) переменной;
- визуализацию — анимационные картинки, составленные из графических примитивов, связанных с переменными программы. Значение переменной может определять координаты, размер или цвет графического объекта. Графические объекты включают векторные геометрические фигуры или произвольные растровые изображения. Визуализация может содержать элементы обратной связи, например кнопки, ползунки и т. д. (см. рис. 3.8, 3.9). С помощью визуализации создается изображение, моделирующее объект управления или систему операторского управления.

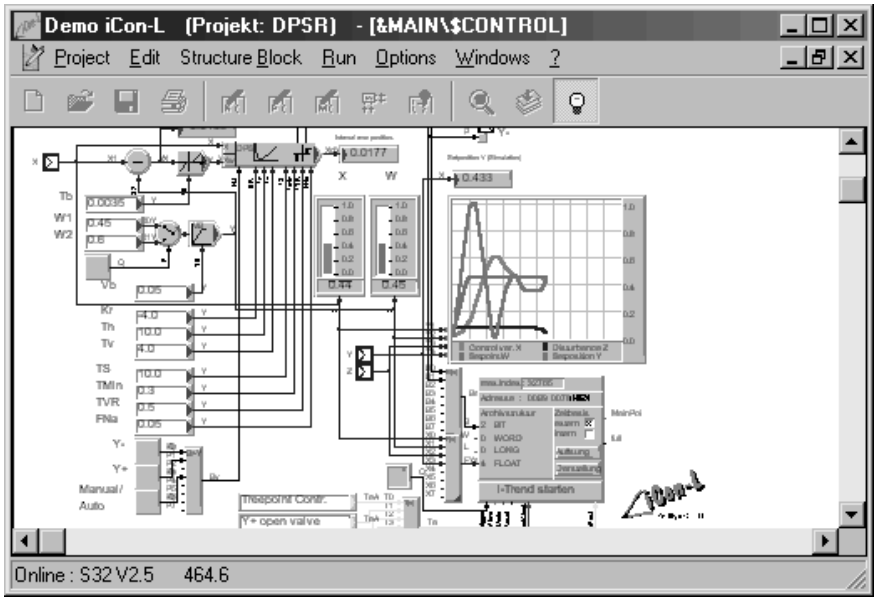

Рис. 3.8. Визуализация в iCon-L

3.2.5. Средства управления проектом

Все программные комплексы обязательно содержат средства управления проектом. Эту задачу решает менеджер проекта, в обязанности которого входит:

- представление всех элементов проекта и общей его структуры в удобном виде (см. рис. 5.10). Создание, удаление, переименование и копирование компонентов. Автоматический вызов соответствующих редакторов для любой глубины вложения программных компонентов. Настройка ресурсов;
- управление процессом трансляции и сборки кода. Настройка опций транслятора;
- сравнение и выборочное слияние нескольких проектов или их версий;
- управление библиотеками. Здесь существуют две задачи. Первая — это включение необходимых библиотек в состав проекта, а вторая — это создание и сопровождение новых библиотек;
- документирование проекта. Документирование проекта в комплексах МЭК-программирования предусматривает распечатку всех данных проекта, включая:
  - ♦ текстовое описание, дата создания и авторские права;
  - ♦ описание переменных и реализацию всех компонентов проекта;
  - ♦ ресурсы проекта — конфигурацию ПЛК, описание глобальных переменных, настройки задач, список и состав библиотечных модулей;
  - ♦ таблицу перекрестных ссылок и стек вызовов;
  - ♦ окно трассировки.

Естественно, нельзя ожидать от системы программирования полного комплекта документации в соответствии с требованиями ЕСКД. Под словами «полная документация» в руководстве по применению системы понимается только то, что по данному печатному документу можно полностью и однозначно восстановить проект.

CoDeSys позволяет составить специальные файлы комментариев на разных языках (русский, английский и т. д.). Благодаря этому можно распечатать несколько разноязычных вариантов документации одного и того же проекта без изменения в тексте программ. Кроме того, система предоставляет возможность настройки формата страницы документа, включая колонтитулы с вашим фирменным логотипом.

Средства восстановления проекта. В реальной жизни нельзя исключать ситуацию, что исходные файлы проекта окажутся утраченными. В это время обязательно возникнет необходимость внести поправки в работу готовой программы. Эта задача имеет три решения.

1. Декомпиляция кода. Исполняемый код считывается из памяти ПЛК и преобразуется в МЭК-программы. Для систем генерирующих машинный код эта задача практически невыполнима. Безусловно, можно дизассемблировать код в IL или ST. Но это ненамного лучше, чем обычное машинно-зависимое дизассемблирование. Структура программы получится отличной от исходного представления. Как правило, разобраться в такой программе сложнее, чем написать заново. Для интерпретирующих систем ситуация значительно лучше. Так, OpenPCS способен восстановить программу из исполняемого кода IL абсолютно адекватно, естественно, с потерей комментариев. Декомпиляция — это крайняя мера. Важное практическое значение она имела во времена преобладания автономных пультов программирования ПЛК и при отсутствии надежных устройств хранения информации.

2. Сжатие всех файлов проекта и сохранение в памяти ПЛК. Современные мощные алгоритмы компрессии и существенное удешевление памяти делают такой подход все более популярным (MULTIPROG, CoDeSys). Безусловно, при наличии достаточного объема памяти ПЛК это наиболее удобный способ архивации.

3. Правильная организация работы. В комплекс разработчика включается утилита для периодической архивации проектов и сохранения на сервере, сменных носителях, в печатном виде и отправки по электронной почте. В архив помещаются исходные файлы, включенные в проект библиотеки, объектные файлы, текстовое описание архива и любые другие нужные файлы. Промежуточные версии проекта не перезаписываются, а хранятся независимо, что позволяет осуществить быстрый откат при выборе неудачного решения. В связи с появлением накопителей большой емкости и надежных перезаписываемых оптических носителей такой подход не имеет технических препятствий.

Средства обеспечения безопасности. Возможность просмотра и модификации проекта закрывается парольным доступом или аппаратным ключом. Посторонний человек не должен иметь возможности читать, распечатывать и модифицировать проект.

Сквозной (по всем программам проекта, разделам объявлений, конфигурации и др.) ***контекстный поиск и замена.***

Средства тестирования «разумности» проекта. Вспомогательные средства, позволяющие отыскать странные и потенциально опасные моменты в программах. Например, объявленные, но не использованные переменные, использование одной области памяти разными переменными или в разных параллельных задачах, присваивание разных значений выходу ПЛК в одном рабочем цикле и т. д. Подобные «трюки» сами по себе не являются ошибками. Но они часто приводят к сложно обнаруживаемым паразитным эффектам. Средства тестирования помогают отыскать тонкие места в программах, не создавая препятствий там, где эти приемы применены осмысленно.

Средства импорта и экспорта проектов в другие комплексы программирования.

Перечисленные выше средства управления проектами позволяют создавать высококачественные проекты с минимумом затрат времени на это.

3.3. Комплекс CoDeSys

Комплекс CoDeSys разработан фирмой 3S (Smart Software Solutions). Это универсальный инструмент программирования контроллеров и встраиваемых систем на языках МЭК 61131-3, не привязанный, к какой-либо аппаратной платформе и удовлетворяющий современным требованиям быстрой разработки программного обеспечения (рис. 3.9).

Ядро системы исполнения CoDeSys написано на языке C. Существует несколько модификаций оптимизированных для различных микропроцессоров (включая PC-совместимые). Для привязки к конкретному ПЛК требуется адаптация, касающаяся низкоуровневых ресурсов — распределение памяти, интерфейс связи и драйверы ввода-вывода.

Среди особенностей данного пакета можно отметить следующее.

- Прямая генерация машинного кода. Генератор кода CoDeSys — это классический компилятор, что обеспечивает очень высокое быстродействие программ пользователя.
- Полноценная реализация МЭК-языков, в некоторых случаях даже расширенная.
- «Разумные» редакторы языков построены таким образом, что не дают делать типичные для начинающих МЭК программистов ошибки.

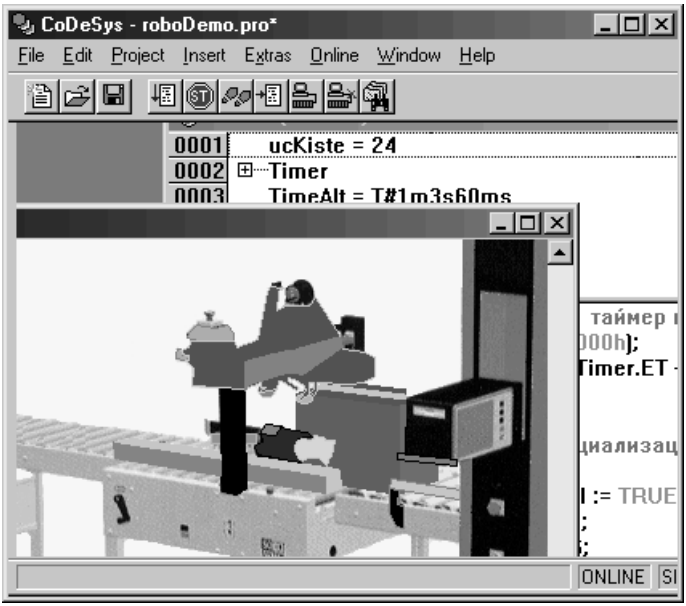

Рис. 3.9. Фрагмент визуализации конвейера в CoDeSys

- Встроенный эмулятор контроллера позволяет проводить отладку проекта без аппаратных средств. Причем эмулируется не некий абстрактный контроллер, а конкретный ПЛК с учетом аппаратной платформы. При подключении реального контроллера (режим online) отладчик работает аналогичным образом.
- Встроенные элементы визуализации дают возможность создать модель объекта управления и проводить отладку проекта без изготовления средств имитации. Существует «операционная» версия CoDeSys. Это компактное приложение, выполняющее только визуализацию, без средств разработки. Во многих простых случаях нет необходимости приобретать отдельно SCADA-систему. Серверы данных (DDE и OPC) также входят в стандартный пакет поставки.
- Очень широкий набор сервисных функций, ускоряющих работу программиста.
- В настоящее время создано более 150 адаптаций комплекса CoDeSys. Фирма 3S не скрывает своих клиентов. Все они открыто взаимодействуют друг с другом и совместно работают над совершенствованием программного инструментария.

Наиболее интересные решения и опыт применения обсуждаются на ежегодном семинаре и учитываются в новых версиях пакета. Благодаря этому и, безусловно, наличию хорошей команды разработчиков и грамотного руководства, пакет развивается исключительно динамично.

- Для CoDeSys доступен адаптированный русский перевод документации, выполненный ПК «Пролог» (см. Интернет-ссылки в конце книги).

В последние годы фирма 3S предпринимает весьма успешные усилия для интегрирования различных инструментов в одном пакете (конфигуратор распределенных систем, специализированный модуль управления перемещением, графический логический анализатор, система управления версиями и др.). Определение систем класса CoDeSys как инструментов программирования ПЛК является, безусловно, слишком скромным.

3.4. Строение комплекса CoDeSys

Базовый состав комплекса программирования ПЛК состоит из двух обязательных частей: системы исполнения и рабочего места программиста. *Система исполнения* функционирует в контроллере и, кроме непосредственно исполнения управляющей программы, обеспечивает загрузку кода прикладной программы и отладочные функции. Естественно, система исполнения должна иметь связь с компьютером *рабочего места программиста*. Как физически организована связь ПК и ПЛК, не столь важно. В простейшем случае ПЛК подключается к компьютеру через стандартный com-порт (RS232) нуль-модемным кабелем. В условиях цеха может использоваться более помехоустойчивый и дальнобойный интерфейс (RS422, RS485 или токовая петля).

В комплексе CoDeSys посредником между средой разработки и ПЛК служит специальное приложение — шлюз связи (gateway). Шлюз связи взаимодействует с интегрированной средой через Windows сокет-соединение, построенное на основе протокола TCP/IP. Такое соединение обеспечивает единообразное взаимодействие приложений, работающих на одном компьютере или в сети (рис. 3.10). Благодаря этому программист может абсолютно полноценно работать на удаленном компьютере. Причем удаленность не ограничивается рамками локальной сети. ПК, выполняющий задачу шлюза связи, может одновременно взаимодействовать с ПК программиста через Интернет и с ПЛК через модемное соединение.

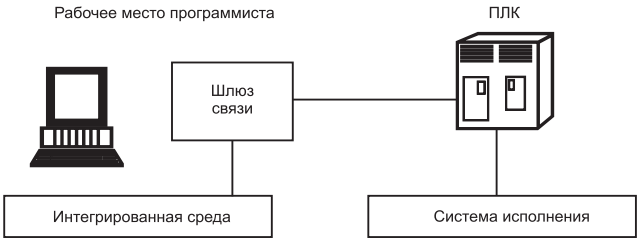

Рис. 3.10. Соединение интегрированной среды программирования с ПЛК

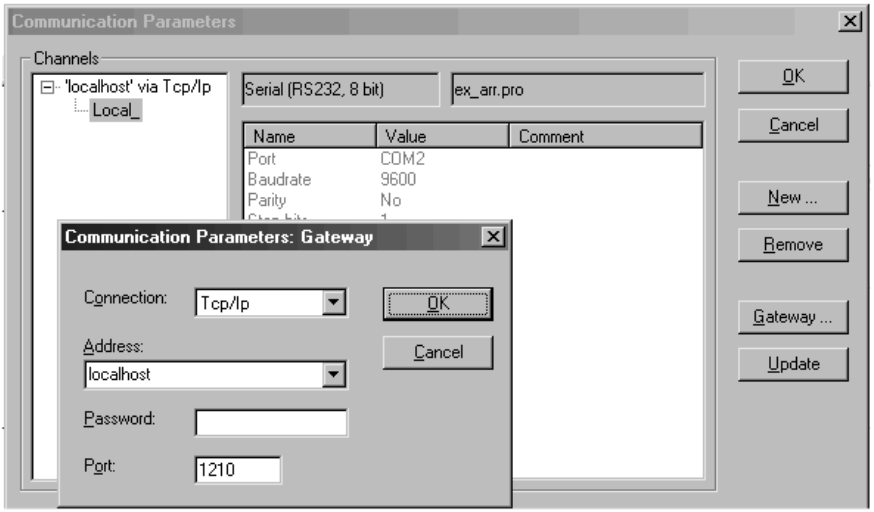

Рис. 3.11. Настройка коммуникационных параметров (CoDeSys)

По умолчанию шлюз связи настроен на локальную работу (local host) и запускается автоматически при установлении связи с ПЛК из интегрированной среды. Для соединения с ПЛК через com-порт достаточно только настроить параметры драйвера интерфейса (рис. 3.11) в соответствии с руководством по применению ПЛК (порт, скорость, контроль паритета и число стоп-бит).

В состав любого комплекса обязательно входит руководство по применению и электронная справочная система. Ассортимент дополнительных приложений CoDeSys включает серверы данных (DDE, OPC), утилиты конфигурирования комплекса, средства управления проектами и версиями, тестовые инструменты, специализированные библиотеки функций и функциональных блоков.

Глава 4. Данные и переменные

Прежде чем переходить к непосредственно изучению МЭК-языков, необходимо познакомиться с общими элементами этих языков. Общие элементы служат единым фундаментом, позволяющим объединить многоязычные компоненты в одном проекте. В главе будут рассмотрены данные, с которыми способен работать стандартный ПЛК, форматы их представления и наиболее общие приемы и тонкости работы с ними.

4.1. Типы данных

Тип данных переменной определяет род информации, диапазон представления и множество допустимых операций. Языки МЭК используют идеологию строгой проверки типов данных. Это означает, что любую переменную можно использовать только после ее объявления. Присваивать значение одной переменной другой можно, только если они обе одного типа. Допускается также присваивание значения переменной совместимого типа, имеющей более широкое множество допустимых значений. В этом случае происходит неявное преобразование типа без потерь. Неявные преобразования типов данных с потерями запрещены. Так, например, логическую переменную, способную принимать только два значения (логические 0 и 1), можно присвоить переменной типа **SINT** (−128...+127), но не наоборот.

При трансляции программы все подобные попытки отслеживаются и считаются грубыми ошибками. Если же это действительно необходимо, то выполнить присваивание с потерями возможно, но только при помощи специальных операторов. Операторы преобразования в МЭК выполняют также и более сложные операции, например преобразование числа или календарной даты в текстовую строку, и наоборот.

Наибольшее разнообразие типов данных в стандарте предусмотрено для представления *целых чисел*. Смысл применения широкого спектра целочисленных переменных заключается в первую очередь в оптимизации кода программы. Скорость вычислений зависит от того, как микропроцессор оперирует с переменными данного типа. Так, вполне очевидно, что 16-разрядный про-

цессор выполняет сложение двух 16-разрядных значений одной командой. Сложение же двух значений 32-разрядных переменных — это подпрограмма из нескольких команд.

Дополнительные задержки могут образовываться за счет мультиплексирования шины данных, связывающих процессор и память, особенностей микросхем памяти и т. д. В общем случае, меньшие по диапазону представляемых значений типы переменных требуют меньше памяти, меньше кода, и вычисления с их участием выполняются значительно быстрее.

Типы данных МЭК разделяются на две категории — элементарные и составные. *Элементарные* или *базовые типы* являются основой для построения составных типов. К *составным типам* относятся перечисления, массивы, структуры, массивы структур и т. д.

4.2. Элементарные типы данных

4.2.1. Целочисленные типы

Целочисленные переменные отличаются различным диапазоном сохраняемых данных и, естественно, различными требованиями к памяти. Подробно данные характеристики представлены в следующей таблице.

Тип	Нижний предел	Верхний предел	Размер, в байтах
BYTE	8 бит		1
WORD	16 бит		2
DWORD	32 бита		4
LWORD	64 бита		8
SINT	−128	127	1
INT	−32768	32767	2
DINT	-2^{31}	$2^{31}-1$	4
LINT	-2^{63}	$2^{63}-1$	8
USINT	0	255	1

Тип	Нижний предел	Верхний предел	Размер, в байтах
UINT	0	65535	2
UDINT	0	$2^{32}-1$	4
ULINT	0	$2^{64}-1$	8

Нижний предел диапазона целых без знака 0, верхний предел определяется как $(2^n) - 1$, где n — число разрядов числа. Для чисел со знаком нижний предел $-(2^{n-1})$, верхний предел $(2^{n-1}) - 1$.

Наименования целых типов данных образуются с применением префиксов, выражающих отношение размера к 16-разрядным словам: S (short *1/2) короткое, D (double *2) двойное, L (long *4) длинное. Префикс U (unsigned) указывает на представление целых без знака.

Переменные типов **BYTE**, **WORD**, **DWORD** и **LWORD** определяются стандартом как битовые строки **ANY_BIT**. Говорить о диапазоне значений чисел для этих переменных вообще некорректно. Они представляют строки из 8, 16 и 32 бит, соответственно. Помимо обращения с такими переменными как к единым целым, их можно использовать побитно.

Целые числа могут быть представлены в двоичной, восьмеричной, десятичной или шестнадцатеричной системе счисления. Числовые константы, отличные от десятичных, требуют указания основания системы счисления перед знаком «#». Например:

двоичное	2#0100_1110
восьмеричное	8#116
шестнадцатеричное	16#4E
десятичное	78

Для обозначения шестнадцатеричных цифр от 10 до 15 используются латинские буквы от A до F.

Символ подчеркивания «_» не влияет на значение и используется исключительно для улучшения зрительного восприятия числа. Например: 10_000, 16#01_88. Подчеркивание можно применять только между цифрами или в конце числа. Два или более подчеркивания подряд применять нельзя.

При начальной инициализации целочисленные переменные получают нулевые значения. Если необходимо задать другие начальные значения, это можно сделать непосредственно при объявлении переменной.

Примеры:

```
VAR
 wVar0, wVar1:  WORD;          (*2 переменных типа WORD*)
 byVar3:        BYTE;          (*тип BYTE начальное значение 0*)
 byVar2:        BYTE := 16#55; (*тип BYTE начальное
                                значение 55h*)
END_VAR
byVar2  := 2#1_0_0_0_1_0_0_0;  (*равносильно 2#1000_1000*)
byVar3  := 2#1_0_0_0__1_0_0_0; (*ошибка*)
```

4.2.2. Логический тип

Логические переменные объявляются ключевым словом **BOOL**. Это означает их принадлежность к алгебре Буля. Они могут принимать только значение логического нуля FALSE (ЛОЖЬ) или логической единицы TRUE (ИСТИНА). При начальной инициализации логическое значение по умолчанию — ЛОЖЬ.

```
VAR
 bVar1:         BOOL := TRUE;
 wVar2:         WORD;
END_VAR
```

При преобразовании значения логической переменной в целую FALSE дает 0, а TRUE 1.

```
wVar2 :=  BOOL_TO_WORD(bVar1);  (*результат 1*)
```

При обратном преобразовании любого целого в логическую переменную истину образует любое ненулевое значение:

```
wVar2 := 0;
bVar1 :=  WORD _TO_ BOOL (wVar2);  (*результат FALSE*)
```

Результаты операций, дающих логическое значение, можно присваивать переменным типа **BOOL**:

```
bVar1 := wVar2 > 5000;
```

По определению **BOOL** — это строка из одного бита, но из соображений эффективности кода при автоматическом распределении памяти транслятором под битовую переменную выделяется, как правило, 1 байт памяти целиком. Переменные типа **BOOL**, связанные с дискретными входами-выходами или определенные с прямым битовым адресом, действительно физически представлены одним битом.

4.2.3. Действительные типы

Переменные действительного типа **REAL** представляют действительные числа в диапазоне $\pm 10^{\pm 38}$. Из 32 бит, занимаемых числом, мантисса занимает 23 бита. В результате точность представления приблизительно составляет 6 — 7 десятичных цифр.

Длинный действительный формат **LREAL** занимает 64 бита. Число содержит 52-битовую мантиссу. Точность представления приблизительно составляет 15 — 16 десятичных цифр. Диапазон чисел длинного действительного $\pm 10^{\pm 307}$.

Числа с плавающей запятой, записываются в формате с точкой: 14.0, –120.2, 0.33_ или в экспоненциальной форме: –1.2E10, 3.1e7.

4.2.4. Интервал времени

Переменные типа **TIME** используются для выражения *интервалов времени*. В отличие от времени суток (TIME_OF_DAY) временной интервал не ограничен максимальным значением в 24 часа.

Числа, выражающие временной интервал, должны начинаться с ключевого слова TIME# или в сокращенной форме T#. В общем случае представление времени составляется из полей дней (d), часов (h), минут (m), секунд (s) и миллисекунд (ms). Порядок представления должен быть именно такой, хотя ненужные элементы можно опускать. Для лучшего зрительного восприятия поля допускается разделять символом подчеркивания. Например:

```
VAR
  TIME1:          TIME := t#10h_14m_5s;
END_VAR
```

Старший элемент может превышать верхнюю границу диапазона представления. Так, если в представлении присутствуют дни или часы, то секунды не могут превышать значения 59. Если се-

кунды стоят первыми, то их значение может быть и большим. Смысл этого правила состоит в том, что если вы хотите выражать интервал, например, исключительно в секундах — пожалуйста. Но если вы задействуете минуты, то для единообразия представления, секунды обязаны соблюдать принятые «правила субординации».

```
TIME1 := t#1m65s;        (*ошибка*)
TIME1 := T#125s;         (*правильно*)
```

Младший элемент можно представить в виде десятичной дроби:

```
TIME1 := T#1.2S;         (*равносильно T#1s200ms*)
```

4.2.5. Время суток и дата

Типы переменных, выражающие время дня или дату, представляются в соответствии с ISO 8601.

Тип	Короткое обозначение	Начальное значение
DATE	**D**	1 января 1970г.
TIME_OF_DAY	**TOD**	00:00
DATE_AND_TIME	**DT**	00:00 1 января 1970г.

Дата записывается в формате «год»—«месяц»—«число». Время записывается в формате «часы»:«минуты»:«секунды».«сотые». Дата определяется ключевым словом **DATE#** (сокращенно **D#**), время дня **TIME_OF_DAY#** (сокращенно **TOD#**), дата и время **DATE_AND_TIME#** (сокращенно **DT#**).

DATE#2002-01-31 или **D#**2002-01-31

TIME_OF_DAY#16:03:15.47 или **TOD#**16:03:15.47

DATE_AND_TIME#2002-01-31-16:03:15.47 или
DT#2002-01-31-16:03:15.47

Все три типа данных физически занимают 4 байта **(DWORD)**. Тип **TOD** содержит время суток в миллисекундах начиная с 0 часов. Типы **DATE** и **DT** содержат время в секундах начиная с 0 часов 1 января 1970 года.

4.2.5. Строки

Тип *строковых переменных* **STRING** определяет переменные, содержащие текстовую информацию. Размер строки задается при объявлении. Например, объявление строки str1, вмещающей до 20 символов, и str2 — до 60 символов:

```
VAR
  str1: STRING(20);
  str2: STRING(60) := 'Протяжка';
END_VAR
```

Если начальное значение не задано, то при инициализации будет создана пустая строка.

Количество необходимой памяти определяется заданным при объявлении размером строки. Для типа **STRING** каждый символ занимает 1 байт (**WSTRING** 2 байта). Строковые константы задаются между одинарных кавычек:

```
str1 := 'Полет нормальный';
```

При необходимости помещения в строку кода, не имеющего печатного отображения, используется знак ($) и следующий за ним код из двух цифр в шестнадцатеричной системе счисления. Для распространенных управляющих терминальных кодов можно применить следующие сокращения.

Обозначение	Код
$$	Знак доллара
$'	Одиночная кавычка
$L или $l	Перевод строки
$N или $n	Новая строка
$P или $p	Перевод страницы
$R или $r	Разрыв строки
$T или $t	Табуляция

4.2.6. Иерархия элементарных типов

Приведенная ниже *иерархия элементарных типов* применяется исключительно для удобства описания программ. Каждое наименование ANY_... объединяет некоторое множество типов. Так, при описании любой битовой операций удобнее указать, что она применима для ANY_BIT, чем перечислять всякий раз допустимые элементарные типы. Применять ANY_ при объявлении переменных, конечно, нельзя.

<table>
<tr><td rowspan="6">ANY</td><td rowspan="2">ANY_NUM</td><td>ANY_INT</td><td>SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT</td></tr>
<tr><td>ANY_REAL</td><td>REAL, LREAL</td></tr>
<tr><td colspan="2">ANY_BIT</td><td>BOOL, BYTE, WORD DWORD, LWORD</td></tr>
<tr><td colspan="3">STRING</td></tr>
<tr><td colspan="3">TIME</td></tr>
<tr><td colspan="2">ANY_DATE</td><td>DATE, TIME_OF_DAY, DATE_AND_TIME</td></tr>
</table>

4.3. Пользовательские типы данных

Описание *пользовательских типов данных* (кроме массивов) должно выполняться на уровне проекта (в CoDeSys на вкладке «Типы данных» — «Организатор Объектов»). Объявление типа всегда начинается с ключевого слова **TYPE** и заканчивается строкой **END_TYPE**.

4.3.1. Массивы

Массивы представляют собой множество однотипных элементов с произвольным доступом. Массивы могут быть многомерными. Размерность массива и диапазоны индексов задаются при объявлении (см. пример задания трехмерного массива):

<*Имя массива*>**:ARRAY**
[<li1>..<hi1>,<li2>..<hi2>,<li3>..<hi3>**] OF** <*тип элемента*>;

где li1, li2, li3 указывают нижние пределы индексов; hi1, hi2 и hi3 — верхние пределы. Индексы должны быть целого типа и

только положительные, отрицательные индексы использовать нельзя.

Примеры объявления массивов:

```
XYbass:   ARRAY [1..10,1..20] OF INT;
TxtMsg:   ARRAY [0..10] OF STRING(32);
Mass1:    ARRAY [1..6] OF SINT := 1,1,2,2,2,2;
Mass2:    ARRAY [1..6] OF SINT := 1,1,4(2);
```

Два нижних примера показывают, как можно выполнить инициализацию элементов массива при объявлении. Оба примера создают одинаковые массивы. В первом примере все начальные значения приведены через запятую. Во втором примере присутствует сокращение N(a,b,c..), которое означает — повторить последовательность a, b, c.. N раз. Многомерные массивы инициализируются построчно:

```
Mass2d:   ARRAY [1..2,1..4] OF SINT := 1,2,3,4,5,6,7,8;
```

Результат инициализации Mass2d показан на рис. 4.1.

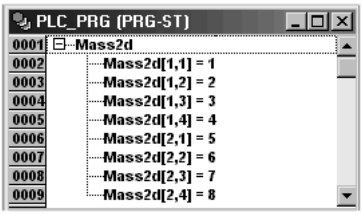

Рис. 4.1. Результат начальной инициализации массива Mass2d

Для доступа к элементам массива применяется следующий синтаксис:

<Имя_массива>[*Индекс1,Индекс2,Индекс3*]

Для двухмерного массива используются два индекса. Для одномерного, очевидно, достаточно одного. Например:

```
XYbass[2,12] := 1;
i := STR_TO_INT(TxtMsg[4]);
```

Если это не принципиально, используйте в массивах нумерацию с 0. В этом случае вычисление физического адреса элемента при исполнении проще. В результате код получается несколько короче.

4.3.2. Структуры

Структуры предназначены для создания новых типов данных на основе элементов разных базовых типов. С переменной типа структура можно обращаться как с единым элементом, передавать в качестве параметра, создавать указатели, копировать и т. д.

В отличие от массивов структура действительно вводит новый тип данных. Это означает, что до применения конкретной переменной нужно выполнить как минимум два объявления. Сначала нужно описать структуру. Описание структуры происходит глобально, на уровне проекта. Описанная структура получает *идентификатор* (имя структуры). Но это еще не переменная, это новый тип данных. Теперь, используя новый идентификатор, нужно объявить одну или сколько угодно переменных, точно так же, как и для базовых типов. Только теперь переменная нового типа получает «телесную оболочку» или, иными словами, конкретное место в памяти данных.

Объявление структуры должно начинаться с ключевого слова **STRUCT** и заканчиваться **END_STRUCT**. Синтаксис объявления выглядит так:

TYPE <*Имя_структуры*>:
  STRUCT
<*Объявление переменной 1*>
…
  <*Объявление переменной n*>
  END_STRUCT
END_TYPE

Пример объявления структуры по имени Trolley:

```
TYPE Trolley:
STRUCT
  Start:        TIME;
  Distance:     INT;
  Load, On:     BOOL;
  Articl:       STRING(16);
END_STRUCT
END_TYPE
```

Объявление в программе переменной Telega1 типа Trolley и начальная инициализация структуры выглядит так:

Telega1: Trolley := (Articl:='Пустой');

Состояние элементов после начальной инициализации Telega1 показано на рис. 4.2.

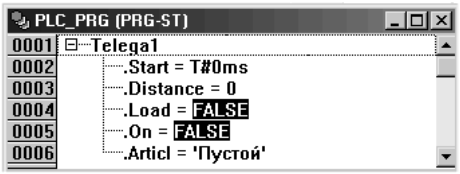

Рис. 4.2. Результат начальной инициализации Telega1

При начальной инициализации не обязательно задавать значения для всех элементов. Элементы, не имеющие явно указанных начальных значений, по умолчанию получат нулевые значения.

Для доступа к элементам структуры используется следующий синтаксис:

<Имя_переменной>.<Имя_элемента>

Например:

Telega1.On := True;

Структуры могут включать другие структуры, массивы и сами образовывать массивы. Пример объявления и инициализации массива структур:

TrolleySet: **ARRAY**[0..2] **OF** Trolley := (Articl := 'T 1'), (Articl := 'T 2'), (Articl := 'T 3');

TrolleySet[i].On := TRUE;

Если структура содержит вложенную структуру, то доступ к элементам вложенной структуры осуществляется с применением составного имени, содержащего две точки:

train.wagon[5]. weight; (*wagon[] вложенный массив структур*)

Поскольку физический размер элементов структуры известен транслятору заранее, обращение к элементу структуры не дает каких-либо накладных расходов в сравнении с простой переменной. Транслятор имеет возможность рассчитать абсолютные адре-

са элементов при компиляции. Естественно, это не относится к массивам структур. Чтобы не иметь проблем при использовании нескольких различных переменных одной структуры, применять прямые адреса в структуре нельзя.

4.3.3. Перечисления

Перечисление позволяет определить несколько последовательных значений переменной и присвоить им наименования. Перечисление — это удобный инструмент, позволяющий ограничить множество значений переменной и усилить контроль при трансляции. Как и структура, перечисление создает новый тип данных, определение которого выполняется на уровне проекта:

```
TYPE <Имя перечисления>:
(<Элемент 0>, < Элемент 1>, ... < Элемент n>);
END_TYPE
```

Объявленная позднее переменная типа *<Имя перечисления>* может принимать только перечисленные значения. При инициализации переменная получает первое из списка значение. Если числовые значения элементов перечисления не указаны явно, им присваиваются последовательно возрастающие числа начиная с 0. Фактически элемент перечисления — это число типа INT с ограниченным набором значений. Если необходимо, значения элементам можно присвоить явно при объявлении типа перечисления. Например:

```
TYPE  TEMPO: (Adagio := 1,Andante := 2,Allegro := 4);
END_TYPE
```

Идентификаторы элементов перечисления используются в программе как значения переменной:

```
VAR
  LiftTemp : TEMPO := Allegro;
END_VAR
```

Если в разные перечисления включены элементы с одинаковыми именами, возникает неоднозначность. Для решения этой проблемы применяется префикс, содержащий перечисление: TEMPO#Adagio. В CoDeSys все наименования элементов перечисления обязаны быть уникальными.

4.3.4. Ограничение диапазона

Тип переменных с *ограниченным диапазоном значений* позволяет определить допустимое множество значений переменной. Объявление типа переменной с ограниченным диапазоном должно происходить непосредственно между ключевыми словами **TYPE** и **END_TYPE**:

TYPE *<Имя>* :
<Целый тип> (<от>..<до>)
END_TYPE

Например:

```
TYPE DAC10:
  INT (0..16#3FF);
END_TYPE
```

Применение переменной с ограничением диапазона покажем на примере:

```
VAR
  dac: DAC10;
END_VAR
dac := 2000;
```

При попытке трансляции данного примера возникает законная ошибка:

```
Error: Type mismatch: Cannot convert '2000' to
'INT(0..1023)'.
```

4.3.5. Псевдонимы типов

Проблема выбора подходящего типа данных не всегда решается легко. Допустим, вы работаете с температурой, замеренной 16-разрядным АЦП. Может ли быть температура только выше нуля или когда-либо потребуется работать в отрицательной области, еще не совсем очевидно. В одном случае нужно использовать тип переменных **UINT**, а в другом — **INT**. Здесь удобно определить новый тип данных:

```
TYPE TEMPERATURA : UINT;
END_TYPE
```

Далее везде в программе вы используете тип **TEMPERATURA** при объявлении переменных. Если вдруг понадобится изменить тип температуры на **INT**, то это легко и быстро можно будет сделать в одном месте.

Аналогичные *псевдонимы типов* удобно создавать для любых часто используемых в программе типов. Например, для массивов или других типов, имеющих длинное и невыразительное определение.

4.3.6. Специфика реализации типов данных CoDeSys

Стандарт только определяет совместимые типы данных, но не требует обязательной поддержки всех типов для всех реализаций систем МЭК-программирования. CoDeSys имеет наиболее полную поддержку стандартных типов. Но даже он в настоящее время (версия 2.2) не поддерживает 64-разрядные целые и текстовые строки Unicode.

Кроме того, ограничение поддерживаемых типов данных возможно даже в рамках одного комплекса программирования для разных контроллеров. Так, восьмиразрядный генератор кода CoDeSys не поддерживает действительные переменные, перечисления и переменные, выражающие время суток и календарную дату. Ограничение поддержки типов диктуется достижением минимальной стоимости при максимальной эффективности ПЛК различных категорий. Так, полная реализация ядра системы исполнения CoDeSys (включая отладочные функции и трассировку значений переменных) для Intel 8051 совместимого микроконтроллера требует всего 6 Кб памяти кода. Естественно, что и код прикладной программы должен быть максимально компактным, для чего приходиться идти на определенные компромиссы. Но при необходимости любые специализированные типы данных можно определить на базе элементарных типов и поддержать при помощи библиотек.

В CoDeSys нет ограничения на способ применения битовых строк. В результате типы **BYTE**, **WORD** и **DWORD** можно применять в операциях, требующих целых без знака (**USINT**, **UINT** и **UDINT**), но не наоборот.

Внутренний формат переменных типа **TIME** стандартом не ограничен. В CoDeSys интервалы времени хранятся в переменной типа **DWORD** в миллисекундах, что обеспечивает представление интервалов почти 50 суток.

Если длина строки **STRING** при объявлении не указана, то принимается значение по умолчанию — 80 символов. В CoDeSys

используются нультерминированные (как в С-компиляторах) строки. То есть под строку всегда заранее выделяется область памяти заданного максимального размера. Любая строка оканчивается нулевым байтом, который не входит состав строки, а служит исключительно для определения конца строки функциями, оперирующими со строками. Пустая строка состоит из единственного нулевого байта. При объявлении строки необходимо задавать размер на единицу больше необходимого для символа «конец строки». Такая форма представления наиболее компактна (всего 1 вспомогательный байт), но, очевидно, не оптимальна в плане быстродействия. Если, например, нужно слить две строки, то функция конкатенации строк обязана сначала найти, где кончается первая строка. В других системах программирования можно встретить реализацию строк в виде структуры, содержащей максимальный размер, текущую длину строки и саму строку (массив байт). Вообще же работа со строками в ПЛК требуется не часто.

Для поддержки проверки значений переменных с ограниченным диапазоном во время работы система исполнения должна предоставлять средства контроля. В CoDeSys эта задача решается достаточно изящно — действия, которые выполняются, при попытке выхода за диапазон определяются программистом. Для этого служат специальные функции (CheckRangeSigned, CheckRangeUnsigned), которые необходимо включить в проект. На входе функции получают три параметра: две границы диапазона и значение. Любая необходимая реакция на нарушение границ (ограничение переменной, индикация ошибки и т. д.) описывается в теле функций контроля.

4.4. Переменные

Каждая *переменная* обязательно имеет наименование и тип. Сущность переменной может быть различной. Переменная может представлять вход или выход ПЛК, данные в оперативной или энергонезависимой памяти. Далее мы рассмотрим правила объявления и некоторые практические сложности и тонкости, возникающие при работе с переменными.

4.4.1. Идентификаторы

Имя переменной (ее идентификатор) должно быть составлено из печатных символов и цифр. Цифру нельзя ставить на первое место. Пробелы в наименовании использовать нельзя. Вместо них

обычно применяется символ подчеркивания. Символ подчеркивания является значимым. Так имена ‘Var1’, ‘Var_1’ и ‘_Var1’ являются различными. Два подчеркивания подряд использовать нельзя. Регистр букв не учитывается. Так ‘VAR1’ и ‘Var1’ одно и то же. Как минимум, 6 первых знаков идентификатора являются значимыми для всех систем программирования.

В CoDeSys такого ограничения нет — все символы наименования являются значимыми. Символы *кириллицы* (русского языка), к сожалению, в идентификаторах применять нельзя. Это ограничение характерно для всех программных систем.

Аналогичные требования относятся и к любым идентификаторам МЭК-программ (компоненты, метки, типы и т. д.).

4.4.2. Распределение памяти переменных

Контроллер с точки зрения МЭК программы имеет несколько областей памяти, имеющих разное назначение.

1. Область входов ПЛК.
2. Область выходов ПЛК.
3. Область прямо адресуемой памяти.
4. Оперативная память пользователя (ОЗУ).

Аппаратные ресурсы ПЛК присутствуют в МЭК-проектах в неявной форме. Размещение переменной в одной из трех первых областей приводит к ее связи с определенной аппаратурой — входами, выходами или переменными системы исполнения (диагностика модулей, настройка параметров ядра и т. д.). Распределение переменных в этих областях определяется изготовителем ПЛК. Привязка к конкретным адресам задается при помощи прямой адресации. Для обеспечения переносимости программного обеспечения прямые адреса нужно использовать только в разделе объявлений. В языках программирования стандарта не предусмотрено операций прямого чтения входов-выходов. Эту работу выполняет система исполнения. При необходимости для низкоуровневого обращения изготовителем ПЛК поставляются специальные библиотеки.

Объявление переменной без префикса AT физически означает выделение ей определенной памяти в области ОЗУ. Распределение доступной памяти ОЗУ транслятор осуществляет автоматически.

Переменные принято разделять на глобальные и локальные по области видимости. *Глобальные переменные* определяются на уровне ресурсов проекта (**VAR_GLOBAL**) и доступны для всех программных компонентов проекта. *Локальные переменные* описыва-

ются при объявлении компонента и доступны только внутри него. Описание любого программного компонента содержит, как минимум, один раздел объявления локальных переменных **VAR**, переменных интерфейса **VAR_INPUT**, **VAR_OUTPUT**, **VAR_IN_OUT** и внешних глобальных переменных **VAR_EXTERNAL** (см. подробнее «Компоненты организации программ»).

Наименования разделов объявления переменных могут содержать дополнительные ключевые слова, уточняющие способ применения.

Ключевое слово	Применение переменной
RETAIN	Переменные нужно разместить в энергонезависимой памяти, сохраняющей значения при выключенном питании. Такая память не является обязательной и присутствует далеко не во всех ПЛК
CONSTANT	Константы, доступные только для чтения

4.4.3. Прямая адресация

Для создания *прямо адресуемой* переменной используется следующее объявление:

имя переменной AT% *прямой адрес* *тип*;

Прямой адрес начинается с буквы, определяющей область памяти:

Символ	Область памяти
I	Область входов
Q	Область выходов
M	Прямо адресуемая память

Далее следует символ, определяющий тип прямого адреса:

Символ	Область памяти
нет	Бит
X	Бит

Символ	Область памяти
B	Байт
W	Слово
D	Двойное слово
L	Длинное слово

Завершает прямой адрес число — составной иерархический адрес, поля которого разделены точкой. В простейшем случае используется два поля адреса: номер элемента и номер бита.

В конце объявления, как и для автоматически размещаемых переменных, необходимо указать тип переменной. При указании адреса одного бита тип переменной может быть только **BOOL**.

В прямом адресе указывается именно номер элемента. Это коренным образом отличается от физических адресов микропроцессора. Если прямой адрес определяет байт, то номер элемента — это номер байта. Если прямой адрес определяет слово, то номер элемента — это номер слова, и, естественно, один элемент занимает два байта. Так, следующие три объявления адресуют один и тот же байт:

```
dwHeat    AT %MD1:    BYTE;
wbHeat    AT %MW2:    BYTE;
byHeat    AT %MB4:    BYTE;
```

Нумерацию элементов памяти для данного примера иллюстрирует следующая таблица.

D	0				1			
W	0		1		2		3	
B	0	1	2	3	4	5	6	7

На случай, если такая схема окажется почему-либо неприемлемой, компилятор CoDeSys имеет специальный флажок, принудительно включающий *байтовую адресацию* для всех типов.

В каждой области памяти адресация элементов начинается с нуля. Физическое размещение прямо адресуемых областей в ОЗУ определяется конфигурацией контроллера. Очевидно, что сопоставление идентификаторов переменных прямым адресам являет-

ся делом, требующим большой аккуратности. Поэтому для сложных модульных контроллеров применяются специальные фирменные конфигураторы, подключаемые к оболочке комплекса программирования и позволяющие графически «собрать» ПЛК и определить все необходимые интерфейсные переменные.

Входы ПЛК — это переменные с прямыми адресами в области **I**. Они доступны в прикладных программах только по чтению. Выходы **Q** — только по записи. Переменные в области **M** доступны по записи и чтению.

В области памяти **M** размещают обычно переменные, которые нельзя однозначно отнести к входам или выходам. Это могут быть диагностические ресурсы модулей, параметры системы исполнения и т. д.

Прямые адреса можно использовать в программах непосредственно:

```
IF %IW4 > 1 THEN ...          (*Значение входа IW4*)
```

Тем не менее все же желательно компактно сосредоточить в проекте все аппаратно-зависимые моменты.

Обратите внимание, что прямая адресация позволяет разместить несколько разнотипных переменных в одной и той же памяти. Например, специально для быстрого обнуления 16-дискретных выходов **(BOOL)** можно использовать переменную типа **WORD**. Или, например, совместить переменную **STRING** и несколько переменных типа **BYTE**, что даст возможность организовать форматирование вывода без применения строковых функций. Поскольку физическое распределение адресов известно на этапе трансляции, компилятор формирует максимально компактный код для таких объединений, чего не удается достичь при работе с элементами массива, где требуется динамическая адресация.

4.4.4. Поразрядная адресация

В стандарте предусмотрена удобная форма работы с отдельными битами переменных типа битовых строк — *поразрядная адресация*. Необходимый бит указывается через точку после идентификатора. Аналогичным образом можно использовать отдельные биты прямоадресуемой памяти. Младшему биту соответствует нулевой номер. Поразрядная нумерация не должна превышать границы соответствующего типа числа.

```
VAR
  a: WORD;
  bStop     AT %IX64.3:     BOOL;
END_VAR

a := 0;
a.3 := 1;           (*или a.3 := TRUE; — результат 2#0000_1000*)
a.18 := TRUE;       (*ошибка, в WORD не может быть бит a.18*)
IF a.15 THEN ...    (*a меньше нуля?*)
```

4.4.5. Преобразования типов

Преобразование типов происходит при присваивании значения переменной одного типа переменной другого типа. Преобразование меняет физическое представление значения в памяти данных, но не должно изменять само значение. Если это невозможно, то преобразование приводит к частичной потере данных. Но в таком случае транслятор требует явного указания необходимости выполнения такой операции.

Рассмотрим сначала работу с целыми числами. Пусть, например, объявлена переменная siVar типа короткое целое **(SINT** 8 бит) и переменная iVar типа целое **(INT** 16 бит). Допустим siVar = 100, а iVar = 1000. Выражение iVar := siVar является вполне допустимым, поскольку числа типа **SINT** являются подмножеством **INT** (iVar примет значение 100). Здесь преобразование типа будет выполнено транслятором автоматически, без каких-либо дополнительных указаний. Обратное присваивание siVar:= iVar приведет к переполнению и потере данных. Заставить транслятор выполнить преобразование с вероятной потерей данных можно только в явной форме при помощи специального оператора siVar := **INT_TO_SINT**(iVar). Результат равен 24 (в шестнадцатеричной форме 1000 это 16#03E8 и только младший его байт перейдет в **SINT**, значение 16#E8 соответствует десятичному числу 24).

Аналогичная ситуация возникает при работе с действительными числами длинного **LREAL** и короткого **REAL** типов.

Операторы явного преобразования базовых МЭК-типов образуют свои наименования из двух частей. Вначале указывается «исходный тип», затем «_TO _» и «тип результата». Например:

```
si := INT_TO_SINT(16#55AA);       (* Результат 16#AA*)
si := TIME_TO_SINT(T#120ms);      (*120*)
```

```
i := REAL_TO_INT(2.7);                    (*Результат 3*)
i := TRUNC(2.7);                          (*Результат 2*)
t := STRING_TO_TIME('T#216ms'); (*Результат T#116ms*)
```

Операция **TRUNC** выполняет отбрасывание дробной части в отличие от преобразования **REAL_TO_INT**, выполняющего округление.

Обратите внимание, что операции преобразования допустимы для любых комбинаций базовых типов, а не только для совместимых типов (например, дату в строку). Так, преобразования <...>_TO_STRING фактически заменяют оператор PRINT, распространенный в языках общего применения.

В конкретной реализации отдельные преобразования могут не поддерживаться или иметь определенные особенности, в первую очередь это относится к преобразованиям строк в другие типы и обратно. Поэтому мы не будем здесь приводить подробные описания всех возможных преобразований. При необходимости используйте руководство по применению системы программирования или оперативную справку.

4.5. Тонкости вычислений

Для неспециалистов в области информатики использование чисел и математических выражений в вычислительной технике может вызвать удивление. При определенных условиях «язык всех наук» упирается в препятствия, которых в классической математике нет и быть не может.

При программировании ПЛК наиболее широко применяются логические и целочисленные переменные. Целочисленные типы имеют максимально допустимое верхнее значение. Это понятно, поскольку для каждого целого типа в памяти выделяется определенное количество байт. Деление на ноль является недопустимой критической ситуацией, вызывающей ошибку микропроцессора. Это тоже понятно, поскольку бесконечно большое число не пригодно для арифметических вычислений. Теперь давайте возьмем переменную любого целого типа, присвоим ей максимально допустимое значение MAX и прибавим 1. Бытовая логика подсказывает, что переменная не изменится. Если в полное ведро добавить одну каплю, то больше уже не станет. Произойдет переполнение, которое должно вызвать ошибку, как и при делении на ноль. Проверим:

```
byX: USINT := 255;

byX := byX + 1;      (*результат 0*)
byX := byX + 1;      (*результат 1*)
byX := byX − 2;      (*результат 255*)
```

На самом деле переполнение приводит к обнулению переменной и не вызывает ошибки: MAX + 1 = 0. При вычитании происходит аналогичное превращение: 0 − 1=MAX. При внимательном изучении полученных результатов мы можем заметить своего рода симметрию относительно нуля. Получается, что алгебраическое равенство (a + b) − b = a справедливо, даже если сумма a + b вызывает переполнение. Дистрибутивный закон (a − b) c = ac − bc также работает:

```
a, b, c: USINT;

a := 100;
b := 50;
c := 3;

x1 := (a − b) * c;      (*результат 150*)
x2 := a*c − b * c;      (*результат 150*)
```

Ассоциативный (сочетательный) закон a + (b − c) = (a + b) − c вы можете проверить самостоятельно.

Получается, что фундаментальные арифметические аксиомы действуют, не взирая на переполнение, даже если переполнение приведет к тому, что сам результат будет арифметически неправильным:

```
(usint#100 + usint#50) * usint#3  =  194;
usint#100 * usint#3 + usint#50 * usint#3  = 194.
```

Равенство (a + b)c = ac + bc справедливо, хотя правильный результат, конечно, должен быть равен 450, а не 194. Фактически числовая ось для целых типов данных свернута в окружность, как в часах.

В примерах выше были использованы *беззнаковые типы данных,* представляющие множество натуральных чисел. Как известно, множество натуральных чисел незамкнуто относительно вычитания. При вычитании могут образовываться отрицательные числа. При необходимости работы с отрицательными значениями используются знаковые типы данных, представляющие множест-

во целых чисел. При одинаковой разрядности максимальное положительное число для знаковых типов вдвое меньше (на 1 разряд), чем для беззнаковых.

Наибольшее распространение для представления отрицательных чисел получил дополнительный код. Для данных в дополнительном коде описанные выше математические закономерности остаются в силе. В знаковых типах с применением дополнительного кода выбирается определенная граница, разделяющая положительные и отрицательные числа. Так, для типа **SINT** граница положительных чисел 127, для **INT** это 32767. Отрицательные числа образуются путем вычитания модуля числа из границы. Ноль в дополнительном коде один, в области положительных чисел. Превышение границы является также обратимым, как и переполнение: sint#127 + 1 = −128, sint# −128 − 1= 127.

Представление значений типов **SINT** и **USINT**, закодированных одинаковыми 8-битными последовательностями в десятичной и двоичной форме, показано в таблице:

USINT	SINT	BIN
0	0	0000_0000
1	1	0000_0001
...	...	...
127	127	0111_1111
128	−128	1000_0000
...	...	...
254	−2	1111_1110
255	−1	1111_1111

Из всего вышеописанного вытекает необходимость учета диапазона возможных значений переменных. Для этого необходимо внимательно проанализировать формулу вычислений, оптимизировать и перегруппировать ее при помощи элементарных алгебраических преобразований. К счастью, обнаружение ошибок переполнения обычно не вызывает трудности. Ошибочный результат никогда не бывает почти правильным. Если есть ошибка в целочисленных вычислениях, то результат, как правило, выглядит абсолютно невероятным.

Еще один существенный факт состоит в том, что множество целых чисел незамкнуто относительно операции деления. Иначе говоря, при делении двух целых образуется рациональное число, которое неизбежно приходится округлять для представления его на множестве целых чисел. За редким исключением, когда возможно деление без остатка, деление в целочисленных выражениях образует операционную погрешность.

Для достижения минимальной вычислительной ошибки прежде всего необходимо попытаться преобразовать формулу с целью минимизации количества операций деления. Рассмотрим пример.

Допустим, необходимо вычислить выражение:

$$X = \frac{a}{b} + \frac{c}{d}.$$

Здесь, очевидно, можно выполнить приведение дробей к общему знаменателю. В результате вместо двух операций деления останется только одна. Программа, выполняющая вычисление «в лоб» и с преобразованием, выглядит так:

```
a,b,c,d: INT;
a := 1; c := 2;
b := 3; d := 3;

x  := a/b + c/d;              (*результат 0*)
x1 := (a * d + c * b)/(b * d);  (*результат 1*)
```

В микропроцессорах операция округления сводится к отбрасыванию дробной части. В результате округленное значение всегда меньше истинного значения. При суммировании округленных значений погрешность накапливается. С точки зрения математики, более корректен метод симметричного округления. Если остаток меньше $1/2$, его отбрасывают, если больше — добавляют единицу. В нашем примере при симметричном округлении $1/3 = 0$, а $2/3 = 1$, в сумме погрешности компенсировались бы, и результат был бы равен 1. К сожалению, симметричное округление сложно реализуется и поэтому не поддерживается микропроцессорами непосредственно (см. реализацию ST-функции, выполняющей деление с симметричным округлением, в разделе примеров.) При программировании вычислений с дробями о свойстве округления забывать нельзя.

При работе с числами в формате с плавающей запятой максимальное и минимальное (машинный ноль) значения переменных являются абсолютными. Так, при попытке увеличить максимально допустимое значение оно не изменится. Алгебраические аксиомы за этими пределами уже не выполняются (см. рис. 4.3).

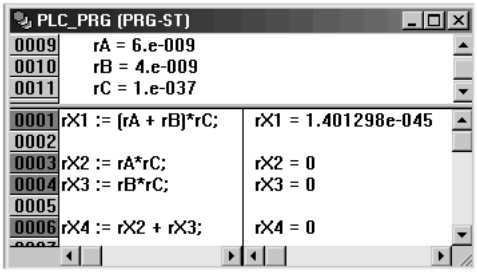

Рис. 4.3. Эффект потери значимости в операциях с действительными числами

Еще один момент, на который необходимо обратить внимание, это использование *констант*. Когда транслятор встречает константу, он выделяет под нее минимально необходимый тип данных. При вычислении выражений константы принимают тип по результату. Для разрешения возможной неоднозначности стандартом предусмотрено явное указание типа констант. Для этого используется префикс типа отделенный от константы значком «#». Например, вы присваиваете целочисленной переменной результаты выражения составленного и коротких целых и констант. Преобразование типов будет происходить неявно. Вопрос только в том — когда? До вычисления выражения все операнды преобразуется в **INT**, или сначала вычисляется выражение, а уже затем преобразуется. Допустим, нужно заставить транслятор вычислить выражение в коротких целых. Это можно сделать, например, так: iVar := USINT#100 – byVar.

4.6. Венгерская запись

При наличии строгой типизации данных очень полезной оказывается возможность узнавать тип переменной по ее наименованию непосредственно в тексте программ. В этом случае некорректное применение переменных бросается в глаза и позволяет избежать многих сложно локализуемых ошибок.

Для этого может использоваться специальная запись имен переменных. Впервые такая запись имен была предложена Чарльзом Симони (Charles Simonyi) и обоснована в его докторской диссертации. Возможно, потому что Симони родился в Будапеште и образованные по его системе наименования причудливы (на первый взгляд), как венгерский язык, за его методикой записи закрепилось название «венгерская запись». В настоящее время Симони является ведущим инженером Microsoft, а венгерская запись стала общепризнанной при программировании под Windows.

Идея венгерской записи заключается в прибавлении к идентификаторам коротких префиксов, определяющих тип и некоторые другие важные характеристики переменной. Префиксы принято записывать строчными буквами, а имя переменной с заглавной буквы. Поскольку венгерская запись «работает» для любых типизированных языков, имеет смысл применить ее и при программировании ПЛК.

Для базовых типов МЭК можно предложить следующие префиксы типов.

Префикс	Тип
b	BOOL
by	BYTE, USINT
si	SINT
w	WORD, UINT
i	INT
dw	DWORD, UDINT
di	DINT
r	REAL
lr	LREAL
st	STRING
t	TIME
td	TIME_OF_DAY
d	DATE
dt	DATE_AND_TIME

Примеры обозначений:

```
bStop:    BOOL;
bySet:    BYTE;
wSize     UINT;
```

«Венгерские» имена сами говорят о корректности их применения. Очевидно, следующее выражение является бессмысленным: bStop := wSize * 2; а выражение bStop := wSize > 2; вполне допустимым.

Уточнить назначение переменной можно добавлением еще одного символа перед префиксом типа:

Префикс	Назначение переменной
a	Составной тип, массив
n	Индекс
c	Счетчик

Для временных переменных можно вообще не утруждать себя придумыванием имен, а использовать только префиксы. Например:

```
aiSample:   ARRAY [1..32] OF INT;
ci:         INT;

FOR ci := 1 TO 32 DO
    siSample[ci] := −1;          (*без комментариев*)
END_FOR
```

К сожалению, некоторые из предложенных префиксов совпадают с зарезервированными словами (BY, AT, D, DT, N, ST). При использовании их в качестве временных переменных вы можете добавить порядковый номер или букву алфавита. Например:

```
byA, byB, by1, by2:   BYTE;
```

Структуры и функциональные блоки образуют имена экземпляров с включением полного или сокращенного наименования типа. Например, *tpUpDelay: TP*;

Символ подчеркивания удобно использовать для индикации способа обращения к переменной. Подчеркивание в начале имени указывает — только чтение. Идентификаторы переменных, соот-

ветствующих входами ПЛК, начинаются символом подчеркивания. Подчеркивание в конце имени указывает — только запись. Идентификаторы выходов заканчиваются символом подчеркивания. Например, *_bInp1, byOut2_*.

Если система обозначений хорошо продумана, то ее применение не вызывает сложности. Единый подход к наименованию очень здорово облегчает чтение программы и позволяет отказаться от излишних комментариев. Уникальные префиксы удобны не только для базовых типов, но и для широко используемых в проекте собственных типов данных и функциональных блоков. Стандарт МЭК не содержит рекомендаций по составлению имен переменных и компонентов программы. Никакого стандартного набора префиксов венгерской записи также нет. Вы можете использовать вышеописанную систему или разработать свою собственную. Главное, чтобы принятая система была понятна всем программистам — участникам проекта.

Описанные правила образования венгерских имен применяются в приведенных ниже примерах. В простых случаях, когда тип переменной не имеет значения или очевиден, венгерская запись не используется.

4.7. Формат BCD

Двоично-кодированный десятичный формат представления BCD (binary coded decimal) представляет собой числа в позиционной десятичной системе, где каждая цифра числа занимает 4 бита. Например, десятичное число 81 будет представлено в виде 2#1000_0001. Арифметические операции с BCD-числами требуют применения специального математического аппарата, малоэффективны в сравнении с обычным двоичным представлением. Но, с другой стороны, BCD оказывается очень удобным при организации клавиатурного ввода и индикации. Например, функции вывода числа на принтер или даже на сегментный индикатор получаются тривиальными (одна одномерная таблица на 10 констант).

Для хранения чисел в формате BCD стандарт МЭК предлагает использовать переменные типов ANY_BIT (кроме BOOL, конечно). Арифметика BCD-вычислений обычно не поддерживается в стандартном комплекте библиотек систем программирования ПЛК. В библиотеке утилит CoDeSys реализованы две простые функции BCD преобразования: BCD_TO_INT и INT_TO_BCD.

Глава 5. Компоненты организации программ (POU)

Данная глава посвящена *компонентам организации программ* и продолжает описание общих элементов стандарта. Компоненты образуют код прикладного программного обеспечения ПЛК. Именно на уровне компонентов доступно совмещение различных языков МЭК. В этой главе будут подробно рассмотрены все возможные компоненты стандарта, способы их определения и использования. В англоязычных документах компоненты организации программ сокращенно обозначаются POU — Program Organization Unit. Чтобы не вызывать неоднозначность, мы далее также будем использовать эту аббревиатуру.

5.1. Определение компонента

Компоненты организации программ являются базовыми элементами, из которых строится код проекта. Аналогичным образом электронные устройства составляются обычно из модулей. Например, магнитофон содержит: усилители записи и воспроизведения, генератор подмагничивания и стирания, блок питания и т. д. Каждый компонент программы имеет собственное наименование, определенный интерфейс и описание на одном из МЭК-языков. Один компонент может вызвать другие компоненты. Вызов самого себя (рекурсия) в стандарте МЭК не разрешена. Комбинировать различные языки в одном проекте можно при описании различных компонентов, но отдельный компонент целиком реализуется на одном языке МЭК. При вызове компонента язык его реализации значения не имеет.

К компонентам организации программ в МЭК-стандарте относятся функции, функциональные блоки и программы. Все они во многом похожи, но имеют определенные особенности и различное назначение.

Компонент обладает свойством *инкапсуляции* — работает как «черный ящик», скрывая детали реализации. Для работы с компонентом достаточно знать его интерфейс, включающий описание входов и выходов. Внутреннее его устройство знать необязатель-

но. В графической форме представления компонент выглядит как прямоугольник с входами слева и выходами справа. Локальные (внутренние) переменные компонента недоступны извне и в графическом представлении не отображаются.

Благодаря инкапсуляции компоненты успешно решают задачу *структурной декомпозиции проекта*. На верхнем уровне представления мы работаем с крупными компонентами. Каждый из них выполняет значительную для данного проекта задачу. Лишние подробности на этом уровне только мешают пониманию проблемы. Раскрывая вложенные компоненты один за другим, мы можем добраться до самого детального представления.

Готовый компонент всегда можно вскрыть, изучить и поправить. Это, конечно, относится только к пользовательским компонентам и открытым библиотекам. Некоторые стандартные компоненты включены в транслятор и не доступны для просмотра и изменения. Это относится и к внешним библиотекам. Внешние библиотеки реализуются в виде объектного кода при помощи внешних средств, например компилятора C или ассемблера. Возможно даже, что компонент реализован не только программно, а использует вспомогательные аппаратные средства, например часы реального времени или математический сопроцессор.

Еще одной задачей, решаемой компонентами, является *локализация имен переменных*. Это означает, что в различных компонентах можно использовать повторяющиеся имена. Так, например, любимую переменную с оригинальным идентификатором «X» можно использовать в каждом компоненте, и всякий раз это будет новая переменная. Область видимости локальных переменных определяется рамками одного компонента. Конечно, можно все переменные проекта объявить глобальными. Модификация и отладка подобных проектов вызывает такие же ощущения, как распутывание «бороды» из лески на удочке во время клева. Ограничение области видимости является обязательным во всех современных системах программирования.

Экземпляры функциональных блоков, объявленные внутри других компонентов, также обладают локальной областью видимости. Программы и функции всегда определены глобально.

5.1.1. Объявление POU

Реализации любого POU всегда должен предшествовать *раздел объявлений*. Объявления функции, функционального блока и программы начинаются соответственно с ключевых слов **FUNCTION**,

FUNCTION_BLOCK и **PROGRAM**. За ним следует идентификатор (имя компонента). Далее определяется интерфейс POU. К *интерфейсу компонента* относятся входы **VAR_INPUT**, выходы **VAR_OUTPUT** и переменные типа вход-выход **VAR_IN_OUT**. Завершают раздел объявлений локальные переменные **VAR**.

В функциях разделы **VAR_OUTPUT** и **VAR_IN_OUT** отсутствуют. Выходом функции служит единственная переменная, совпадающая с именем функции. Тип возвращаемого значения указывается при определении идентификатора через двоеточие.

Например: **FUNCTION** iNearby : **INT**

Структура раздела объявлений POU показана в таблице.

Тип POU	Функция	Функциональный блок	Программа
	FUNCTION имя: **ТИП**	**FUNCTION_BLOCK** имя	**PROGRAM** имя
Интерфейс	**VAR_INPUT**	**VAR_INPUT**	**VAR_INPUT**
	—	**VAR_OUTPUT**	**VAR_OUTPUT**
	—	**VAR_IN_OUT**	**VAR_IN_OUT**
Локальные переменные	**VAR**	**VAR**	**VAR**

Все разделы переменных являются не обязательными. Так нет ничего удивительного в программе, которая не требует координации работы с другими программами. Интерфейс ей не нужен, и раздел объявлений будет содержать только локальные переменные VAR.

5.1.2. Формальные и актуальные параметры

Интерфейс компонента образуется входными и выходными переменными. Интерфейсные входные переменные называют *формальными параметрами*. При использовании компонента его формальные параметры связываются с *актуальными параметрами*. И наконец, при вызове параметры компонента приобретают актуальные или *текущие значения*. Эти понятия необходимы для избежания двусмысленности при описании техники работы с компонентами.

Поясним их различия на примере. Возьмем стандартный блок R_TRIG. Он имеет вход с названием CLK. Мы будем использовать его в программе, в которой определена некая подходящая переменная, например bPulse. При вызове блока из нашей программы мы подаем bPulse на вход CLK. Далее программа компилируется и загружается в контроллер. Переменная bPulse приобретает некоторое значение, например TRUE. Вход CLK, естественно, тоже будет иметь значение TRUE. Здесь отличия уже практически очевидны. CLK — это формальный параметр, bPulse — актуальный параметр, а TRUE — фактическое значение. С формальными параметрами приходится иметь дело при проектировании POU и описании его интерфейса. Актуальные параметры работают при использовании компонента. Текущие значения рождаются только в «железе» в процессе выполнения.

5.1.3. Параметры и переменные компонента

При объявлении POU вы можете встретить следующие заголовки:

Формальные входные параметры ***VAR_INPUT***

Передаются POU по значению путем копирования. При вызове блока такой переменной можно присвоить значение другой переменной (совместимого типа), константы или выражения. Любые изменения такой переменной внутри POU никак не отображаются на данные вызывающего компонента. Применяется в любых POU. Могут иметь значения по умолчанию. Отражаются в графическом представлении с левой стороны компонента.

Формальные выходные параметры ***VAR_OUTPUT***

Отражают результаты работы компонента. Передаются из POU по значению путем копирования. Чтение значения выходов обычно имеет смысл после выполнения блока. Вне компонента параметры **VAR_OUTPUT** доступны только по чтению. Не используются в функциях, поскольку функция имеет только одно возвращаемое значение. Могут иметь начальные значения. Отражаются в графическом представлении справа.

Параметр типа ***VAR_IN_OUT***

Этот параметр одновременно является входом и выходом. Передача переменной экземпляру блока выполняется по ссылке. Это означает, что внешняя переменная как бы сама работает внутри блока на правах внутренней переменной. В компонент передается

только адрес ее расположения в памяти данных. Для переменной **VAR_IN_OUT** нельзя:

- использовать ее в функциях;
- присваивать начальное значение;
- обращаться как к элементу структуры данных, через точку;
- присваивать константу, как актуальный параметр.

Присваивание внешней переменной для **VAR_IN_OUT** можно производить только при вызове блока.

Важнейшим свойством **VAR_IN_OUT** является отсутствие копирования внешних данных. Параметры **VAR_INPUT** и **VAR_OUTPUT** могут оперировать с массивами и структурами, но всякий раз при обращении к компоненту будет происходить полное копирование данных. Это может отнимать много времени. Присваивание одного массива другому для **VAR_IN_OUT** означает фактически переключение компонента с одного массива на другой. Локальная копия данных в этом случае не создается.

Как и глобальные переменные, параметры **VAR_IN_OUT** нарушают идеологию независимости компонентов. Правильный компонент не должен иметь возможности испортить чужую память. Поэтому применять их нужно очень аккуратно и только в случаях, когда это действительно необходимо.

*Локальные переменные **VAR***

Доступны только внутри компонента, вне компонента доступа нет. Могут иметь начальные значения. Для функций локальные переменные размещаются в динамической памяти (обычно в стеке). По окончании работы функции память освобождается и может использоваться в других функциях. В программах и экземплярах функциональных блоков переменные **VAR** сохраняют свои значения между вызовами программ и экземпляров. В графическом представлении компонента локальные переменные не отражаются.

5.2. Функции

Функция — это программный компонент, отображающий множество значений входных параметров на выход. Функция всегда *возвращает* только одно значение. При объявлении функции указывается тип возвращаемого значения, имя функции и список входных параметров. Вызов функции производится по имени с указанием значений входных параметров. Функция может испо-

льзоваться в математических выражениях наряду с операторами и переменными.

Функция не имеет внутренней памяти. Это означает, что функция с одними и теми же значениями входных параметров всегда возвращает одно и то же значение. Функция — это чистый код. Многократное использование функции не приводит к повторному включению кода функции при компоновке. Реализация функции присутствует в коде проекта только один раз. Всякий раз при вызове функции процессор исполняет один и тот же поименованный код. Функция может иметь локальные (временные) переменные. Но при окончании своей работы функция освобождает локальную память.

Тип функции (тип возвращаемого значения) может быть любым из числа стандартных типов данных или типов созданных пользователем. Тело функции может быть описано на языках IL, ST, LD или FBD. Использовать SFC нельзя. Из функции можно вызывать библиотечные функции и другие функции текущего проекта. Вызывать функциональные блоки и программы из функций нельзя.

5.2.1. Вызов функции с перечислением значений параметров

В прародителях языка ST — языках Паскаль и C вызов функции производится по имени с перечислением в скобках списка актуальных входных параметров, через запятую, слева направо. Аналогичный способ приемлем и в языке ST. Например:

```
y := MUX(0, x1, x2);          (*Возвращает нулевой вход — x1*)
```

Здесь интересно обратить внимание на то, что наименования параметров нам не нужны. При перечислении параметров важно только соблюсти правильную последовательность в соответствии с определением в объявлении функции. В графических языках порядок входных параметров задан направлением сверху вниз (рис. 5.1).

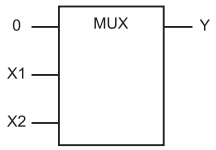

Рис. 5.1. Графическое отображение вызова функции

Некоторые функции могут иметь равноценные параметры, тогда их порядок очевидно безразличен. Например:

```
y := MAX(x1, x2);  (*Возвращает наибольшее из значений
                     входных параметров*)
y := MAX(x2, x1);  (*Результат тот же*)
```

5.2.2. Присваивание значений параметрам функции

Второй способ вызова функции предусматривает непосредственное присваивание значений параметрам функции по именам:

```
stHhello := CONCAT(STR1:='Добрый ', STR2:='день');
```

Это равносильно:

```
stHhello := CONCAT('Добрый ', 'день');
```

или:

```
stHello := CONCAT('Добрый ', STR2:='день');
```

Если в программе уже определена переменная с именем, совпадающим с наименованием входного параметра (STR1 := 'Добрый';), вызов может вызывать недоумение:

```
stHello := CONCAT(STR1:=STR1, STR2:='день');
```

На самом деле тут все правильно: слева от знака присваивания — параметр функции, справа — переменная.

Описанный способ вызова функции предполагает возможность задавать параметры в произвольном порядке и опускать некоторые из них. Текущая версия CoDeSys не обеспечивает функциям такую возможность. Единственный смысл такой нотации — в универсальности, приемлемой для функциональных блоков и программ.

Передача параметров функции всегда происходит путем копирования. При любом способе вызова функция получает локальные копии значений переменных.

5.2.3. Функции с переменным числом параметров

Для многих функций трудно предугадать, сколько значений нужно будет обработать в конкретном случае. Например, для функции **AND** можно ограничиться двумя входами и использовать «лесенку» вызовов функций для обработки большего числа переменных (рис. 5.2).

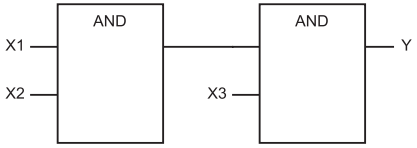

Рис. 5.2. Соединение двухвходовых AND

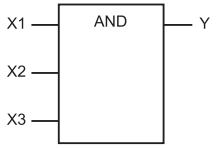

Рис. 5.3. Расширяемая реализация AND

На рис. 5.2 представлена не очень красивая конструкция. Было бы значительно удобнее иметь «расширяемую» функцию, которая могла бы адаптироваться под переменное число параметров. Такая реализация показана на рис. 5.3.

Стандарт МЭК действительно предусматривает такую возможность. В текстовых языках расширение производится добавлением переменных в конец списка параметров:

```
y := MUX(x_n, x1, x2, x3, x4, x5);
```

По причине сложности реализации транслятора переменное число параметров в пользовательских функциях не используется.

5.2.4. Операторы и функции

Операторы — это символы определенных операций. Но их можно определить и как функции, наделенные определенными привилегиями. Во-первых, код для операторов транслятор создает сам и не требует подключения каких-либо библиотек. Во-вторых, многие операторы имеют особые формы записи в выражениях ST. Например, математические операторы (сложение, вычитание, умножение и деление) имеют традиционное символьное представление в текстовых языках (+, –, *, /). В графических языках операторы выглядят как обычные функции.

В принципе, можно обходиться без символьного представления операторов. Например:

```
Y := SUB(MUL(4,x),3);
```

Но символьное представление в ST выглядит значительно лучше:

```
Y := 4 * x – 3;
```

Математики пишут еще короче:

```
Y = 4x – 3.
```

Все три записи равноценны по смыслу. Символьные выражения понятнее и дают возможность более сконцентрироваться на сути выражения, а не на форме его представления.

При работе с операторами необходимо обращать внимание на наличие символьной формы представления. Так, для математических и логических операторов в языке ST, как правило, допускается только символьное представление. Выражение Y := **AND**(x1, x2) вызовет ошибку компиляции. Необходимо писать так: Y := x1 **AND** x2;. Если оператор не имеет символьного представления, то на него распространяются обычные правила вызова функций. Например: y := **SQRT**(x);.

Обратите внимание, что имена входных параметров для операторов в описании не заданы. Это означает, что вызывать такие функции в ST можно только перечислением параметров.

5.2.5. Перегрузка функций и операторов

Существует достаточно много функций, имеющих смысл для переменных разного типа. Например, функция **MAX** возвращает наибольшее из входных значений. Очевидно, что код команд микропроцессора, оперирующих с переменными типа **SINT** и **REAL**, должен быть разным, но с точки зрения языков МЭК это одна и та же функция. Автоматическая генерация разного кода для одной функции в зависимости от типов переменных называется *перегрузкой*. Реализация перегрузки пользовательских функций сложна для трансляции и спорна. Перегрузка операторов прозрачна для компилятора с точки зрения контроля типов. В пользовательских функциях это может приводить к сложно локализуемым ошибкам.

Многие стандартные функции и операторы поддерживают перегрузку. Тип самой функции определяется требованием совместимости с входными типами. Так, для функции **MAX** с входными параметрами типа **INT** выход будет типа **INT**.

5.2.6. Пример функции

Рассмотрим пример функции целого типа *Nearby_int*, возвращающей ближайшее к образцу *pattern* значение из двух входных *in1* и *in2*.

Объявление:

```
FUNCTION Nearby_int : INT
VAR_INPUT
```

```
        pattern, in1, in2:      INT;
END_VAR
```

На языке ST тело функции будет таким:

```
IF ABS(in1 – pattern) < ABS(in2 – pattern) THEN
    Nearby_int := in1;
ELSE
    Nearby_int := in2;
END_IF
```

Локальные переменные в данном примере не использованы, соответственно секция объявлений **VAR…END_VAR** отсутствует.

На языке IL этот алгоритм можно выразить так:

```
  LD          in2
  SUB         pattern
  ABS
  ST          tmp
  LD          in1
  SUB         pattern
  ABS
  LT          tmp
  NOT
  JMPC        ret_in2
  LD          in1
  ST          Nearby_int
  RET
ret_in2:
  LD          in2
  ST          Nearby_int
```

Здесь использована промежуточная переменная *tmp*, которую необходимо объявить:

```
VAR
  tmp:INT;
END_VAR
```

Аналогичную функцию в FBD удобно реализовать с применением бинарного мультиплексора. Такой пример представлен на рис. 5.4.

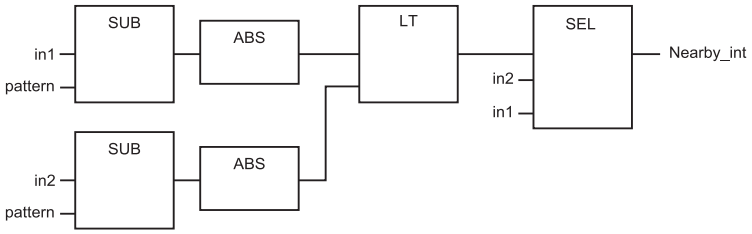

Рис. 5.4. Реализация функции *Nearby_int* в FBD

5.2.7. Ограничение возможностей функции

Можно ли создать функцию с внутренней памятью? Можно, если пойти на хитрость — использовать в функции глобальную переменную, ее значение будет сохраняться при повторном вызове. При этом сама функция будет обладать уникальностью глобальной переменной. Так, можно сделать функцию счетчик, подсчитывающую число вызовов функции в глобальной переменной. Создать несколько независимых счетчиков на основе такой функции невозможно. Кроме того, можно обратиться из функции к неким аппаратным средствам, посредством прямо адресуемых переменных. Например, к системному таймеру. В этом случае функция будет возвращать разные значения для одинаковых входных данных. Но на самом деле, любой аппаратный ресурс или глобальная переменная не принадлежат самой функции. По сути, для функции это те же формальные параметры. Разница только в том, что законные параметры для функции готовит вызывающий код, а значения глобальных переменных она добывает сама.

Локальные переменные функции, имеющие явно заданные начальные значения, получат их всякий раз в начале работы функции. Если начальные значения не заданы, ситуация может быть различной. Транслятор может обнулять их принудительно или не тратить на это время и оставлять случайные значения. Это зависит от реализации. Некоторые генераторы кода имеют специальный флажок в настройках, предоставляющий выбор программисту — скорость или дополнительная страховка от случайных ошибок.

Обычно транслятор размещает локальные переменные и параметры функции в стеке. Но это не всегда так. Возможно, что из соображений оптимизации или в силу аппаратных ограничений

(некоторые процессоры имеют очень маленький стек) для функции будет отведено постоянное место в статической области данных. Тогда может получиться, что локальные переменные будут сохранять свои значения между вызовами. Это экзотическое исключение из общего правила, и использовать его крайне опасно.

Компоненты SFC требуют некоторой памяти данных для запоминания своего текущего состояния. Поэтому запрет на их использование в функциях не удивителен.

С экземплярами функциональных блоков ситуация похожая. Если экземпляр функционального блока создать в локальной памяти функции, то его переменные будут принимать начальные значения при каждом вызове функции. Иногда это не существенно. CoDeSys не ограничивает такую возможность, следуя закону Кейсэра: «Можно сделать защиту от дурака, но только от неизобретательного».

По определению, функция возвращает одно значение, но это очень легко обойти. Тип функции может быть составным, например структурой. Здесь необходимо иметь в виду, что, прежде чем использовать данные этой структуры, ее придется присвоить некоторой однотипной переменной. Такое присвоение выполняется транслятором, путем побайтного копирования. Фактически это цикл, скрытый от программиста за простым знаком равенства. Понятно, что при копировании структура или массив требуют времени пропорционально своему размеру. Аналогичная ситуация складывается и при передаче составных типов в качестве параметра функции.

Функция обязательно должна иметь хотя бы один входной параметр и обязана возвращать значение. Пустой тип, обозначающий отсутствие передачи значений (тип VOID в C++), стандартом не предусмотрен. Если возвращать все же нечего, используйте тип **BOOL**. Вы можете даже не присваивать значения выходной переменной функции. По умолчанию функция будет возвращать FALSE. Возврат **BOOL** дает мизерные затраты кода для любого типа процессора.

Определенные трудности с применением функций возникают в языке LD. Здесь иногда приходится искусственно дополнять функцию дополнительным входным параметром EN (ENable), разрешающим или запрещающим выполнение функции. Подробнее см. раздел «Расширение возможностей LD».

Среди компонентов МЭК, функция представляет «легкую весовую категорию». Примером хороших функций являются стандар-

тные функции и операторы. Число их входных параметров обычно не превышает трех.

Если при реализации функции возникли вышеописанные сложности, обычно лучшим решением является перевод ее в более «серьезную» весовую категорию — функциональный блок.

5.2.8. Функции в логических выражениях

Применение функций в *логических выражениях* имеет одну тонкость. Логическое выражение не всегда обязательно вычислять целиком. Возможно, что по некоторой начальной части выражения уже можно сделать вывод об итоговом значении. Оптимизирующий компилятор достаточно «разумен» для таких действий. То есть, если логическое выражение содержит функции, то нельзя гарантировать, что все они будут вызваны.

Например:

IF func1(x) **OR** func2(x) **THEN** ...

Если func1(x) возвращает TRUE, то func2(x) вызываться не будет.

Еще один наглядный пример на языке FBD показан на рис. 5.5.

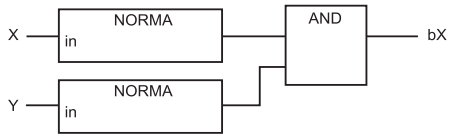

Рис. 5.5. Функция в логическом выражении

Здесь нельзя гарантировать, что функция NORMA будет вызвана 2 раза. Если первый вызов дает FALSE, то понятно, что дальше логическое выражение можно не проверять.

Когда функции, участвующие в логических выражениях, не делают никакой посторонней работы, ничего страшного в описанном нет. Поэтому крайне не желательно в логических функциях выполнять запись выходов ПЛК или глобальных переменных. Не игнорируйте данное предупреждение, поскольку «отловить» такую ошибку очень сложно. Если даже описанный эффект не проявляется, нет гарантии, что ошибка не возникнет в другой программной среде или с новой версией компилятора.

5.3. Функциональные блоки

Функциональный блок — программный компонент, отображающий множество значений входных параметров на множество выходных. После выполнения экземпляра функционального блока все его переменные сохраняются до следующего выполнения. Следовательно, функциональный блок, вызываемый с одними и теми же входными параметрами, может производить различные выходные значения. Сохраняются все переменные, включая входные и выходные. Так, если мы вызовем экземпляр функционального блока, не определяя значения некоторых входных параметров, он будет использовать ранее установленные значения. Возможность задания переменного числа входных значений заложена по определению и не требует каких-либо дополнительных усилий. Извне доступны только входы и выходы функционального блока, получить доступ к внутренним переменным блока нельзя.

С позиций *объектно-ориентированного программирования* (ООП) функциональные блоки — это объекты, великолепно реализующие инкапсуляцию, т. е. сокрытие деталей реализации. Объединение кода и данных в «одном флаконе» роднит функциональные блоки с классами ООП. Возможность *наследования* и *полиморфизм*, к сожалению, пока отсутствуют.

5.3.1. Создание экземпляра функционального блока

Прежде чем использовать функциональный блок, необходимо создать его экземпляр. Эта операция аналогична по смыслу объявлению переменной. Описав новый блок, мы фактически создали новый тип данных подобный структуре. Каждый функциональный блок может иметь любое количество экземпляров. Так, различные экземпляры блока «таймер» совершенно независимы друг от друга. Каждый из них имеет собственные настройки и живет собственной жизнью.

Каждый экземпляр функционального блока имеет свой собственный идентификатор и свою область в статической памяти данных. Объявление еще одного экземпляра блока приводит к выделению еще одной области в памяти данных. Код, очевидно, как и для функции, остается общим (рис. 5.6).

Экземпляр функционального блока создается в разделе объявлений переменных функционального блока, программы или в разделе глобальных переменных проекта. Как и переменные, он должен получить уникальный идентификатор. Например, созда-

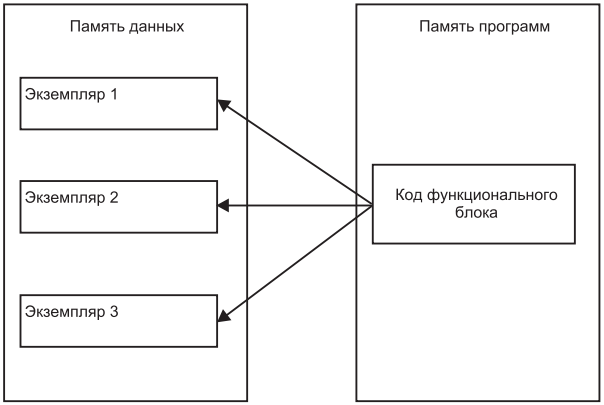

Рис. 5.6. Распределение памяти для экземпляров функциональных блоков

ние экземпляра стандартного функционального блока «инкрементный счетчик» с идентификатором ctuTimeMeter выглядит так:

```
ctuTimeMeter: CTU;
```

Очевидно, что создавать экземпляры можно только для известных системе блоков. Это библиотечные блоки или блоки, ранее реализованные пользователем. С точки зрения транслятора, создание экземпляра означает выделение необходимой памяти для размещения переменных блока.

Экземпляр функционального блока можно не только вызывать, но и использовать в качестве входных переменных других функциональных блоков.

Функциональным блоком иногда называют экземпляр функционального блока, для краткости. В данной книге такие неоднозначные сокращения применяться не будут. Позволим себе только называть иногда функциональный блок просто блоком, а экземпляр функционального блока — экземпляром.

5.3.2. Доступ к переменным экземпляра

После создания экземпляра функционального блока можно сразу начать работать с его данными. При этом совсем не обязательно вызывать его. Обращаться к переменным экземпляра можно так же, как к элементам структуры данных, через точку:

```
ctuTimeMeter.RESET := FALSE;
ctuTimeMeter.PV := 100;
x := ctuTimeMeter.CV;
```

Входы экземпляра блока доступны для записи и чтения извне. Выходы — только для чтения. Изменять значения выходов можно только из тела блока, извне нельзя. Транслятор отслеживает такие попытки и выдает сообщение об ошибке.

5.3.3. Вызов экземпляра блока

Вызывать экземпляр функционального блока с перечислением параметров, как функцию, нельзя. Значения входных переменных должны присваиваться непосредственно. В текстовых языках входные переменные перечисляются в скобках, после имени экземпляра. Присваивание входных значений выполняется операцией ‘:=’.

На языке **ST:**

```
ctuTimeMeter (RESET := FALSE);
```

На языке **IL:**

```
CAL     ctuTimeMeter(RESET := FALSE)
```

Специальный символ ‘=>’ позволяет получить значения выходов после выполнения блока:

```
ctuTimeMeter (RESET := FALSE, CU := Inp1, CV => x);
```

При вызове экземпляра можно определить только необходимые параметры, причем в произвольном порядке. В графических языках неиспользуемые входы и выходы экземпляра блока просто остаются не подключенными — PV на рис. 5.7.

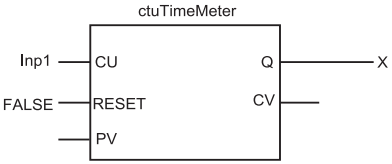

Рис. 5.7. Вызов экземпляра функционального блока (FBD)

В языке ST при отсутствии параметров пустые скобки в ST после имени экземпляра ставить не нужно:

```
ctuTimeMeter();      (*Лишние скобки*)
```

Использовать экземпляры функциональных блоков в выражениях нельзя, но можно использовать их входы и выходы:

```
X := ctuTimeMeter.PV − ctuTimeMeter.CV + 1;
```

Вы можете определить значения входов заранее и вызвать экземпляр функционального блока вообще без параметров:

На языке **ST:**

```
ctuTimeMeter.RESET := FALSE;
ctuTimeMeter;
```

На языке **IL:**

```
LD      FALSE
ST      ctuTimeMeter.RESET
CAL     ctuTimeMeter
```

Нужно обратить внимание на то, что МЭК не поощряет использование элементов данных отдельно от вызова экземпляра блока, поскольку это может привести к проблемам при использовании экземпляра блока в многозадачных проектах. С другой стороны, многократное повторное присваивание входных значений увеличивает размер кода и снижает эффективность программы.

5.3.4. Инициализация данных экземпляра

При описании блока в разделе объявлений можно явно присвоить начальные значения переменным. Например:

```
FUNCTION_BLOCK SyncSwitch
VAR_INPUT
   ... ... ...
   Sync:        BOOL := TRUE;
```

При создании экземпляра функционального блока SyncSwitch входная переменная Sync получит значение TRUE. Если начальные значения не заданы, используются нулевые значения.

Экземпляр функционального блока может потребовать индивидуальной инициализации, отличной от той, которая определена при реализации. Установку начальных значений переменных проще всего выполнить при создании экземпляра. Значения, заданные при создании экземпляра, сильнее значений, заданных при реализации блока.

```
SyncSw1: SyncSwitch := (Sync := FALSE);
```

Теперь переменная SyncSw1.Sync получит начальное значение **FALSE**, несмотря на значение, указанное в объявлении блока.

Физически начальные значения переменные получают еще до первого использования экземпляра. Операция начальной инициализации переменных производится по сбросу, который выполняется непосредственно после загрузки проекта в память ПЛК, по команде отладчика или при перезапуске контроллера.

Некоторые трансляторы имеют опцию отключения инициализации по умолчанию с целью ускорения запуска ПЛК. В этом случае полагаться на то, что переменные, не имеющие явно указанных начальных значений, будут получать одинаковые значения при перезапуске системы, нельзя.

Возможны случаи, когда экземпляру функционального блока нужна разумная инициализация. Например, для настройки блока необходимо провести некоторые вычисления. Специальной процедуры инициализации в функциональных блоках не предусмотрено. Здесь придется потратить на инициализацию один или несколько первых циклов выполнения экземпляра. Окончание сложной процедуры инициализации индицируют обычно выходом готовности (ENO). Часто удобно применить для инициализации действие (см. ниже) и сосредоточить контроль над инициализацией в одном месте (обычно в шаге Init SFC диаграммы). Такой метод позволяет проводить инициализацию данных, экземпляров блоков и программ в необходимой последовательности и взаимосвязи. В большинстве же практических случаев для блоков, требующих определенной настройки, оказывается достаточным ввести несколько специальных входов (уставок). Так сделано во всех стандартных блоках.

5.3.5. Тиражирование экземпляров

При необходимости получить копию экземпляра функционального блока можно использовать оператор присваивания:

```
SyncSw, SyncSw2: SyncSwitch;
(*работаем с экземпляром SyncSw1*)
... ... ...
```

На языке **ST:**

```
SyncSw2 := SyncSw1;
```

На языке **LD** реализация этого примера показана на рис. 5.8.

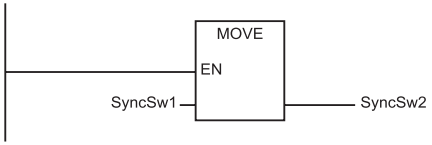

Рис. 5.8. Присваивание экземпляров функционального блока

Оператор присваивания выполняет побайтное копирование данных экземпляра блока по аналогии со структурами данных. Очевидно, что такая техника копирования подходит только для экземпляров одного и того же функционального блока.

5.3.6. Особенности реализации и применения функциональных блоков

Входные переменные внутри блока доступны для записи. Это вызывает определенный соблазн для программиста. Так, например, входную переменную удобно применить в качестве счетчика итераций, если она как раз и отражает число нужных повторений. Это позволит избежать создания дополнительной локальной переменной. Поскольку при вызове экземпляра блока вводная переменная должна получить новое значение, ничего страшного на первый взгляд нет. Вызов экземпляра не обязан сопровождаться присваиванием значений всем формальным параметрам. Возможно, в какой-то момент вы решите, что входной параметр уже определен, и можно не задавать его повторно. В результате значение параметра будет равно тому значению, которое он имел при работе в роли локальной переменной, при предыдущем вызове экземпляра. Конечно, можно придумать много примеров, когда изменение значения входной переменной безопасно. Но все же использовать такой прием нужно исключительно обдуманно и осторожно. В общем случае при реализации блока входные переменные нужно рассматривать как константы.

Применение глобальных переменных в функциональных блоках вызывает те же проблемы, что и в функциях. Экземпляры блока перестают быть независимыми. Изменения переменной, выполненные одним экземпляром, проявят себя совсем в другом месте. Иногда это действительно необходимо, но в обычной практике желательно ограничивать такое применение глобальных переменных.

Если экземпляры функционального блока используют глобальную переменную только для чтения, то никаких побочных яв-

лений возникнуть не может. Аналогичная ситуация возникает при применении прямоадресуемых переменных. С входами проблем нет. Единственное ограничение — это ухудшение возможностей блока в плане его повторного применения. При использовании блока в другой программе или проекте прямые адреса придется поправить. Хорошее решение этой проблемы дают шаблонные переменные.

5.3.7. Шаблонные переменные

Шаблонные переменные или, как их иногда называют, *конфигурационные переменные* (в CoDeSys variable configuration) являются частично определенными прямоадресуемыми переменными. Прямой адрес заменяется звездочкой. Определение полного адреса шаблонной переменной дается в специальном разделе ресурсов.

Рассмотрим, как это делается на примере. Создадим блок SHAMAN, который будет иметь шаблонную переменную bMarvel. Для нашего примера достаточно раздела объявлений:

```
FUNCTION_BLOCK SHAMAN
VAR
bMarvel     AT %I*    : BOOL;
END_VAR
```

Далее создадим два экземпляра блока SHAMAN в главной программе:

```
PROGRAM PLC_PRG
VAR
Shaman1:            SHAMAN;
Shaman2:            SHAMAN;
END_VAR
```

Для того чтобы окончательно «разобраться» с шаблонными переменными экземпляров, их нужно прописать в ресурсах проекта:

```
VAR_CONFIG
PLC_PRG.Shaman1.bMarvel     AT %IX1.0 : BOOL;
PLC_PRG.Shaman1.bMarvel     AT %IX1.1 : BOOL;
END_VAR
```

Типы данных шаблонной переменной, указанные в объявлении блока и при настройке адреса в ресурсах, обязаны совпадать.

5.3.8. Пример функционального блока

В качестве простого примера реализуем блок синхронного переключателя SyncSwitch. Алгоритм его работы следующий: выход переключателя Q принимает значения, равные входу Start, но переключение выхода разрешено только при Sync := TRUE. Графически это отражено на рис. 5.9.

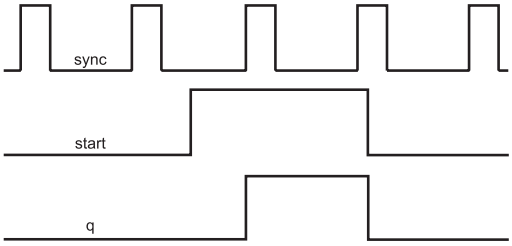

Рис. 5.9. Диаграмма работы синхронного переключателя

Условие включения выхода выражается уравнением: Q = start **AND** sync, а условие выключения Q = **NOT** start **AND** sync. Значение выхода должно сохранятся между синхроимпульсами, поэтому использовать здесь функцию нельзя. На языке IL блок SyncSwitch можно реализовать так:

```
FUNCTION_BLOCK SyncSwitch
VAR_INPUT
   Sync:        BOOL;
   Start:       BOOL;
END_VAR
VAR_OUTPUT
   Q:           BOOL;
END_VAR

   LD           Sync
   AND          Start
   S            Q

   LD           Sync
   ANDN         Start
   R            Q
```

Такой блок полезен при реализации «безударного» переключения в цепи переменного тока. Импульсы синхронизации должны

отражать интервалы, когда мгновенное значение сетевого напряжения близко к нулю. Тогда переключение силовой цепи с выхода SyncSwitch будет происходить без броска тока.

Конечно, практическое значение нашего нового блока несколько меркнет, если вспомнить о стандартном *доминантном переключателе* — **SR**. С ним задача решается в одну строчку (ST):

SR_1(SET1:= Start **AND** Sync, RESET:= Sync, Q1=> Q);

5.3.9. Действия

В функциональных блоках МЭК очень не хватает возможности выполнять несколько различных операций. Особенно, если блок содержит объемные данные. Можно, конечно, сделать дополнительный вход и по нему анализировать, что мы хотим от блока. В CoDeSys эта проблема решена самым естественным способом. Функциональные блоки и программы можно дополнять действиями. Действие работает внутри блока с полным правом доступа ко всем данным. Его можно вызвать как из тела блока, так и извне. Действие указывается через точку после названия экземпляра блока и может иметь список значений входов и выходов. При вызове действия из тела блока наименование экземпляра, естественно, не требуется. Действие не имеет собственных данных и использует входы, выходы и локальные переменные блока. Язык реализации действия может быть произвольным.

Для примера дополним вышеописанный блок SyncSwitch действием EmergencyBreak. Пускай вызов данного приводит к мгновенному безусловному выключению выхода. Для определения действия нужно выбрать блок в органайзере объектов CoDeSys и дать команду **«Add Action»**. Опишем действие на языке ST: Q := FALSE; Это все. Вызвать данное действие из ST-программы можно так:

SyncSwitch1.EmergencyBreak(Q => q);

В графических языках прямоугольник, представляющий данное действие, будет иметь заголовок SyncSwitch1.EmergencyBreak.

Обратите внимание, что окно редактора для действий не имеет раздела объявлений. Компоненты программ с действиями имеют раскрывающиеся списки действий в органайзере объектов. Список действий в разделе объявлений POU не отражается.

Действия аналогичны методам класса в C++. Термин «действие», пожалуй, даже понятнее, чем «метод». Своим происхожде-

нием действия обязаны SFC. В CoDeSys действия можно использовать как подпрограммы.

5.4. Программы

Программа — глобальный программный элемент, отображающий множество значений входных параметров на множество выходных. Программа очень похожа на функциональный блок. Из всех программных компонентов МЭК-программа самый крупный. При помощи программ определяется верхний уровень проекта и реализуется управление многозадачностью. Программы являются глобальными компонентами и объявляются на уровне ресурсов.

5.4.1. Использование программ

Обращение к переменным и вызов программы ничем не отличается от работы с экземпляром функционального блока:

```
AuxiliaryPrg(Active := TRUE, Q => q);
```

Правильный с позиций стандарта проект должен включать одну или несколько программ, ассоциированных с задачами. Число функций и функциональных блоков, как правило, значительно больше.

5.5. Компоненты в CoDeSys

Создание нового программного компонента в CoDeSys выполняется командой **«Add Object...»**. Команда открывает диалоговую панель создания компонента. Вам нужно будет только придумать имя, выбрать язык реализации и указать, что именно требуется создать — функцию, блок или программу. Шаблон компонента, включающий раздел объявлений, создается автоматически. Писать ключевые слова объявлений вручную не приходится. Вся рутинная работа, как и должно быть, выполняется компьютером.

Объявления библиотечных POU содержатся в самих файлах библиотек, которые необходимо подключать к проекту при помощи менеджера библиотек (*Library Manager*).

Иерархический список всех компонентов проекта CoDeSys содержится на страничке «POUs» менеджера проекта *Object Organizer* (рис. 5.10). Для редактирования компонента достаточно выбрать его из списка двойным щелчком мыши. Описание и реали-

зация компонента представляется в отдельном окне. Окна базируются на соответствующем редакторе, текстовом или графическом, в зависимости от языка реализации компонента. Выбор необходимого редактора происходит автоматически.

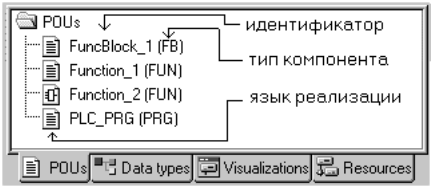

Рис. 5.10. Список компонентов проекта CoDeSys

В CoDeSys есть интересная возможность автоматически конвертировать компонент с одного языка на другой. Автоматический перевод возможен на языки IL, FBD и LD. Конвертирование выполняется командой **«Project» «Object Convert»**.

При создании шаблона нового компонента CoDeSys не прописывает автоматически заголовок **VAR_IN_OUT** в разделе объявлений (и правильно делает). При необходимости его нужно создавать вручную.

В графическом представлении компонента CoDeSys отражает параметры **VAR_IN_OUT** слева, снабжая их специальным значком ▷ «треугольник». В некоторых системах такие параметры отображаются одновременно с двух сторон графического блока. То есть соединительная линия как бы проходит через изображение компонента.

Глобальные переменные **VAR_GLOBAL**, **VAR_ACCESS** и константы в CoDeSys объявляются только в ресурсах. Они существуют в единственном экземпляре и доступны на запись и чтение для всех POU проекта. Обычно глобальные переменные, которые используются внутри компонента, перечисляются в его разделе объявлений под заголовком **VAR_EXTERNAL**. В CoDeSys это делать необязательно.

Функцию с переменным числом параметров в текущей версии CoDeSys создать нельзя. Но стандартные операторы **ADD**, **MUL**, **AND**, **OR**, **XOR** и **MUX** являются расширяемыми. То есть для них допустимо создавать дополнительные входы.

Перегрузка пользовательских функций в CoDeSys не поддерживается. В случае необходимости рекомендуется создать несколько одинаковых функций для разных типов параметров.

Здесь удобно использовать имена функций с суффиксами типов. Например:

Nearby_sint, Nearby_int, Nearby_real

Суффикс типа хорошо бросается в глаза, и смысл его очевиден. Венгерская запись также успешно решает эту проблему:

siNearby, iNearby, rNearby

Обратите внимание, что в отличие от функционального блока в CoDeSys экземпляров программы не существует. Все программы определены глобально и существуют только в единственном экземпляре. Ограничений на количество и способ использования программ в CoDeSys нет.

При создании нового проекта CoDeSys автоматически создает программу с именем PLC_PRG. В однозадачных проектах PLC_PRG является главной программой. Цикл выполнения пользовательской задачи сводится к циклическому вызову PLC_PRG, которая должна содержать вызовы всех прочих необходимых компонентов.

Глава 6. Структура программного обеспечения ПЛК

В этой главе представлен верхний уровень модели МЭК программирования ПЛК. Описаны средства, реализующие выполнение программ и многозадачность, дано понятие о ресурсах и элементах конфигурации. Как и в других главах, общие понятия будут детализированы на примере их реализации в комплексе CoDeSys.

6.1. Задачи

Назначение *задач* состоит в управлении работой программ проекта, исполняемых одним процессором. Как и программа, каждая задача должна иметь собственный уникальный идентификатор. Задачи подразделяются на циклические и разовые (single). Выполнение *разовой задачи* запускается по фронту логической триггерной переменной. Циклические задачи выполняются через заданные интервалы времени. Каждая задача может включать вызов одной или нескольких программ. Если программа имеет входные параметры **(VAR_INPUT)**, то они задаются в описании задачи. Все программы одной задачи выполняются в одном рабочем цикле ПЛК.

Определение задач в системах программирования МЭК выполняется по-разному. Это может быть текстовое описание или графическое представление. CoDeSys содержит специальный инструмент — менеджер задач (Task configuration), представляющий задачи и их программы в виде иерархического дерева (см. рис. 6.1).

В любом проекте всегда существует, как минимум, одна задача. По умолчанию это *циклическая задача*, вызываемая в каждом рабочем цикле ПЛК. В CoDeSys она включает единственную программу PLC_PRG.

Каждая задача обладает определенным приоритетом. Приоритет определяется числом от 0 до 32. Чем меньше число, тем выше приоритет. Если две или более задачи должны получить управление одновременно, то побеждает задача с более высоким приоритетом. Так, если две циклические задачи всегда совпадают по времени и имеют разные приоритеты, то задача с меньшим приори-

тетом не будет работать вовсе. При одинаковом приоритете управление получает задача, имеющая большее время ожидания. То есть две равно приоритетные задачи будут работать поочередно.

В системе исполнения CoDeSys реализована *невытесняющая многозадачность*. Это означает, что любая задача, даже более приоритетная, дает доработать текущей задаче до конца одного рабочего цикла. Работа циклических задач является аппаратно независимой. Механизм управления задачами ПЛК всегда опирается на аппаратный системный таймер, поэтому нельзя гарантировать, что система исполнения обеспечит точность вызова задач до миллисекунды. Как правило, минимальная дискретность временного интервала, на которую вы можете рассчитывать, — 10 мс. Когда речь идет о медленных для ПЛК задачах, интервалы времени измеряются десятыми долями секунды. При таких интервалах время одного рабочего цикла несоизмеримо мало, поэтому механизма невытесняющей многозадачности оказывается достаточно для обеспечения высокой относительной точности работы циклических задач.

Например, задача поддержания рабочего давления воздуха в ресивере при помощи включения и выключения компрессора. Двигателю компрессора необходимо не менее 5 секунд на разгон, а ощутимое увеличение давление произойдет минимум через 1,5...2 минуты. Очевидно, что «дергать» компрессор чаще, чем в 2 минуты, бессмысленно и вредно.

На рис. 6.1 показано определение трех задач. Задача T1 имеет приоритет, равный 10, и должна вызываться через 200 мс. Задача T2 имеет наименьший приоритет, равный 20, и должна вызываться в каждом цикле, не занятом другими задачами. Задача T3 имеет самый высокий приоритет, равный 1, и должна вызываться через 800 мс.

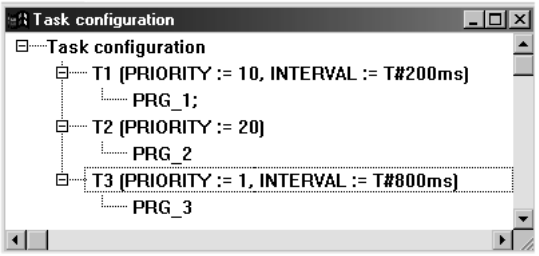

Рис. 6.1. Менеджер задач CoDeSys

При работе на ПЛК, имеющим время рабочего цикла около 55 мс (такой цикл дает Windows эмулятор, привязанный к тикам системного таймера компьютера), временная диаграмма исполнения вышеописанных задач будет выглядеть, как показано на рис. 6.2.

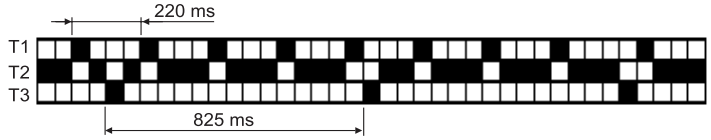

Рис. 6.2. Временная диаграмма исполнения задач T1, T2, T3

Обратите внимание, что тестирование многозадачных проектов требует осознанной работы с отладчиком. Отладчик должен быть настроен на необходимую задачу (в CoDeSys команда **«Set Debug Task»**). Только тогда точки останова будут корректно работать, т. е. останов будет происходить именно при работе данной задачи.

6.2. Ресурсы

С точки зрения стандарта МЭК *ресурс* это один процессор, снабженный собственной системой исполнения. То есть одна или несколько задач загружается в ресурс и исполняется им. В CoDeSys (и не только) применяется понятие проект — все прикладное программное обеспечение, обеспечивающее работу конкретного приложения. Слово же «ресурсы» употребляется во множественном числе и определяет набор аппаратно зависимых деталей проекта. То есть проект включает аппаратно независимые реализации программ (функции, функциональные блоки и их локальные данные) и требующие настройки ресурсы. Ресурсы содержат:

- определение глобальных и прямо адресуемых переменных;
- конфигурацию ПЛК;
- установки целевой системы исполнения (тип микропроцессора, распределение памяти, порядок байт в слове, параметры сети и т. д.);
- менеджер задач.

Сюда же включаются и дополнительные фирменные инструменты, зависящие от особенностей реализации конкретной системы исполнения. Это модуль трассировки переменных, терминал для работы с фирменными командами ядра ПЛК (PLC Browser),

конфигуратор сети и т. д. Это необязательные элементы, их наличие и тонкости работы зависят от конкретной реализации аппаратуры и встроенного программного обеспечения ПЛК.

6.3. Конфигурация

Стандарт МЭК вводит еще одно понятие более высокого уровня, чем ресурс, — это конфигурация. *Конфигурация* — это множество ресурсов, взаимодействующих определенным образом (сконфигурированных). В одной системе может быть несколько интеллектуальных ресурсов, каждый из которых обладает собственным процессором, памятью и системой исполнения. Каждый из них можно программировать. Это могут быть реальные модули (возможно, удаленные) или виртуальные машины, эмулируемые одним процессором. Все они имеют доступ к определенным наборам входов-выходов и координируют свою работу посредством глобальных переменных, расположенных в общедоступной памяти.

Что касается понятия проекта в CoDeSys, то оно непосредственно связано с одним аппаратным ресурсом. То есть для программирования каждого ресурса (в рамках МЭК-конфигурации) должен быть создан отдельный проект. Поскольку система программирования универсальна, то здесь не делается различий для взаимодействия ресурсов и конфигураций. Техника же построения распределенных систем существенно зависит от конкретной реализации сети ПЛК. Это может быть, например, высокоуровневое взаимодействие через переменные общего доступа **(VAR_ACCESS)**, обмен (MAP MMS) сообщениями с помощью библиотеки функциональных блоков стандарта МЭК 61131-5 или работа с удаленными модулями ввода-вывода (CANopen).

С переводом слова «конфигурация» (configuration) в англоязычных описаниях систем программирования МЭК существует неоднозначность. Так, слово «конфигурация», как это описано выше, понимается как существительное (как конфигурация морского дна). Кроме того, есть еще процесс конфигурирования, который также называют «configuration». Так, объект «PLC configuration» на вкладке ресурсов проекта CoDeSys — это инструмент, выполняющий конфигурирование ПЛК.

Глава 7. Языки МЭК

Эта глава начинается с краткого анализа особенностей создания прикладного программного обеспечения ПЛК и представления языков МЭК. Далее каждый из пяти языков МЭК будет рассмотрен более подробно. Специфика реализации языков, существующие ограничения и пути развития будут продемонстрированы на примере транслятора комплекса CoDeSys.

7.1. Проблема программирования ПЛК

Как было описано выше, ПЛК функционирует циклически — чтение входов, выполнение прикладной программы и запись выходов. В результате прикладное программирование для МЭК ПЛК существенно отличается от традиционной модели, применяемой при работе на языках высокого уровня ПК. Рассмотрим в качестве иллюстрации простейшую задачу: необходимо запрограммировать мерцающий световой индикатор. Очевидно, что алгоритм должен быть примерно такой:

1) включить выход;
2) выдержать паузу;
3) выключить выход;
4) выдержать паузу;
5) переход к шагу 1 (начало программы);
6) конец программы.

Реализованная по этому алгоритму программа для ПЛК работать не будет. Во-первых, она содержит бесконечный цикл. Весь код прикладной программы выполняется от начала и до конца в каждом рабочем цикле. Любая прикладная программа ПЛК является частью рабочего цикла и должна возвращать управление системе исполнения. Поэтому шаг 5 «переход на начало программы» лишний.

Если в нашем алгоритме удалить «переход на начало», программа будет работать. Хотя и не так, как задумано. Выход всегда будет оставаться в выключенном состоянии, поскольку физически установка значений выходов производится по окончании прикладной программы один раз. Промежуточные изменения

значений выходов не отображаются на аппаратные средства. Конечно, значение переменной будет изменяться многократно, но определяющим выход станет только последнее значение.

Что еще плохо для ПЛК в данном алгоритме, так это задержка времени. Вполне вероятно, что, кроме мерцания одним выходом, ПЛК должен будет выполнять еще и другую работу. То есть программу необходимо будет дополнять. Но если контроллер занят ожиданием, то в данном алгоритме это означает, что ничего иного он делать не сможет. Значит, выдержку времени необходимо организовать иначе. Достаточно засечь время и заняться другими делами, контролируя периодически часы. Здесь нет ничего особенного. Так поступает обычно и большинство людей в ожидании назначенного часа.

С учетом приведенных соображений алгоритм мерцающего индикатора для ПЛК должен быть таким:

1. Проверить таймер, если время паузы вышло, то:
   а. Инвертировать выход (включить, если выключен, и наоборот);
   б. И начать отсчет новой паузы;
2. Конец программы.

Несмотря на описанные сложности, алгоритм получился в итоге проще. Так и должно быть. Технология ПЛК специально ориентирована на подобные задачи.

Одна из возможных практических реализаций мерцающего индикатора (с двумя таймерами) будет представлена в главе «Примеры программирования», пример «Генератор импульсов».

7.1.1. ПЛК как конечный автомат

Чтобы писать хорошие программы для ПЛК, нужно научиться думать определенным образом. Секрет состоит в том, чтобы представлять себе контроллер не как машину, последовательно выполняющую команды программы, а как *конечный автомат*.

В любом автомате существует множество входов (X), множество выходов (Y) и множество возможных состояний (S). В нашем случае это конечные множества, поскольку число входов-выходов ПЛК ограничено, так же как и объем памяти переменных (определяющих возможные состояния). Начальное состояние ($s_0 \in S$) однозначно определено. Автомат работает по тактам, для ПЛК это рабочий цикл. В каждом такте значения входов известны. Значения выходов определяются (функция выходов λ) значениями вхо-

дов и текущим состоянием. Реакция автомата зависит только от текущего состояния без предыстории, т. е. не важно, как он пришел в данное состояние. Вместе с тем текущее состояние также изменяется по тактам, автомат переходит в новое состояние (функция переходов δ). В теории автоматов описанные шесть объектов $A = \{X, Y, S, s_0, \lambda, \delta\}$ принято называть конечным автоматом Мили.

Мы не будем более подробно углубляться в теорию автоматов. Достаточно понять принцип работы конечного автомата. Классическая сфера применения ПЛК — это программная реализация автоматов. Именно это и обусловило подход к программированию ПЛК. Контроллер вычисляет программно заданную функцию выходов и функцию переходов. В каждом рабочем цикле ПЛК выполняет расчет новых значений для выходов, которые необходимо изменить. В итоге классическая прикладная программа ПЛК оказывается более похожей на вычисление по формуле.

Типовым вводным примером дискретных автоматов с памятью является блок управления стиральной машиной. Базовые механизмы машины включают: клапан подачи воды, нагреватель, привод барабана, помпу слива воды и таймер. Каждому механизму можно сопоставить логическую переменную. Все возможные состояния машины определяются, таким образом, множеством значений переменных. Переход из одного состояния в другое происходит под воздействием входных сигналов. Заметьте, что таймер здесь является самостоятельным блоком. Сигнал окончания выдержки времени является обычным входом.

Несколько расширив понятие автомата, мы можем рассматривать переходы как функции событий. События не обязательно должны быть связаны с входами, это достаточно абстрактное понятие. Тогда окончание таймаута можно будет просто понимать как событие, причем совершенно не важно, как конкретно реализован сам таймер. Модель такой системы удобно изобразить в виде направленного графа состояний (state charts). Состояния отображаются овалами, содержащими значения набора переменных, а переходы — направленными дугами (рис. 7.1). Диаграммы состояний очень эффективный инструмент проектирования и анализа автоматов.

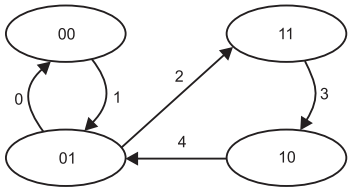

Рис. 7.1. Граф состояний для двух переменных

Техническая база для построения автоматов весьма широка. Это механические узлы, пневматические элементы, реле или логические микросхемы и т. д. Но в отличие от любых других реализаций автоматов технология ПЛК обеспечивает быстрое и исключительно гибкое решение. Безусловно, при построении автоматов на базе программируемых логических матриц и микропроцессоров перепрограммирование также возможно, но значительно более трудоемко. Это можно сделать только при наличии соответствующего оборудования и специальной подготовки.

Реально возможности ПЛК существенно превышают конечные автоматы. Далеко не все, что можно сделать на ПЛК, вписывается в рамки конечных автоматов. Это функции управления по времени, математическая обработка данных, регулирование и т. д. Тем не менее применение формализма конечных автоматов позволяет значительно упростить процесс проектирования.

Причем это относится не только к ПЛК. Подобный подход лежит в основе универсального моделирующего языка Unified Modeling Language (UML) [37]. Пакет расширения Stateflow матричной системы компьютерной математики MATLAB обеспечивает построение анимационных диаграмм состояния моделей различных устройств и систем [34]. Он позволяет выполнять ситуационное моделирование в дополнение к имитационному моделированию, выполняемому мощным пакетом расширения системы MATLAB-Simulink [35].

Только хорошо проработанные технические спецификации проекта позволяют программисту выполнить свою работу качественно и в срок. Неточности в техническом задании или плохо проработанная модель системы неизбежно выливаются в многократные переделки и затяжную отладку. Описание задачи в виде словесного алгоритма и рисунков на бумаге всегда оставляет возможность упустить детали. Самое обидное, что, как правило, детали эти внешне очень просты, но приводят к пересмотру всей структуры построения управляющей программы. Например, забыли кнопку аварийной блокировки или фиксацию промежуточных положений для настройки механики и т. д. Для выявления подобных тонкостей необходим действующий прототип системы и соответственно средства, позволяющие его построить.

Проектирование и отработка модели с применением вспомогательных CASE-инструментов, безусловно, решают эту проблему, но с ростом сложности проектов сроки реализации возрастают слишком резко. Поэтому одной из важнейших задач при созда-

нии языков ПЛК и комплексов программирования является возможность реализовывать прототипы без применения дополнительных средств. Причем это должен быть действующий прототип, а не просто картинка. Высокоуровневая модель, пусть созданная даже из пустых блоков, должна работать так, чтобы ее можно было продемонстрировать заказчику, обсудить и отработать. Далее прототип должен непосредственно стать скелетом готовой программы, без какой-либо специальной переделки. Только так программист получит возможность сразу писать правильную и красивую программу, а не переписывать ее вплоть до превращения в «лоскутное одеяло».

7.2. Семейство языков МЭК

7.2.1. Диаграммы SFC

В семействе МЭК-языков SFC (*Sequential Function Chart*) диаграммы стоят особняком, а точнее, выше по отношению к остальным четырем языкам. Диаграммы SFC являются высокоуровневым графическим инструментом. Благодаря SFC идея превращения модели системы в законченную программу стала реальностью. В отличие от применения вспомогательных средств моделирования SFC дает действующий непосредственно в ПЛК прототип.

Сети Петри

Оригинальный метод формального описания дискретных систем был предложен Карлом Адамом Петри в 1962 году. Он опирается на разделение системы или отдельных ее частей на множество простых позиций. Позиция описывает состояние части системы. Причем состояние понимается здесь достаточно гибко, это может быть состояние оборудования, процесса или программы. Переходы между позициями происходят при выполнении определенных условий. Графически позиция отображается в виде окружности (см. рис. 7.2). Переходам соответствуют отрезки, соединенные с позициями направленными дугами. Каждая позиция способна обладать *маркером* и передавать его другим позициям по исходящим дугам. Маркеры отображается в виде жирной точки. Допускается одновременное присутствие нескольких маркеров. К переходу приходит одна или несколько дуг, идущих от разных позиций. От перехода также могут отходить несколько исходя-

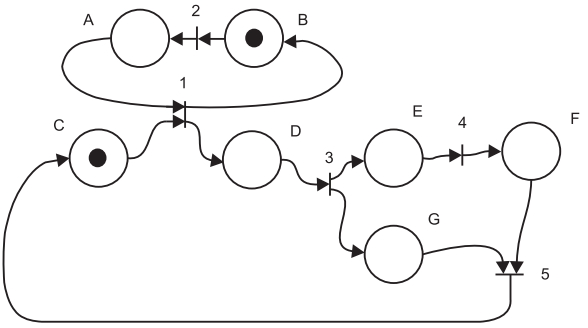

Рис. 7.2. Сеть Петри

щих дуг, ведущих к разным позициям. Проверка условия перехода (разрешение) производится, только если хотя бы одна из его исходных позиций владеет маркером.

Существенным моментом сети Петри является то, что несколько позиций могут одновременно иметь маркеры. То есть сеть описывает процессы, работающие параллельно и взаимосвязанно. Состояние сети определяется совокупностью позиций, владеющих маркерами.

Описанная сеть позиций и переходов является простейшим вариантом сетей Петри. Именно она и послужила прототипом для разработки SFC.

Заслуга первой практической реализации языка этапов и переходов для ПЛК принадлежит французским фирмам. Совместная работа изготовителей ПЛК и объединения пользователей привела к появлению национального стандарта «Графсет» и международного стандарта МЭК 848 (1988 г.). МЭК 61131-3 заимствовал «Графсет» с некоторыми доработками.

В настоящее время отождествлять SFC с сетями Петри уже нельзя. Пути их развития оказались разными, причем не только в отдельных деталях, но и идеологически. Так, в сетях Петри маркер наделили свойствами объекта, а переходы получили способность воспринимать свойства маркера. Это так называемые «цветные» сети Петри, маркеры в которых имеют разные типы и разные цвета. В современном SFC маркер является не более чем квитанцией — нарядом на определенную работу. Позиции же SFC приобрели большую силу и имеют возможность выполнять определенные наборы действий. Чтобы понять разницу, представьте себе коллектив людей, передающих друг другу пакеты (маркеры).

В первом случае действия людей зависят от содержимого пакета, они принимают решение — кому передать, оставить у себя и т. д. Во втором случае действия людей определяются их профессиональными обязанностями и не должны зависеть от содержимого пакета. Так, например, действия маляра должны быть одинаковыми (приготовить поверхность, развести краску, покрасить) для забора и для слитка золота. То есть SFC развивается как инструмент описания производственных операций, а сети Петри расширяют способности моделирования дискретных систем.

SFC-диаграммы

В отличие от сетей Петри дуги в SFC имеют выраженную направленность сверху вниз и отражаются прямыми линиями. Позиции в SFC называют *шагами* или *этапами*. На диаграмме они отражаются в виде прямоугольников. Благодаря такому «кубизму» существует возможность реализации диаграмм в символах псевдографики (рис. 7.3). Задать несколько стартовых шагов в SFC нельзя, только один шаг диаграммы является начальным.

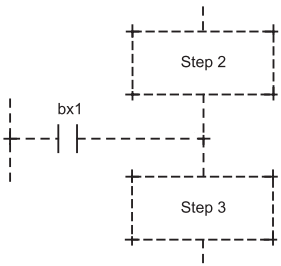

Рис. 7.3. SFC-диаграмма, выполненная символами псевдографики (условие перехода — язык IL)

Графическая диаграмма SFC состоит из шагов и переходов между ними. Разрешение перехода определяется условием. С шагом связаны определенные действия. Описания действий выполняются на любом языке МЭК. Сам SFC не содержит каких-либо управляющих команд ПЛК. Действия могут быть описаны и в виде вложенной SFC-схемы. Можно создать несколько уровней подобных вложений, но в конечном счете действия нижнего уровня все равно необходимо будет описать на IL, ST, LD или FBD.

Заметим сразу, что даже упрощенная реализация SFC требует неявно некоторой памяти для кодирования маркеров позиций.

Требование параллельности приводит к невозможности компактно закодировать все состояния. То есть невозможно использовать одну переменную, которая содержит единственный номер активной позиции. Каждая позиция требует собственных признаков активности. Механизм кодирования может быть достаточно изощренным, но, к счастью, его реализация ложится исключительно на плечи разработчиков системы программирования. На прикладном уровне задумываться о деталях реализации не приходится. Единственное, что нужно учесть, это небольшой дополнительный расход памяти данных и кода.

Из-за необходимости внутренней памяти только функциональные блоки и программы могут быть реализованы в SFC, функции такой возможности лишены.

Целью применения SFC является разделение задачи на простые этапы с формально определенной логикой работы системы. SFC дает возможность быстрого построения прототипа системы без программирования. Причем для отработки верхнего уровня не требуется детальное описание действий, так же как и привязка к конкретным аппаратным средствам.

7.2.2. Список инструкций IL

Язык IL (Instruction list) дословно — список инструкций. Это типичный ассемблер с аккумулятором и переходами по меткам. Набор инструкций стандартизован и не зависит от конкретной целевой платформы. Поскольку IL самый простой в реализации язык, он получил очень широкое распространение до принятия стандарта МЭК. Точнее, не сам IL, а очень похожие на него реализации. Практически все производители ПЛК Европы создавали подобные системы программирования, похожие на современный язык IL. Существуют примеры реализации команд и на основе русскоязычных аббревиатур [8]. Наибольшее влияние на формирование современного IL оказал язык программирования STEP контроллеров фирмы Siemens. Язык IL позволяет работать с любыми типами данных, вызывать функции и функциональные блоки, реализованные на любом языке. Таким образом, на IL можно реализовать алгоритм любой сложности, хотя текст будет достаточно громоздким.

В составе МЭК-языков IL применяется при создании компактных компонентов, требующих тщательной проработки, на которую не жалко времени. При работе с IL гораздо адекватнее, чем с другими языками, можно представить, как будет выглядеть от-

транслированный код. Благодаря чему, IL выигрывает там, где нужно достичь наивысшей эффективности. К компиляторам это относится в полной мере. В системах исполнения с интерпретатором промежуточного кода выигрыш не столь значителен.

7.2.3. Структурированный текст ST

Язык ST (Structured Text) — это язык высокого уровня. Синтаксически ST представляет собой несколько адаптированный язык Паскаль. Вместо процедур Паскаля в ST используются компоненты программ стандарта МЭК.

Для специалистов, знакомых с языком C, освоение ST также не вызовет никаких сложностей. В качестве иллюстрации сравним эквивалентные программы на языках ST и C:

ST:

```
WHILE Counter<>0 DO
   Counter := Counter-1;
   Var1 := Var1*2;

   IF Var1 > 100 THEN
       Var1 := 1;
       Var2 := Var2 + 1;
   END_IF
END_WHILE
```

C:

```
while (Counter— ! = 0)
{
Var1 *= 2;

if(Var1 > 100)
       {
       Var1 = 1;
       ++Var2;
       }
}/*while*/
```

В большинстве комплексов программирования ПЛК язык ST по умолчанию предлагается для описания действий и условий переходов SFC. Это действительно максимально мощный тандем, позволяющий эффективно решать любые задачи.

7.2.4. Релейные диаграммы LD

Язык релейных диаграмм LD (Ladder Diagram) или *релейно-контактных схем* (РКС) — графический язык, реализующий структуры электрических цепей. РКС — это американское изобретение. В начале 70-х гг. XX в. релейные автоматы сборочных конвейеров начали постепенно вытеснятся программируемыми контроллерами. Некоторое время те и другие работали одновременно и обслуживались одними и теми же людьми.

Так появилась задача прозрачного переноса релейных схем в ПЛК. Различные варианты программной реализации релейных схем создавались практически всеми ведущими производителями ПЛК. Благодаря простоте представления РКС обрел заслуженную популярность, что и стало основной причиной включения его в стандарт МЭК.

Слова «релейная логика» звучат сегодня достаточно архаично, почти как «ламповый компьютер». Тем более в связи с созданием многочисленных быстродействующих и надежных бесконтактных (в частности, оптоэлектронных) реле и мощных переключающих приборов, таких как мощные полевые транзисторы, управляемые тиристоры и приборы IGBT [36]. Но, несмотря на это, релейная техника все еще очень широко применяется.

Графически LD-диаграмма представлена в виде двух вертикальных шин питания. Между ними расположены цепи, образованные соединением контактов (см. рис. 7.5). Нагрузкой каждой цепи служит реле. Каждое реле имеет контакты, которые можно использовать в других цепях.

Логически последовательное (И), параллельное (ИЛИ) соединение контактов и инверсия (НЕ) образуют базис Буля. В результате LD идеально подходит не только для построения релейных автоматов, но и для программной реализации комбинационных логических схем. Благодаря возможности включения в LD функций и функциональных блоков, выполненных на других языках, сфера применения языка практически не ограничена.

7.2.5. Функциональные диаграммы FBD

FBD (Function Block Diagram) — это графический язык программирования. Диаграмма FBD очень напоминает принципиальную схему электронного устройства на микросхемах (см. рис. 7.16). В отличии от LD «проводники» в FBD могут проводить сигналы (передавать переменные) любого типа (логический, аналоговый, время и т. д.). Иногда говорят, что в релейных схемах соединительные проводники передают энергию. Проводники FBD тоже передают энергию, но в более широком смысле. Здесь слово «энергия» применимо в том смысле, в котором им оперируют не электрики, а экстрасенсы. Очевидно, что шины питания и контакты здесь уже не эффективны. Шины питания на FBD диаграмме не показываются. Выходы блоков могут быть поданы на входы других блоков либо непосредственно на выходы

ПЛК. Сами блоки, представленные на схеме как «черные ящики», могут выполнять любые функции.

FBD-схемы очень четко отражают взаимосвязь входов и выходов диаграммы. Если алгоритм изначально хорошо описывается с позиции сигналов, то его FBD-представление всегда получается нагляднее, чем в текстовых языках.

7.3. Язык линейных инструкций (IL)

7.3.1. Формат инструкции

Текст на IL — это текстовый список последовательных инструкций. Каждая инструкция записывается на отдельной строке. Инструкция может включать 4 поля, разделенные пробелами или знаками табуляции:

Метка: Оператор Операнд Комментарий

Метка инструкции не является обязательной, она ставится только там, где нужно. Оператор присутствует обязательно. Операнд необходим почти всегда. Комментарий — необязательное поле, записывается в конце строки. Ставить комментарии между полями инструкции нельзя. Пример IL-программы:

```
METKA1:  LD      Sync      (*пример IL*)
         AND     Start
         S       Q
(*для красоты метку можно поставить в отдельную строку*)
METKA2:
         LD      2         (* y = 2 + 2 *)
         ADD     2
         ST      y
```

Для лучшего восприятия строки IL выравнивают обычно в колонки по полям.

Редактор CoDeSys выравнивает текст автоматически. Помимо этого, редактор «налету» выполняет синтаксический контроль и выделение цветом. Так, корректно введенные операторы выделяются голубым цветом. К сожалению, черно-белая печать книги лишает читателя удовольствия наблюдать цветные выделения во всех листингах программ этой книги.

7.3.2. Аккумулятор

Абсолютное большинство инструкции IL выполняют некоторую операцию с содержимым аккумулятора (см. его определение чуть ниже). Операнд, конечно, тоже принимает участие в инструкции, но результат опять помещается в аккумулятор. Например, инструкция **SUB 10** отнимает число 10 от значения аккумулятора и помещает результат в аккумулятор. Команды сравнения сравнивают значение операнда и аккумулятора, результат сравнения ИСТИНА или ЛОЖЬ вновь помещается в аккумулятор. Команды перехода на метку способны анализировать аккумулятор и принимать решение — выполнять переход или нет.

Аккумулятор IL является универсальным контейнером, способным сохранять значения переменных любого типа.

В аккумулятор можно поместить значение типа **BOOL**, затем **INT** или **REAL**, транслятор не будет считать это ошибкой. Такая гибкость не означает, что аккумулятор способен одновременно содержать несколько значений разных типов. Только одно, причем тип значения также фиксируется в аккумуляторе. Если операция требует значение другого типа, транслятор выдаст ошибку.

В стандарте МЭК вместо термина «аккумулятор» используется термин «результат» (result). Так, инструкция берет «текущий результат» и формирует «новый результат». Тем не менее почти все руководства по программированию различных фирм широко используют термин «аккумулятор».

7.3.3. Переход на метку

Программа на IL выполняется подряд, сверху вниз. Для изменения порядка выполнения и организации циклов применяется *переход на метку*. Переход на метку может быть безусловным **JMP** — выполняется всегда, независимо от чего-либо. Условный переход **JMPC** выполняется только при значении аккумулятора ИСТИНА. Переход можно выполнять как вверх, так и вниз. Метки являются локальными, другими словами, переход на метку в другом POU не допускается.

Переходы нужно организовывать достаточно аккуратно, чтобы не получить бесконечный цикл:

```
LD      1
ST      Counter
```

```
loop1:
   (*тело цикла*)

   LD                          Counter
   ADD    1
   ST     Counter
   LE     5
   JMP    loop1
```

В примере показана реализация цикла на 5 повторений с одной очевидной ошибкой. Вместо безусловного перехода **JMP** должен быть **JMPC.**

В системах с интерпретатором IL или промежуточным кодированием время выполнения перехода оказывается зависимым от направления и расстояния. В CoDeSys команда безусловного перехода транслируется в одну машинную команду процессора и выполняется очень быстро. Ограничений на число переходов в CoDeSys нет.

Но это не означает целесообразность создания больших монолитных IL-программ с множеством переходов. Такие программы очень сложно читать и не легко сопровождать.

7.3.4. Скобки

Последовательный порядок выполнения команд IL можно изменять при помощи *скобок*. Открывающая скобка ставится в инструкции после операции. Закрывающая скобка ставится в отдельной строке. Инструкции, заключенные в скобки, выполняются в первую очередь. Результат вычисления инструкций в скобках помещается в дополнительный аккумулятор, после чего выполняется команда, содержащая открывающую скобку. Например:

```
LD      5
MUL     (2
SUB     1
)
ST      y          (*y  =  5 * (2 – 1) = 5*)

LD      5
MUL     2
SUB     1
ST      y          (*y = 5 * 2 – 1 = 9*)
```

Скобки могут быть *вложенными*. Каждое вложение требует организации некоего временного аккумулятора. Это вызывает неоднозначность при выходе из блока скобок командами **JMP**, **RET**, **CAL** и **LD**. Применять эти команды в скобках нельзя.

7.3.5. Модификаторы

Добавление к мнемонике некоторых операторов символов — *модификаторов* **'C'** и **'N'** модифицирует смысл инструкции.

Символ **'N'** (negation) вызывает инверсию значения операнда до выполнения инструкции. Операнд должен быть типов **BOOL, BYTE**, **WORD** или **DWORD**.

Символ **'C'** (condition) добавляет проверку условий к командам перехода, вызова и возврата. Команды **JMPC**, **CALC**, **RETC** будут выполняться только при значении аккумулятора ИСТИНА. Добавление символа **'N'** приводит к сравнению условия с инверсным значением аккумулятора. Команды **JMPCN**, **CALCN**, **RETCN** будут выполняться только при значении аккумулятора ЛОЖЬ. Модификатор **'N'** без **'C'** не имеет смысла в данных операциях и не применяется.

7.3.6. Операторы

Стандартные операторы IL с допустимыми модификаторами представлены в таблице.

Оператор	Модификатор	Описание
LD	N	Загрузить значение операнда в аккумулятор
ST	N	Присвоить значение аккумулятора операнду
S		Если аккумулятор ИСТИНА, установить логический операнд (ИСТИНА)
R		Если аккумулятор ИСТИНА, сбросить логический операнд (ЛОЖЬ)
AND	N, (	Поразрядное И
OR	N, (	Поразрядное ИЛИ

Оператор	Модификатор	Описание
XOR	N, (	Поразрядное ИЛИ
NOT		Поразрядная инверсия аккумулятора
ADD	(	Сложение
SUB	(	Вычитание
MUL	(	Умножение
DIV	(	Деление
MOD	(	Деление по модулю
GT	(	>
GE	(	>=
QE	(	=
NE	(	< >
LE	(	<=
LT	(	<
JMP	CN	Переход к метке
CAL	CN	Вызов функционального блока
RET	CN	Выход из POU и возврат в вызывающую программу.

Операторы S и R применяются только с операндами типа **BOOL**. Прочие операторы работают с любыми переменными базовых типов.

Приведенный список содержит операторы, поддерживаемые в обязательном порядке. Трансляторы кода CoDeSys для различных аппаратных платформ реализуют различные подмножества дополнительных операторов.

7.3.7. Вызов функциональных блоков и программ

Вызвать экземпляр функционального блока или программу в IL можно с одновременным присваиванием переменных. Например:

```
CAL     CTD_1(CD := TRUE, LOAD := FALSE, PV := 10)
LD      CTD_1.CV
ST      y
```

Аналогичный вызов можно выполнить с предварительным присваиванием значений входных переменных:

```
LD      TRUE
ST      CTD_1.CD
LD      FALSE
ST      CTD_1.LOAD
LD      10
ST      CTD_1.PV
CAL     CTD_1
LD      CTD_1.CV
ST      y
```

7.3.8. Вызов функции

При *вызове функции* с перечислением параметров в IL существует одна немаловажная особенность. В качестве первого параметра используется аккумулятор:

```
LD      TRUE
SEL     3,4
```

На ST это равносильно вызову **SEL** (TRUE,3,4). Очевидно, при вызове функции или оператора с одним параметром список параметров вообще не нужен:

```
LD      ivar1
INT_TO_BOOL
ST      bvar1
```

7.3.9. Комментирование текста

Язык IL является языком низкого уровня. Поэтому тексты IL всегда достаточно велики по объему. Обычно при написании программы алгоритм кажется настолько понятным, что не нуждается в *комментариях*. Однако текст чужой программы без комментариев понять очень тяжело. Нередко бывает, что и самому автору программы через год-другой после ее написания трудно в ней

разобраться. Возможность комментировать каждую строку не означает, что так и нужно поступать. Правильно составленное пояснение должно пояснять суть проблемы и идею решения, а не описывать сами команды. Например: (*Игнорировать колебания до 5 единиц*) — плохой комментарий. (*Колебания замеров до 5 единиц являются шумом*) — значительно лучше.

Транслятор IL CoDeSys допускает многострочные комментарии. Целостное пояснение всегда воспринимается лучше, чем короткие обрывочные комментарии в строках инструкций.

Комментарии МЭК имеют один существенный недостаток. Если при отладке понадобится временно отключить часть исходного текста, то проще всего его целиком закомментировать. Здесь обычно и возникает проблема вложенных комментариев. Вложенные комментарии не столь страшны для транслятора CoDeSys, сколько доставляют неудобства при правке текста. Строковые комментарии (вводимые с помощью ; в ассемблере и // в C++), к сожалению, стандартом МЭК не предусмотрены.

7.3.10. IL в режиме исполнения

В *режиме исполнения* CoDeSys автоматически превращает окно редактора в окно мониторинга. Значения всех переменных отображаются справа от команд IL и доступны для изменения. Вы можете установить или сбросить точку останова и прошагать программу по одной команде. Интересные возможности для IL предоставляет режим Flow control CoDeSys. В этом режиме в окне мониторинга подсвечиваются номера строк, которые исполнялись в предыдущем рабочем цикле, и отображаются соответствующие значения аккумулятора (рис. 7.4).

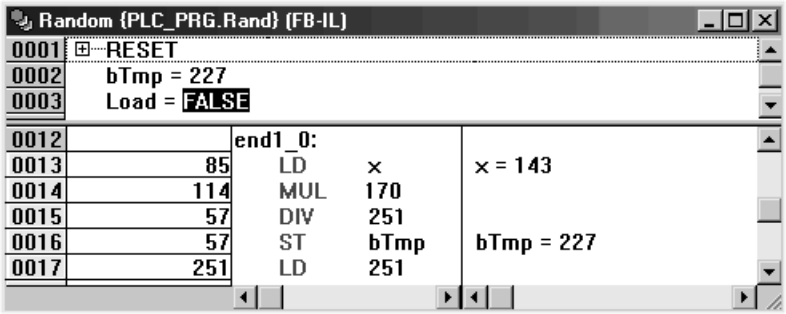

Рис. 7.4. IL в режиме исполнения

7.4. Структурированный текст (ST)

7.4.1. Выражения

Основой ST-программы служат выражения. Результат вычисления выражения присваивается переменной при помощи оператора «:=», как и в Паскале. Каждое выражение обязательно заканчивается точкой с запятой «;». Выражение состоит из переменных констант и функций, разделенных операторами:

```
iVar1 := 1 + iVar2 / ABS(iVar2);
```

Стандартные операторы в выражениях ST имеют символьное представление, например математические действия: +, –, *, /, операции сравнения и т. д.

Помимо операторов, элементы выражения можно отделять пробелами и табуляциями для лучшего восприятия. В текст могут быть введены комментарии. Везде, где допустимы пассивные разделители, можно вставлять и комментарии:

```
iVar1 := 1 + (*получить знак*)  iVar2 / ABS(iVar2); (*проверка на 0 была выше*)
```

Несколько выражений можно записать подряд в одну строку. Но хорошим стилем считается запись одного выражения в строке. Длинные выражения можно перенести на следующую строку. Перенос строки равноценен пассивному разделителю.

Выражение может включать другое выражение, заключенное в скобки. Выражение, заключенное в скобки, вычисляется в первую очередь.

Тип выражения определяется типом результата вычислений:

```
bAlarm := byInp1 > byInp2 AND byInp1 + byInp2 <> 0 OR bAlarm2;
```

7.4.2. Порядок вычисления выражений

Вычисление выражения происходит в соответствии с правилами *приоритета операций*. Первыми выполняются операции с наивысшим приоритетом.

В порядке уменьшения приоритета операции располагаются так: выражение в скобках; вызов функции; степень **EXPT**; замена знака (–); отрицание **NOT** ; умножение, деление и деление по модулю **MOD**; сложение и вычитание (+, –); операции сравнения

(<, >, <=, >=); равенство (=); неравенство (<>); логические операции **AND**, **XOR** и **OR**.

Приоритет операций в выражениях очень важен. В первую очередь с математической точки зрения:

```
X := 2 + 2 * 2;        (* = 6*)
X := (2 + 2) * 2;      (* = 8*)
```

Здесь результаты очевидны. Умножение имеет более высокий приоритет, чем сложение, и выполняется раньше. Скобки меняют порядок вычислений, и результат оказывается другим.

При составлении выражений обязательно необходимо учитывать возможный диапазон изменения значений и типы переменных. Ошибки, связанные с переполнением, возникают в процессе выполнения и не могут быть обнаружены транслятором.

```
iVar1:      SINT;
siVar2:     SINT := 120;
siVar1 := 120 - siVar2 + 20;      (*120 – 120 = 0,  0 + 20 = 20*)
siVar1 := 120 - (siVar2 + 20);    (*120 + 20 = –116, 120 + 116 = –20*)
```

7.4.3. Пустое выражение

Пустое выражение состоит из точки с запятой «;» . Для точки с запятой транслятор не генерирует никакого кода. Если случайно поставить лишнюю «;», это не вызовет ошибки. Единственное осмысленное применение пустого выражения — это обеспечение правильности языковых конструкций. Например, может потребоваться оттранслировать проект, содержащий еще не реализованный POU. Для корректной трансляции достаточно написать в теле POU один пустой оператор. Еще один пример, где пустой оператор оказывается весьма кстати, это условие **IF**, не содержащее раздел **THEN**:

```
IF x = Threshold THEN
    ;                              (*все хорошо*)
ELSIF x > Threshold THEN
    bMarker := bMarker – 1;        (*шаг вниз*)
ELSE
    bMarker := bMarker + 1;        (*шаг вверх*)
END_IF;   (*лишняя ; – поставлена по Паскалевской привычке*)
```

7.4.4. Оператор выбора IF

Оператор выбора позволяет выполнить различные группы выражений в зависимости от условий, выраженных логическими выражениями. Полный синтаксис оператора IF (если) выглядит так:

```
IF  <логическое выражение IF>
THEN
  <выражения IF> ;
[
ELSIF  <логическое выражение ELSEIF 1>
THEN
  < выражения ELSEIF 1> ;
 …
ELSIF  <логическое выражение ELSEIF n>
THEN
  < выражения ELSEIF n> ;
ELSE
  < выражения ELSE> ;
]
END_IF
```

Если <логическое выражение IF> ИСТИНА, то выполняются выражения первой группы — <выражения IF>. Прочие выражения пропускаются, альтернативные условия не проверяются.

Часть конструкции в квадратных скобках является необязательной и может отсутствовать.

Если <логическое выражение IF> ЛОЖЬ, то одно за другим проверяются условия ELSIF. Первое истинное условие приведет к выполнению соответствующей группы выражений. Прочие условия **ELSIF** анализироваться не будут. Групп **ELSIF** может быть несколько или не быть совсем.

Если все логические выражения дали ложный результат, то выполняются выражения группы **ELSE**, если она есть. Если группы **ELSE** нет, то не выполняется ничего.

В простейшем случае оператор **IF** содержит только одно условие:

```
IF bReset THEN
  iVar1 := 1;
  iVar2 := 0;
END_IF
```

На первый взгляд конструкция IF с несколькими группами ELSIF выглядит сложной, но на самом деле оказывается достаточно выразительной:

```
IF bReset THEN
  iVar1 := 1;
  ELSIF  byLeft < 16 THEN
    iVar1 := 2;
  ELSIF  byLeft < 32 THEN
    iVar1 := 3;
  ELSIF  byLeft < 64 THEN
    iVar1 := 4;
ELSE
  bReset := TRUE;
END_IF
```

7.4.5. Оператор множественного выбора CASE

Оператор множественного выбора **CASE** позволяет выполнить различные группы выражений в зависимости от значения одной целочисленной переменной или выражения. Синтаксис:

```
CASE <целочисленное выражение> OF
    <значение 1>:
          <выражения 1> ;

    <значение 2> , <значение 3> :
          <выражения 3> ;

     <значение 4>..<значение 5> :
          <выражения 4> ;
      ...
[
ELSE
       <выражения ELSE> ;
]
END_CASE
```

Если значение выражения совпадает с заданной константой, то выполняется соответствующая группа выражений. Прочие условия не анализируются (<значение 1>: <выражения 1> ;).

Если несколько значений констант должны соответствовать одной группе выражений, их можно перечислить через запятую (<значение 2> , <значение 3> : <выражения 3> ;).

Диапазон значений можно определить через двоеточие (<значение 4>..<значение 5> : <выражения 4> ;).

Группа выражений **ELSE** является необязательной. Она выполняется при несовпадении ни одного из условий (<выражения ELSE> ;).

Пример:

```
CASE byLeft/2 OF
0,127:
        bReset := TRUE;
        Var1 := 0;
16..24:
        Var1 := 1;
ELSE
        Var1 := 2;
END_CASE
```

Значениями выбора **CASE** могут быть только целые константы, переменные использовать нельзя. Одинаковые значения в альтернативах выбора задавать нельзя, даже в диапазонах. Так, следующий пример обязан вызвать ошибку при трансляции:

```
CASE byLeft OF
20:     Var1 := 0;
16..24: Var1 := 1;
END_CASE
```

Безусловно, оператор **CASE** «слабее» оператора **IF**, не имеющего подобных ограничений. Но формат **CASE** не только выразителен для программиста, но и более эффективен. Использование целочисленных констант позволяет транслятору выполнить оптимизацию кода, часто весьма существенную.

7.4.6. Циклы WHILE и REPEAT

Циклы **WHILE** и **REPEAT** обеспечивают повторение группы выражений, пока верно условное логическое выражение. Если условное выражение всегда истинно, то цикл становится бесконечным.

Синтаксис **WHILE**:

WHILE <Условное логическое выражение> **DO**
  <Выражения — тело цикла>
END_WHILE

Условие в цикле WHILE проверяется до начала цикла. Если логическое выражение изначально имеет значение ЛОЖЬ, тело цикла не будет выполнено ни разу.

Синтаксис REPEAT:

REPEAT
  <Выражения — тело цикла >
UNTIL <Условное логическое выражение>
END_REPEAT

Условие в цикле REPEAT проверяется после выполнения тела цикла. Если логическое выражение изначально имеет значение ЛОЖЬ, тело цикла будет выполнено один раз.

Пример:

```
ci := 64;
WHILE ci > 1 DO
    Var1 := Var1 + 1;
    ci := ci/2;
END_WHILE
```

Правильно построенный цикл WHILE или REPEAT обязательно должен изменять переменные, составляющие условие окончания в теле цикла, постепенно приближаясь к условию завершения. Если этого не сделать, цикл не закончится никогда.

Старайтесь не использовать точное равенство и неравенство для прекращения цикла. Иначе есть вероятность ошибочно проскочить граничное условие. Лучше использовать условия больше и меньше. В следующем примере ошибка хорошо видна исключительно благодаря его простоте:

```
ci := 1;
WHILE ci <> 100 DO
    Var1 := Var1 + 1;
    ci := ci + 10;
END_WHILE
```

Очевидно, счетчик ci при начальном значении 1 и приращении 10 никогда не станет равным 100.

Для реализации минимального времени выполнения цикла необходимо избегать в теле цикла и в условном выражении вычислений, которые можно было сделать заранее. Такие вычисления повторяются в цикле, всякий раз отнимая время. Например:

```
WHILE ci < 5 + x DO
    Var := Var1 + 2*x*x + 1;
    ci := ci + 1;
END_WHILE
```

Данный цикл можно оптимизировать по скорости:

```
iMax := 5+x;
iPoly := 2*x*x + 1;
WHILE ci < iMax DO
     Var := Var1 + iPoly;
     ci := ci + 1;
END_WHILE
```

7.4.7. Цикл FOR

Цикл **FOR** обеспечивает *заданное* количество повторений группы выражений. Синтаксис:

```
FOR <Целый счетчик> := <Начальное значение>
TO <Конечное значение>
[BY <Шаг>] DO
  <Выражения — тело цикла>
END_FOR
```

Перед выполнением цикла счетчик получает начальное значение. Далее тело цикла повторяется, пока значение счетчика не превысит конечного значения. Счетчик увеличивается в каждом

цикле. Начальное и конечное значения и шаг могут быть как константами, так и выражениями.

Счетчик изменяется после выполнения тела цикла. Поэтому если задать конечное значение меньшее начального, то при положительном приращении цикл не будет выполнен ни разу. При одинаковых начальном и конечном значениях тело цикла будет выполнено один раз.

Часть конструкции **BY** в скобках необязательна, она определяет шаг приращения счетчика. По умолчанию счетчик увеличивается на единицу в каждой итерации. В качестве счетчика можно использовать переменную любого целого типа. Пример:

```
Var1 := 0;
FOR cw := 1 TO 10  DO
   Var1 := Var1 + 1;
END_FOR
```

Данный цикл будет выполнен 10 раз и соответственно Var1 будет иметь значение 10.

Шаг изменения счетчика итераций может быть и отрицательным. Начальное условие в этом случае должно быть больше конечного. Цикл будет закончен, когда значение счетчика станет меньше конечного значения. Например:

```
Var1 := 0;
FOR ci := 10 TO 1 BY -1 DO
   Var1 := Var1 + 1;
END_FOR
```

Цикл FOR исключительно удобен для итераций с заранее известным числом повторов. Причем, чтобы создать бесконечный цикл FOR, нужно хорошо постараться. Можно, например, попытаться задать нулевой шаг приращения (в CoDeSys это не помогает) или сбросить счетчик в теле цикла.

Для построения правильного цикла достаточно соблюдать два простых формальных требования:

- не изменяйте счетчик цикла и условие окончания в теле цикла. Счетчик и переменные образующие конечное условие в цикле, можно использовать только для чтения;
- не задавайте в качестве конечного условия максимальное для типа переменной счетчика значение. Так, если для однобайтного целого без знака задать константу 255, то условие

окончания не будет выполнено никогда. Цикл станет бесконечным.

В CoDeSys счетчик изменяется в теле цикла всегда, включая завершающую итерацию, когда условие окончания уже достигнуто. Но в стандарте такие тонкости не оговорены. С целью оптимизации по скорости транслятор может обойти изменение счетчика лишний раз. Поэтому не рекомендуется использовать значение счетчика вне тела цикла. Переносимость такой программы гарантировать нельзя. Избегайте использовать цикл FOR со сложными условиями окончания и в случаях, когда по окончании цикла необходимо определить причину окончания. Например, когда цикл может быть прерван оператором EXIT и есть необходимость узнать, сколько итераций в действительности было выполнено (см. ниже пример с EXIT).

Второй момент, который необходимо учитывать при создании переносимой программы, это отрицательное приращение счетчика. К сожалению, далеко не все системы программирования поддерживают такую возможность.

7.4.8. Прерывание итераций операторами EXIT и RETURN

Оператор EXIT, помещенный в теле циклов WHILE, REPEAT и FOR, приводит к немедленному окончанию цикла. Хороший стиль программирования призывает избегать такого приема, но иногда он весьма удобен. Рассмотрим, например, поиск элемента массива с определенным значением (x). Проще всего организовать линейный перебор при помощи цикла FOR:

```
bObtained:= FALSE;
FOR cN := 1 TO MaxIndex  DO
    IF x = aX[cN] THEN
        Index      := cN;
        bObtained := TRUE;
        EXIT;
    END_IF
END_FOR
IF bObtained THEN    (*элемент найден, его индекс — Index*)
```

Для вложенного цикла оператор EXIT завершает только «свой» цикл, внешний цикл будет продолжать работу. Например:

```
FOR y := 0 TO 9  DO
  FOR x := 0 TO 99  DO (* обрабатываем строку массива Arr[y][x]*)
    ;
    IF ... THEN EXIT;     (*'хвост' строки обрабатывать не
                           надо, переходим к следующей*)
  END_FOR
END_FOR
```

При необходимости завершения внешнего цикла по условию, возникшему во вложенном цикле, можно использовать пару синхронизированных операторов **EXIT**:

```
bBreakY := FALSE;
FOR y := 0 TO 9  DO
  FOR x := 0 TO 99  DO
    ;
    IF ... THEN bBreakY := TRUE; EXIT;  (*прервать обработку*)
  END_FOR
  IF bBreakY THEN EXIT;
END_FOR
```

Оператор **RETURN** осуществляет немедленный возврат из POU. Это единственный способ прервать вложенные итерации без введения дополнительных проверок условий. Оператор **RETURN** выполняется очень быстро, фактически это одна машинная команда процессора. Но не стоит им злоупотреблять. Поскольку в тексте компонента, имеющего, например, 50 выходов, разобраться весьма не просто.

Иногда бывает удобно создать безусловный цикл, а условия выхода формировать в теле цикла с использованием **EXIT**. Например, могут потребоваться несколько равновероятных, но невзаимосвязанных условий выхода из цикла. Создать безусловный (бесконечный) цикл в ST проще всего так: **WHILE** TRUE **DO...**

7.4.9. Итерации на базе рабочего цикла ПЛК

Использовать для условия выхода из циклов **WHILE** и **REPEAT** входы, выходы или другие аппаратно-зависимые переменные ПЛК нельзя. Данные переменные не изменяют своих значений в пределах одного рабочего цикла пользовательской программы, поэтому цикл всегда будет бесконечным. Если подобная

необходимость все же есть, используйте для итераций рабочий цикл ПЛК. Выполнение итерации задается простым условием **IF**.

Аналогично можно поступать при необходимости построения длительных циклов. Например, инициализация или копирование больших массивов данных. «Размазывание» объемных операций на множество рабочих циклов является стандартным приемом, позволяющим избежать нежелательного замедления других, параллельно выполняемых задач.

Так, инициализация данных функционального блока с выставлением сигнала готовности может выглядеть так:

```
ENO:          BOOL := FALSE;
niCounter:    INT  := 1000;
aiVar:    ARRAY[0..999] OF INT;

IF niCounter = 0 THEN
   ENO := TRUE;
   …                              (*основная работа блока*)
ELSE
   niCounter := niCounter – 1;
   aiVar[niCounter] := GetInitVal(niCounter);
                                  (*сложная инициализация*)
END_IF
```

7.4.10. Оформление текста

Оформление текстов ST-программ может быть совершенно произвольным. Расположение операторов и выражений в строке не влияет на правильность программ. Но очень важно выработать свой собственный стиль и строго придерживаться его. Важнейшую роль в оформлении играют *отступы* в начале строк. Отступы зрительно объединяют строки, содержащие выражения одного уровня вложения. Текст, выровненный в виде лесенки, каждая ступенька которой относится к одному циклу или условию, читается легко. Несмотря на возможность горизонтальной прокрутки в редакторе, желательно, чтобы по ширине текст помещался на одной странице. Не стоит располагать несколько выражений в одну строку. Ничего страшного нет в том, что текст окажется растянутым по вертикали: лаконичные выражения и даже пустые строки только помогают зрительному анализу (см. пример ниже).

```
FOR icY := 0 TO 8  DO
   FOR icX := 0 TO 16  DO
      IF iaPos[icY,icX] > iLevel THEN
         iBalance := iBalance + 1;
      ELSE
         IF iaPos[icY,icX] < iLevel THEN
            iBalance := iBalance − 1;
         END_IF
      END_IF
   END_FOR
   iLevel := iLevel *2;
END_FOR
```

Плохо оформленный ST-текст читать крайне тяжело, даже редактор с цветовым выделением инструкций здесь не спасает. Мало того, ошибки в схеме отступов способны совершенно сбить с толку:

```
FOR icY := 0 TO 8  DO
   FOR icX := 0 TO 16  DO
         IF iaPos[icY,icX] > iLevel THEN
         iBalance := iBalance + 1;
         ELSE
   IF iaPos[icY,icX] < iLevel THEN
         iBalance := iBalance − 1;
   END_IF
   END_IF
   END_FOR
iLevel := iLevel *2;
END_FOR
```

Для оформления ST текстов вполне применимы рекомендации, которые можно встретить в литературе по программированию на Паскале и C. Обратите внимание, что в ST отсутствуют пресловутые программные скобки (в Паскале: begin, end; в C: {}). Вместо них каждое выражение языка имеет собственную концовку **(WHILE .. END_WHILE, IF .. END_IF)**. То есть закрывающая программная скобка является информативной. Зрительно такой текст воспринимается явно лучше. При создании сложных вло-

жений в языке C закрывающие скобки часто расположены сплошной лесенкой. В таких случаях опытные программисты применяют краткие комментарии после каждой закрывающей скобки. Комментарии подсказывают, с чего начат данный уровень отступа. Например: *(*FOR x*)*. Это хороший прием, но при грамотном применении отступов в строках ST такая необходимость возникает значительно реже, чем в C и Паскале.

7.5. Релейные диаграммы (LD)

7.5.1. Цепи

Релейная схема представляет собой две вертикальные шины питания, между ними расположены горизонтальные цепи, образованные контактами и обмотками реле. Количество контактов в цепи произвольно, реле одно. Если последовательно соединенные контакты замкнуты, ток идет по цепи и реле включается (в примере на рис. 7.5 Lamp1). При необходимости можно включить параллельно несколько реле, последовательное включение не допускается.

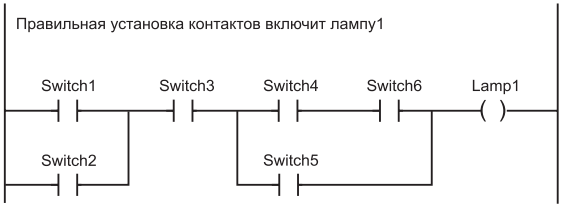

Рис. 7.5. Схема LD из одной цепи

В LD каждому контакту ставится в соответствие логическая переменная, определяющая его состояние. Если контакт замкнут, то переменная имеет значение ИСТИНА. Если разомкнут — ЛОЖЬ. Имя переменной пишется над контактом и фактически служит его названием.

Последовательное соединение контактов или цепей равноценно логической операции И. Параллельное соединение образует монтажное ИЛИ.

Цепь может быть либо замкнутой **(ON)**, либо разомкнутой **(OFF)**. Это как раз и отражается на обмотке реле и соответственно на значении логической переменной обмотки (ИСТИНА/ЛОЖЬ).

Приведенная на рис. 7.5 схема эквивалентна выражению:

Lamp1 := (Switch1 **OR** Switch2) **AND** Switch3 **AND** ((Switch4 **AND** Switch6) **OR** Switch5);

Зрительное восприятие LD-диаграмм должно быть интуитивно понятным. Для России этому несколько мешает принятая система условных графических обозначений, базирующаяся на американском стандарте NEMA. Преимущество таких обозначений состоит в возможности применения символов псевдографики для построения LD-диаграмм.

Сопоставление обозначений базовых элементов LD и обозначений ЕСКД приведено в таблице.

LD	ЕСКД	Обозначение
		Нормально разомкнутый контакт
		Нормально замкнутый контакт
		Обмотка реле

Контакт может быть инверсным — нормально замкнутым. Такой контакт обозначается с помощью символа |/| и замыкается, если значение переменной ЛОЖЬ. Происхождение этого обозначения связано с русской буквой И (инверсия), которую американцы вписывали в контакт (шутка). Инверсный контакт равнозначен логической операции НЕ.

Переключающий контакт образуется комбинацией прямого и инверсного контактов (см. пример на рис. 7.6).

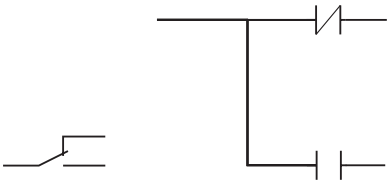

Рис. 7.6. Переключающий контакт

Обмотки реле также могут быть инверсными, что обозначается символом (/). Если обмотка инверсная, то в соответствующую логическую переменную копируется инверсное значение состояния цепи.

7.5.2. Реле с самофиксацией

Помимо обычных реле, в релейных схемах часто применяются поляризованные реле. Такое реле имеет две обмотки, переключающие его из одного положения в другое. Переключение производится импульсами тока. При отключении тока питания поляризованное реле остается в заданном положении, что реализует элементарную ячейку памяти.

В LD такое реле реализуется при помощи двух специальных обмоток **SET** и **RESET**. Обмотки типа **SET** обозначаются буквой **S** внутри круглых скобок **(S)**. Обмотки типа **RESET** обозначаются буквой **R**. Если соответствующая обмотке **(S)** переменная принимает значение ИСТИНА, то сохраняет его бесконечно. Вернуть данную переменную в ЛОЖЬ можно только обмоткой **(R)**.

Очевидно, что полной аналогии с поляризованным реле программно достичь невозможно. Даже если значение логического выхода сохраняется в энергонезависимой памяти, состояние самой электрической цепи при выключенном питании ПЛК определяется его схематикой. Фиксация безопасного положения аппаратуры при аварии питания системы управления может быть достигнута только аппаратно.

Условие выключения реле не всегда равносильно отсутствию условия включения. Благодаря **(R)** и **(S)** обмоткам условия включения и выключения реле можно формировать совершенно независимо, причем в любой цепи и сколько угодно раз. Обмотки **(R)** и **(S)** обеспечивают фиксацию условий управления, что необходимо при реализации автоматов с памятью.

Конечно, самофиксацию несложно организовать и на простом реле, используя дополнительный контакт в цепи питания. Пример этого представлен на рис. 7.7.

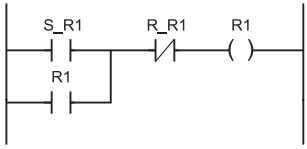

Рис. 7.7. Реле с самофиксацией

Контакт S_R1 включает, а R_R1 выключает реле R1, благодаря контакту R1 реле получает питание после размыкания S_R1. Применение **SET/RESET**-обмоток не дает ничего принципиально нового, но делает LD-диаграмму проще и красивее.

7.5.3. Порядок выполнения и обратные связи

Идеология релейных схем подразумевает параллельную работу всех цепей. Ток во все цепи подается одновременно.

В LD решение диаграммы выполняется последовательно слева направо и сверху вниз. В каждом рабочем цикле однократно выполняются все цепи диаграммы, что и создает эффект параллельности работы цепей. Любая переменная в рамках одной цепи всегда имеет одно и то же значение. Если даже реле в цепи изменит переменную, то новое значение поступит на контакты только в следующем цикле. Цепи расположенные ниже, получат новое значение переменной сразу. Цепи расположенные выше — только в следующем цикле. Строгий порядок выполнения схемы очень важен. Случайный или даже истинно параллельный порядок выполнения цепей мог бы приводить к эффекту «гонок», встречающемуся в электронных схемах с триггерами. Благодаря жесткому порядку выполнения LD-диаграммы сохраняют устойчивость при наличии обратных связей.

В приведенной на рис. 7.8 схеме включение **Key** вызовет мгновенное (в том же цикле) включение **P2** и отключение **P3**. Реле **P1** будет включено только в следующем цикле, причем даже если **Key** уже в обрыве (ЛОЖЬ).

Используя вышеописанный принцип цикличности выполнения LD-диаграмм, очень легко построить генератор единичных

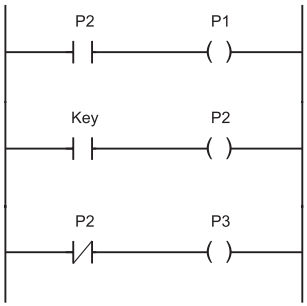

Рис. 7.8. LD диаграмма с обратной связью

импульсов. Пример такого построения дан на рис. 7.9. Период импульсов на реле **P1** будет равен удвоенной длительности рабочего цикла ПЛК.

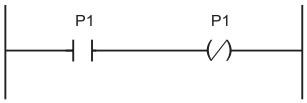

Рис. 7.9. Генератор единичных импульсов

7.5.4. Управление порядком выполнения

Порядок выполнения цепей диаграммы можно принудительно изменять, используя метки (labels) и переходы (jumps).

Метку можно ставить только в начало цепи. Имена меток подчинены правилам наименования переменных. Для наглядности можно закончить метку двоеточием. Двоеточие не образует новой метки. Так, **M1:** и **M1** это одно и то же. Цепь может иметь только одну метку и один переход. Переход равнозначен выходному реле и выполняется, если выходная переменная имеет значение ИСТИНА. Переход может быть инверсным, в этом случае он выполняется при значении цепи ЛОЖЬ. Используя переход, можно пропустить выполнение части диаграммы. Пропущенные цепи не сбрасываются, а именно не выполняются — замирают в том положении, в котором были ранее. Переход вверх допускается и позволяет создавать циклы (рис. 7.10). Проверка условий окончания цикла, естественно, лежит на совести программиста.

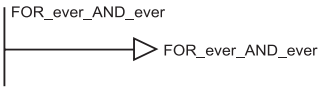

Рис. 7.10. Простейший бесконечный цикл

Идеологически переходы противоречат аналогии LD с релейными схемами, нарушая законы электрических цепей. В схеме LD с переходами разобраться бывает сложно. Желательно не заниматься управлением порядком выполнения LD-диаграммы в ней самой, а использовать для этого более выразительные средства. Например, разделить LD-диаграмму на модули (действия), а порядок выполнения описать в SFC.

Специальный переход **RETURN** прекращает выполнение LD диаграммы. Если **RETURN** встречается в основной программе (**PLC_PRG**), рабочий цикл прерывается. В функциях и функциональных блоках происходит возврат в место вызова. Иными словами, использование перехода **RETURN** аналогично по смыслу оператору **RETURN** в текстовых языках.

7.5.5. Расширение возможностей LD

В LD-диаграмму можно вставить функции и функциональные блоки. Функциональные блоки должны иметь логические вход и выход. На рис. 7.11 показан пример организации цикла на 10 повторов на базе функционального блока декрементный счетчик. Первая цепь загружает счетчик числом повторов. Вторая цепь — генератор единичных импульсов. Третья — декрементный счетчик с проверкой условия окончания цикла. Тело цикла на рисунке не показано.

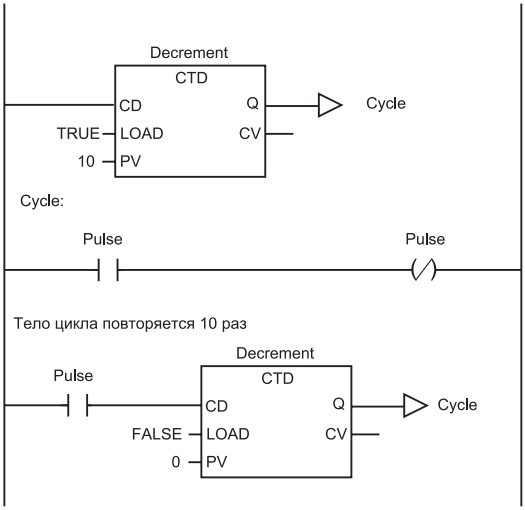

Рис. 7.11. Цикл на 10 повторов на базе функционального блока CTD

Для включения в диаграмму функций в них искусственно вводится добавочный логический вход, обозначаемый **EN** (Enable) (см. пример на рис. 7.12). Логическое значение на входе **EN** разрешает или запрещает выполнение функции. Сама функция не терпит никаких изменений при добавлении входа **EN**.

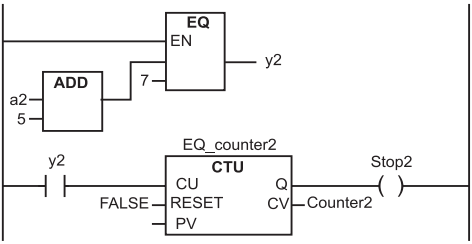

Рис. 7.12. Использование оператора EQ, управляемого по входу EN

В первой редакции стандарт МЭК определял контакты и обмотки, управляемые фронтами импульсов: контакт |P| и обмотка (P) переднего фронта, контакт |N| и обмотка (N) заднего фронта. В настоящее время поддержка таких контактов и обмоток не является обязательной, так как аналогичную цепь легко можно построить при помощи функциональных блоков **R_TRIG** и **F_TRIG**.

7.5.6. Особенности реализации LD в CoDeSys

Нетрудно заметить, что длина LD-цепей различна в зависимости от сложности. Например, пусть первая цепь состоит из 50 контактов, а вторая из двух. В этом случае шины питания на схеме должны быть широко разнесены, чтобы вместить все соединения первой цепи. Вторая цепь будет выглядеть неоправданно растянутой. Эта особенность РКС всегда вызывала сложности при реализации программирующих станций ПЛК. Если цепь не вмещалась в один экран, ее разрывали и переносили остаток ниже. Аналогично выглядят предложения данного абзаца, выровненные по ширине. Перенос легко читается для одного провода, но цепь может быть разветвленной.

Стандарт МЭК допускает не изображать общую правую шину вообще и выравнивать цепи влево для лучшего зрительного восприятия. Графический редактор CoDeSys не ограничивает возможную ширину LD-цепи и не требует применения переносов. Сложные цепи изображаются слитно и для работы с ними необходимо пользоваться горизонтальной прокруткой экрана (см. рис. 7.13). Правая шина изображается вертикальными отрезками в пределах каждой цепи и выравнивается влево, цепи пронумерованы и разделены горизонтальными линиями.

Время выполнения одной LD-цепи не постоянно. Если процессор обнаружил, что цепь разомкнута, то уже нет смысла анализи-

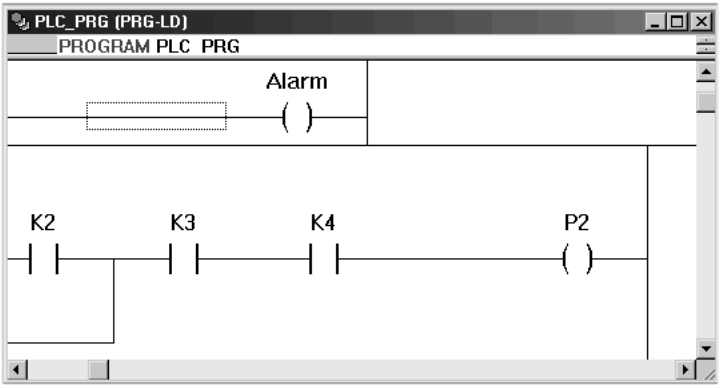

Рис. 7.13. Правый край двух цепей различной длины

ровать ее далее, можно сразу присвоить значение FALSE выходной переменной. Компилятор кода CoDeSys так и делает.

Для функциональных блоков CoDeSys позволяет графически подключать только один вход в логическую цепь. Стандарт не накладывает таких ограничений. Например, в системе MULTIPROG первая цепь, показанная на рис. 7.11, может выглядеть как на рис. 7.14.

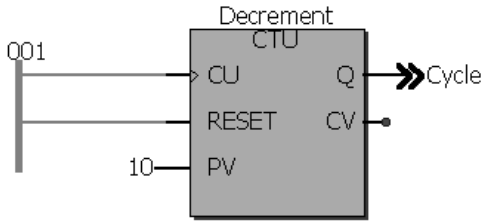

Рис. 7.14. Изображение декрементного счетчика в MULTIPROG

Вход RESET здесь получает значение ИСТИНА непосредственно от шины питания. Такая схема нагляднее, но приемлема только для логических переменных. Аналоговые переменные все равно не вписываются в идеологию LD и требуют непосредственное присваивание.

Функция или функциональный блок в LD рассматриваются как исполнительное устройство — аналог реле. Применение функции фактически означает конец LD-диаграммы и начало FBD. Правила построения LD-диаграмм здесь уже не работают.

Помимо входа **EN**, стандарт предлагает определять для функций и дополнительный выход **ENO** (Enable Out), показывающий дальнейшее прохождение тока в цепи. Выход **ENO** появился в редакции стандарта 1993 года, но не является обязательным. В CoDeSys такая возможность не поддерживается. Выход **ENO** должен служить для индикации ошибок в функции. В CoDeSys контроль ошибок исполнения реализован иначе.

По определению функция имеет только один выход. Благодаря этому функции можно использовать в выражениях ST. Функции с дополнительным выходом **ENO** решают одну проблему, но создают другую.

7.5.7. LD-диаграммы в режиме исполнения

В режиме *Online* обмотки реле, контакты и проводники, находящиеся в состоянии **On** (под током), окрашены голубым (цвета по умолчанию) цветом. CoDeSys позволяет менять значения логических переменных (ИСТИНА/ЛОЖЬ) непосредственно в графической диаграмме двойным щелчком мыши на имени переменной. Значения входов-выходов функциональных блоков отображаются числовыми значениями.

Точка останова может устанавливаться только целиком на цепь. Для установки или сброса точки останова необходимо щелкнуть мышью по номеру цепи. В режиме останова номер цепи подсвечен красным. Пошаговое — по одной цепи выполнение достигается командами **«Step over»** и **«Step in»**.

7.6. Функциональные блоковые диаграммы (FBD)

7.6.1. Отображение POU

Диаграмма FBD строится из компонентов, отображаемых на схеме прямоугольниками. Входы POU изображаются слева от прямоугольника, выходы справа. Внутри прямоугольника указывается тип POU и наименования входов и выходов. Для экземпляра функционального блока его наименование указывается сверху, над прямоугольником. В графических системах программирования прямоугольник компонента может содержать картинку, отражающую его тип. Размер прямоугольника зависит от чис-

ла входов и выходов и устанавливается графическим редактором автоматически. Пример графического представления экземпляра Blinker функционального блока BLINK дан на рис. 7.15.

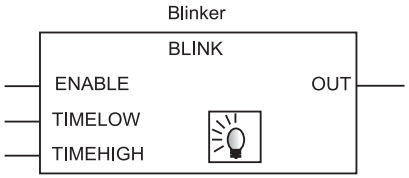

Рис. 7.15. Графическое представление экземпляра функционального блока

Программа в FBD не обязательно должна представлять большую единую схему. Как и в LD, диаграмма образуется из множества цепей, которые выполняются одна за другой.

В CoDeSys все цепи одного POU отображаются в едином графическом окне, пронумерованные и разделенные горизонтальными линиями (рис. 7.16). Значения переменных, вычисленные в одной цепи, доступны в последующих цепях сразу в том же рабочем цикле.

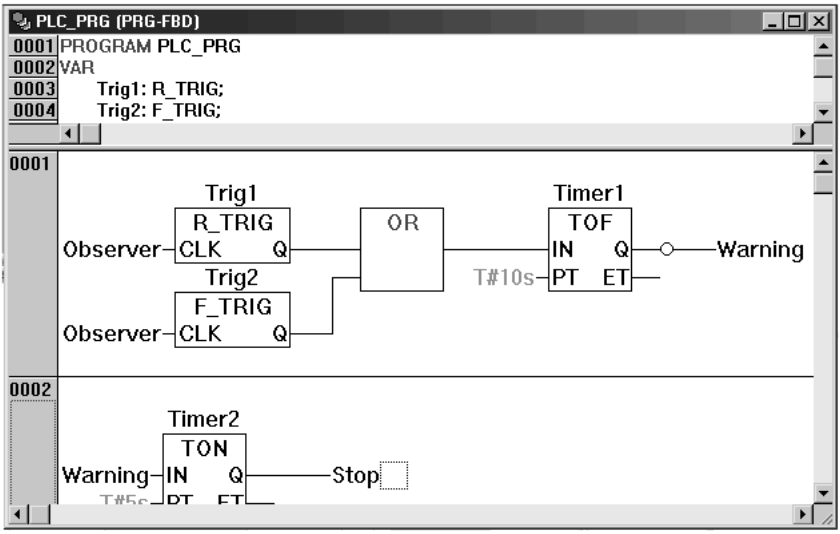

Рис. 7.16. Диаграмма FBD из двух цепей

7.6.2. Соединительные линии

Прямоугольники POU в FBD соединены линиями связи. *Соединения* имеют направленность слева направо. Вход блока может быть соединен с выходом блока, расположенного слева от него. Помимо этого, вход может быть соединен с переменной или константой. Соединение должно связывать переменные или входы и выходы одного типа. В отличие от компонента переменная изображается на диаграмме без прямоугольной рамки. Ширина соединительной линии в FBD роли не играет. Стандарт допускает использование соединительных линий разной ширины и стиля для соединений разного типа.

7.6.3. Порядок выполнения FBD

Выполнение FBD-цепей идет слева направо, сверху вниз. Блоки, расположенные левее, выполняются раньше. Блок начинает вычисляться только после вычисления значений всех его входов. Дальнейшие вычисления не будут продолжены до вычисления значений на всех выходах. Другими словами, значения на всех выходах графического блока появляются одновременно. Вычисление цепи считается законченным только после вычисления значений на выходах всех входящих в нее элементов.

В некоторых системах программирования пользователь имеет возможность свободно передвигать блоки с сохранением связей. В этом случае ориентироваться нужно исходя из порядка соединений. Редактор FBD CoDeSys автоматически расставляет блоки в порядке выполнения.

7.6.4. Инверсия логических сигналов

Инверсия логического сигнала в FBD изображается в виде окружности на соединении, перед входом или переменной. Инверсия не является свойством самого блока и может быть легко добавлена или отменена непосредственно в диаграмме. В CoDeSys это делается командой **«Negate»**. На рис. 7.16 выход Q экземпляра функционального блока **TOF** инвертируется перед присвоением его значения переменной Warning.

7.6.5. Соединители и обратные связи

Соединители (connectors) представляют собой поименованное соединение, которое можно разорвать и перенести в следующую цепь. Такой прием может понадобиться при ограниченной шири-

не окна редактора FBD. В CoDeSys ширина окна не ограничена, поэтому соединители здесь не нужны.

Стандарт не запрещает соединения, идущие с выхода блока на свой вход или вход ранее исполняемых блоков. Обратная связь не образует цикл, подобный **FOR**, просто некоторое вычисленное значение поступит на вход при следующем вызове диаграммы. Фактически это означает неявное создание переменной, которая сохраняет свое значение между вызовами диаграммы. Для устранения неоднозначности необходимо присвоить безопасное начальное значение переменной обратной связи. Но как это сделать для переменной, которая не объявлена в явной форме, непонятно?

В редакторе FBD CoDeSys обратные соединения запрещены. Для создания обратной связи используйте явно объявленную внутреннюю переменную.

При необходимости переноса или разветвления соединения в другие цепи также необходимо использовать промежуточные локальные переменные.

7.6.6. Метки, переходы и возврат

Порядок выполнения FBD-цепей диаграммы можно принудительно изменять, используя метки и переходы, точно так же, как и в релейных схемах.

Метка ставится в начале любой цепи, являясь, по сути, названием данной цепи. Цепь может содержать только одну метку. Имена меток подчинены общим правилам наименования идентификаторов МЭК. Графический редактор автоматически нумерует цепи диаграммы. Эта нумерация применяется исключительно для документирования и не может заменять метки.

Переход обязательно связан с логической переменной и выполняется, если переменная имеет значение ИСТИНА. Для создания безусловного перехода используется константа ИСТИНА, связанная с переходом. Метки и переходы в FBD представлены в примере, показанном на рис. 7.17. Обратите внимание на последнюю цепь на рис. 7.17 — она является пустой. Пустая цепь обозначается единственной константой TRUE.

Оператор возврата **RETURN** можно использовать в FBD так же, как и переход на метку, т. е. в связке с логической переменной. Возврат приводит к немедленному окончанию работы программного компонента и возврату на верхний уровень вложений. Для основной программы это начало рабочего цикла ПЛК.

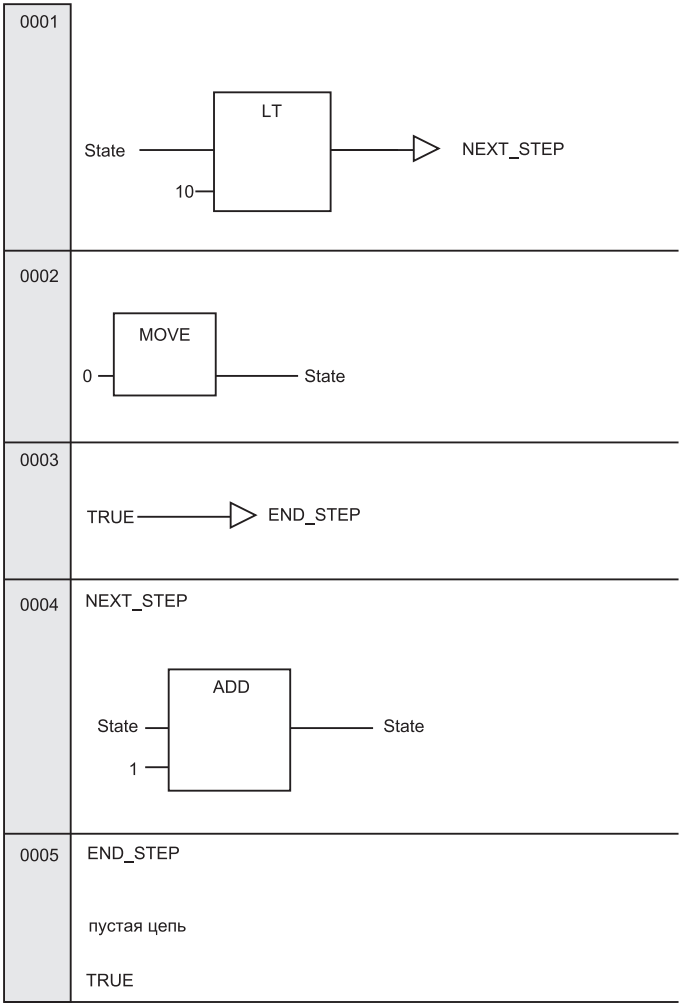

Рис. 7.17. Метки и переходы в FBD

7.6.7. Выражения ST в FBD

CoDeSys позволяет записывать выражения ST на входе графических блоков. Такой прием расширяет стандартный FBD и часто оказывается достаточно удобным. Компактная форма представления выражений облегчает запись и чтение функциональных диаграмм.

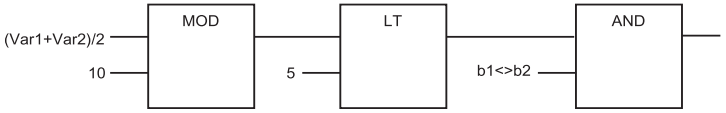

Рис. 7.18. Фрагмент FBD-диаграммы с выражениями

7.7. Последовательные функциональные схемы (SFC)

7.7.1. Шаги

Любая SFC-схема составляется из элементов, представляющих шаги и условия переходов (см. рис. 7.19). Шаги показаны на схеме прямоугольниками. Реальная работа шага (действия) описывается в отдельном окне системы программирования и не отражается на диаграмме. О назначении шага SFC говорит только его название или, если этого не достаточно, краткое текстовое описание (комментарий).

Шаги на схеме могут быть пустыми, что не вызывает ошибки при компиляции проекта. Пустые шаги являются нормой при применении программирования сверху вниз, характерного для SFC. Определить действия, соответствующие шагу, можно в любое время. Нет ничего удивительного, если пустые шаги останутся и в законченном проекте. Задачей пустого шага является ожидание перехода.

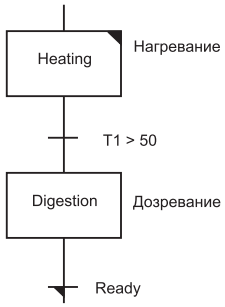

Рис. 7.19. Шаги и переходы

7.7.2. Переходы

Ниже шага на соединительной линии присутствует горизонтальная черта, обозначающая переход (см. пример на рис. 7.19).

Условием перехода может служить логическая переменная, логическое выражение, константа или прямой адрес.

Переход выполняется при соблюдении двух условий:

1) переход разрешен (соответствующий ему шаг активен);

2) условие перехода имеет значение TRUE.

Простые условия отображаются непосредственно на диаграмме справа от черты, обозначающий переход. В CoDeSys на диаграмме можно записывать только выражения на языке ST (T1 > 50 на рис. 7.19).

Для громоздких условий применяется другой подход. Вместо условия на диаграмме записывается только идентификатор перехода. Само же условие описывается в отдельном окне с применением языка IL, ST, LD или FBD.

На рис. 7.20—7.23 показаны четыре возможных представления перехода Ready на разных языках.

Переменные или прямые адреса используются в условии перехода только для чтения. В условном выражении перехода нельзя вызывать экземпляры функциональных блоков и использовать операцию присваивания.

Признаком того, что идентификатор перехода на диаграмме является отдельно реализованным условием, а не простой логической переменной, служит закрашенный угол перехода (см. рис. 7.19).

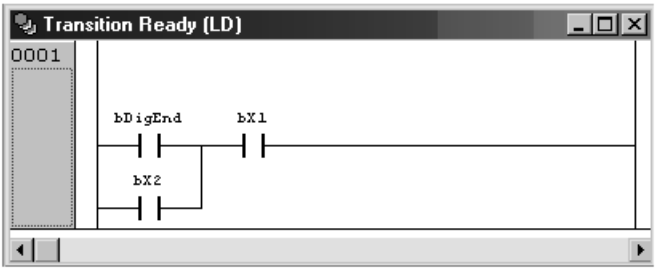

Рис. 7.20. Переход Ready (LD)

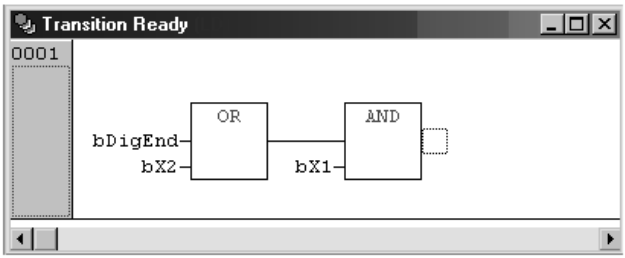

Рис. 7.21. Переход Ready (FBD)

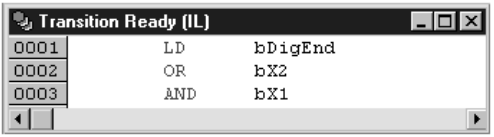

Рис. 7.22. Переход Ready (IL)

Рис. 7.23. Переход Ready (ST)

В качестве условия перехода может быть задана логическая константа. Если задано TRUE, то шаг будет выполнен однократно, за один рабочий цикл, далее управление перейдет к следующему шагу. Если задано условие FALSE, то шаг будет выполняться бесконечно.

7.7.3. Начальный шаг

Каждая SFC-схема начинается с шага, выделенного графически двойными вертикальными линиями или по всему периметру. Это — *начальный шаг* (рис. 7.24). Наименование начального шага может быть произвольным (по умолчанию Init). Начальный шаг присутствует обязательно, хотя и может быть пустым.

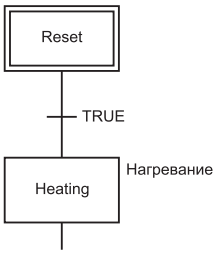

Рис. 7.24. Начальный шаг Reset

7.7.4. Параллельные ветви

Несколько ветвей SFC могут быть параллельными (рис. 7.25). Признаком параллельных ветвей на схеме является двойная горизонтальная линия. Каждая параллельная ветвь начинается и заканчивается шагом. То есть условие входа в параллельность всегда одно, условие выхода тоже одно на всех.

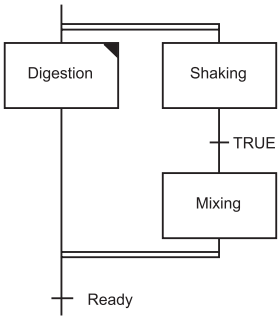

Рис. 7.25. Параллельные ветви

Параллельные ветви выполняются теоретически одновременно. В жизни это означает — в одном рабочем цикле, слева направо.

Условие перехода, завершающее параллельность, проверяется только в случае, если в каждой параллельной ветви активны последние шаги.

В данном примере (рис. 7.25) *Shaking* будет выполнен однократно, далее *Digestion* и *Mixing* будут работать параллельно до выполнения условия *Ready*.

7.7.5. Альтернативные ветви

Несколько ветвей SFC могут быть *альтернативными ветвями*. Признаком альтернативных ветвей на схеме является одинарная горизонтальная линия. Каждая альтернативная ветвь начинается и заканчивается собственным условием перехода. Проверка альтернативных условий выполняется слева направо. Если верное условие найдено, то прочие альтернативы не рассматриваются. В альтернативных ветвях всегда работает только одна из ветвей, поэтому ее окончание и будет означать переход к следующему за альтернативной группой шагу.

В данном примере (рис. 7.26) альтернатива *Stop* оценивается первой. Шаги *Move_Dwn* и *Move_Up* имеют шанс стать активными, только если *Stop* равен FALSE.

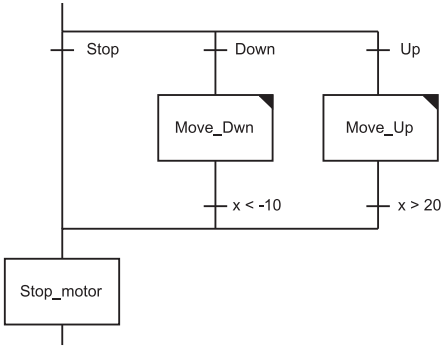

Рис. 7.26. Альтернативные ветви

При создании альтернативных ветвей желательно задавать взаимоисключающие условия. В этом случае вероятность допустить ошибку при анализе или в процессе доработки диаграммы значительно ниже.

7.7.6. Переход на произвольный шаг

В общем случае SFC-схема выполняется сверху вниз. Стандартом допускается создание *переходов на произвольный шаг*. Для этого применяются соединительные линии с промежуточными стрелками или поименованные переходы. То есть переход выполняется на шаг, имя которого указано под стрелкой. В англоязычных источниках переход на произвольный шаг называется «прыжок» (jump).

В примере, показанном на рис. 7.27, шаги *Move_Dwn* и *Move_Up* последовательно активируют друг друга. Заметьте, что условие *Stop* при этом проверяться не будет, шаги *Move_Dwn* и *Move_Up* соединены в логическое кольцо, имеющее 2 варианта входа, но ни одной возможности выхода. Маркер активности будет перемещаться исключительно в этом кольце.

Прыжок из одной ветви параллельного блока наружу вызывает эффект размножения маркера. Прыжок внутрь параллельного блока нарушает параллельность ветвей. Подобных трюков необходимо избегать.

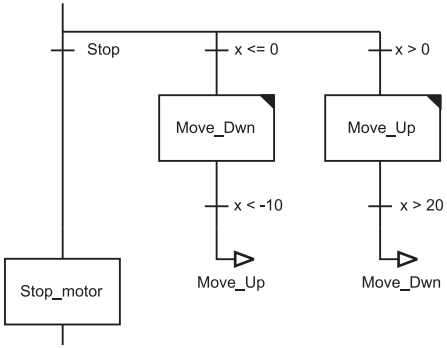

Рис. 7.27. Перекрестные поименованные переходы

7.7.7. Упрощенный SFC

Помимо рассмотренной ниже стандартной МЭК-технологии связи шагов и действий, в CoDeSys реализована упрощенная реализация (easy mode SFC). Смысл ее заключается в применении более простого, компактного и быстрого последовательного SFC-исполнителя. Помимо этого, сами диаграммы получаются компактнее и часто проще для понимания. Безусловно, возможности упрощенной реализации несколько уже — нельзя включать и выключать *действия* в разных шагах и управлять активностью действий по времени.

Действия могут быть трех классов — текущее, входное и выходное. Графически действия на диаграмме никак не отображаются, их редактирование выполняется в отдельных окнах. В упрошенной реализации действия принадлежат шагу. То есть действие нельзя вызвать из другого шага или откуда-либо еще. Можно считать, что каждый прямоугольник шага при его увеличенном рассмотрении содержит 3 раздела, соответствующие трем возможным действиям. Если шаг удалить, то и все его действия будут утрачены. Не удивительно, что такие действия не требуют отдельных идентификаторов и называются по именам шагов.

Для создания нового или редактирования существующего действия в CoDeSys достаточно щелкнуть мышкой по прямоугольнику шага. Это приведет к открытию соответствующего редактора или вызову диалога создания нового действия, если шаг еще не описан.

На рис. 7.28. показан момент определения действия шага Digestion (Дозревание).

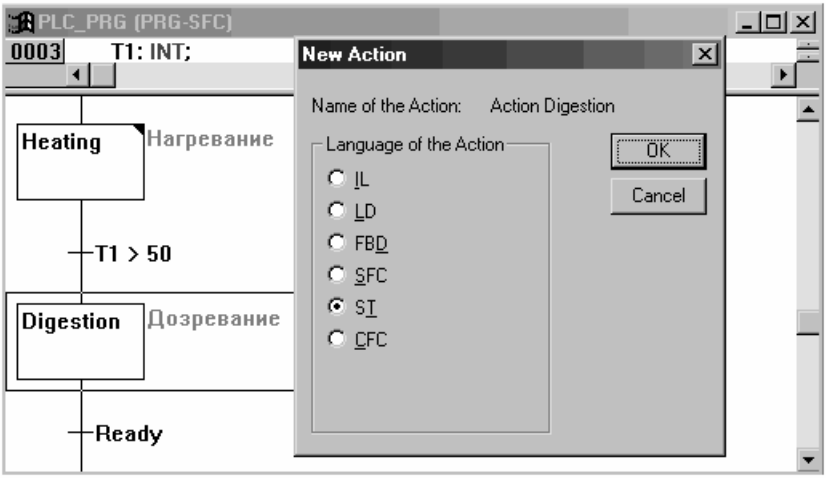

Рис. 7.28. Создание нового описания шага

Шаги, содержащие действие, на схеме отличаются тем, что верхний правый угол прямоугольника закрашен. Пока шаг активен, текущее действие будет выполняться один раз в каждом рабочем цикле.

Входные и выходные действия

Весьма вероятен случай, когда определенные действия нужно выполнить в шаге только один раз (рис. 7.29). Например, включить нагрев в начале активности шага и выключить при переходе на другой шаг. С этой целью и предусмотрены входное и выходное действия. Входное действие обозначается сегментом **'E'** (Entry) в нижнем левом углу прямоугольника шага и выполняется однократно при активизации шага. Выходное обозначается сегментом **'X'** (eXit) в нижнем левом углу прямоугольника шага. Выходное действие выполняется однократно при завершении работы шага.

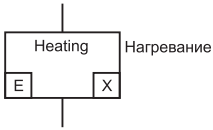

Рис. 7.29. Шаг с входным и выходным действиями

Механизм управления шагом

Для каждого шага CoDeSys создает две логические переменные. Допустим, шаг называется Step1. Для него будут определены переменные _Step1 и Step1. Объявления переменных происходят неявно, т. е. в разделе объявлений никаких дополнительных записей делать ненужно. Переменная с лидирующим подчеркиванием (_Step1) получает значение **TRUE**, когда шаг активируется (входное условие выполнено), и сбрасывается при деактивации (сразу при выполнении выходного условия). Переменная без подчеркивания (Step1) отстает на один рабочий цикл, т. е. получает значение TRUE после выполнения входного действия и сбрасывается после выполнения выходного действия. Комбинации двух этих переменных (_Step1, Step1) последовательно образуют 4 возможных состояния шага: не выполняется (00), входное действие (10), текущее действие (11 и 10), выходное действие (01).

Данные переменные можно использовать для определения активности шага, например, с целью синхронизации параллельных ветвей (рис. 7.30). Так, в следующем примере шаг Step4 не может быть закончен раньше, чем Step2.

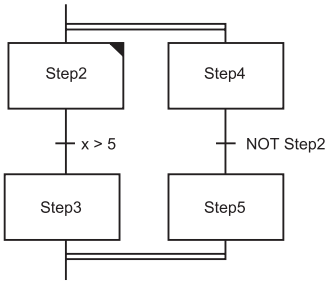

Рис. 7.30. Синхронизация шагов

Для доступа к переменным шага вне данного компонента и из отладчика их необходимо объявить как логические переменные. Предварительная установка переменных (_Step) шагов в отладчике дает возможность быстро перейти к отладке какого-либо определенного состояния системы. Обращаться к переменным шага на запись в пользовательской программе нельзя. Это приведет к тому, что SFC-диаграмма будет работать совсем не так, как следует из ее графического представления, что, безусловно, противоречит самой идее SFC. На самом деле, запись не запрещена, но использовать эту возможность желательно только в целях отладки.

Описанный упрощенный механизм SFC продолжает работать и при использовании МЭК шагов. То есть CoDeSys позволяет в любое время подключить полный SFC исполнитель, без переделки того, что уже реализовано.

7.7.8. Стандартный SFC

Вышеописанная упрощенная техника настраивает на то, что изначально определяются шаги, которые наполняются определенным содержимым в процессе работы над проектом.

При применении МЭК-действий подход несколько иной. Сначала определяются действия (виды работ), которые должна выполнять система, а затем уже составляется диаграмма, в которой определяется их порядок и взаимосвязь. Каждое действие сопоставляется одному или нескольким шагам. Причем вполне возможно, что некоторое действие должно запускаться в одном шаге и останавливаться в другом. Также возможно, что начатое действие должно закончить свою работу вообще независимо ни от каких шагов. Например, начав движение, кабина лифта должна как минимум доехать до ближайшего этажа и выпустить пассажиров, даже если дана команда на окончание работы.

Действия МЭК показываются на SFC-диаграмме в виде прямоугольников, расположенных справа от шага и привязанных к нему графически. Пример шагов, содержащих действия, показан на рис. 7.31.

Существенно важным здесь является то, что одно и тоже действие можно многократно использовать в разных шагах. Так, в данном примере шаги *Cooling* (охлаждение) и *Drying* (сушка) используют действие *air_cooling* (воздушный обдув). В отличие от описанных выше упрощенных действий, действия МЭК не принадлежат конкретному шагу, а являются самостоятельными программными элементами SFC-компонента.

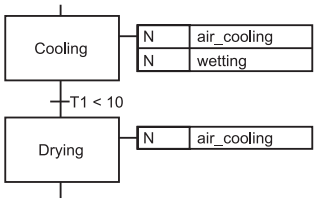

Рис. 7.31. Действия

Идентификаторы действий должны быть уникальны в пределах компонента POU и не должны совпадать с идентификаторами шагов и переходов.

7.7.9. Классификаторы действий

Прямоугольник, отображающий действие, содержит в левой части специальное поле — *классификатор* (см. рис. 7.31). Классификатор (qualifier) определяет способ влияния активного шага на данное действие.

Возможны следующие классификаторы:

- **N** — несохраняемое действие (Non-stored, рис. 7.32). Данное действие будет выполняться в каждом рабочем цикле, пока активен шаг.

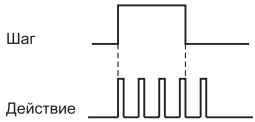

Рис. 7.32. Несохраняемое действие

- **P** — импульс (Pulse, рис. 7.33). Действие выполняется один раз при активации и второй раз после деактивации шага.

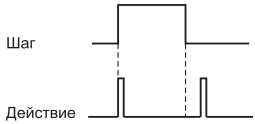

Рис. 7.33. Импульсное действие

- **S** — сохраняемое (Stored, рис. 7.34). Действие активируется и остается активным до сброса. Действие продолжит выполняться в каждом цикле даже тогда, когда шаг уже не активен.

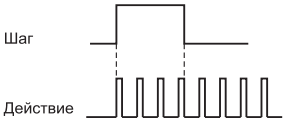

Рис. 7.34. Сохранение действия

- **R** — сброс (Reset, рис. 7.35). Действие деактивируется.

Рис. 7.35. Сброс действия

- **L** — ограниченное по времени (time Limited). Действие активируется вместе с шагом и остается активным на заданное время, но не дольше, чем шаг. На рис. 7.36 показаны два возможных случая. В первом случае действие деактивируется по истечении времени, во втором — по причине деактивации шага.

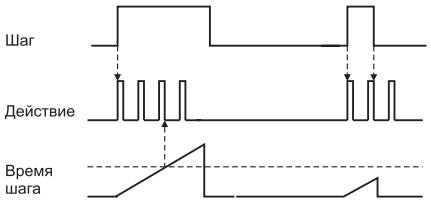

Рис. 7.36. Ограниченное по времени действие

- **SL** — сохраняемое и ограниченное по времени (Stored and time Limited, рис. 7.37). Действие активируется вместе с шагом и остается активным заданное время, вне зависимости от активности шага. Действие можно деактивировать досрочно из другого шага с классификатором R.

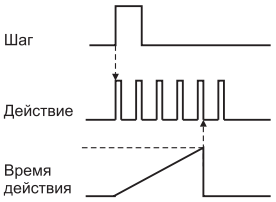

Рис. 7.37. Сохраняемое и ограниченное по времени действие

- **D —** отложенное (Delayed, рис. 7.38). Действие активируется через заданное время после активации шага и остается актив-

ным, пока активен шаг. Если шаг окажется активным меньше заданного времени, то действие не будет активировано.

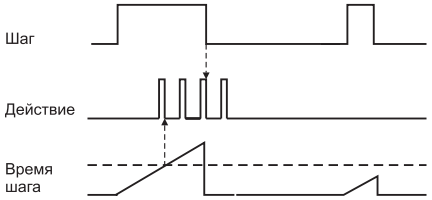

Рис. 7.38. Отложенное действие

- **DS —** отложенное сохраняемое (Delayed and Stored, рис. 7.39). Действие активируется через заданное время после активации шага и остается активным до сброса. Если шаг активен меньше заданного времени, то действие не будет активировано. При параллельном выполнении сброса в процессе отсчета времени (в другом шаге с классификатором R) действие не будет активироваться.

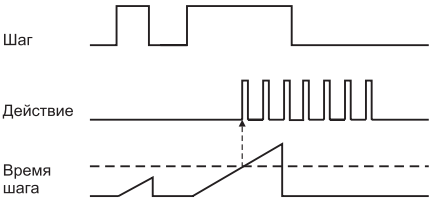

Рис. 7.39. Отложенное сохраняемое действие

- **SD** — сохраняемое отложенное (Stored and time Delayed, рис. 7.40). Действие активируется через заданное время после активации шага, даже если шаг уже не активен. Но если в процессе отсчета задержки активации выполнить сброс

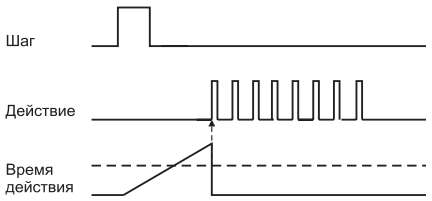

Рис. 7.40 Сохраняемое отложенное действие

(в другом шаге с классификатором R), то активация не произойдет. Активированное действие остается активным до сброса.

Классификаторы L, D, SD, DS и SL требуют указания константы времени в формате TIME. Например: D T#10s.

Как видно из приведенных выше временных диаграмм, каждое активное действие выполняется еще один раз уже после деактивации.

На рис. 7.41 как раз показан момент, когда шаг Init уже утратил активность, но действие A_0 выполняется последний раз. Это необходимо для того, чтобы действия могли отработать потерю активности и выполнить некоторые завершающие операции.

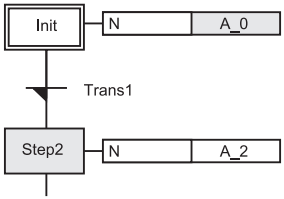

Рис. 7.41. Деактивация действия

Определить состояние деактивации можно внутри действия путем анализа признака активности (подробнее см. раздел 7.7.12). Такая необходимость возникает особенно часто при реализации импульсных действий, когда разные операции нужно выполнить в начале и в конце шага. Например:

```
IF Step1.X THEN
    iACount := iACount + 1;    (*Счетчик активных циклов*)
ELSE
    iDCount := iDCount + 1;    (*Счетчик деактиваций*)
END_IF
```

7.7.10. Действие — переменная

Действие стандартного SFC не обязательно должно что-либо делать. В качестве имени действия можно указать логическую переменную, внутреннюю или внешнюю. Переменная (на рис. 7.42 bMoveUp) будет соответствовать состоянию активности действия, классификаторы будут работать.

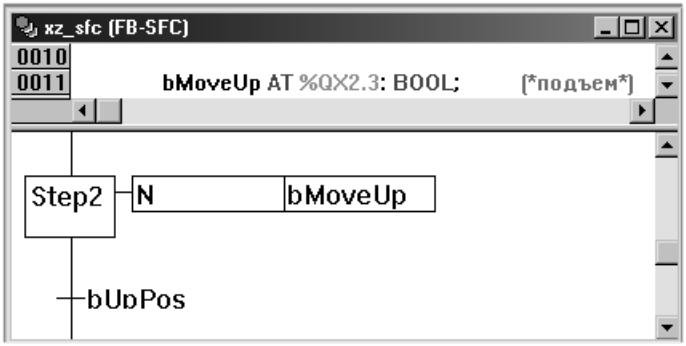

Рис. 7.42. Переменная действия

Такая переменная гораздо полезнее, чем может показаться на первый взгляд. Вполне вероятно, что необходимо управлять одним дискретным выходом, а вся необходимая логика уже отражена в SFC-диаграмме. Никакого дополнительного программирования здесь ненужно. Кроме того, переменные действий часто применяются для синхронизации различных ветвей диаграммы или программ всего проекта.

7.7.11. Механизм управления действием

Классификаторы определяют достаточно сложные возможности управления работой действия. Причем очевидно, что действие так же, как и шаг, должно иметь внутреннюю память состояния и логику управления, причем даже более развитую. Для реализации такого управления в каждое действие неявно включается экземпляр функционального блока SFCActionControl (рис. 7.43).

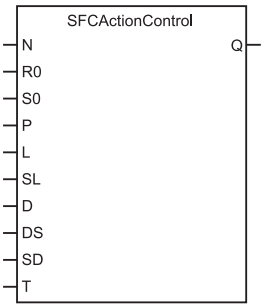

Рис. 7.43. Графическое представление SFCActionControl

Функциональный блок имеет входы, соответствующие классификаторам. Если какой-либо активный шаг включает данное действие, то это приводит к присваиванию логической единицы соответствующему входу SFCActionControl. После обработки всех активных шагов по состоянию выхода Q система исполнения принимает решение о необходимости вызова данного действия. Все действия вызываются один раз в рабочем цикле. Порядок выполнения активных действий зависит от реализации исполнителя.

В CoDeSys идентификаторы действий автоматически сортируются в менеджере проекта в алфавитном порядке, в процессе исполнения эта последовательность сохраняется. Интерфейс функционального блока SFCActionControl описан в библиотеке IECSFC.lib CoDeSys, которая обязательно должна быть включена в проект при использовании SFC с МЭК-действиями. Входы блока имеют одну тонкость — вместо классификаторов **S** и **R** используются идентификаторы **S0** и **R0**. Это сделано для исключения конфликта с ключевыми словами.

Функциональный блок SFCActionControl реализуется из стандартных компонентов МЭК, как это показано на рис. 7.44.

Как видно из реализации, сброс R0 является доминирующим входом. При наличии сброса выход Q безусловно приобретает значение FALSE. Прочие входы после соответствующего анализа объединены по ИЛИ. Как уже было сказано, действие вызывается всякий раз при Q = TRUE и последний раз после перехода Q из TRUE в FALSE.

Описание функционального блока SFCActionControl дано здесь для лучшего понимания механизма работы действий. На самом деле вы не найдете ни объявления экземпляров, ни его реализации в проекте. Логика работы действий скрыта от глаз и является заботой системы исполнения SFC.

В CoDeSys существует все же лазейка, позволяющая получить доступ к блоку управления действием. Экземпляр SFCActionControl доступен под именем AC в каждом МЭК-действии.

```
IF A_3.AC.Q THEN
   ...;                              (*действие активно?*)
END_IF
```

Если вы правильно поняли механизм МЭК-действий, то ошибочность примеров представленных на двух следующих ри-

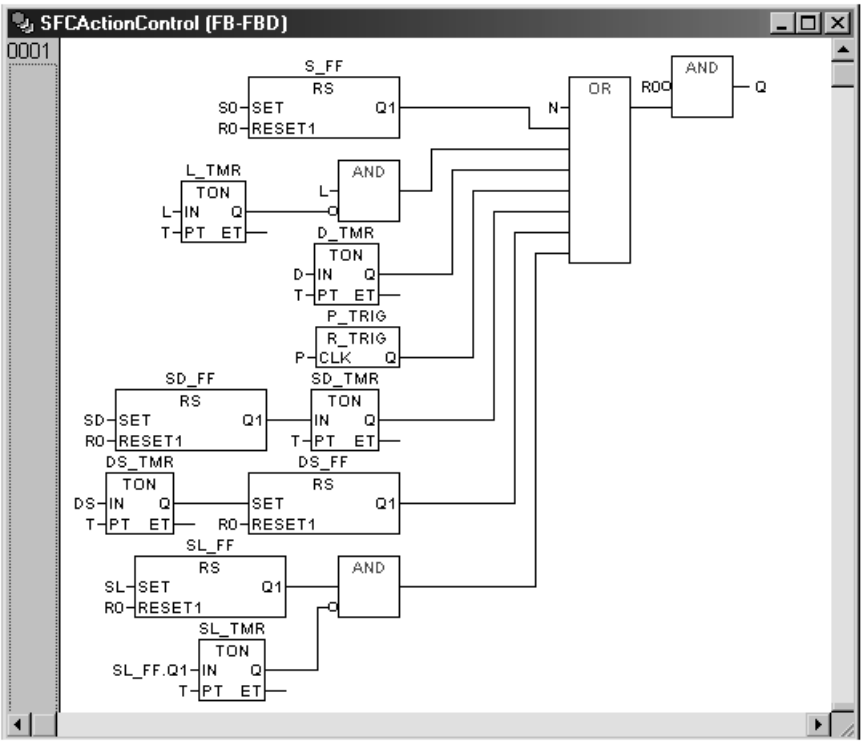

Рис. 7.44. Реализация функционального блока SFCActionControl (FBD)

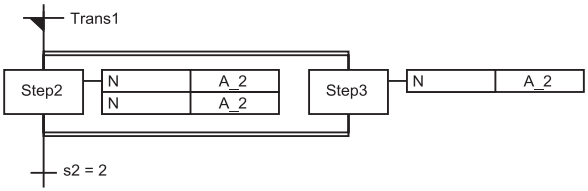

Рис. 7.45. Попытка вызвать действие три раза в одном цикле

сунках должна быть для вас очевидной. Начнем с примера, показанного на рис. 7.45.

Данная конструкция абсолютно бессмысленна. Выполнение действия A_2 трижды разрешается, но само действие все равно будет выполнено один раз в рабочем цикле.

Теперь рассмотрим пример, представленный на рис. 7.46. Здесь действие A_2 два раза разрешается, затем сбрасывается. В результате выполняться оно не будет.

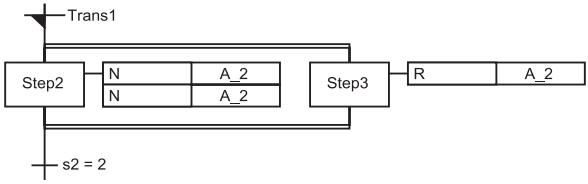

Рис. 7.46. Попытка вызвать действие 2 раза в одном цикле и запретить

7.7.12. Внутренние переменные шага и действия

Для каждого МЭК-шага неявно объявлена структура из двух доступных по чтению переменных. Первая переменная типа **BOOL** носит «оригинальное» название X и является *признаком активности шага*. По смыслу она равноценна логической переменной шага <StepName> в упрощенной реализации SFC. Логическая единица является признаком активности шага. Вторая переменная типа **TIME** называется T и указывает *время активности шага*.

Доступ к переменным шага возможен через имя шага и точку, как к данным структуры или экземплярам функционального блока — <StepName>.x. Вне программного компонента необходимо использовать «трехэтажную» конструкцию, начинающуюся с имени POU. Например:

IF MoveCtrl.Moving.X <> TRUE **THEN** ...

Переменные шага можно использовать в условиях переходов. Пример применения такого приема представлен на рис. 7.47.

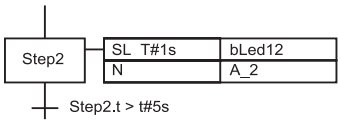

Рис. 7.47. Пятисекундный шаг

Переменная X также доступна и для шагов с аналогичным смыслом. Фактически <ActionName>.x это общедоступная копия выхода Q SFCActionControl.

7.7.13. Функциональные блоки и программы SFC

Применение SFC в объемных компонентах позволяет сократить время выполнения и соответственно время реакции системы. При помощи шагов монолитная программа разбивается на короткие фрагменты, выполняющиеся в разных рабочих циклах ПЛК. В других языках МЭК реализация объемных и в тоже время быстрых программ требует дополнительного кодирования механизма поэтапного выполнения. Проблемы с ростом времени рабочего цикла при усложнении программы особенно заметны в LD. SFC стимулирует к равномерному распределению вычислительной мощности процессора практически без дополнительных усилий программиста.

Реализация функциональных блоков и программ в SFC имеет существенную особенность. Отсутствуют первая и последняя инструкции. Оператор **RETURN** также не используется. Программа как бы не имеет конца. Каждый вызов SFC POU равноценен выполнению одного цикла. Что конкретно будет выполнять POU, зависит от его предыдущего состояния.

Принудительно вернуть компонент в начальное состояние можно только путем сброса ПЛК. Принудительная активация начального шага в SFC не означает автоматический сброс компонента. Она приведет только к тому, что кроме текущих активных действий активным еще станет и начальный шаг. Начальный шаг не содержит скрытых действий. Он не запрещает другие шаги и действия. Ситуация демонстрируется примером на рис. 7.48. Здесь параллельная ветвь с пустым шагом (Idle) передает активность начальному шагу. Другая параллельная ветвь S1—S2 бесконечно зациклена сама на себя. Она продолжает спокойно работать независимо ни от чего. Кроме того, не забывайте, что еще существуют действия с памятью.

Отработка реакции на все необходимые события, включая экстренные, должна быть предусмотрена в SFC явным образом. Перевод системы в начальное или безопасное состояние предусматривает для ПЛК установку заданного положения исполнительных механизмов и управление ими. Нажатие аварийной клавиши, обесточивающей исполнительные механизмы, должно корректно отрабатываться программным обеспечением. Установка и поддержание безопасного состояния системы — это такая же работа, как и нормальное функционирование. Не стоит для этих целей использовать программный сброс ПЛК, тем более что такая функция в стандартных библиотеках отсутствует.

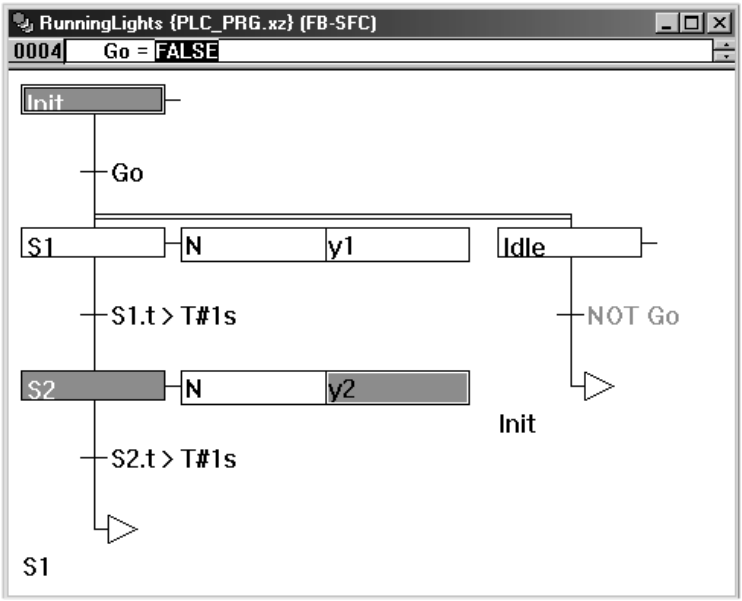

Рис. 7.48. Замкнутая ветвь SFC

В CoDeSys экстренный сброс SFC-программ и функциональных блоков все же возможен. Проблема решается с помощью специальных системных флагов (SFCInit, SFCReset), управляющих работой SFC-исполнителя.

7.7.14. Отладка и контроль исполнения SFC

В различных системах программирования для отладки SFC-диаграмм применяются следующие приемы:

- анимация активных шагов и действий;
- принудительная установка активных шагов;
- блокировка проверки или фиксация условий переходов;
- пропуск выполнения заданных шагов;
- блокировка выполнения действий;
- мониторинг времени активности шагов.

Общим методом отслеживания работы SFC-диаграмм в реальном масштабе времени является мониторинг и трассировка переменных шагов и действий, специально созданных вспомогательных логических флагов и счетчиков активности.

Расширенные отладочные функции реализуются в системах программирования различными способами и зависят от системы исполнения. Поэтому мы не будем рассматривать их здесь детально. Ограничимся одним простым примером мониторинга активности шагов в CoDeSys.

Условия перехода могут зависеть от переменных, связанных с различными входными воздействиями, датчиками, сообщениями сети, логическими вычислениями и т. д. Если некоторый шаг «пролетает» слишком быстро или наоборот «тормозит», не всегда так просто понять причину сбоя. Тем более что это не обязательно программная ошибка. Исходя из здравого смысла, можно примерно оценить допустимые пределы времени активности шага. Так, процесс перемещения кабины лифта с одного этажа на другой не может занимать 100 мс, но и не должен занимать несколько часов. В CoDeSys для каждого шага можно задать временные границы. Нарушение границ возбуждает специальный флаг ошибки. Помимо этого, отладочная система позволяет определить шаг, где произошла ошибка, и условие, приведшее к нарушению.

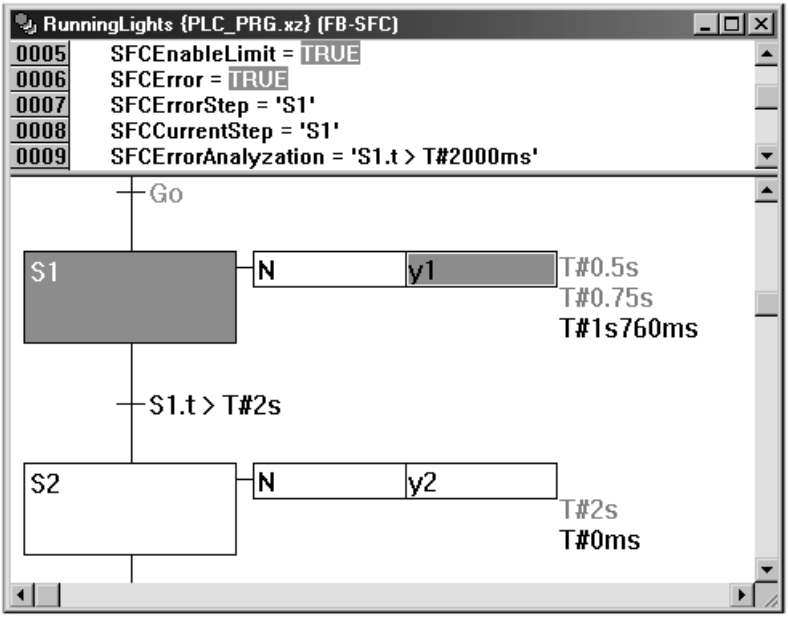

Рис. 7.49. Диагностика «торможения» шага в CoDeSys

На рис. 7.49 показана SFC-диаграмма в режиме исполнения. Шаг S1 по условию (вероятно, ошибочно) активен дольше, чем задано атрибутами шага (не более T#0.75s). Атрибуты времени показаны справа от шага серым цветом. Текущее время — черным цветом. В окне объявлений вы можете увидеть флаги, сигнализирующие ошибку, шаг и диагноз — условие, задерживающее выполнение.

Флаги с точки зрения отладки, возможно, не являются самым удобным средством, но зато они универсальны. Анализ флагов и управляющих переменных SFC-исполнителя позволяет при необходимости создать в прикладной программе собственный модуль контроля и ранней диагностики «разумности» системы управления.

Глава 8. Стандартные компоненты

В этом разделе описаны наиболее широко применяемые стандартные операторы, функции и функциональные блоки. Описанные компоненты присутствуют во всех без исключения комплексах МЭК-программирования.

В конкретных реализациях возможны незначительные отличия. Для информации о включенных в состав поставки конкретного комплекса библиотечных компонентах используйте руководство по применению. Руководства по применению обычно пишут люди, настолько владеющие предметом, что им трудно представить, что тут вообще может быть что-то непонятно. Поэтому перед применением нового компонента в своей программе желательно детально исследовать его работу на простых примерах.

Еще раз напомним, что для начинающих при первом знакомстве досконально разбирать все тонкости стандартных компонентов не обязательно. Для начала достаточно иметь общее представление, чтобы суметь найти решение, когда возникнет такая необходимость.

8.1. Операторы и функции

8.1.1. Арифметические операторы

Почти все арифметические операторы имеют символьную форму для записи в выражениях языка ST. В других языках МЭК используются вызовы операторов в виде функции.

Оператор	Символ	Действие	Типы параметров
ADD	+	Сложение	ANY_NUM, TIME
SUB	–	Вычитание	ANY_NUM, TIME
MUL	*	Умножение	ANY_NUM, TIME
DIV	/	Деление	ANY_NUM, TIME

Оператор	Символ	Действие	Типы параметров
MOD	**MOD**	Остаток от деления	ANY_INT
EXPT		Возведение в степень	IN1 ANY_NUM IN2 ANY_INT
MOVE	:=	Присваивание	ANY

Арифметические операторы являются перегружаемыми: тип результата операции определяется типом операндов.

В графических языках блоки **MUL** и **ADD** можно расширять, т. е. добавлять произвольное число параметров (см. раздел «Функции с переменным числом параметров»).

Пример ST:

```
ix1:= (ix1 + ix2 + ix3) / 3;
```

Реализация этого же примера в FBD представлена на рис. 8.1.

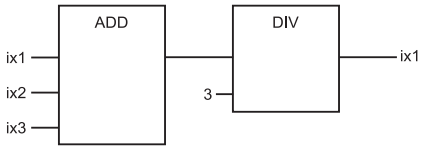

Рис. 8.1. Пример графического представления арифметических блоков

Переменные типа **TIME** можно складывать между собой, вычитать. Одну переменную типа **TIME** можно умножать и делить на число. Результат во всех случаях будет иметь тип **TIME**.

Операция **MOD** применима только на множестве целых чисел. Смысл выражения OUT := IN1 **MOD** IN2 можно раскрыть на языке ST так:

```
IF (IN2 = 0) THEN      OUT := 0 ;
ELSE                   OUT := IN1 – (IN1/IN2) * IN2 ;
END_IF
```

Операция OUT := **EXPT**(IN1/IN2) означает $OUT = IN1^{IN2}$. Параметр IN2 должен быть целого типа.

Операция **MOVE** может иметь только один параметр совместимого типа. В явном виде **MOVE** встречается только в графических языках. В IL присваивание значения одной переменной или константы другой переменной выполняется парой инструкций **LD, ST.**

8.1.2. Операторы битового сдвига

Операторы сдвига применимы для типов **ANY_BIT.** Все они имеют 2 параметра:

OUT := **ОПЕРАТОР**(IN,N);

Оператор	Действие
SHL	Побитный сдвиг операнда IN влево на N бит, с дополнением нулями справа
SHR	Побитный сдвиг операнда IN вправо на N бит, с дополнением нулями слева
ROR	Циклический сдвиг операнда IN вправо на N бит, старшие биты замещаются младшими
ROL	Циклический сдвиг операнда IN влево на N бит, младшие биты замещаются старшими

Пример применения операций сдвига в графической схеме представлен на рис. 8.2.

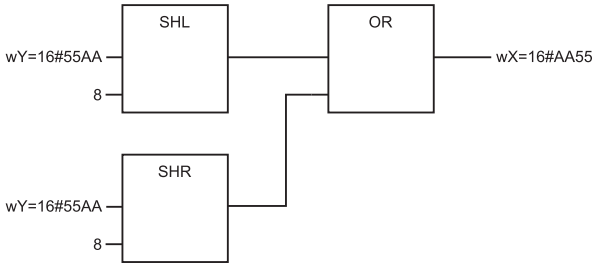

Рис. 8.2. Перестановка байт в слове (режим online)

В языке ST этот пример будет выглядеть так:

wX := **SHL**(wY,8) **OR SHR**(wY,8);

8.1.3. Логические битовые операторы

Битовые операторы применимы для типов **ANY_BIT**.

Оператор	Действие
AND	Побитное И
OR	Побитное ИЛИ
XOR	Побитное исключающее ИЛИ
NOT	Побитное НЕ

Оператор **NOT** имеет только один параметр.

В FBD блоки **AND**, **OR** и **XOR** можно расширять, т. е. добавлять произвольное число входных параметров (см. раздел «Функции с переменным числом параметров»). Операция **NOT** для проводников типов **BOOL** (инверсия) обозначается в виде окружности (см. пример на рис. 8.3).

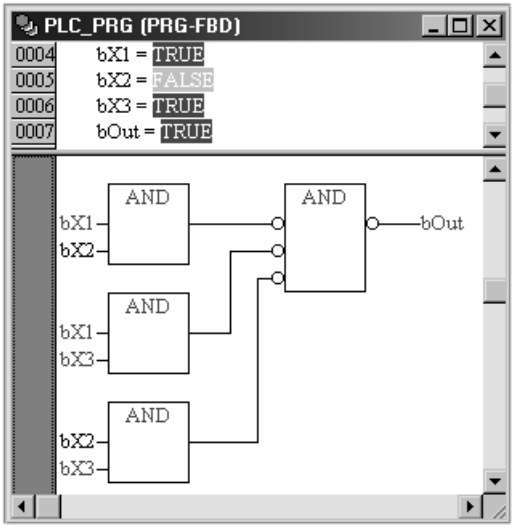

Рис. 8.3. Блок голосования два из трех (на логическом базисе И—НЕ)

В языке LD логические операции И, ИЛИ для типа **BOOL** заменяются монтажными соединениями. Операция **AND** представ-

ляется последовательным соединением контактов, а операция **OR** параллельным соединением (монтажное ИЛИ) (см. пример на рис. 8.4).

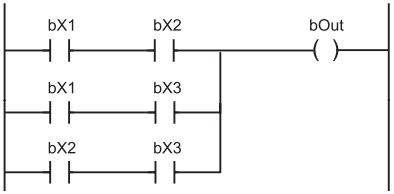

Рис. 8.4. Блок голосования два из трех (IL)

В выражениях ST логические операции записываются в виде операторов:

bOut := (bX1 **AND** bX2) **OR** (bX1 **AND** bX3) **OR** (bX2 **AND** bX3);

В IL логические операторы приобретают форму команд:

```
LD    bX1
AND   bX2
OR    ( bX1
AND   bX3
)
OR    ( bX2
AND   bX3
)
ST    bOut
```

8.1.4. Операторы выбора и ограничения

Операторы выбора и ограничения представлены в таблице.

Текстовый формат	Действие	Типы параметров
OUT:= **SEL**(G, IN0, IN1)	Бинарный выбор: OUT:= IN0 при G = FALSE OUT:= IN1 при G = TRUE	IN0, IN1: ANY G: BOOL
OUT := **MAX**(IN0, IN1)	Наибольшее из значений	ANY

Текстовый формат	Действие	Типы параметров
OUT := **MIN**(IN0, IN1)	Наименьшее из значений	ANY
OUT := **LIMIT**(Min, IN, Max)	Ограничитель: OUT:=**MIN**(**MAX**(IN,Min),Max)	
OUT := **MUX**(K, IN_0, …, $IN_{(K-1)}$)	Мультиплексор: OUT := IN_K.	IN_0, …, $IN_{(K-1)}$):ANY K: ANY_INT

В CoDeSys **MAX** и **MIN** оперируют только с двумя параметрами, стандартом предусматривается расширяемая реализация. Мультиплексор **MUX** является расширяемым (см. раздел «Функции с переменным числом параметров» и пример на рис. 8.5).

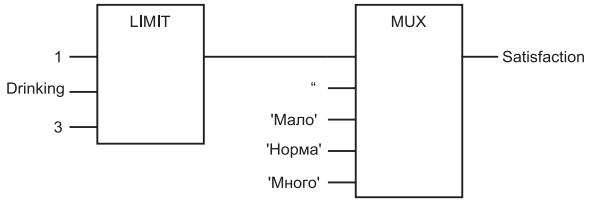

Рис. 8.5. Пример применения ограничителя и мультиплексора

8.1.5. Операторы сравнения

Следующие операторы реализуют операции сравнения.

Оператор	Символ	Действие
GT	>	Больше
GE	>=	Больше или равно
EQ	=	Равно
LE	<=	Меньше или равно
LT	<	Меньше
NE	<>	Не равно

Интересно, что стандарт допускает расширение операторов сравнения. Например, выражение bOut := iX1 > iX2 > iX3 вполне

допустимо. На практике обычно реализуют операторы сравнения на 2 входа.

Операторы сравнения принимают 2 параметра любого типа, тип возвращаемого значения **BOOL**. Действие описывается по отношению к первому параметру. Так, bOUT := IN1 > IN2; будет иметь значение TRUE, если IN1 больше IN2.

Если необходимость сравнения нескольких величин все же возникает, необходимо использовать несколько операторов сравнения, объединенных по **И**. Сравнение трех переменных на ST можно записать так:

bOut := iX1 > iX2 **AND** iX1 > iX3 **AND** iX2 > iX3;

В графическом представлении используется многовходовый блок **AND**. На рис. 8.6 дан пример на сравнение трех переменных.

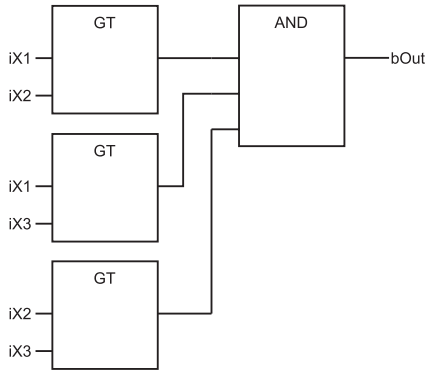

Рис. 8.6. Сравнение трех переменных iX1 > iX2 > iX3 (FBD)

Сравнение текстовых строк производится на основании значений кодов символов.

8.1.6. Математические функции

Стандартные математические функции представлены следующими операторами.

Оператор	Действие	Типы параметров
ABS	Абсолютное значение числа	ANY_NUM
SQRT	Квадратный корень числа	ANY_REAL

Оператор	Действие	Типы параметров
LN	Натуральный логарифм числа	ANY_REAL
LOG	Десятичный логарифм числа	ANY_REAL
EXP	Экспонента	ANY_REAL
SIN	Синус	ANY_REAL
COS	Косинус	ANY_REAL
TAN	Тангенс	ANY_REAL
ASIN	Арксинус	ANY_REAL
ACOS	Арккосинус	ANY_REAL
ATAN	Арктангенс	ANY_REAL

Единственная функция способная работать с целочисленными значениями **ABS**, все прочие функции возвращают результат только в формате с плавающей запятой.

Стандартные тригонометрические функции работают в радианах.

Пример ST:

```
lrY := EXPT(SIN(lrX),2) + EXPT(COS(lrX),2); (*Результат 1,
                                              при любом x*)
```

8.1.7. Строковые функции

Строковые функции представлены следующими инструкциями.

INT := **LEN(STR)** Возвращает длину строки
STR := **LEFT**(STRING **STR**, INT **SIZE**) Возвращает левую часть STR размером SIZE
STR := **RIGHT**(STRING **STR**, INT **SIZE**) Возвращает правую часть STR размером SIZE
STR := **DELETE**(STRING **STR**,INT **LEN**,INT **POS**) Возвращает STR, удалив LEN символов с позиции POS

INT := **LEN(STR)** Возвращает длину строки
STR := **MID**(STRING **STR**, INT **LEN**, INT **POS**) Возвращает часть STR с позиции POS длиной LEN
STR := **CONCAT**(STRING **STR1**, STRING **STR2**) Возвращает конкатенацию строк STR := STR1 + STR2
STR := **INSERT**(STRING **STR1**, STRING **STR2**, INT **POS**) Возвращает STR1 со вставленной STR2 в позицию POS
STR := **REPLACE**(**STR1**, STRING **STR2**, INT **LEN**, INT **POS**) Возвращает STR1, заменив LEN символов, с позиции POS на STR2
INT := **FIND**(STRING **STR1**, STRING **STR2**) Возвращает позицию STR2 в строке STR1. Если STR2 не найдена, возвращает 0

Нумерация позиций в строке начинается с 1.

8.2. Стандартные функциональные блоки

8.2.1. Таймеры

Таймеры ПЛК принципиально отличаются от таймеров, применяемых в языках общего применения. В языках программирования компьютеров существуют функции задержки (delay, sleep), которые приводят к приостановке выполнения программы на заданное время. Таймера, способного приостановить работу ПЛК, в стандарте МЭК нет. Представьте себе, что на один вход контроллера поступает некоторый сигнал. На второй вход поступает тот же сигнал, но через аппаратный модуль задержки. Именно так работают стандартные таймеры. Временная задержка влияет только на формирование выходных сигналов и не вызывает никакого замедления в программе.

Для правильной работы таймеров необходима аппаратная поддержка. Все экземпляры функциональных блоков таймеров «засекают» время (в CoDeSys во внутренней локальной переменной StartTime), пользуясь общими часами. При проектировании ПЛК достаточно иметь один аппаратный таймер-счетчик, увеличивающийся с постоянной частотой. Аппаратный счетчик должен иметь

достаточную разрядность, чтобы исключить возможность переполнения за один рабочий цикл ПЛК.

Нельзя полагаться на то, что повторный вызов экземпляра функционального блока в одном рабочем цикле даст различные результаты. Значения программных таймеров могут обновляться при вызове экземпляра функционального блока или синхронно с обновлением входов. Это зависит от реализации системы исполнения. Не используйте в своих программах циклы (WHILE, REPEAT) с условием окончания итераций по таймеру.

TP генератор импульса

TP			
IN	**BOOL**	**Q**	**BOOL**
PT	**TIME**	**ET**	**TIME**

Запуск таймера происходит по фронту импульса на входе IN. Вход PT задает длительность формируемого импульса. После запуска таймер не реагирует на изменение значения входа IN. Выход ET отсчитывает прошедшее время. При достижении ET значения PT счетчик останавливается, и выход Q сбрасывается в 0.

Временная диаграмма работы таймера TP показана на рис. 8.7.

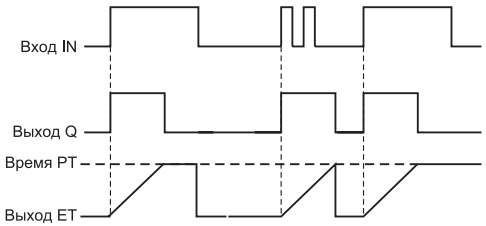

Рис. 8.7. Временная диаграмма работы таймера TP

На рис. 8.8 показано простейшее применение блока TP в качестве генератора коротких прямоугольных импульсов. Длительность паузы задается таймером. Начальное состояние bx = 0. В первом цик-

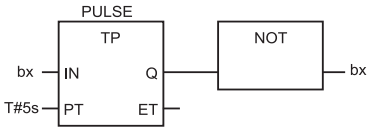

Рис. 8.8. Пример использования блока TP

ле bx получит значение 1 благодаря инвертору **NOT**. Так формируется фронт запуска, который поступает на вход IN таймера в третьем цикле. Инвертор формирует фронт запуска по каждому спаду выхода таймера.

TOF таймер с задержкой выключения

TOF			
IN	BOOL	Q	BOOL
PT	TIME	ET	TIME

По фронту входа IN выход Q устанавливается в TRUE. Сброс счетчика ET и начало отсчета времени происходит по каждому спаду входа IN. Выход Q будет сброшен через заданное PT время после спада входного сигнала. Если во время отсчета вход IN будет установлен в TRUE, то отсчет приостанавливается. Таким образом, выход Q включается по фронту, а выключается логическим нулем продолжительностью не менее PT.

Временная диаграмма работы таймера TOF показана на рис. 8.9.

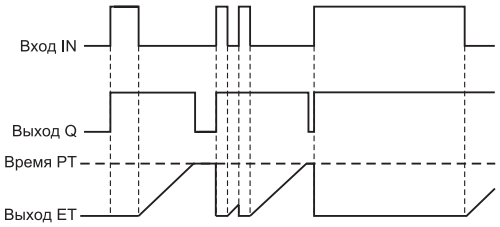

Рис. 8.9. Временная диаграмма работы таймера TOF

TON таймер с задержкой включения

TON			
IN	BOOL	Q	BOOL
PT	TIME	ET	TIME

По фронту входа IN выполняется обнуление счетчика и начинается новый отсчет времени. Выход Q будет установлен в **TRUE**

через заданное PT время, если IN будет продолжать оставаться в состоянии **TRUE**. Спад входа IN останавливает отсчет и сбрасывает выход Q в **FALSE**. Таким образом, выход Q включается логической единицей продолжительностью не менее PT, а выключается по спаду входа IN.

Временная диаграмма работы таймера TON показана на рис. 8.10.

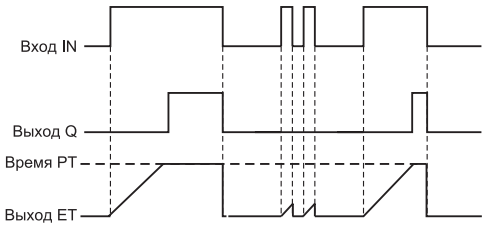

Рис. 8.10. Временная диаграмма работы таймера TON

RTC часы реального времени

RTC			
EN	**BOOL**	Q	**BOOL**
PDT	**DATE_AND_TIME**	CDT	**DATE_AND_TIME**

При создании экземпляра блока, пока вход EN равен **FALSE**, выход Q равен **FALSE**, а выход CDT равен DT#1970-01-01-00-00:00:00. По переднему фронту EN в часы загружается начальное время PD и начинается отсчет. Пока часы работают, выход Q = **FALSE**. Если EN перейдет в **FALSE**, CDT сбросится в начальное значение.

Описанная реализация блока RTC вызывает массу сложностей. Очевидно, часы должны продолжать свою работу при выключенном питании ПЛК. Коррекция хода часов отдельных экземпляров функционального блока RTC должна осуществляться на прикладном уровне. Многие ПЛК имеют аппаратно реализованные часы реального времени. Доступ к аппаратным часам гораздо проще организовать через прямо адресуемые ячейки памяти, чем поддерживать блок RTC.

Кроме всего прочего, программирование задач управления по часам реального времени достаточно коварно. Иногда часы могут

идти назад! Например, при коррекции или переходе на зимнее время. В настоящее время функциональный блок RTC исключен из стандарта МЭК.

8.2.2. Триггеры

Работу триггеров SR и RS легче всего понять по аналогии с электрическими устройствами. Представьте себе электрический пускатель. Для переключения ему нужны 2 кнопки «ПУСК» и «СТОП». Кнопки не имеют механической фиксации, переключение выполняется коротким нажатием кнопок. Пускатель сам фиксирует свое состояние.

Именно так работают триггеры SR и RS. Их поведение отличается только при одновременном нажатии обеих кнопок. В блоке доминантной установки SR побеждает «ПУСК». В блоке доминантного сброса RS побеждает «СТОП».

SR переключатель с доминантой включения

SR			
SET1	**BOOL**	Q1	**BOOL**
RESET	**BOOL**		

Блок SR имеет два устойчивых состояния Q1 = **TRUE** и Q1 = **FALSE**. На языке ST работа блока описывается выражением:

Q1 = (**NOT** RESET **AND** Q1) **OR** SET1.

Вход SET1 включает выход, вход RESET — выключает. При одновременном воздействии обоих входов вход SET1 является доминантным.

RS переключатель с доминантой выключения

RS			
SET	**BOOL**	Q1	**BOOL**
RESET1	**BOOL**		

Блок SR имеет два устойчивых состояния Q1 = TRUE и Q1 = FALSE. На языке ST работа блока описывается выражением:

Q1 = **NOT** RESET1 **AND** (Q1 **OR** SET).

Вход SET включает выход, вход RESET1 — выключает. При одновременном воздействии обоих входов вход RESET1 является доминантным.

8.2.3. Детекторы импульсов

Детекторы импульсов предназначены для применения в случае, когда требуется реакция не на состояние дискретного сигнала, а на его изменение.

R_TRIG детектор переднего фронта

R_TRIG			
CLK	**BOOL**	Q	**BOOL**

Функциональный блок R_TRIG генерирует единичный импульс по переднему фронту входного сигнала.

Реализация блока требует одной внутренней переменной M: **BOOL** := FALSE. На языке ST блок реализуется так:

Q := CLK **AND NOT** M;

M := CLK;

Выход Q устанавливается в TRUE, если в предыдущем цикле вход CLK был равен FALSE, а в текущем цикле он уже имеет значение TRUE. При следующем вызове функционального блока выход сбрасывается в FALSE. Переменная **M** запоминает значение CLK в предыдущем цикле.

Если на вход CLK подать константу TRUE, то при перезапуске ПЛК на выходе Q будет сформирован единичный импульс. Аналогично, если вход CLK связан с аппаратурой и уже имеет значение TRUE, экземпляр R_TRIG сформирует ложный единичный импульс при первом вызове. Если бы переменная M имела начальное значение TRUE, то ложного импульса не возникало бы. В случае, когда это явление не желательно, можно создать собственный безопасный детектор фронта или применить пустой вызов экземпляра при начальной инициализации. Такое поведение детекторов

фронтов не является ошибкой, поскольку во многих случаях начальный импульс оказывается желательным.

F_TRIG детектор заднего фронта

F_TRIG			
CLK	**BOOL**	Q	**BOOL**

Функциональный блок F_TRIG генерирует единичный импульс по заднему фронту входного сигнала.

Реализация блока требует одной внутренней переменной M: **BOOL** := FALSE. На языке ST блок реализуется так:

```
Q := NOT CLK AND NOT M;
M := NOT CLK;
```

Из сравнения двух реализаций очевидно, что блок F_TRIG превращается в R_TRIG включением на входе инвертора **NOT**.

Обратите внимание на примечание к блоку R_TRIG. Блок F_TRIG также обладает свойством формировать ложный импульс при перезапуске.

8.2.4. Счетчики

Теперь рассмотрим реализацию счетчиков.

CTU инкрементный счетчик

CTU			
CU	**BOOL**	Q	**BOOL**
RESET	**BOOL**		
PV	**WORD**	CV	**WORD**

По каждому фронту на входе CU значение счетчика (выход CV) увеличивается на 1. Выход Q устанавливается в TRUE, когда счетчик достигнет или превысит заданный PV порог. Логическая единица на входе сброса (RESET = TRUE) останавливает счет и обнуляет счетчик (CV := 0).

CTD декрементный счетчик

CTD			
CD	**BOOL**	Q	**BOOL**
LOAD	**BOOL**		
PV	**WORD**	CV	**WORD**

По каждому фронту на входе CD счетчик (выход CV) уменьшается на 1. Выход Q устанавливается в TRUE, когда счетчик достигнет нуля. Счетчик CV загружается начальным значением, равным PV по входу LOAD = TRUE.

CTUD инкрементный / декрементный счетчик

CTUD			
CU	**BOOL**	QU	**BOOL**
CD	**BOOL**	QD	**BOOL**
RESET	**BOOL**		
LOAD	**BOOL**		
PV	**WORD**	CV	**WORD**

По значению входа RESET = TRUE счетчик CV сбрасывается в 0. По значению входа LOAD = TRUE счетчик CV загружается значением равным PV.

По фронту на входе CU счетчик увеличивается на 1. По фронту на входе CD счетчик уменьшается на 1 (до 0).

Выход QU равен TRUE, если CV >= PV, иначе FALSE.

Выход QD равен TRUE, если CV = 0, иначе FALSE.

8.3. Расширенные библиотечные компоненты

В этом разделе описано несколько широко распространенных функций и функциональных блоков, реализованных практически всеми комплексами программирования ПЛК. В CoDeSys описанные блоки включены в состав библиотеки утилит (UTILS). Библиотека реализована как внутренняя, т. е., написана исключительно на языках МЭК (ST) и доступна для редактирования.

8.3.1. Побитовый доступ к целым

Для побитового доступа к целым используются следующие инструкции.

EXTRACT

Чтение бита. Функция EXTRACT типа **BOOL** имеет 2 параметра: **DWORD** X и **BYTE** N. Возвращает TRUE, если бит номер N в числе X равен 1, в противном случае — FALSE. Нумерация бит начинается с 0.

PUTBIT

Запись бита. Функция типа **DWORD** имеет 2 параметра: **DWORD** X, **BYTE** N и **BOOL** B. Функция возвращает X с установленным в 1 битом N, если B равен TRUE. В противном случае заданный бит принимает значение 0.

PACK

Упаковка значений восьми логических переменных в байт. Функция PACK получает восемь параметров B0, B1, ..., B7 типа **BOOL**. Возвращаемое значение типа **BYTE** содержит побитно упакованные значения входных параметров.

UNPACK

Распаковка байта в восьми логических переменных. Функциональный блок PUTBIT имеет восемь входов типа **BOOL** и выход типа **BYTE**, выполняет обратную по отношению к PACK распаковку.

Пример функционального блока Turn, изменяющего порядок бит в байте на обратный:

```
FUNCTION_BLOCK Turn
VAR_INPUT
   In: BYTE;
END_VAR
VAR_OUTPUT
   Out: BYTE;
END_VAR
VAR
   B0,B1,B2,B3,B4,B5,B6,B7: BOOL;
   UNPACKER: UNPACK;
END_VAR
```

```
UNPACKER(B:=In, B0=>B7, B1=>B6, B2=>B5, B3=>B4,
B4=>B3, B5=>B2, B6=>B1, B7=>B0);
Out := PACK(B0, B1, B2, B3, B4, B5, B6, B7);
```

8.3.2. Гистерезис

Функциональный блок HYSTERESIS (см. рис. 8.11) реализует компаратор, обладающий эффектом гистерезиса.

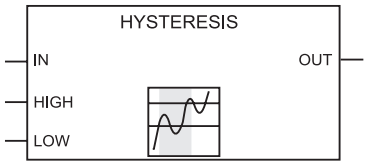

Рис. 8.11. Функциональный блок HYSTERESIS

Если значение входа IN превышает порог HIGH, выход OUT приобретает значение FALSE. Если значение IN меньше порога LOW, выход OUT равен TRUE. В пределах от LOW до HIGH выход функционального блока HYSTERESIS не изменяется. Реализация блока на языке ST очень проста:

```
IF  IN < LOW THEN
            OUT := TRUE;
END_IF
IF IN > HIGH THEN
            OUT := FALSE;
END_IF
```

Все три входа IN, LOW и OUT — типа **INT**, выход OUT типа **BOOL**.

Применение функционального блока HYSTERESIS демонстрирует рис. 8.12.

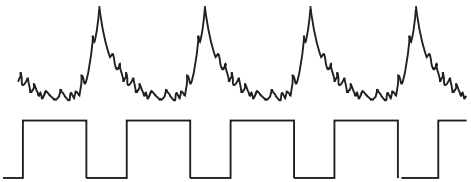

Рис. 8.12. Селекция импульсов с помощью HYSTERESIS

8.3.3. Пороговый сигнализатор

Функциональный блок LIMITALARM отслеживает соответствие значения входа заданному диапазону. Результат формируется с помощью логических выходов — меньше, больше, норма. Величина входного сигнала IN сравнивается с верхним HIGH и нижним LOW порогами. Все входные переменные целого типа со знаком **(INT)**. Три логических **(BOOL)** выхода сообщают результат сравнения:

- выход O сигнализирует о повышенном значении (IN > HIGH);
- выход U сигнализирует о пониженном значении (IN < LOW);
- выход IL сообщает о допустимом значении (LOW<= IN >= HIGH).

На рис. 8.13 проиллюстрирована работа LIMITALARM. Здесь амплитуда входного синусоидального сигнала равна 1000 единиц. Верхний порог — 600, нижний порог — – 600.

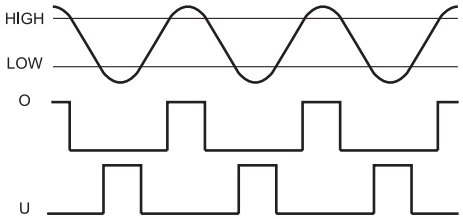

Рис. 8.13. Иллюстрация работы функционального блока LIMITALARM

8.3.4. Ограничение скорости изменения сигнала

Функциональный блок RAMP ограничивает скорость изменения сигнала. RAMP имеет 1 выход OUT и 5 входов:

IN	входной сигнал
ASCEND	максимальное нарастание
DESCEND	максимальный спад
TIMEBASE	время нарастания/спада
RESET	сброс (BOOL)

Если новое значение входа по сравнению с предыдущим выросло меньше, чем на ASCEND, или уменьшилось в пределах DESCEND, сигнал беспрепятственно передается на выход. В случае слишком быстрого роста или спада сигнала его изменение ограничивается. TIMEBASE задает время, за которое определяется изменение. Мгновенное изменение выхода рассчитывается так, чтобы за заданный интервал не превысить установленные пороги. Если TIMEBASE равен t#0s, то в качестве интервала используется один цикл вызова экземпляра функционального блока.

Сброс (RESET := TRUE) вызывает мгновенное присваивание выходу входного значения. После снятия сброса отслеживание изменений пойдет с текущего значения.

В библиотеке UTILS функциональный блок реализован дважды: для сигнала типа INT (RAMP_INT) и REAL (RAMP_REAL) (см. пример на рис. 8.14).

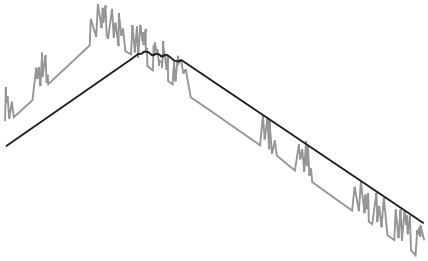

Рис. 8.14. График входного сигнала и выхода RAMP

8.3.5. Интерполяция зависимостей

Функциональный блок CHARCURVE (рис. 8.15) выполняет кусочно-линейную интерполяцию зависимостей, заданных вектором значений узловых точек. Напоминаем, что интерполяция означает вычисление значений зависимости, заданной узловыми точками в промежутках между этими точками.

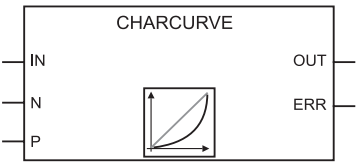

Рис. 8.15. Функциональный блок CHARCURVE

Пусть входное значение x принадлежит интервалу $X_i < x < X_{i+1}$. Тогда y лежит на отрезке прямой, соединяющем соответствующие узловые точки (X_i,Y_i) и (X_{i+1},Y_{i+1}). Значение y определяется по формуле прямой:

$$y = Y_i + \frac{Y_{i+1} \text{ с } Y_i}{X_{i+1} \text{ с } X_i}(x \text{ с } X_i).$$

В функциональном блоке CHARCURVE для удобства вычисления используется интервал (X_{i-1},Y_{i-1}) и (X_i,Y_i). То есть интерполяция опирается не на начальную, а на конечную узловую точку интервала. В этом случае формула выглядит так:

$$y = Y_i \text{ с } \frac{Y_i \text{ с } Y_{i\text{с}1}}{X_i \text{ с } X_{i\text{с}1}}(X_i \text{ с } x).$$

Вектор узловых точек CHARCURVE должен быть представлен в виде массива **ARRAY** P[0..10], где P — структура типа POINT, состоящая из двух переменных INT — *X* и *Y*. Вход N **(BYTE)** указывает число узловых точек, которое не должно превышать 11, что соответствует десяти отрезкам интерполяции. Данные подаются на вход IN **(INT)** и после преобразования поступают на выход OUT (INT).

Узловые точки должны быть отсортированы в порядке возрастания значений X. Выход ERR дает диагностику возможных ошибок применения функционального блока CHARCURVE:

ERR	Расшифровка
0	Ошибок нет
1	Массив точек отсортирован неверно
2	Входное значение лежит за пределом области определения: IN < P[0].X или IN > P[N-1].X
4	Недопустимое значение N: N < 2 или N > 11

При обнаружении ошибки выход OUT принимает значение равное нулю.

Применение блока CHARCURVE подробно проиллюстрировано в примере «Линеаризация измерений».

8.3.6. Дифференцирование

Функциональный блок DERIVATE осуществляет численное дифференцирование входного сигнала (рис. 8.16).

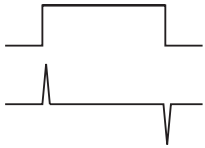

Рис. 8.16. Дифференцирование прямоугольного импульса

Производную сигнала можно вычислить, зная две ординаты и приращение времени. Разделив приращение сигнала на приращение времени, получим приближенно значение производной. Для получения точного значения необходимо использовать бесконечно малое приращение времени, что невозможно на практике. Расчет производной по двум точкам дает хорошие результаты при точных значениях ординат. При работе с оцифрованными результатами измерений обязательно присутствует погрешность квантования и, как правило, другие высокочастотные шумы различной природы. Незначительное отклонение ординаты одной из двух точек существенно влияет на результат. Поэтому функциональный блок DERIVATIVE использует более совершенную формулу, полученную на основе интерполяционного полинома по четырем ординатам.

Описанный эффект представлен на рис. 8.17. На графике сигнала (жирная линия) показаны четыре узловые точки. Поскольку мы говорим о сигнале, по оси абсцисс отложено время, по оси ор-

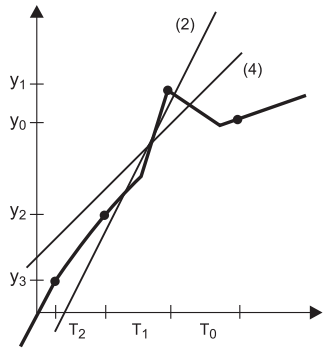

Рис. 8.17. Дифференцирование по двум и по четырем точкам

динат — уровень сигнала. Точки пронумерованы так, что больший индекс соответствует более старым значениям. Здесь Y_0 — текущая ордината, Y_1 — предыдущая. Аналогично: T_0 — это интервал времени между Y_1 и Y_0 и т. д. Отклонение ординаты Y_1 вверх имитирует на рисунке шум реального сигнала. Рассчитанная по двум точкам Y_1 и Y_2 производная примерно равна 2,03. Прямая линия с таким тангенсом угла наклона обозначена на рисунке цифрой (2). Прямая (4) соответствует расчету производной по четырем точкам (tg = 0,99).

Входящий в библиотеку утилит CoDeSys функциональный блок DERIVATIVE рассчитывает значение производной по следующей формуле:

$$OUT := (3(Y_0 - Y_3) + Y_1 - Y_2) / (3T_2 + 4T_1 + 3T_0).$$

Интервалы времени между замерами необязательно должны быть равными.

Ординаты точек запоминаются за четыре последних вызова экземпляра функционального блока. При каждом очередном вызове экземпляра значения сдвигаются:

$Y_3 = Y_2$, $Y_2 = Y_1$, $Y_1 = Y_0$.
$T2 = T1$, $T1 = T0$.

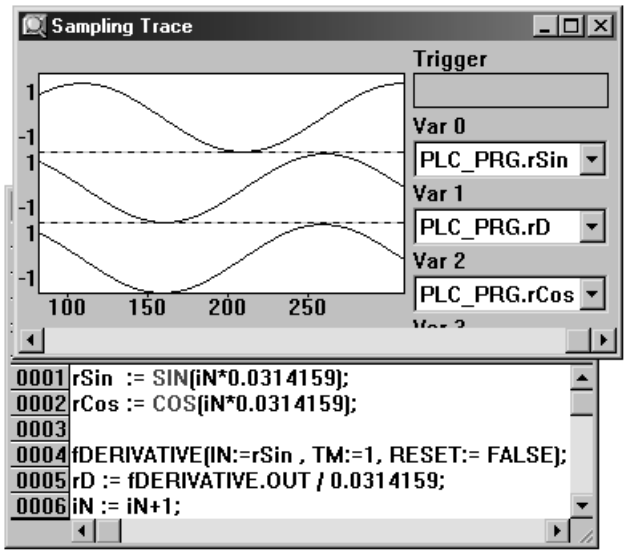

Рис. 8.18. Проверка работы функционального блока DERIVATIVE

Функциональный блок DERIVATIVE имеет 3 входа:

IN	**REAL**	входное значение
TM	**DWORD**	приращение времени (в мс)
RESET	**BOOL**	сброс

Значение производной дает единственный выход OUT типа **REAL**. Для приведенной выше формулы Y_0 это IN, а T_0 соответственно TM. Во время сброса (RESET = TRUE) и при начальной инициализации экземпляра (первом вызове) ординаты Y_3, Y_2 и Y_1 принимают значение IN, выход OUT равен 0.

Применение функционального блока DERIVATIVE иллюстрирует следующий пример. Результаты трассировки по циклам и окно текста примера (ST) показаны на рис. 8.18. Как известно, производная от функции синуса — косинус, что подтверждается кривыми трассировки.

Малозаметная тонкость функционального блока DERIVATIVE вызвана тем, что он опирается на формулу центрированной разности. Фактически он вычисляет приближенное значение производной для точки соответствующей центру интервала T_1. В результате выход задержан по отношению к входу на один с половиной цикл вызова экземпляра.

8.3.7. Интегрирование

Функциональный блок INTEGRAL вычисляет приближенное значение определенного интеграла входного сигнала (см. рис. 8.19).

Функциональный блок INTEGRAL имеет 3 входа:

IN	**REAL**	входное значение
TM	**DWORD**	приращение времени (в мс)
RESET	**BOOL**	сброс

Значение интеграла дает выход OUT. Выход OVERFLOW **(BOOL)** сигнализирует о переполнении максимального значения переменной OUT типа **REAL**.

Рис. 8.19. Интегрирование треугольного импульса

Перед началом вычислений необходимо выполнить сброс (RESET). Далее заданный интервал интегрирования разбивается на несколько малых частей. Значение интеграла вычисляется путем циклического суммирования. При каждом вызове экземпляра функционального блока он получает на вход новое значение IN и соответствующее ему приращение времени TM. Интеграл рассчитывается приближенно методом прямоугольников (см. рис. 8.20). Значение интеграла приближенно равно сумме площадей прямоугольников. На языке ST реализация данного функционального блока (без сброса) описывается одним выражением : OUT := OUT + IN * TM. Сброс выполняется установкой значения входа RESET в TRUE, он вызывает обнуление суммы и снимает признак переполнения.

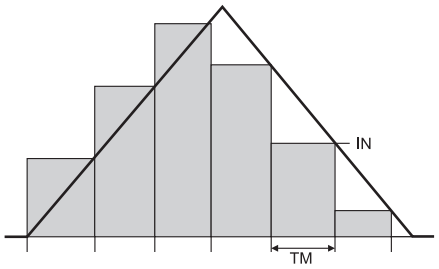

Рис. 8.20. Принцип работы функционального блока INTEGRAL

Работу функционального блока INTEGRAL иллюстрирует следующий пример. Проинтегрируем от 0 до 2π сигнал, заданный формулой AC = 10 + 4 sin x + 2 sin 2x. Реализация примера на языке ST и окно трассировки показаны на рис. 8.21. Интервал интегрирования [0,2π] разбит на 200 частей.

Обратите внимание, что функциональные блоки DERIVATIVE и INTEGRAL реализуют только математический алгоритм и не занимаются самостоятельно замером приращения времени (как, например, RAMP). Эту операцию необходимо выполнять внешни-

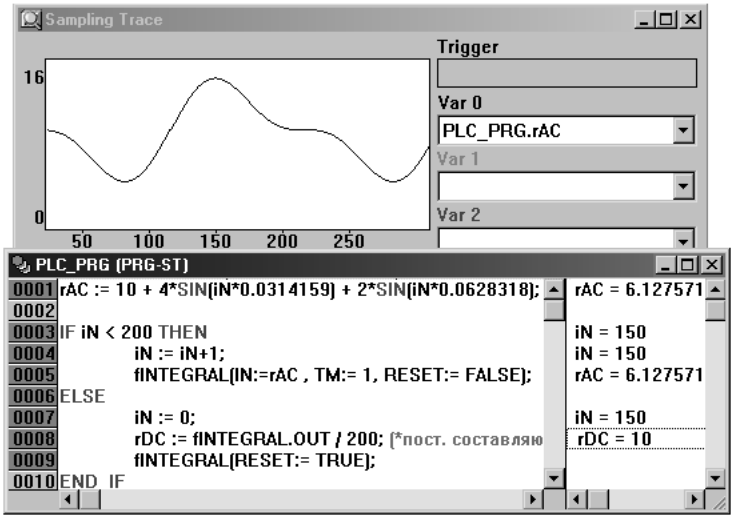

Рис. 8.21. Выделение постоянной составляющей сигнала

ми средствами. В описаниях величина TM задана в миллисекундах, но это только лишь отражает наиболее распространенную область применения. На самом деле ничего не мешает работе функционального блока INTEGRAL с произвольным масштабом времени (секунды, часы и т. д.). Мало того, вообще не обязательно ограничиваться интегрированием по времени. Например, вы можете применить INTEGRAL для расчета расхода топлива за отрезок пути по мгновенным значениям.

8.3.8. ПИД-регулятор

Пропорционально-интегрально-дифференциальный регулятор (ПИД) — это наиболее широко применяемый тип автоматических регуляторов с обратной связью.

Рассмотрим кратко, как работает ПИД-регулятор. Предположим, некоторый объект имеет вход, позволяющий управлять им, и датчик, измеряющий реакцию объекта (выходная переменная). Кроме того, на объект действуют различные возмущающие факторы. В результате выходная переменная может изменяться даже при постоянном входном задании. Разница задания (эталонного входного сигнала) и выходной переменной образует ошибку управления $e(t)$. Задачей регулятора является автоматическое из-

менение входного воздействия $y(t)$ так, чтобы свести последствия возмущений к минимуму.

В зависимости от объекта управления выходной переменной может быть напряжение, давление, температура, скорость перемещения и т. д. В общем случае выходная переменная реагирует на изменение воздействия по некоторому сложному закону с запаздыванием. Математическая модель такого объекта представляет собой систему дифференциальных уравнений. На основании модели объекта можно найти оптимальный закон регулирования. Но это достаточно сложная задача. В большинстве практических случаев промышленной автоматики применяется универсальный ПИД-регулятор. Для него закон регулирования формируется настрой трех констант, рассчитанных на основании модели объекта или подобранных опытным путем. Уравнение ПИД-регулятора имеет вид:

$$y(t) = Y_0 + Kp(e(t) + \frac{1}{Tn}\int_0^t e(t) + Tv\frac{de(t)}{dt}),$$

где Y_0 — значение при нулевой ошибке, $e(t)$ — сигнал ошибки, Kp — коэффициент передачи, Tn — постоянная интегрирования, Tv — постоянная дифференцирования.

На рис. 8.22 показаны результаты трассировки системы с регулятором на основе функционального блока PID при единичном скачке сигнала управления. Верхняя кривая соответствует переходной характеристике объекта управления. Вторая кривая — выходная переменная в системе с ПИД-регулятором. Нижняя кривая — сигнал управления с выхода регулятора.

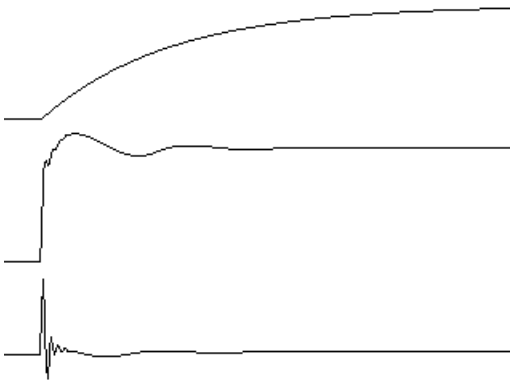

Рис. 8.22. Пример применения функционального блока ПИД

Пропорциональный канал регулятора формирует постоянное управляющее воздействие при постоянном значении ошибки. Если же приложенное к объекту воздействие оказывается недостаточным, ошибка сохраняется бесконечно долго. На рис. 8.23 показано регулирование объекта, обладающего трением, который «трогается с места» только тогда, когда величина управления превысит значение единицы. В данном же случае скачок управляющего задания вдвое меньше.

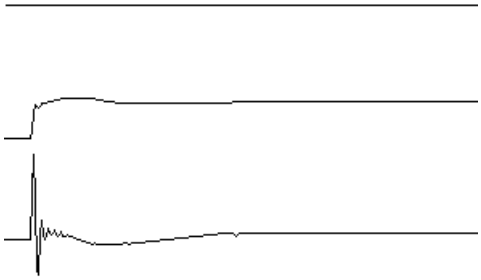

Рис. 8.23. Регулирование объекта с трением

Интегрирование даже малой постоянной ошибки приводит к постоянному увеличению управляющего воздействия во времени. Тем самым достигается точность регулирования при изменении характеристик объекта. Дифференциальный канал ПИД-регулятора улучшает динамические характеристики, компенсируя запаздывание фазы управляющего сигнала.

Интерфейс функционального блока ПИД-регулятора определяется следующими входами:

ACTUAL	**REAL**	сигнал обратной связи
SET_POINT	**REAL**	задание
KP	**REAL**	коэффициент передачи
TN	**DWORD**	постоянная интегрирования (msec)
TV	**DWORD**	постоянная дифференцирования (msec)
Y_OFFSET	**REAL**	стационарное значение
Y_MIN	**REAL**	минимальное допустимое значение

Y_MAX	**REAL**	максимальное допустимое значение
MANUAL	**BOOL**	ручной режим
RESET	**BOOL**	сброс

и выходами:

Y	**REAL**	управляющее воздействие
LIMITS_ACTIVE	**BOOL**	признак достижения пороговых значений
OVERFLOW	**BOOL**	признак ошибки переполнения

Схематично функциональный блок можно представить следующим образом (рис. 8.24).

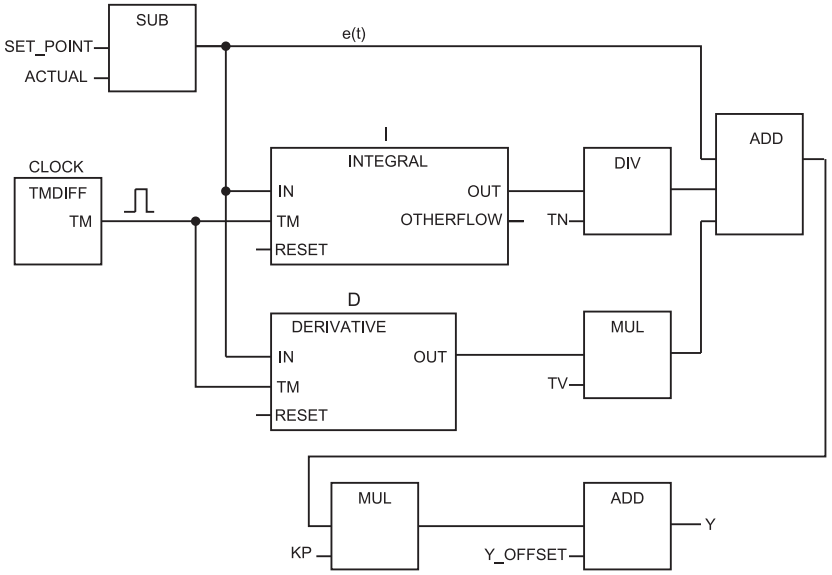

Рис. 8.24. Схема построения функционального блока ПИД

Помимо основной функции ПИД-регулирования, в CoDeSys реализации данного функционального блока предусмотрен ручной режим и ограничение диапазона выходной переменной регулятора. В ручном режиме регулятор выключен, значение Y автомати-

чески не регулируется. Если заданы пороги допустимого значения (Y_MAX > Y_MIN), регулятор автоматически ограничивает значение управляющего воздействия и адаптирует рост интегральной составляющей.

Пропорционально-интегральный (ПИ) регулятор получается из функционального блока ПИД при TV := 0. Для пропорционального (П) и пропорционально-дифференциального (ПД) законов регулирования библиотека утилит CoDeSys включает функциональный блок PD.

Глава 9. Примеры программирования

В этой главе будут подробно рассмотрены примеры программирования с применением языков МЭК. Примеры расположены в порядке возрастания сложности. В заголовках примеров сокращенно указаны примененные языки и форма реализации. Сокращение PRG говорит о том, что пример реализован как программа в форме законченного проекта. Другие примеры реализованы в форме компонентов, FUN — функция, FB — функциональный блок. Все иллюстрации получены на основе реализации примеров в комплексе CoDeSys.

9.1. Генератор импульсов (PRG LD)

Генератор прямоугольных импульсов с заданными длительностями импульса и паузы показан на рис. 9.1. Это элементарный пример LD с функциональными блоками.

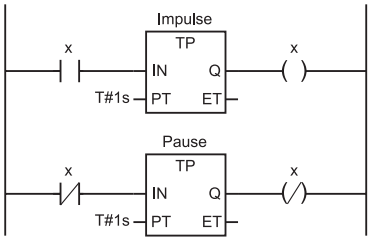

Рис. 9.1. Генератор прямоугольных импульсов

Таймер *Impulse* запускается по переднему фронту *x* и сбрасывает *x* по окончании заданного времени, запуская, таким образом, таймер *Pause*. Конец паузы взводит переменную *x*, которая и запускает новый импульс (в следующем рабочем цикле). Переменная *x* является выходом генератора.

Безусловно, задачу формирования прямоугольных импульсов можно решить с применением единственного таймера (см. пример ШИМ или стандартный блок BLINK CoDeSys). Путем дора-

ботки цепей данного примера можно формировать импульсы, зависящие от более сложных условий, чем одиночные выходы таймеров.

Выходы ET-таймеров дают линейно нарастающие значения времени. Если преобразовать их к целому типу, то получится генератор треугольных импульсов.

9.2. Последовательное управление по времени (PRG LD, SFC)

Используя блоки таймеров, несложно организовать последовательное переключение выходов с фазами заданной продолжительности и произвольным фазовым сдвигом. Программа, формирующая три последовательных интервала по 1 секунде, показана на рис. 9.2.

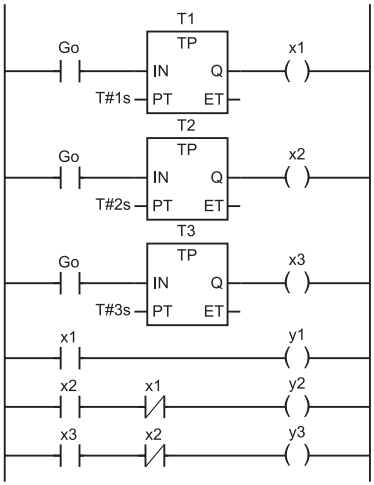

Рис. 9.2. Последовательное управление по времени в LD

По фронту входа Go одновременно запускаются все три таймера T1 — T3. Каждый таймер отмеряет момент окончания соответствующей фазы. Три нижних цепи выделяют выходы Y1 — Y3, соответствующие фазам управления. Все переменные программы должны быть объявлены как **BOOL**. Процесс работы схемы наглядно отражает экран трассировки CoDeSys (рис. 9.3).

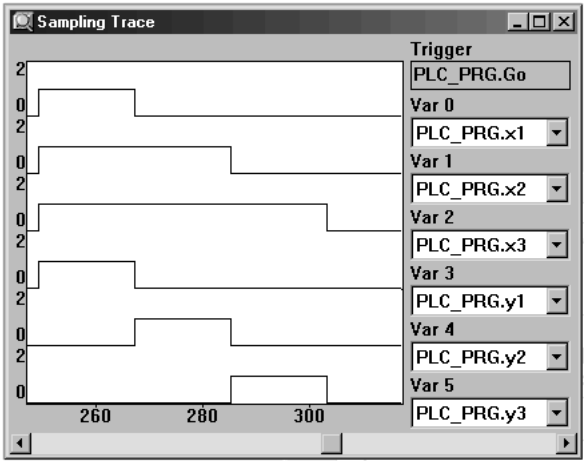

Рис. 9.3. Трассировка схемы последовательного управления

В данную схему несложно добавить цепь автоматического перезапуска. Тогда схема будет работать циклически, как «бегущие огни». Если же вы настроены более серьезно, назовем нашу программу модулем управления двигателем с электронной коммутацией обмоток статора.

Рис. 9.4. Цепь автоматического перезапуска

Данный пример можно реализовать значительно проще с применением SFC. В разделе объявлений нужно объявить всего три логических переменных.

```
y1, y2, y3: BOOL;
Go:         BOOL := TRUE;
```

Никаких вспомогательных переменных не требуется. Алгоритм реализуется дословно. После окончания работы шага S1 запускается шаг S2, за ним S3. Каждый шаг работает заданное время. Шаг Init — пустой, он ожидает разрешения работы — Go. Действия y1, y2, y3 связаны с логическими переменными. Программа целиком отражена в SFC-диаграмме на рис. 9.5.

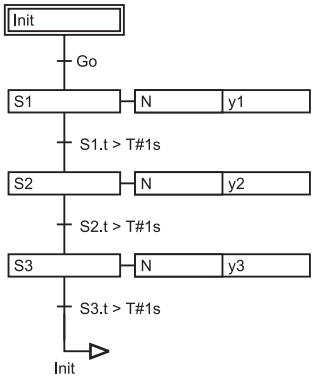

Рис. 9.5. Трехфазная коммутация в SFC

9.3. Кодовый замок (PRG LD)

Пример демонстрирует замену релейного автомата программной реализацией на ПЛК без переработки алгоритма работы устройства.

На рис. 9.6 приведена принципиальная электрическая схема кодового замка на электромагнитных реле. Для открывания замка необходимо набрать код последовательным нажатием кнопок К2, К3, К4, К5. Первую кнопку кода К2 нужно нажимать одновременно с кнопкой дверного звонка К1. Все промежуточные реле Р2 — Р5 работают с самофиксацией, одновременно освобождая реле предыдущей цепи. Ошибочное нажатие кнопок К6 — К9 или открывание двери ($К_д$) сбрасывают замок в исходное состояние. Таким образом, «свой» открывает замок с

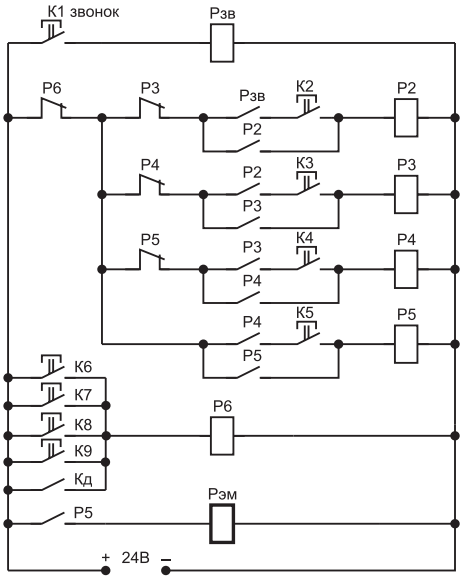

Рис. 9.6. Принципиальная схема кодового замка

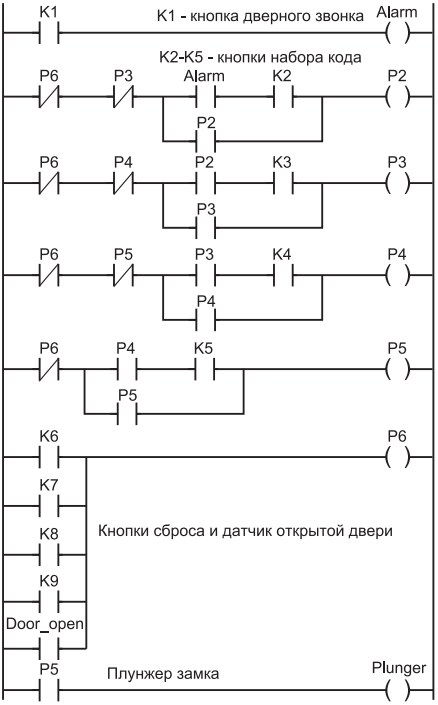

Рис. 9.7. LD-диаграмма кодового замка

очень коротким звонком, «чужой» вынужден поднять трезвон при попытках подбора кода. Одновременное нажатие всех кнопок К2 — К9 вызывает сброс в исходное состояние. Секретность цифр кода достигается распайкой К2 — К5 к различным клавишам наборного поля. Силовые цепи звонка громкого боя и электромагнитного плунжера на схеме не показаны.

Для реализации кодового замка на контроллере необходим ПЛК-имеющий 10 дискретных входов К1 — Кд и 2 выхода: звонок и электромагнитный плунжер. Реализующая логику управления диаграмма LD показана на рис. 9.7. Легко заметить, что диаграмма практически повторяет принципиальную схему. Единственное отличие в том, что контакты реле Р6 разделены на несколько цепей. Это диктуется правилами построения диаграмм и не влияет на работу схемы.

9.4. Динамический знаковый индикатор (FUN LD, ST)

Смысл динамической индикации заключается в экономии выходов контроллера. Один индикатор, состоящий из семи сегментов, требует для управления семь выходов. Допустим, необходимо построить табло, состоящее из N однотипных индикаторов. Если использовать статическую индикацию, то понадобится 7 * N выходов. Соединим параллельно одинаковые сегменты всех индикаторов и добавим ключи позволяющие управлять подачей питания на каждый из них отдельно. Тогда достаточно иметь 7 + N выходов для вывода информации на любой из индикаторов. Если поочередно зажигать все индикаторы достаточно быстро, то за счет послесвечения и инерции зрения образуется статическая картина.

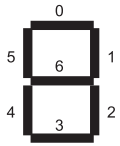

Рис. 9.8. Нумерация сегментов индикатора

Для управления индикатором будем использовать выход типа **BYTE**, где семи младшим битам сопоставлены сегменты, как показано на рис. 9.8.

Рассмотрим для примера табло, состоящее из двух индикаторов. Отображаемая информация — это восьмиразрядное число, которое нужно показать в шестнадцатеричном формате. Построение генератора, управляющего питанием индикаторов, не представляет сложности. Главная задача — создать функцию, которая должна формировать байт управления сегментами из двух параметров: номер индикатора (HD) и индицируемое значение (IN). Для выбора из двух индикаторов достаточно переменной типа **BOOL**. Значение входа HD равное ИСТИНА выбирает старший индикатор.

Раздел объявлений функции:

```
FUNCTION DinLight7 : BYTE
VAR_INPUT
   IN:     BYTE;
   HD:     BOOL;
END_VAR
VAR CONSTANT
   LightCodes: ARRAY[0..15] OF BYTE :=
         16#3F, 16#06, 16#5B, 16#4F,
```

```
        16#66, 16#6D, 16#7D, 16#07,
        16#7F, 16#6F, 16#77, 16#7C,
        16#39, 16#5E, 16#79, 16#71;
END_VAR
```

Изображение нужного шестнадцатеричного знака дают коды таблицы LightCodes. Если в качестве индекса массива использовать число от 0 до F, то значение элемента обеспечит высвечивание соответствующей цифры. Вся логика функции сводится к выделению нужной тетрады байта и использованию его в качестве индекса LightCodes. Если нужна младшая цифра, то старшая тетрада числа просто обнуляется. Для выбора старшей тетрады содержимое байта сдвигается на 4 вправо. Реализация функции на языке ST будет выглядеть так:

```
IF HD THEN
   IN := SHR (IN,4);
ELSE
   IN := IN AND 16#0F;
END_IF

DinLight7_ST := LightCodes[IN];
```

Функцию DinLight7 также несложно реализовать в LD, если использовать аналогичные блоки. Пример диаграммы дан на рис. 9.9.

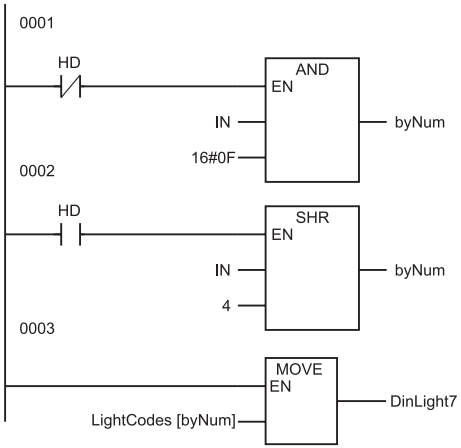

Рис. 9.9. LD-реализация функции DinLight7

На рис. 9.10 показан рабочий момент тестовой программы для проверки работы динамического индикатора. В окне Indicator изображена визуализация панели. Сегменты индикаторов окрашиваются красным цветом при единичных значениях соответствующих битов byLight1 и byLight2. Так, например, первый сегмент (см. рис. 9.8) правого индикатора включается по переменной PLC_PRG.byLight1.1. Кнопка «+» устанавливает в TRUE логическую переменную bAddButton, что вызывает последовательное увеличение значений x в тестовой программе.

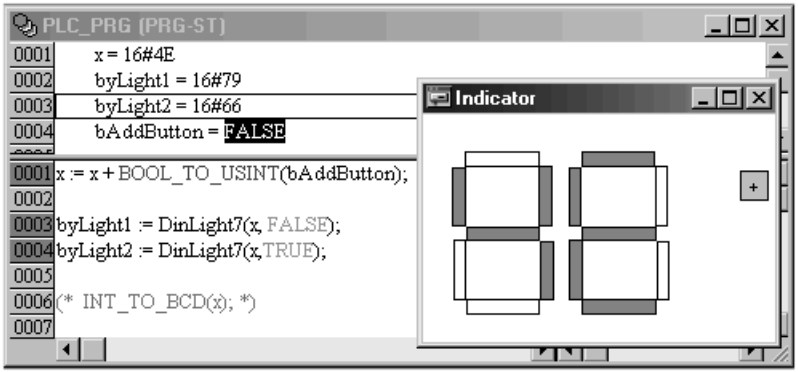

Рис. 9.10. Тестовая программа и визуализация индикатора

Обратите внимание, что вместо отображения шестнадцатеричного формата наша индикаторная панель способна отображать десятичные числа в диапазоне 0..99. Для этого достаточно преобразовать число формат BCD при вызове функции. Например:

Light := DinLight7(**INT_TO_BCD**(byNum), bPulse);.

9.5. Целочисленное деление с симметричным округлением (FUN ST)

Реализованная ниже функция выполняет деление в целых числах с симметричным округлением. Если точный результат деления ближе к большему целому, он округляется вверх, в противном случае — вниз. Найдем, например, частное 18/10. Точный результат 1,8. Стандартное целочисленное деление дает результат 1, относительная погрешность составляет 44%. Наша функция дает результат 2, относительная погрешность составляет 11%.

Задача суммирования целочисленных значений, вычисленных по формуле, включающей деление встречается достаточно часто, например, для определения среднего значения за заданный отрезок времени. При суммировании симметрично округленных чисел точность суммы повышается. Происходит это за счет компенсации абсолютных погрешностей, которые равновероятно имеют положительное и отрицательное значения. Округление методом отбрасывания остатка всегда дает числа с недостатком. В результате при суммировании ошибка накапливается.

Функция имеет 2 параметра — числитель и знаменатель.

```
FUNCTION ISMDIV : INT
VAR_INPUT
   iNum:      INT;
   iDen:      INT;
END_VAR
VAR
   iExcess:   INT;
   bSign:     BOOL;
END_VAR

(*определить знак результата*)
IF (iNum < 0) XOR (iDen < 0) THEN
   bSign := TRUE;
ELSE
   bSign := FALSE;
END_IF

(*округлять проще абсолютные значения*)
ISMDIV        := ABS(iNum / iDen);        (*частное*)
iExcess       := ABS(iNum MOD iDen);      (*и остаток*)

IF iExcess >=  ABS(iDen) - iExcess THEN
   ISMDIV := ISMDIV + 1;          (*ост. не меньше половины*)
END_IF

IF bSign THEN ISMDIV := -ISMDIV; END_IF
```

Результат работы функции хорошо заметен на графиках (см. рис. 9.11).

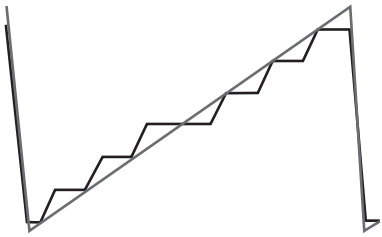

Рис. 9.11. График деления y := x/3, где x изменяется от −10 до +10

На рис. 9.11 показаны результаты деления на 3 пилообразного сигнала с амплитудой 10 единиц. Прямая отображает точное значение, а лесенка — обычное целочисленное деление с округлением вниз. Четко видно, что график результатов смещен вверх в отрицательной области и вниз в положительной.

Широкая горизонтальная полка нулевого уровня соответствует значениям −2, −1, 0, 1, 2 исходного сигнала. Графики получены путем трассировки в CoDeSys. Если отражать только полученные целые значения, то, естественно, никаких непрерывных линий быть не должно. Должны быть точки, соответствующие входным и выходным значениям функции, как показано на рис. 9.12.

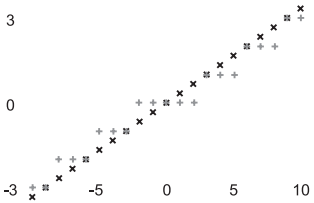

Рис. 9.12. Точечный график деления y := x/3 (× — точные значения, + — y)

С помощью трассировки нельзя непосредственно построить зависимости $y = f(x)$. Трассировка — это развертка значений во времени. Поэтому приходится программно задавать приращение x на единицу в каждом шаге. В результате x приобретает смысл дискретного времени, и временная зависимость будет совпадать с функциональной. В окне трассировки CoDeSys намеренно выполняет интерполяцию дискретных значений, имитируя непрерывность

времени. В результате график (рис. 9.13) приближается к картинке, которую можно получить на экране осциллографа, подключенного к аналоговым выходам реального ПЛК.

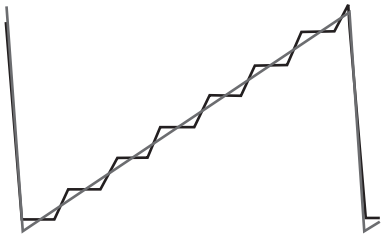

Рис. 9.13. График деления y := ISMDIV(x, 3), где x изменяется от −10 до +10

Кривые на рис. 9.13 отражают работу функции ISMDIV. Из графика видно, что операционная ошибка деления распределена симметрично относительно точных значений.

Реализованная функция ISMDIV выполняет симметричное округление с одним исключением. Если остаток составляет ровно половину от частного, симметрия нарушается. Для этого случая вы можете доработать функцию округления в соответствии с правилом Гаусса. Коррекция результата вверх при половинном остатке должна проводиться, только для четных чисел. Округление вверх и вниз в этом случае равновероятно, что и является задачей симметричного округления.

9.6. Генератор случайных чисел (FB ST)

Функциональный блок генерирует неповторяющиеся целые числа, принадлежащие интервалу [1..250], с равномерным распределением. Для формирования очередного числа применяется формула вычетов:

$$X_i = bX_{ic1} \text{ с } aE(b X_{ic1}/a),$$

где a и b — целые константы, E — функция, возвращающая целую часть аргумента. Очевидно, что при точном вычислении (без E) формула всегда должна давать 0. «Случайность» результата достигается исключительно за счет целочисленного округления [17]. После генерации определенного количества разных чисел последовательность в точности повторяется. Период повторе-

ния последовательности определяется выбором констант a и b. В нашем примере b является простым числом 251, константа a равна 170. Выбранные константы очень удачны. Период нашего генератора составляет 250. Фактически генератор перебирает почти все допустимые значения заданного интервала без повторов. Формируемая последовательность зависит от начального значения X. При одинаковых начальных значениях генерируемые последовательности совпадают. Начальное значение последовательности может быть произвольным из интервала от 1 до 250.

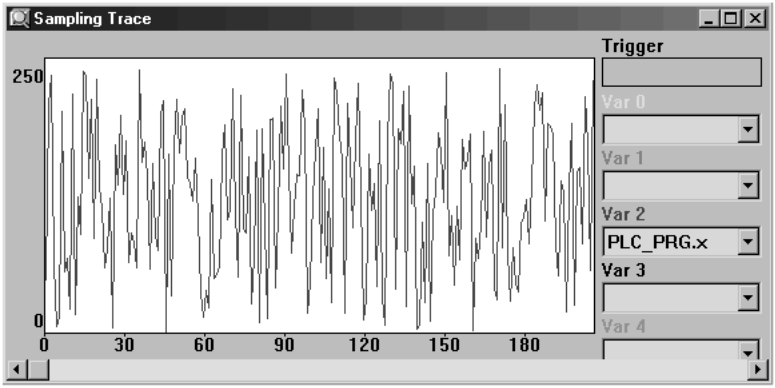

Рис. 9.14. Трассировка работы генератора случайных чисел

Функциональный блок имеет один выход x — генерируемое значение и два входа. Вход x0 — начальное значение. По фронту входа Load блок загружает начальное значение. Если начальное значение не задано, по умолчанию используется число 1.

```
FUNCTION_BLOCK Random
VAR_INPUT
   Load:    BOOL;
   x0:      USINT;
END_VAR
VAR_OUTPUT
   x:       USINT := 1;
END_VAR
VAR
   RESET: R_TRIG;
```

```
END_VAR

Reset(CLK := Load);
IF Reset.Q THEN x := x0; END_IF
x := x * 170 – 251 *(x * 170/251);
```

Трассировка 200 циклов работы генератора случайных чисел показана на рис. 9.14. На практике для формирования разных случайных последовательностей в качестве случайного начального числа можно использовать мгновенное значение нескольких младших разрядов системного таймера ПЛК.

9.7. Очередь FIFO (FB ST)

Блок FIFO реализует *очередь* — первым пришел, первым ушел. Работа блока проста. Если запись разрешена (bWREn = TRUE), то входной байт (byWR) помещается в конец очереди. Если чтение разрешено (bRDEn = TRUE), то блок отдает первый по очереди байт (byRD). Выход bReady говорит о наличии очереди. При попытке чтения из пустой очереди (bReady = FALSE) byRD должен быть равен 0.

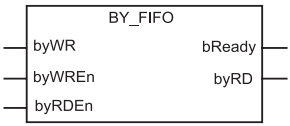

Рис. 9.15. Графическое представление экземпляра функционального блока FIFO

На рис. 9.16. показано применение блока FIFO для временного сдвига сигнала. Первые пять циклов чтение данных запрещено. Далее чтение разрешается, и в каждом цикле вызова блока происходит запись нового и чтение старого значения отсчета сигнала. Верхняя кривая — входной сигнал, нижняя кривая — сигнал, задержанный на 5 рабочих циклов ПЛК.

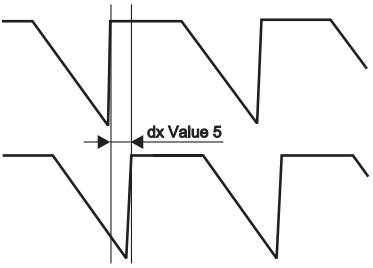

Рис. 9.16. Задержка сигнала блоком FIFO

Рассмотрим пример блока с размером очереди до 16 байт. Поскольку никаких преобразований данных в блоке не происходит, в него можно безопасно помещать любые однобайтные данные.

```
FUNCTION_BLOCK BY_FIFO
VAR_INPUT
   byWR:    BYTE;          (*записываемые данные*)
   bWREn:   BOOL;          (*разрешение записи*)
   bRDEn:   BOOL;          (*разрешение чтения*)
END_VAR
VAR_OUTPUT
   bReady:  BOOL;          (*очередь жива*)
   byRD:    BYTE;          (*читаемые данные*)
END_VAR
VAR
   byBuf:   ARRAY[0..15] OF BYTE;
   nWRPos:  WORD := 0;     (*позиция записи*)
   nRDPos:  WORD := 0;     (*позиция чтения*)
END_VAR

IF bWREn THEN              (*запись в FIFO*)
   byBuf[nWRPos] := byWR;

   IF nWRPos > 14 THEN     (*WRPos движется по кольцу*)
         nWRPos := 0;
   ELSE
         nWRPos := nWRPos + 1;
   END_IF
END_IF
IF nWRPos = nRDPos THEN    (*FIFO пуст?*)
   bReady :=-FALSE;
   byRD   := 0;
ELSE
   bReady := TRUE;
```

```
    IF bRDEn THEN                    (*чтение из FIFO*)
        byRD := byBuf[nRDPos];

        IF nRDPos > 14 THEN          (*RDPos движется по кольцу*)
                nRDPos := 0;
        ELSE
                nRDPos := nRDPos + 1;
        END_IF
    END_IF
END_IF
```

Алгоритм опирается на два кольцевых индекса, указывающих позиции чтения и записи. При достижении верхней границы буферного массива индекс обнуляется. Запись не должна обгонять чтение. В этом случае очередь будет перезаписана. Корректная обработка такой ситуации в данном примере не предусмотрена.

9.8. Быстрая очередь FIFO (FB ST)

Данный текст реализации мог быть написан только полным «чайником», тем не менее он работает лучше приведенного выше примера. Быстрота достигается за счет того, что индексы позиций записи и чтения не имеют проверки верхней границы. Фактически текст содержит только запись и чтение, никаких вспомогательных операций. Трюк состоит в использовании для индексов беззнакового целого типа и достаточно большого массива. Поскольку BYTE#255 + BYTE#1 = 0, все получается автоматически. Раздел объявлений входов и выходов аналогичен примеру байтового FIFO-буфера. Локальные переменные должны объявляться так:

```
VAR
    byBuf:    ARRAY[0..255] OF BYTE;
    nWRPos:   BYTE := 0;          (*позиция записи*)
    nRDPos:   BYTE := 0;          (*позиция чтения*)
END_VAR
```

Выход готовности очереди ликвидирован в жертву быстродействию. О корректности применения блока должна заботиться вызывающая программа.

```
IF bWREn THEN                    (*запись в FIFO*)
    byBuf[nWRPos] := byWR;
```

```
    nWRPos := nWRPos + 1;
END_IF

IF bRDEn THEN                    (*чтение из FIFO*)
    byRD := byBuf[nRDPos];
    nRDPos := nRDPos + 1;
END_IF
```

Дальнейшего ускорения можно достичь, если выбросить входы разрешения bWREn и bRDEn, а запись и чтение выполнить отдельными действиями.

9.9. Фильтр «скользящее среднее» (FB ST)

Фильтр «скользящее среднее» (рис. 9.17) является простейшим в реализации цифровым фильтром. Тем не менее он дает очень хорошие результаты при подавлении шумов и высокочастотных помех.

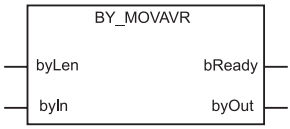

Рис. 9.17. Графическое представление экземпляра функционального блока Move Average

Алгоритм работы фильтра основан на усреднении нескольких последних значений сигнала. Математически это выглядит так:

$$y[i] = \frac{1}{N} \sum_{j=0}^{N \text{ c } 1} x[i \text{ c } j],$$

где $x[]$ — входной сигнал, $y[]$ — выходной сигнал, N — ширина фильтра или число усредняемых замеров.

Формула предполагает, что замеры сигнала собраны в массиве. При вычислении значения $y[i]$ используется $x[\text{i}]$ и $N-1$ предыдущих значений. Очевидно, при обработке сигнала в режиме реального времени, выходной сигнал будет запаздывать по отношению к входу.

Рисунок 9.18 демонстрирует действие фильтра. Верхняя кривая — входной сигнал. Это прямоугольные импульсы, модулированные по амплитуде гармонической помехой, с периодом в два

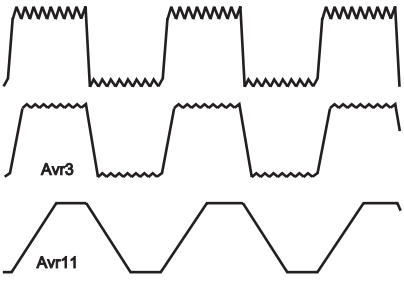

Рис. 9.18. Фильтр «скользящее среднее»

цикла. Средняя и нижняя кривая — это результаты обработки входного сигнала фильтром «скользящее среднее», шириной 3 и 11 циклов:

Формула требует только $N - 1$ последних значений, более старые значения интереса не представляют. Поэтому для реализации алгоритма оказывается подходящим буфер типа FIFO. Если фильтр широкий, то вычисления суммы в каждом цикле будут отнимать много времени. К счастью, это легко преодолеть, если использовать предыдущее значение суммы. Достаточно вычесть из старой суммы лишнее значение $x[j - N]$ и прибавить новое $x[j]$, все промежуточные значения уже содержатся в сумме. Такой алгоритм называется рекурсивным.

Функциональный блок фильтра имеет два входа: byIn — входной сигнал, byLen — ширина фильтра и два выхода: byOut — входной сигнал, bReady — готовность фильтра. Готовность появляется после накопления первоначальной суммы, необходимой для вывода фильтра в рабочий режим. Блок фильтра использует FIFO из примера, приведенного выше. Раздел объявлений выглядит так:

```
FUNCTION_BLOCK BY_MOVAVR
VAR_INPUT
   byLen:  BYTE;
   byIn:   BYTE;
END_VAR
VAR_OUTPUT
   bReady: BOOL := FALSE;
   byOut:  BYTE;
END_VAR
VAR
   byShift: BYTE := 0;
   byFIFO: BY_FIFO := (bWREn := TRUE , bRDEn := FALSE);
   wSum:   WORD;
END_VAR
```

Обратите внимание, что при 8-разрядном входном сигнале для суммы используется переменная типа **WORD**. Ширина фильтра не должна превышать 257 исходя из максимально возможного значения **BYTE** входа (255 * 257 = 65535). В противном случае может произойти переполнение суммы. В нашем примере ширина фильтра ограничивается размером FIFO.

```
IF byShift < byLen THEN
(*разогрев — начальный набор суммы*)
    byFIFO(byWR := byIn);
    wSum  := wSum  + byIn;
    byShift := byShift + 1;

   IF byShift = byLen THEN
(*очередь собралась достаточная*)
        bReady := TRUE;
        byFIFO.bRDEn := TRUE;
    END_IF
ELSE                                (*полет нормальный*)
    byFIFO(byWR := byIn);
    wSum := wSum − byFIFO.byRD + byIn;
END_IF
byOut := WORD_TO_BYTE(wSum / byShift);
```

Проверка byShift = byLen введена для оптимизации. Благодаря этому разрешение чтения из FIFO и установка готовности выполняется один раз. В рабочем режиме время на это не тратится. До накопления полной очереди усредняется то, что есть. Благодаря этому фильтр включается плавно.

9.10. Медианный фильтр (FB ST)

Медианный фильтр по алгоритму реализации похож на фильтр «скользящее среднее», в нем также используется усреднение нескольких последних значений сигнала. Но выходом фильтра является не само усредненное значение, а ближайшее к среднему входное значение. Таким образом, фильтр выбирает наиболее «подходящее» значение из нескольких последних. Медианный фильтр не формирует плавные изменения и используется в случаях, когда важно сохранить крутизну фронтов сигнала.

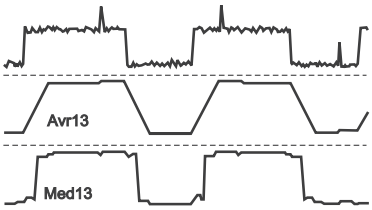

Рис. 9.19. Усредняющий и медианный фильтры

Рисунок 9.19 демонстрирует действие фильтра. Верхняя кривая — входной сигнал. Это прямоугольные импульсы, искаженные импульсной помехой и шумом. Модель шума получена с применением функционального блока «генератор случайных чисел». Средняя кривая — это результаты обработки входного сигнала фильтром «скользящее среднее», шириной 13 циклов. Нижняя кривая — это выход медианного фильтра аналогичной ширины.

Как видно из рисунка, медианный фильтр хорошо подавляет короткие (в сравнении с шириной фильтра) импульсы, но плохо справляется с шумами. Усредняющий фильтр хорошо работает в обоих случаях, но существенно заваливает фронты сигнала. Заметьте, что в отличие от скользящего среднего медианный фильтр является нелинейным, т. е. Med(x + y) <> Med(x) + Med(y).

Для реализации данного фильтра придется несколько доработать блок FIFO. Нужно «научить» его считывать элементы очереди без удаления. Доработка не влияет на базовые операции чтения и записи. В разделе локальных переменных блока нужно добавить объявление:

```
wNextPos: WORD;
```

К реализации блока добавьте 2 действия. Действие **ClrPos** устанавливает позицию просмотра очереди на позицию чтения:

```
wNextPos := wRDPos;
```

Действие **GetNext** читает очередной элемент очереди без удаления:

```
byRD := byBuf[wNextPos];
IF wNextPos > 14 THEN    (*NextPos движется по кольцу*)
   wNextPos := 0;
ELSE
   wNextPos := wNextPos + 1;
END_IF
```

Текст функционального блока «медианный фильтр» выглядит так:

```
FUNCTION_BLOCK BY_MEDIAN
VAR_INPUT
   byLen:   BYTE;
   byIn:    BYTE;
END_VAR
VAR_OUTPUT
   bReady: BOOL := FALSE;
   byOut:   BYTE;
END_VAR
VAR
   byShift: BYTE := 0;
   byFIFO: BY_FIFO := (bWREn:=TRUE , bRDEn:=FALSE);
   wSum:    WORD;              (*скользящая сумма*)
   cby,                        (*счетчик цикла*)
   byX,                        (*значение из очереди*)
   byAvr,                      (*скользящее среднее*)
   byDifference,               (* минимальная абс.
                                  разность*)
   byXdif:  BYTE;              (*абс. разность*)
END_VAR

IF byShift < byLen THEN        (*прогрев — начальный набор
                                  суммы*)
   wSum := wSum + byIn;
   byFIFO(byWR :=byIn);
   byShift := byShift + 1;
   IF byShift = byLen THEN     (*трогаемся потихоньку*)
      bReady := TRUE;
      byFIFO.bRDEn := TRUE;
   END_IF
ELSE                           (*едем*)
   byFIFO(byWR := byIn);
   wSum := wSum − byFIFO.byRD + byIn;
```

```
    byAvr := WORD_TO_BYTE(wSum / byLen);

    byFIFO.ClrPos;                    (*поиск медианы*)
    byDifference := 255;

    FOR cby := 1 TO byLen DO
        byFIFO.GetNext;
        byX := byFIFO.byRD;
        byXdif := MAX(byX,byAvr) − MIN(byX,byAvr);
        IF byXdif < byDifference THEN
            byDifference := byXdif;
            byOut := byX;
        END_IF
    END_FOR
END_IF
```

Вычисление среднего значения здесь ничем не отличается от реализации, описанной для фильтра «скользящее среднее». Далее в очереди просматриваются последовательно byLen значений. Для каждого кандидата byX оценивается абсолютное значение его отличия от среднего byXdif. Из просмотренного хвоста очереди выбирается ближайший к среднему элемент. Алгоритм реализован «в лоб» без оптимизации.

Обратите внимание, что цифровые фильтры можно использовать только при наличии аналогового фильтра на входе АЦП. Высокочастотные составляющие сигнала, для которых не выполняются условия теоремы Котельникова, не ослабляются цифровым фильтром. Хуже того, цифровой фильтр выделит некоторое низкочастотное колебание, не имеющее ничего общего с исходным сигналом. Данный эффект называется перекрытием спектров (подробнее см. [13], [20]). Эффект представлен на рис. 9.20. Верхняя кривая — периодический сигнал с периодом в 8 раз короче цикла работы фильтров. Две нижние кривые — выходы усредняющего и медианного цифровых фильтров.

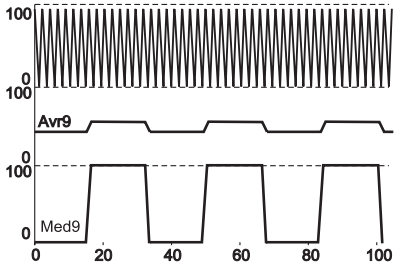

Рис. 9.20. Перекрытие спектров

9.11. Линеаризация измерений (PRG ST)

Данный пример демонстрирует методику применения функционального блока CHARCURVE.

Рассмотрим для конкретности практическую задачу. Необходимо измерить количество топлива в баке емкостью 220 л. Бак имеет неправильную геометрическую форму. Потенциометрический датчик подключен к аналоговому десятиразрядному входу ПЛК. Необходимо получить значения в литрах, относительная погрешность алгоритма уровня не должна превышать 5%.

Определить аналитическое выражение в данном случае слишком сложно. Будем использовать зависимость, снятую опытным путем. Для этого можно применить ПЛК и CoDeSys в режиме мониторинга входных переменных. Будем считать, что датчик уровня уже подключен к контроллеру.

Для обработки исходных данных и оптимального выбора вектора узловых точек в данной задаче удобно использовать математические комплексы [11] либо специализированные научные калькуляторы [12]. Но давайте умышленно ограничимся более доступным офисным инструментом — Microsoft Excel.

В первую очередь нужно снятые экспериментально данные, занести в таблицу Excel и провести сглаживание (например, методом скользящего среднего). На рис. 9.21 показана сглаженная зависимость в форме таблицы и графика. Это стандартная точечная диаграмма со сглаживающими линиями (сплайны).

Таблица узловых точек получается следующим образом:

- первая и точка берется из исходной таблицы, для нулевого аргумента;
- далее нужно определить несколько (5 — 10) точек по графику «на глаз». Это нужно для того, чтобы не переопределять ряд отображаемых Excel данных всякий раз при подгонке очередной точки;
- теперь строим точечную диаграмму для полученной таблицы. Это должен быть второй ряд данных в том же окне диаграммы, стандартная точечная диаграмма без сглаживания;
- для исходного ряда данных нужно включить режим отображения вертикальных планок относительной погрешности по Y (5%);
- далее последовательно корректируем узловые точки так, чтобы отрезки вписывались в полосу, ограниченную планками погрешности. При необходимости добавляем дополни-

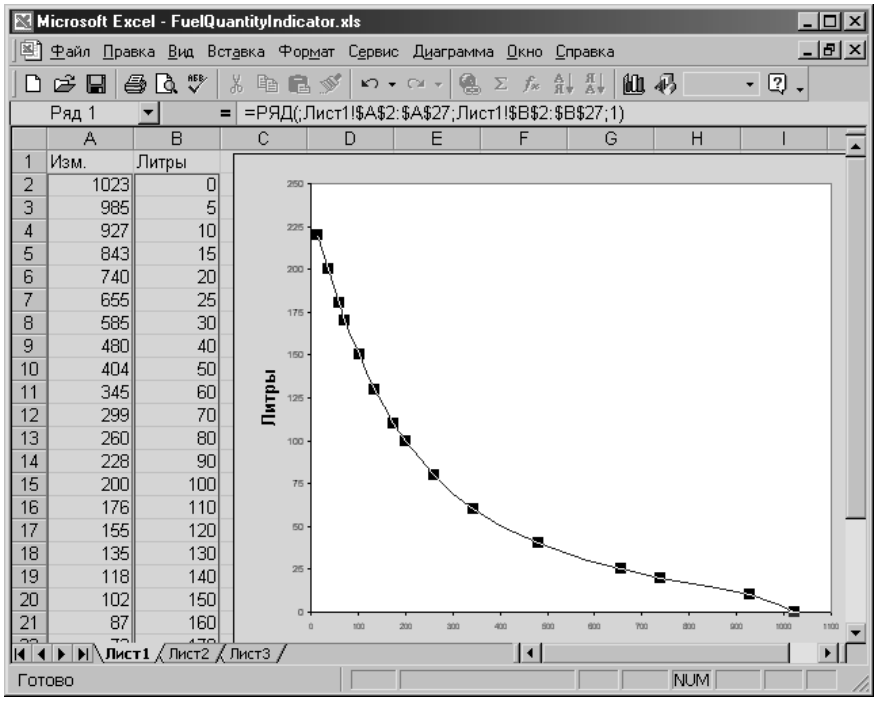

Рис. 9.21. График зависимости объема топлива от значений АЦП

тельные точки. Окно диаграммы необходимо растянуть до удобного масштаба.

Увеличенный фрагмент графической диаграммы, демонстрирующий описанный процесс, показан на рис. 9.22.

Полученные результаты необходимо проверить в CoDeSys. Для этого достаточно использовать режим эмуляции ПЛК. Тестовая программа на языке ST будет выглядеть так:

```
PROGRAM PLC_PRG
VAR
   curvLIN: CHARCURVE;
   x,y: INT;
   BP:ARRAY[0..10] OF POINT :=
         (X := 0,Y := 220),(X := 10,Y := 220),
         (X := 160,Y := 112),(X := 300,Y := 68),
         (X := 480,Y := 39),(X := 700,Y := 22),
```

```
            (X := 960,Y := 8),(X := 1023,Y := 0);
END_VAR

IF x > 999 THEN
   x := 0;
ELSE
   x := x+5;
END_IF
curvLIN(IN := x, N := 8, P := BP, OUT => y);
```

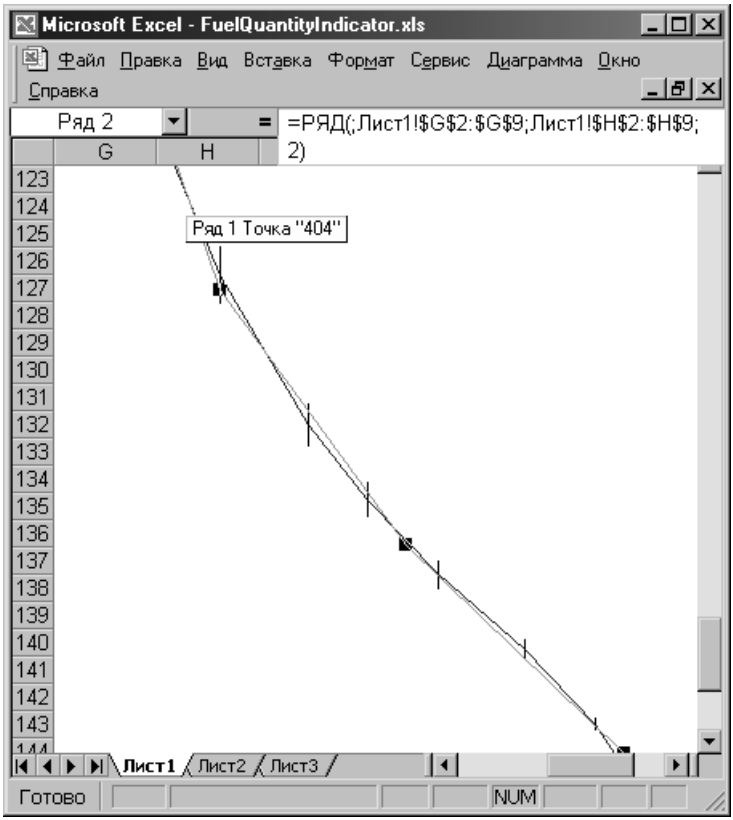

Рис. 9.22. Подбор узловых точек

Результат трассировки тестовой программы показан на рис. 9.23. По оси *X* здесь указаны циклы работы программы,

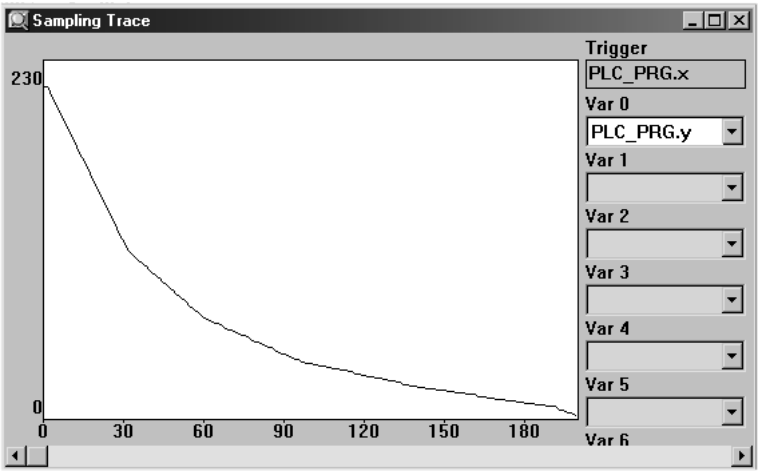

Рис. 9.23. Трассировка зависимости

один цикл увеличивает аргумент на 5 (x := x + 5). Вид полученного графика соответствует исходной зависимости (рис. 9.21). Это свидетельствует об отсутствии принципиальных ошибок в решении.

Использованный здесь графический метод подбора значений, конечно, нагляден, но не дает гарантии достижения заданной точности. При необходимости вы можете записать в Excel формулу кусочно-линейной интерполяции (см. описание CHARCURVE) и просчитать значения погрешности во всей области определения. Для 10 разрядного АЦП это интервал [0—1023].

9.12. Широтно-импульсный модулятор на базе таймера (FB IL)

Блок широтно-импульсного модулятора (ШИМ) генерирует импульсы, коэффициент заполнения которых можно изменять. Блок можно применить для аналогового управления с использованием дискретного выхода. Время реакции объекта управления должно быть значительно больше периода ШИМ-импульсов. Это может быть, например, электрический нагреватель, лампа накаливания, двигатель постоянного тока, стрелочный индикатор и т. д.

В нашем примере период импульсов равен 100 мс. Коэффициент заполнения byRate лежит в пределах от 0 до 100%. На

рис. 9.24 показаны формируемые блоком ШИМ импульсы при byRate равном 30, 50 и 70.

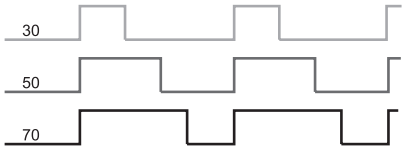

Рис. 9.24. Выходы широтно-импульсного модулятора

Функциональный блок построен на базе стандартного таймера TP. Входной параметр типа **SINT** преобразуется в **TIME**. При использовании периода равного 100 длительность импульса равна byRate, длительность паузы равна 100 – byRate.

```
FUNCTION_BLOCK  PWMTP
VAR_INPUT
   byRate:     USINT;
END_VAR
VAR_OUTPUT
   bOUT:       BOOL;
END_VAR
VAR
   PERIOD:     TP;
   tPulse:     TIME;
   tPause:     TIME;
END_VAR

   CAL         PERIOD
   LD          PERIOD.Q
   EQ          FALSE
   RETCN                    (*нет изменений*)

   LD          bOUT         (*кончился импульс или  пауза?*)
   EQ          TRUE
   JMPC        pause
   LD          TRUE         (*импульс*)
   ST          bOUT

   LD          byRate
```

```
    SINT_TO_TIME
    ST          PERIOD.PT
    JMP         oscillation
pause:
    LD          FALSE           (*пауза*)
    ST          bOUT
    LD          100
    SUB         byRate
    SINT_TO_TIME
    ST          PERIOD.PT
oscillation:
    CAL   PERIOD(IN := FALSE)  (*для запуска таймера*)
    CAL   PERIOD(IN := TRUE)   (*нужен импульс*)
```

9.13. Управление реверсивным приводом (FB SFC)

На рис. 9.25 показана схема реверсивного электропривода двигателя постоянного тока без обратной связи с управлением разгоном и торможением по времени. Схема содержит следующие элементы управления:

- **Pwr —** силовой пускатель, подает напряжение питания;
- **Rew —** блок реверса. Если блок реверса включен, то провода питания соединяются перекрестно, обеспечивая изменение полярности. Изменять направление вращения можно только при остановленном двигателе и выключенном питании Pwr;
- **Start —** цепь разгона. Обеспечивает плавный старт двигателя без перегрузки. Включается на заданное время при пуске;
- **Break —** блок торможения. Подключает нагрузку к вращающемуся в режиме генератора двигателю, обеспечивая электромагнитное торможение. Включается на заданное время после отключения питания. При включенном питании включать торможение нельзя.

Как видно из описания, алгоритм управления получается не слишком простым. Задача заключается в том, чтобы создать функциональный блок, имеющий два входа — включить привод (On) и реверс (Direction). Переключать входы можно в любое время и в любом порядке. Входы tStart и tBrake задают время разгона и

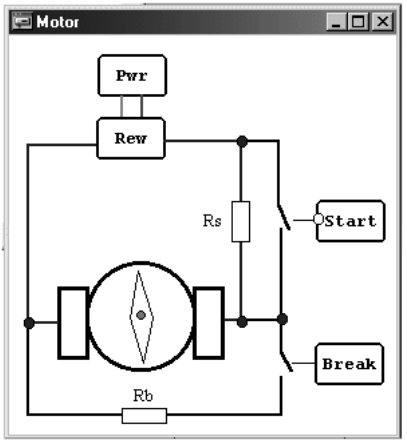

Рис. 9.25. Визуализация схемы привода в CoDeSys

торможения соответственно (по умолчанию 5 секунд). Блок имеет 4 выхода, управляемых вышеописанными силовыми элементами.

Функциональный блок ReversibleEngine реализован в виде SFC-диаграммы (рис. 9.26). Шаги и условия переходов реализованы на ST.

```
FUNCTION_BLOCK ReversibleEngine
VAR_INPUT
   Direction, On: BOOL;
   tStart:        TIME := t#5s;      (*время разгона*)
   tBrake:        TIME := t#5s;      (*время торможения*)
END_VAR
VAR_OUTPUT
   Power,                            (*включение*)
   Reversal,                         (*реверс*)
   Starting,                         (*разгон*)
   Braking:       BOOL;              (*тормоз*)
END_VAR
VAR
   Tm:            TON;          (*таймер разгона и тормоза*)
END_VAR
```

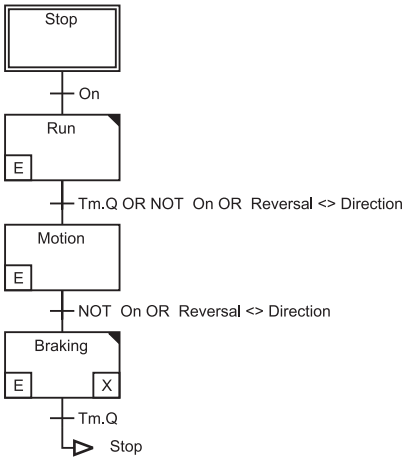

Рис. 9.26. SFC-диаграмма блока управления реверсивным приводом

Стоп

Начальный шаг (Stop) не делает ничего, точнее говоря, ждет включения On.

```
TRANSITION FROM Stop TO Run:= On
END_TRANSITION
```

Разгон

Шаг, выполняющий режим разгона (Run), должен быть активен не более времени tStart. Если во время разгона произойдет выключение или изменение направления, шаг должен закончиться досрочно. Таким образом, формируется условие перехода:

```
TRANSITION FROM Run TO Motion:=
        Tm.Q OR NOT On OR Reversal <> Direction
END_TRANSITION
```

При активации шага взводится таймер, включается выход стартового режима, общее питание и выход реверса в зависимости от нужного направления. Вся работа во время активности шага сводится к вызову экземпляра таймера:

```
STEP Run:           (*разгон*)
        Tm;
END_STEP
```

```
ENTRY_ACTION
      Tm(IN := FALSE, PT:= tStart);      (*запуск таймера*)
      Tm.IN          := TRUE;
      Starting       := TRUE;
      Power          := TRUE;
      Reversal       := Direction;
END_ACTION
```

Движение

Шаг, отвечающий за режим движения (Motion), активен, пока не произойдет выключение или изменение направления. Условие перехода от движения к торможению:

```
TRANSITION FROM Motion TO Braking:=
      NOT On OR Reversal <> Direction
END_TRANSITION
```

Если шаг разгона был прерван досрочно, то данное условие перехода также будет ИСТИНА, двигатель перейдет к отработке торможения.

При активации шага нужно выключить разгон. Во время движения, кроме ожидания условий перехода, ничего делать не требуется.

```
STEP Motion:                         (*режим*)
END_STEP

ENTRY_ACTION
      Starting := FALSE;
END_ACTION
```

Торможение

Торможение всегда отрабатывает заданное время по таймеру и переходит к начальному шагу:

```
TRANSITION FROM Braking TO Stop:=
      Tm.Q
END_TRANSITION
```

Начальное действие шага взводит таймер, отключает питание и включает торможение. Во время отработки торможения никаких действий, кроме контроля времени, делать ненужно. Завер-

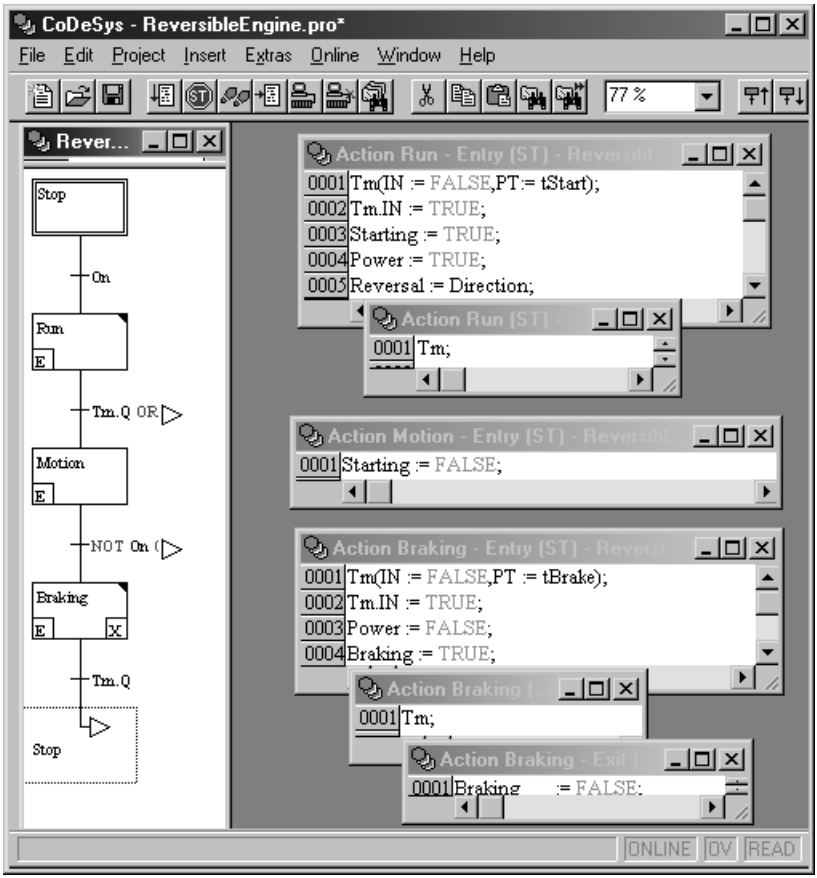

Рис. 9.27. Полное представление функционального блока ReversibleEngine

шается шаг отключением торможения. По окончании шага все выходы выключены, что соответствует начальному состоянию.

```
STEP Braking: (*торможение*)
        Tm;
END_STEP

ENTRY_ACTION
        Tm(IN    := FALSE, PT := tBrake);
        Tm.IN    := TRUE;
        Power    := FALSE;
```

```
        Braking := TRUE;
END_ACTION

EXIT_ACTION
        Braking := FALSE;
END_ACTION
```

Благодаря применению входных и выходных действий алгоритм работы блока получился достаточно эффективным. Кроме контроля таймера, шаги «Разгон» и «Торможение» не содержат никаких действий, повторяющихся в каждом рабочем цикле. Продолжительные шаги «Стоп» и «Движение» заняты только проверкой условий перехода.

Печатное представление действий SFC-диаграммы выглядит несколько громоздко. В среде CoDeSys каждое действие отражается в отдельном окне без ключевых слов (END_ACTION, EXIT_ACTION). Помимо этого, благодаря масштабированию диаграммы и выбору необходимого расположения окон анализировать работу диаграммы достаточно удобно (рис 9.27).

9.14. Сравнение языков с позиции минимизации кода (IL, ST, FBD, LD)

Какой из языков программирования МЭК позволяет создавать наиболее компактный код? Ответ на этот вопрос достаточно прост. Размер кода не зависит от языка реализации.

Перевод программы с одного языка на другой не имеет никакого смысла с точки зрения оптимизации. Применение меток и переходов в графических языках дает возможность формального (машинного) перевода разветвленных ST- и IL-программ в графические языки. Конечно, такая графическая диаграмма более напоминает IL, где текстовые инструкции заменены картинками. Код, генерируемый такими программами, совпадает до бита.

Единственным исключением является SFC. Проверка условий переходов, управление активностью шагов в общем случае требует достаточного развитого механизма управления. Поэтому код с применением SFC всегда имеет определенные издержки. Но в этом нет ничего удивительного. Язык SFC предназначен для быстрой разработки верхнего уровня. Дополнительный код является платой за универсальность, наглядность и быстроту реализации проекта.

Более компактного кода можно достичь только за счет ручной переработки алгоритма. Правильное использование разных языков МЭК требует разного подхода и разного образа мыслей. В результате эффективность кода зависит исключительно от качества проработки и от степени владения программистом данным языком.

Безусловно, размер кода — это самое последнее требование к ПЛК-программе. Необходимость минимизации может быть вызвана недостатком памяти программ ПЛК. Кроме того, переработка алгоритма, направленная на уменьшение кода, как правило, приводит к поиску оптимального алгоритма и соответственно к увеличению производительности системы.

Вышеизложенные утверждения не сложно проверить на практике. Давайте сделаем это на базе примера «Управление реверсивным приводом». Попробуем реализовать пример на разных языках и разными способами без применения SFC.

Все примеры реализованы в CoDeSys (версия 2.2) и откомпилированы для двух широко распространенных микропроцессорных платформ: 8-разрядное семейство Intel 8051 и 16-разрядное Infineon 16x. Приведенные данные о размере кода (в байтах) включают только размер функционального блока и его модуля начальной инициализации переменных. Код тестовой программы, вспомогательных функций генератора кода и библиотек не учитывался.

9.14.1. Программирование последовательности состояний (ST, IL)

При реализации алгоритмов, базирующихся на последовательных состояниях в универсальных языках (C, ассемблер), приходится строить селектор состояний. Во многих случаях достаточно иметь одну переменную, кодирующую состояние программного модуля. Шаг, соответствующий определенному состоянию, должен включать проверку условия перехода и соответствующее изменение селекторной переменной. Одновременно можно выполнить выходное действие, если оно требуется. Входные действия (терминология SFC) потребуют отдельного состояния с безусловным переходом к основному шагу. При необходимости перехода к произвольному шагу селекторной переменной нужно присвоить необходимое значение. Если переход приводит к следующему шагу, то достаточно увеличить селекторную переменную на единицу.

Псевдокод программного компонента, реализующего (язык ST) такой алгоритм, будет выглядеть так:

```
VAR
   state := 0;                    (*начальное состояние*)
END_VAR

CASE state OF
   0:      действие;

           если условие перехода не истина, то закончить
           возврат (return) ;
           выходное действие;

   1:      входное действие шага 2;

   2:      действие;

           если условие ветвления истина, state = 'номер
           шага', возврат;

           если условие перехода не истина, то закончить
           возврат;
           выходное действие;
   … … …
ELSE
           действие по умолчанию;
           state := 0;
           возврат;
END_CASE

Увеличить state на 1;             (*выполняется, если не было
                                  возврата*)
```

Используя описанный метод, переведем функциональный блок Reversible Engine полностью на язык ST. Безусловно, в таком преобразовании нет иного практического смысла, кроме нашего «спортивного интереса» — достижения минимального кода.

Раздел объявлений переменных необходимо дополнить одной локальной переменной State целого типа. Пронумеруем по поряд-

ку все действия диаграммы. Переменная State будет отражать номер активного действия. Алгоритм блока не изменится, если входные действия шагов выполнять в предыдущем цикле, что дает сокращение необходимых состояний. Соответствие ST текста и SFC-действий указано в комментариях.

```
CASE State OF
1: (*STEP Run*)
   Tm;
   IF Tm.Q = FALSE AND On AND Reversal = Direction
   THEN RETURN; END_IF

   (*EXIT Run*)
   Starting := FALSE;

2: (* STEP Motion*)
   IF  On = TRUE AND  Reversal =  Direction THEN
   RETURN; END_IF


   (*EXIT Motion*)
   Power    := FALSE;
   Braking := TRUE;
   Tm(IN := FALSE, PT := tBrake);
   Tm.IN := TRUE;

3: (*5: STEP Braking*)
   Tm;
   IF NOT Tm.Q THEN RETURN; END_IF

   (*EXIT Braking*)
   Braking := FALSE;
ELSE
   State     := 0;      (*На всякий случай*)

   (*STEP Stop*)
   IF NOT On THEN RETURN; END_IF

   (*EXIT Stop*)
   Tm(IN := FALSE, PT:= tStart);
   Tm.IN := TRUE;
   Starting := TRUE;
```

```
    Power    := TRUE;
    Reversal := Direction;
END_CASE

State := State +1;
```

Шаг «стоп» является шагом по умолчанию и соответствует нулевому значению селекторной переменной. Выражение State := 0; не является обязательным, но придает блоку устойчивость при случайных изменениях селекторной переменной. Другим способом защиты может служить применение переменной с ограниченным диапазоном или перечисление. Данная реализация функционального блока работает совершенно аналогично SFC, что несложно проверить в режиме эмуляции ПЛК CoDeSys.

Переведем теперь функциональный блок Reversible Engine ST на язык IL. Перевод выполнен вручную с незначительной очевидной оптимизацией.

```
    LD    State          (*Селектор состояний State*)
    EQ    1
    JMPC  case1
    LD    State
    EQ    2
    JMPC  case2
    LD    State
    EQ    3
    JMPC  case3
    LD    0              (*На всякий случай*)
    ST    State

case0:  (*STEP Stop*)
    LDN   On             (*IF NOT On THEN RETURN*)
    RETC
          (*EXIT Stop*)
    CAL   Tm(IN := FALSE, PT := tStart)
    LD    Direction
    ST    Reversal       (*Reversal:= Direction;*)
    LD    TRUE
    ST    Starting       (*Starting := TRUE;*)
```

```
    ST      Power           (*Power := TRUE;*)
    ST      Tm.IN           (*Tm.IN := TRUE*)
    JMP     NextState
case1:      (*STEP Run*)
    CAL     Tm
    LDN     Tm.Q            (*IF Tm.Q = FALSE AND On  AND
                             Reversal = Direction THEN RETURN*)
    AND     On
    AND     ( Reversal
    EQ      Direction
    )
    RETC
            (*EXIT Run*)
    LD      FALSE
    ST      Starting        (*Starting := FALSE;*)
    JMP     NextState
case2:      (*STEP Motion*)
    LD      On              (*IF  On = TRUE AND  Reversal =
                             Direction THEN RETURN*)
    AND     ( Reversal
    EQ      Direction
    )
    RETC
            (*EXIT Motion*)
    CAL     Tm(IN := FALSE, PT := tBrake)
    LD      TRUE
    ST      Tm.IN
    ST      Braking
    LD      FALSE
    ST      Power
    JMP     NextState
case3:      (*STEP Braking*)
    CAL     Tm
    LDN     Tm.Q
    RETC
```

```
                (*EXIT Braking*)
    ST          Braking             (*A=FALSE*)

NextState:                          (*State := State +1;*)
    LD          State
    ADD         1
    ST          State
```

На рис. 9.28 показана сравнительная диаграмма размера кода функционального блока Reversible Engine исходной SFC-реализации и переведенных ST- и IL-реализаций.

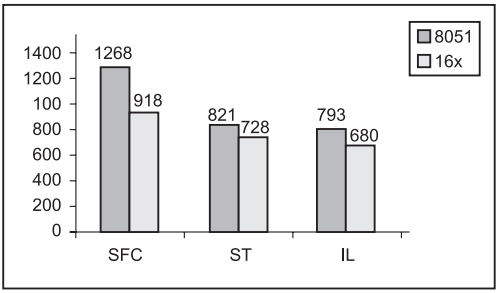

Рис. 9.28. Размер кода Reversible Engine в последовательной реализации

Уменьшение размера кода IL по сравнению с ST достигнуто за счет оптимизации селектора переходов и операции присваивания логических значений. (Оператор LD TRUE и за ним несколько операций присваивания. В языке C существует аналогичная конструкция: Starting = Power = Tm.IN = TRUE. В ST так делать нельзя.)

Непосредственный перевод данного алгоритма в LD FBD приводит к громоздким, некрасивым диаграммам, которые компилируются в аналогичный код.

9.14.2. Параллельное решение в виде логических выражений (FBD, LD, ST, IL)

Для создания «правильной» FBD-диаграммы необходимо обратиться к первоисточнику — условиям задачи и реализовать ее заново, с чистого листа.

Раздел объявлений входных и выходных переменных определяет интерфейс функционального блока и неизменен для всех реализаций. Объявление локальных переменных должно выглядеть так:

```
VAR
    TmRun:    TON;
    TmBrake:  TOF;
END_VAR
```

Будем использовать 2 независимых таймера: TmRun — для разгона и TmBrake — для торможения (рис. 9.29).

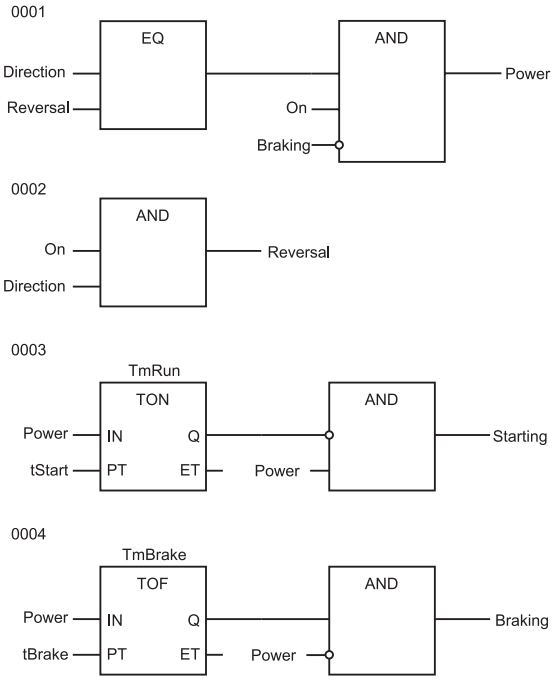

Рис. 9.29. Reversible Engine в FBD

Первая цепь определяет условие включения выхода питания Power. Питание подано, если направление не изменялось (Reversal = Direction), присутствует сигнал включения (On) и торможение закончено (NOT Braking).

Вторая цепь переключает выход реверса при отключенном питании. За отключением питания при изменении направления следит цепь 1.

Цепь 3 запускает таймер разгона по фронту питания. Если питание выключается, разгон будет также выключен.

Цепь 4 запускает таймер торможения по заднему фронту (выключению) питания. Элемент AND предотвращает возможность включения тормоза при включенном питании.

Фактически диаграмма дословно выполняет условия задачи. Метки и переходы не применяются, схема имеет очень компактное представление.

Данную FBD-схему несложно преобразовать в LD. Для того чтобы LD-схема была более похожей на действительно релейную диаграмму, желательно не использовать функциональный блок сравнения (рис. 9.30). Хотя это, конечно, приведет к увеличению кода.

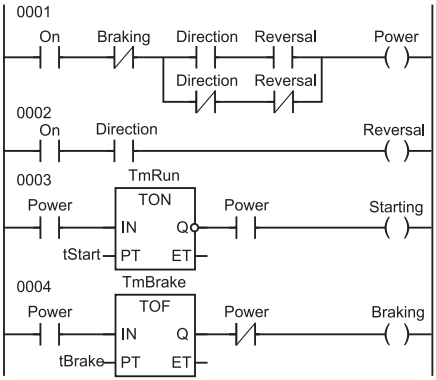

Рис. 9.30. Reversible Engine в LD

Также несложно преобразовать FBD-диаграмму в уравнения, записанные на языке ST.

```
Power := (Direction = Reversal) AND On AND NOT Braking;
Reversal := On AND Direction;
TmRun(IN := Power, PT := tStart);
Starting :=  NOT TmRun.Q AND Power;
TmBrake(IN := Power, PT := tBrake);
Braking := TmBrake.Q AND NOT Power;
```

На языке IL функциональный блок будет выглядеть так:

```
LD      Direction
EQ      Reversal
AND     On
ANDN    Braking
ST      Power

LD      On
AND     Direction
ST      Reversal

CAL     TmRun(IN := Power, PT := tStart)
LD      TmRun.Q
NOT
AND     Power
ST      Starting

CAL     TmBrake(IN := Power, PT := tBrake)
LD      TmBrake.Q
ANDN    Power
ST      Braking
```

Результаты трансляции (рис. 9.31) подтверждают приведенные выше утверждения о том, что размер кода не зависит от языка реализации.

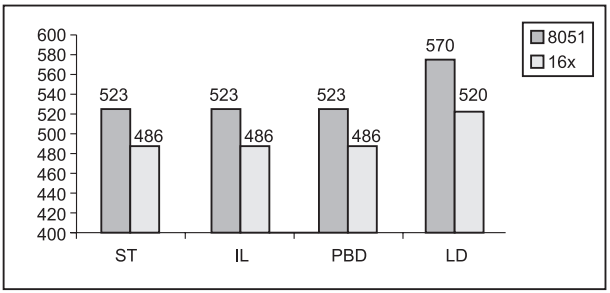

Рис. 9.31. Размер кода Reversible Engine в параллельной реализации

9.14.3. Функциональный блок против программы

Из приведенных выше диаграмм видно, что 16-разрядный код оказывается компактнее 8-разрядного. Объясняется это особенностями компиляции функциональных блоков. Функциональный блок может иметь экземпляры, т. е. с точки зрения процессора разные данные для одного и того же кода. При вызове экземпляр функционального блока получает через стек единственный параметр — адрес размещения в памяти «своих» данных. В результате компилятор вынужден генерировать косвенные обращения к данным. Адреса области переменных в 8051 16-разрядные, что и вызывает рост кода при динамическом вычислении адресов переменных.

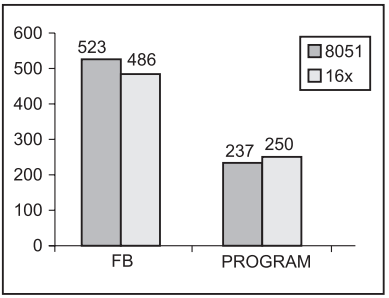

Рис. 9.32. Размер кода Reversible Engine для функционального блока и программы

В CoDeSys программы не могут создавать экземпляры, благодаря чему адреса данных программ известны уже в процессе трансляции и появляется возможность отказаться от косвенной адресации. В результате преобразование функционального блока в программу дает достаточно существенную экономию кода (рис. 9.32).

Список литературы

1. Lewis R.W., Programming industrial control systems using IEC 1131-3/ Revised ed. The Institution of Electrical Engineers. London, United Kingdom, 1998.

2. John Karl-Heinz, Tiegelkamp M. IEC 61131-3: Programming Industrial Automation Systems. Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Tools. Springer-Verlag Berlin Heidelberg, 2001.

3. Monari P.D., Bonfatti F., Sampieri U., IEC 1131-3: Programming methodology. Software engineering methods for industrial automated systems., CJ International, France, 1999.

4. User Manual for PLC Programming with CoDeSys 2.2, 3S — Smart Software Solutions GmbH. Kempten, 2002.

5. IEC DIS 1131-3 Programable Controllers — Programming Languages, Draft International Standard, International Electrotechnical Commission. February, 14. 1992.

6. Мишель Ж., Лоржо К., Эспьо Б. М. Программируемые контроллеры. М.: Машиностроение, 1986.

7. Мишель Ж. Программируемые контроллеры: архитектура и применение. М.: Машиностроение, 1992.

8. Микропроцессорная Система Технических Средств (МСТС). Руководство по программированию. Смоленск: ОАО НПО «Техноприбор».

9. Карпов Ю. Г. Теория автоматов. СПб.: Питер. — 2002. — 224 с.

10. Шалыто А. А. Логическое управление. Методы аппаратной и программой реализации алгоритмов. СПб.: Наука, 2000. — 780 с.

11. Дьяконов В. П. Компьютерная математика. Теория и практика. М.: Нолидж. — 2001. — 1296 с.

12. Дьяконов В. П. Современные зарубежные микрокалькуляторы. М.: Солон-Р. — 2002. — 400 с.

13. Баскаков С. И. Радиотехнические цепи и сигналы. М.: Высшая школа, — 2000. — 462 с.

14. Нортон П., Йао П. Программирование на Borland C++ в среде Windows. К.: Диалектика, — 1993.

15. Бесекерский В. А., Изранцев В. В. Системы автоматического управления с микроЭВМ. М.: Наука. — 1987. — 320 с.

16. Филлипс Ч., Харбор Р. Системы управления с обратной связью. М.: Лаборатория Базовых Знаний, — 2001 — 616 с.

17. Хемминг Р. В. Численные методы. М.: Наука, — 1968. — 400 с.

18. Каханер Д., Моулер К., Нэш С. Численные методы и программное обеспечение. М.: Мир, 2001. — 575 с.

19. Зельдович Я. Б., Яглом И. М. Высшая математика для начинающих физиков и техников. М: Наука, 1982 г., 512 с.

20. Steven W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing., California Technical Publishing, 1997.

21. Structuring Program Development with IEC 61131-3. Eelco van der Wal, Managing Director PLCopen (www.plcopen.org).

22. MULTIPROG wt Manual, English release 2.0, 1998, Klöpper und Wiege Software GmbH.

23. ProConOS Manual, English release 3.0, 1998, Klöpper und Wiege Software GmbH.

24. Bernd Pelzer, Realtime Extensions for Windows NT: ProConOS NT — Realtime Software PLC on Windows NT. Klöpper und Wiege Software GmbH.

25. OpenPCS Programming System Short Introduction, Version 4.0 English, 2000, Infoteam Software GmbH.

26. SoftControl V2.3 PLC Programming System, Help, Softing GmbH.

27. ISAGRAF V3.3, User's guide, 1999, CJ International.

28. iCon-L Open Programming System for Industry Automation V3.0, Help, MPS&AT / ProSign GmbH.

29. Programming with STEP 7 V5.0, Release 02, 1999, Siemens AG.

30. Working with STEP 7 V5.0, Release 02, 1999, Siemens AG.

31. G. Frey, L. Litz (Eds.). Formal Methods in PLC Programming IEEE SMC 2000, Nashville, TN, 8-11 October 2000.

32. Manfred Werner. Version 2.3 von CoDeSys: Offenes Komlettsystem, SPS-Magazin, Ausgabe 10/2002.

33. Roland Wagner. Ein Programmiertool für die ganze Anlage, A&D NEWSLETTER 11/2002.

34. Дьяконов В. П. MATLAB 6/6.1/6.5, Simulink 4/5. Основы применения. Полное руководство пользователя. М.: Солон-Пресс. — 2002. — 768 с.

35. Дьяконов В. П. Simulink 4. Специальный справочник. СПб.: Питер. — 2002. — 528 с.

36. Дьяконов В. П., Максимчук А. А., Ремнев А. М., Смердов В. Ю. Энциклопедия устройств на полевых транзисторах. М.: Солон-Р. — 2002. — 512 с.

37. Джозеф Шмуллер. Освой самостоятельно UML за 24 часа. М: Вильямс. — 2002. — 352 с.

38. Konrad Etschberger, Contloller Area Network. Basics, Protocols, Chips and Applications, IXXAT Press. Germany, 2001.

Интернет-ссылки

- Международная Электротехническая Комиссия
  http://www.iec.ch/
- PLC Open
  http://www.plcopen.org/
- 3S Smart Software Solutions
  http://www.3s-software.com
- ПК «Пролог»
  http://www.prolog.smolensk.ru/
- Frenzel + Berg Elektronik
  http://www.frenzel-berg.de/
- CJ International
  http://www.isagraf.com/
- Klopper und Wiege Software GmbH
  http://www.kw-software.de/
- Infoteam Software GmbH
  http://www.infoteam.de/
- Softing GmbH
  http://www.softing.com/
- ProSign (Process Design) GmbH
  http://www.pro-sign.de/
- Чарльз Симони
  http://www.edge.org/3rd_culture/bios/simonyi.html
- Карл Адам Петри (домашняя страница)
  http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri_eng.html
- Мир сетей Петри
  http://www.daimi.aau.dk/PetriNets
- Главы из книги [2]
  http://www.fen-net.de/karlheinz.john/Bookview.htm
- Книга [20]
  http://www.dspguide.com

Приложение. Перевод специальных терминов и сокращений

Action	Действие
BCD binary coded decimal	Двоично-кодированный десятичный формат представления чисел
Coil	Обмотка реле
Contact	Контакт
Current Result, CR, accumulator, accu	Аккумулятор
FALSE	Логический ноль, ЛОЖЬ
FB Function block	Функциональный блок
FBD Function Block Diagram	Диаграмма функциональных блоков
Feedback	Обратная связь
Function	Функция
IEC International Electrotechnical Commission	МЭК Международная Электротехническая Комиссия
IL Instruction List	Список инструкций
Instantiation	Создание экземпляра
Jump	Переход
Kernel	Ядро системы исполнения ПЛК
Label, mark	Метка
LD Ladder Diagram	Релейная диаграмма

MMI Man Machine Interface	Человеко-машинный интерфейс
Modifier	Модификатор
Network	Цепь
PC Personal Computer	ПК Персональный компьютер
PID Proportional Integral Derivative	ПИД Пропорционально интегрально дифференциальный регулятор
PLC Programmable Logic Controller	ПЛК Программируемый логический контроллер
POU Program Organization Unit	Компонент организации программ
Program	Программа
SCADA Supervisory Control And Data Acquision System	Система сбора данных и оперативного диспетчерского управления
SFC Sequential Function Chart	Последовательная функциональная схема
ST Structured Text	Структурированный текст
Task	Задача
TRUE	Логическая единица, ИСТИНА
Workbench	Рабочее место программиста

Оглавление

Предисловие научного редактора 3
Введение 6
Предупреждения 10

Глава 1. Программируемые контроллеры 11
1.1. Определение ПЛК 11
1.2. Входы-выходы 14
1.3. Режим реального времени и ограничения на применение ПЛК 15
1.4. Условия работы ПЛК 16
1.5. Интеграция ПЛК в систему управления предприятием . 17
1.6. Доступность программирования 20
1.7. Программный ПЛК 20
1.8. Рабочий цикл 21
1.9. Время реакции 23
1.10. Устройство ПЛК 25
1.10.1. Системное и прикладное программное обеспечение 27
1.10.2. Контроль времени рабочего цикла 27

Глава 2. Стандарт МЭК 61131 28
2.1. Открытые системы 28
2.2. Целесообразность выбора языков МЭК 29
2.3. Простота программирования и доходчивое представление 30
2.4. Единые требования в подготовке специалистов 31

Глава 3. Инструменты программирования ПЛК 32
3.1. Комплексы проектирования МЭК 61131-3 32
3.2. Инструменты комплексов программирования ПЛК . . 35
3.2.1. Встроенные редакторы 36
3.2.2. Текстовые редакторы 36
3.2.3. Графические редакторы 37

3.2.4. Средства отладки 41
3.2.5. Средства управления проектом 44
3.3. Комплекс CoDeSys 46
3.4. Строение комплекса CoDeSys 48

Глава 4. Данные и переменные. 50
4.1. Типы данных 50
4.2. Элементарные типы данных. 51
4.2.1. Целочисленные типы 51
4.2.2. Логический тип 53
4.2.3. Действительные типы 54
4.2.4. Интервал времени 54
4.2.5. Время суток и дата 55
4.2.5. Строки 56
4.2.6. Иерархия элементарных типов 57
4.3. Пользовательские типы данных 57
4.3.1. Массивы 57
4.3.2. Структуры 59
4.3.3. Перечисления 61
4.3.4. Ограничение диапазона 62
4.3.5. Псевдонимы типов 62
4.3.6. Специфика реализации типов данных CoDeSys . . 63
4.4. Переменные 64
4.4.1. Идентификаторы 64
4.4.2. Распределение памяти переменных 65
4.4.3. Прямая адресация 66
4.4.4. Поразрядная адресация 68
4.4.5. Преобразования типов 69
4.5. Тонкости вычислений 70
4.6. Венгерская запись 74
4.7. Формат BCD 77

Глава 5. Компоненты организации программ (POU) . . 78
5.1. Определение компонента 78
5.1.1. Объявление POU 79
5.1.2. Формальные и актуальные параметры 80
5.1.3. Параметры и переменные компонента 81

5.2. Функции 82
5.2.1. Вызов функции с перечислением значений параметров 83
5.2.2. Присваивание значений параметрам функции . . 84
5.2.3. Функции с переменным числом параметров . . . 84
5.2.4. Операторы и функции 85
5.2.5. Перегрузка функций и операторов 86
5.2.6. Пример функции 86
5.2.7. Ограничение возможностей функции 88
5.2.8. Функции в логических выражениях 90
5.3. Функциональные блоки 91
5.3.1. Создание экземпляра функционального блока . . 91
5.3.2. Доступ к переменным экземпляра 92
5.3.3. Вызов экземпляра блока 93
5.3.4. Инициализация данных экземпляра 94
5.3.5. Тиражирование экземпляров 95
5.3.6. Особенности реализации и применения функциональных блоков 96
5.3.7. Шаблонные переменные 97
5.3.8. Пример функционального блока 98
5.3.9. Действия 99
5.4. Программы 100
5.4.1. Использование программ 100
5.5. Компоненты в CoDeSys 100

Глава 6. Структура программного обеспечения ПЛК . . 103
6.1. Задачи 103
6.2. Ресурсы 105
6.3. Конфигурация 106

Глава 7. Языки МЭК 107
7.1. Проблема программирования ПЛК 107
7.1.1. ПЛК как конечный автомат 108
7.2. Семейство языков МЭК 111
7.2.1. Диаграммы SFC 111
7.2.2. Список инструкций IL 114
7.2.3. Структурированный текст ST 115

7.2.4. Релейные диаграммы LD 115
7.2.5. Функциональные диаграммы FBD 116
7.3. Язык линейных инструкций (IL) 117
7.3.1. Формат инструкции 117
7.3.2. Аккумулятор 118
7.3.3. Переход на метку 118
7.3.4. Скобки 119
7.3.5. Модификаторы 120
7.3.6. Операторы 120
7.3.7. Вызов функциональных блоков и программ . . . 121
7.3.8. Вызов функции 122
7.3.9. Комментирование текста. 122
7.3.10. IL в режиме исполнения 123
7.4. Структурированный текст (ST) 124
7.4.1. Выражения 124
7.4.2. Порядок вычисления выражений. 124
7.4.3. Пустое выражение 125
7.4.4. Оператор выбора IF 126
7.4.5. Оператор множественного выбора CASE. 127
7.4.6. Циклы WHILE и REPEAT 129
7.4.7. Цикл FOR 130
7.4.8. Прерывание итераций операторами EXIT и RETURN 132
7.4.9. Итерации на базе рабочего цикла ПЛК 133
7.4.10. Оформление текста. 134
7.5. Релейные диаграммы (LD) 136
7.5.1. Цепи 136
7.5.2. Реле с самофиксацией 138
7.5.3. Порядок выполнения и обратные связи 139
7.5.4. Управление порядком выполнения 140
7.5.5. Расширение возможностей LD 141
7.5.6. Особенности реализации LD в CoDeSys 142
7.5.7. LD-диаграммы в режиме исполнения 144
7.6. Функциональные блоковые диаграммы (FBD) 144
7.6.1. Отображение POU 144
7.6.2. Соединительные линии 146
7.6.3. Порядок выполнения FBD 146

7.6.4. Инверсия логических сигналов 146
7.6.5. Соединители и обратные связи 146
7.6.6. Метки, переходы и возврат 147
7.6.7. Выражения ST в FBD 148
7.7. Последовательные функциональные схемы (SFC) . . 149
7.7.1. Шаги 149
7.7.2. Переходы 149
7.7.3. Начальный шаг 151
7.7.4. Параллельные ветви 152
7.7.5. Альтернативные ветви 152
7.7.6. Переход на произвольный шаг 153
7.7.7. Упрощенный SFC 154
7.7.8. Стандартный SFC 157
7.7.9. Классификаторы действий 158
7.7.10. Действие — переменная 161
7.7.11. Механизм управления действием 162
7.7.12. Внутренние переменные шага и действия . . . 165
7.7.13. Функциональные блоки и программы SFC . . . 166
7.7.14. Отладка и контроль исполнения SFC 167

Глава 8. Стандартные компоненты 170
8.1. Операторы и функции 170
8.1.1. Арифметические операторы 170
8.1.2. Операторы битового сдвига 172
8.1.3. Логические битовые операторы 173
8.1.4. Операторы выбора и ограничения 174
8.1.5. Операторы сравнения 175
8.1.6. Математические функции 176
8.1.7. Строковые функции 177
8.2. Стандартные функциональные блоки 178
8.2.1. Таймеры 178
8.2.2. Триггеры 182
8.2.3. Детекторы импульсов 183
8.2.4. Счетчики 184
8.3. Расширенные библиотечные компоненты 185
8.3.1. Побитовый доступ к целым 186
8.3.2. Гистерезис 187

8.3.3. Пороговый сигнализатор 188
8.3.4. Ограничение скорости изменения сигнала 188
8.3.5. Интерполяция зависимостей 189
8.3.6. Дифференцирование 191
8.3.7. Интегрирование 193
8.3.8. ПИД-регулятор 195

Глава 9. Примеры программирования 200
9.1. Генератор импульсов (PRG LD) 200
9.2. Последовательное управление по времени (PRG LD, SFC) 201
9.3. Кодовый замок (PRG LD) 203
9.4. Динамический знаковый индикатор (FUN LD, ST) 205
9.5. Целочисленное деление с симметричным округлением (FUN ST) 207
9.6. Генератор случайных чисел (FB ST) 210
9.7. Очередь FIFO (FB ST) 212
9.8. Быстрая очередь FIFO (FB ST) 214
9.9. Фильтр «скользящее среднее» (FB ST) 215
9.10. Медианный фильтр (FB ST) 217
9.11. Линеаризация измерений (PRG ST) 221
9.12. Широтно-импульсный модулятор на базе таймера (FB IL) 224
9.13. Управление реверсивным приводом (FB SFC) 226
9.14. Сравнение языков с позиции минимизации кода (IL, ST, FBD, LD 231
9.14.1. Программирование последовательности состояний (ST, IL) 232
9.14.2. Параллельное решение в виде логических выражений (FBD, LD, ST, IL) . . . 237
9.14.3. Функциональный блок против программы . . . 241

Список литературы 242

Интернет-ссылки 245

Приложение
Перевод специальных терминов и сокращений 246

Серия «Библиотека инженера»

Игорь Викторович Петров

Программируемые контроллеры. Стандартные языки и приемы прикладного проектирования

Под ред. проф. В. П. Дьяконова

Ответственный за выпуск
В. Митин

Макет и верстка
Н. Бармина

Обложка
Е. Жбанов

ООО «СОЛОН-Пресс»
123242, Москва, а/я 20
Телефоны:
(095) 254-44-10, 252-36-96, 252-25-21
E-mail: Solon-Avtor@coba.ru

Распространение
ООО «Альянс-книга»
(095) 258-91-94

ООО «СОЛОН-Пресс»
127051, г. Москва, М. Сухаревская пл., д. 6, стр. 1 (пом. ТАРП ЦАО)
Формат 60×88/16. Объем 16 п. л. Тираж 1500
ООО «Пандора-1»
Москва, Открытое ш., д. 28
Заказ №