7. *Языки программирования промышленных контроллеров* (ПРК)

Прикладное программное обеспечение (ПО) современных ПРК, имеющих встроенную операционную систему (ОС), может быть разработано как с использованием *традиционных инструментальных программных средств* (языки Си, Паскаль и т.д.), так и на основе *специализированных языковых средств*.

Традиционная технология требует от разработчика прикладного ПО знаний не только в области алгоритмических языков программирования, но и особенностей ОС, а также аппаратных возможностей данного ПРК и организации системы ввода-вывода. При этом разработанное ПО будет *привязано* только к данному типу ПРК и *не может быть перенесено* на другую аппаратно-программную платформу.

Специальные платформо-независимые языки программирования ПРК могут быть применены для различных типов ПРК и разработанные с их помощью программы переносимы, т.к. языки этого типа стандартизированы. Под эгидой Международной Электротехнической Комиссии (МЭК/IEC) разработан стандарт **IEC 1131-3**, специфицирующий ***5*** языков программирования ПРК: три *графических* (SFC, FBD, LD) и два *текстовых* (ST, IL).

В них заложена методология структурного программирования, позволяющая пользователю представить автоматизируемый процесс в наиболее простой и понятной форме.

На рисунке 7.1 отображены области применения языков программирования ПРК.

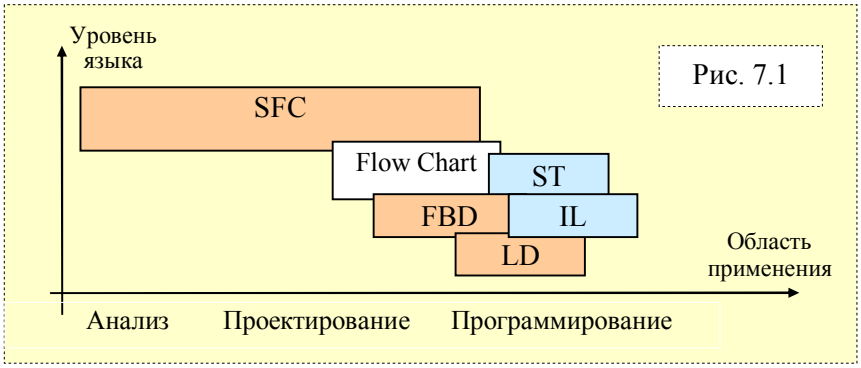


Рис. 7.1

Современные интегрированные многоязыковые системы программирования ПРК позволяют осуществлять *разработку* и *тестирование* прикладных программ для ПРК; обеспечивают единую среду разработки программ для различных типов систем на нескольких языках (в основном, работают под управлением ОС Windows и); соответствуют стандартам графического интерфейса пользователя Microsoft Windows. Они предоставляют также возможность создавать библиотеки функциональных блоков, которые можно использовать в прикладных программах.

Встроенный программный *эмулятор работы* ПРК позволяет производить отладку программ без подключения к реальному ПРК, используются также эмуляторы, с помощью которых можно моделировать состояния сигналов модулей ввода-вывода.

Одной из первых реализаций указанного выше стандарта стала инструментальная система ISaGRAF, разработанная компанией CJInternational (Франция).

Язык последовательных функциональных схем - **SFC** (Sequential Function Chart)

Характер многих *технических объектов и технологических процессов* предполагает выполнение отдельных операций последовательно. В этих случаях разработчику ПО очень важно иметь средство программирования, которое позволяло бы наглядно воспроизводить структуру объекта/процесса. Именно таким средством и является язык SFC. Это высокоуровневый *графический язык*, предназначенный для использования на этапе *проектирования*. Его основой является *математический аппарат* ***сетей Петри*** (СП), позволяющий описать процессы в форме двудольных ориентированных графов (***рис.*** 7.2). Здесь: P_i – позиции, T_j – переходs, $M(P_i)$ – разметка.

Срабатывание какого-либо перехода T_j в размеченной сети ведет к смене разметки.

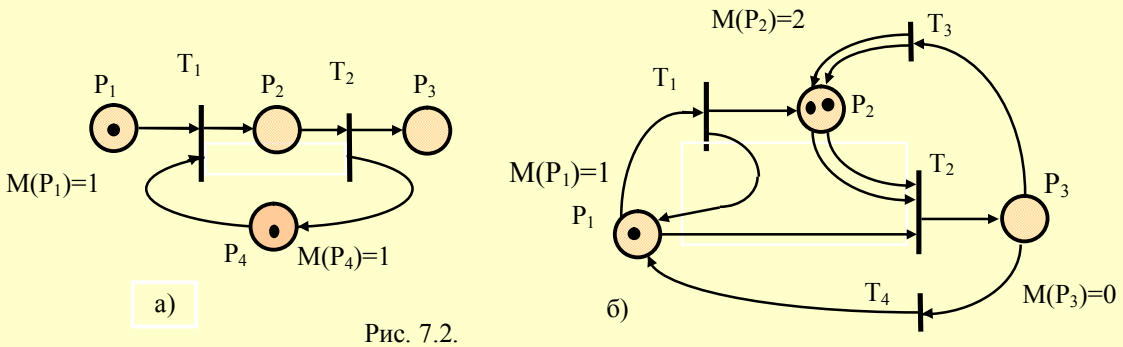


Рис. 7.2.

Функционированием (срабатыванием) размеченной СП называется процесс изменения разметки (начиная от M_0), осуществляемый по правилам:

Естественно, что ***при формировании языка* SFC** интерпретация и реализация некоторых положений теории сетей Петри изменилась.

Язык SFC предназначен для использования *на этапе проектирования* ПО и позволяет описать *блок-схему* программы, т.е. логику ее работы на уровне последовательных *шагов* и условных *переходов*. Он обеспечивает общую структуризацию и координацию функций управления последовательными процессами или машинами и механизмами.

SFC-программа состоит из элементов двух типов: *шагов* (*steps*) и *переходов* (*transitions*), которые могут включать в себя ***элементы других языков***. В соответствии с состоянием внутренних ссылок и входов-выходов логика *шага* может обрабатываться или игнорироваться, т.е. в каждый текущий момент времени *шаг* может быть активным или пассивным.

Логические структуры, связанные с *шагом***,** обрабатываются до тех пор, пока не произойдет событие, предписывающее ПРК перейти к обработке другого *шага***.** С помощью языка SFC автоматизируемый процесс представляется в виде совокупности определённых последовательных *шагов* (автономных ситуаций), разделённых (связанных) *переходами*. Каждому переходу *сопоставлено* ***логическое условие*****,** а *шагу* **– *совокупность действий***.

Условные графические обозначения компонентов SFC-программ приведены на ***рис.*** 7.3: *Начальный шаг; Шаг; Переход; Переход на другой шаг* (*длинный переход*); *Макрошаг* и др**.**

На рис. 7.4-7.6 приведены некоторые программы, разработанные на языке SFC в среде ISaGRAF. *Действия* внутри шагов могут описываться более детально на других языках (ST, IL, LD, FBD).

SFC-***программа*** – это графически представленная совокупность *шагов* и *переходов*, соединенных *направленными связями*. По умолчанию, ориентация линии - *сверху вниз*. Некоторые части программы могут быть отделены и представлены в основной схеме одним специальным символом – *макрошагом*. *Основное правило* при построении схем: шаги не могут следовать подряд; переходы тоже не могут следовать подряд.

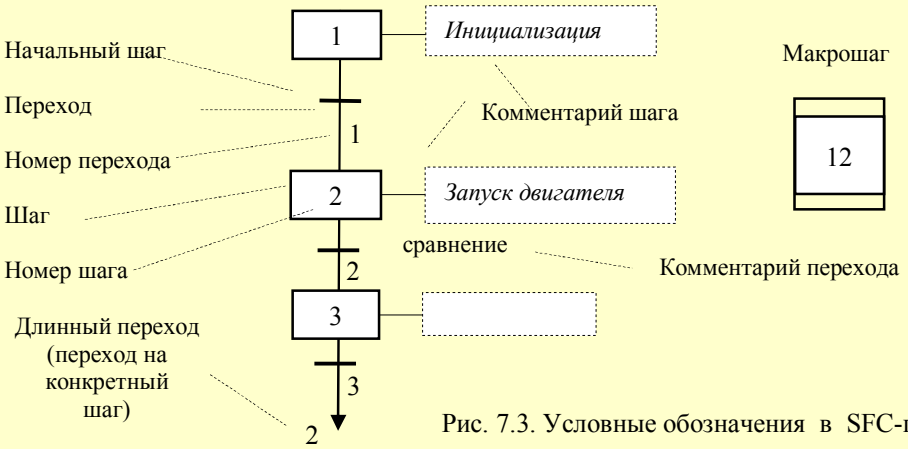


Рис. 7.3. Условные обозначения в SFC-программах

Программирование на SFC обычно разделяется на 2 различных уровня: *уровень* 1 - показывает графически блок-схемы, номера ссылок на шаги, переходы и комментарии, присоединённые к ним; описание *шагов* и *переходов* дается внутри прямоугольников, присоединенных к символам шага и перехода, в виде свободного *комментария* (который не является частью языка); *уровень* 2 – программирование действий внутри шага или условий, присоединённых к переходу, на языке ST или IL; подпрограммы, написанные на других языках (FDB, ST, LD или IL) могут обращаться к этим действиям или переходам.

Начальная ситуация описывается начальным шагом; после запуска программы автоматически активизируются (выделяются) все начальные шаги.

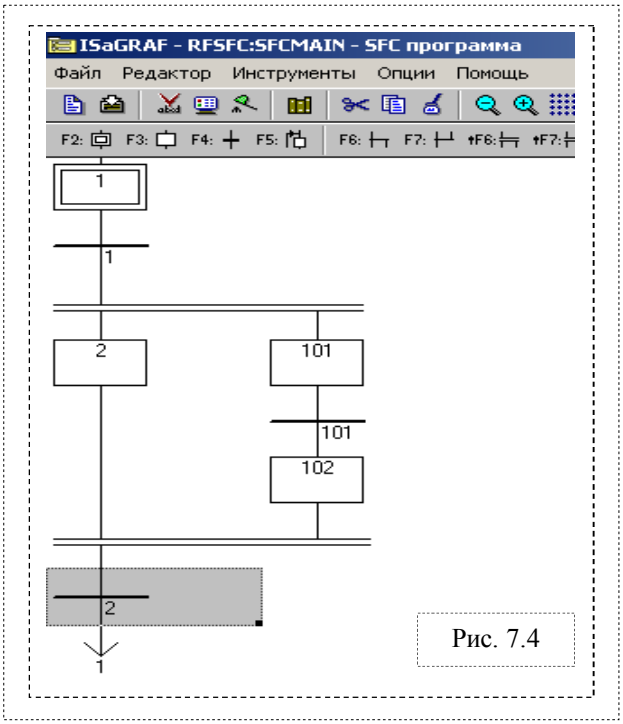


Рис. 7.4

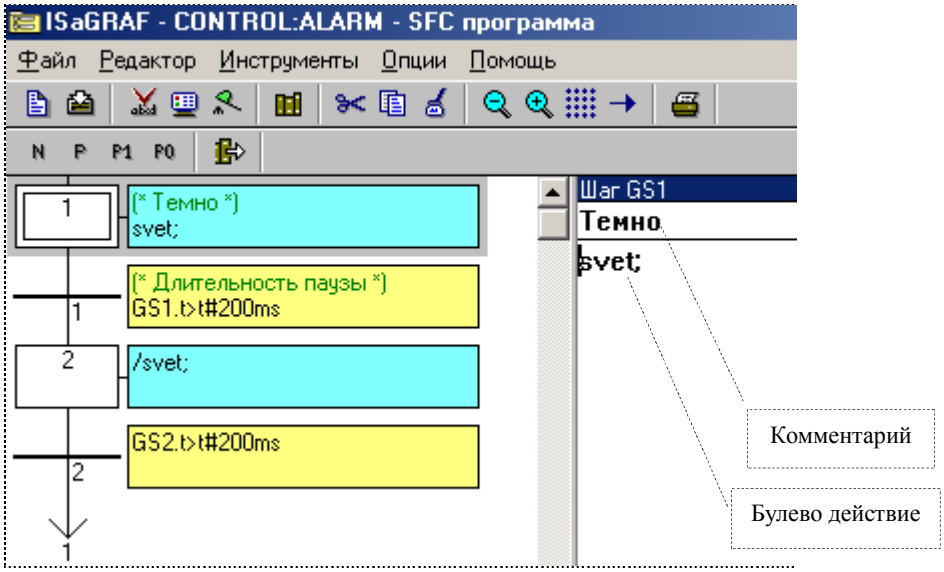


Рис. 7.5.

Атрибуты шага

У *шага* имеются *атрибуты*, которые могут быть использованы в любом другом языке:

- **GSnnn.x** – характеризует его *активность* (логическая *переменная*);

- **GSnnn.t** – характеризует *продолжительность* (*время*) его активного состояния (*таймер*), здесь **nnn** – номер шага.

Действия внутри шагов и условия, соответствующие переходам

На втором уровне программирования осуществляется детальное *описание действий*, которые выполняются во время *активности шага*, и *условий*, которые соответствуют переходам. По умолчанию языком программирования *второго уровня* является язык ST.

Переходы программируются чаще всего простыми булевыми выражениями, а *шаг* может содержать несколько различных типов действий (булевы, импульсные, не сохраняемые, SFC-действия).

Действия внутри шагов могут быть следующих типов:

- *Булевы действия и SFC-действия* (управление дочерними SFC-программами), описываемые с помощью ограниченных текстовых возможностей самого языка SFC (рис. 7.6);
- *"Pulse" и "Non-stored" - импульсные действия*, программируемые на ST и IL (рис. 7.7);
- *вызов подпрограмм*, написанных на любом языке кроме SFC.

В одном и том же шаге может быть описано несколько действий одного или разных типов.

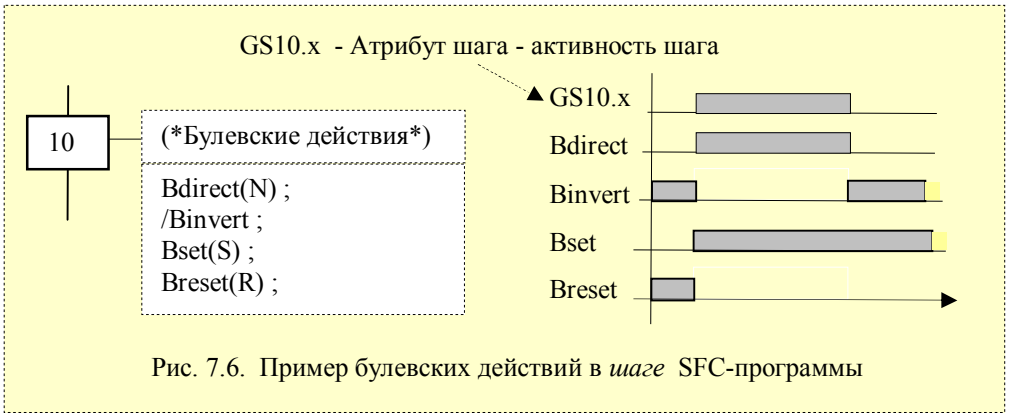


Рис. 7.6. Пример булевских действий в *шаге* SFC-программы

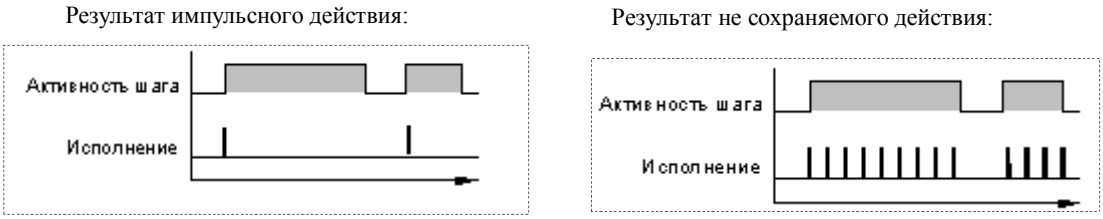


Рис. 7.7. Пример импульсных действий в *шаге* SFC-программы

Условия, соответствующие переходам: Переход 2-уровня - это *булево выражение*. Для того чтобы задать его на языке ST, нужно просто ввести булево выражение в соответствии с синтаксисом ST.

***Типы параллельных соединений*: *схождения и расхождения* (*рис*. 7.8, 7.9)**

Расхождения – это множественные связи *от одного* шага или перехода *ко многим*.
Схождения – это множественные связи *от более чем одного* шага или перехода *к одному* другому.

При обозначении *схождений и расхождений* используются *одиночные* или *двойные* линии. *Альтернативные расхождения и схождения* обозначаются *одиночными горизонтальными* линиями. *Расхождение альтернативное* (*альтернативные ветви*) – это множественная связь *от одного шага* к *нескольким переходам*. Активной становится *одна* из ветвей (в зависимости от активности того или иного перехода). *Проверка активности переходов осуществляется слева направо.*

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 таких ветвях всегда работает только одна из них, поэтому ее окончание и будет означать переход к следующему за альтернативной группой шагу.

При создании альтернативных ветвей желательно задавать взаимоисключающие условия.

Схождение альтернативное – используется для того, чтобы объединить несколько ветвей SFC, начавшихся из альтернативного расхождения.

Параллельные расхождения (параллельные ветви) и схождения - обозначаются *двойными горизонтальными* линиями.

Каждая параллельная ветвь начинается и заканчивается шагом. Т.е. условие входа в параллельность всегда одно, условие выхода тоже всегда одно на всех.

Параллельные ветви выполняются теоретически одновременно. Практически – в одном рабочем цикле, слева направо.

Условие перехода, завершающее параллельность, проверяется только в случае, если в каждой параллельной ветви активны последние шаги.

Расхождение параллельное – это множественная связь *от одного* перехода *к нескольким шагам*. Она соответствует параллельному функционированию процесса.

Схождение параллельное – это множественная связь *от нескольких шагов* к одному и тому же переходу. Используется, чтобы объединить несколько ветвей SFC, начавшихся их параллельного расхождения.

Рис. 7.8. Фрагменты программ, написанных на языке SFC (*альтернативное* расхождение)

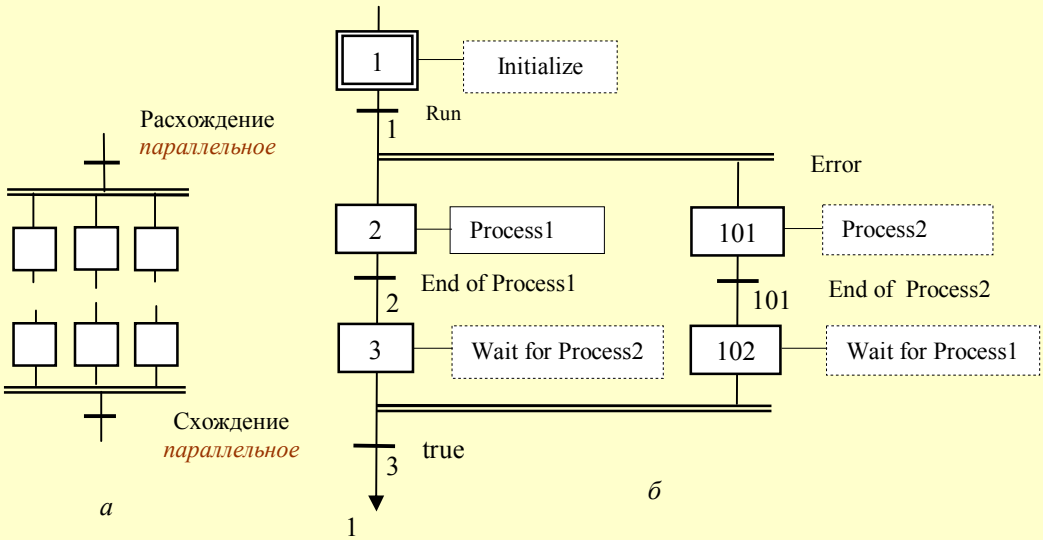


Рис. 7.9. *Параллельное* расхождение и схождение: а) условное обозначение; б) *пример* SFC-программы

Иерархия программы SFC. В системе ISaGRAF каждая SFC-программа может *управлять* (запускать, уничтожать, и т.д.) другими программами на этом же языке (SFC), которые в таком случае называют *дочерними* программами той программы, которая ими управляет (рис. 7.10). SFC-

программы объединяются в *иерархическое дерево* на основе “родственных” отношений (*родитель-наследник*).

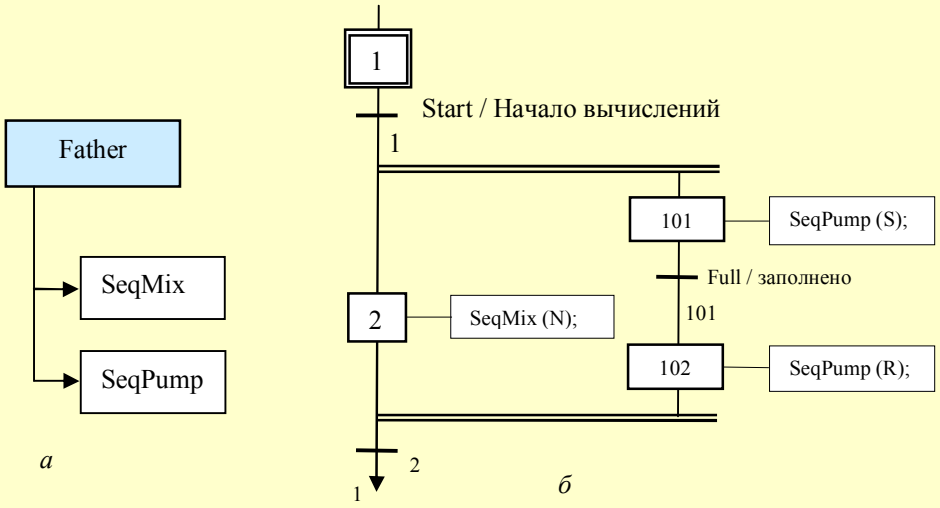


Рис. 7.10. Главная и дочерние SFC-программы:
а) иерархическое SFC-дерево; б) родительская программа, использующая SFC-действия

Язык функциональных блок-схем - FBD (Function Block Diagram)

FBD – *графический* язык - применяется для построения комплексных процедур, состоящих из различных функциональных библиотечных блоков – арифметических, тригонометрических, регуляторов, мультиплексоров и т.д. Наиболее подходит для управления непрерывными процессами и регулирования.

При этом *осуществляется*: *представление функций посредством блоков, связанных между собой; соединения между выходами функциональных блоков в явном виде могут отсутствовать; выход блока может соединяться со входами одного или нескольких блоков.*

Объектами языка FBD являются:

- элементарные функции и элементарные функциональные блоки (ФБ); они находятся в библиотеке; логика их работы (программа) написана на языке С и не может быть изменена в редакторе FBD; изменять можно только их параметры;
- функции и ФБ *пользователя*; конструируются пользователем из элементов языка FBD.

Разработка программы осуществляется с помощью *графического редактора* посредством формирования блок-схемы из перечисленных выше компонентов, которые объединяются друг с другом либо посредством внешних (фактических) параметров (переменные, соответствующие входам и выходам), т.е. *таблично*, либо непосредственно линиями связи – *графическими связями*.

Последовательность (очередность) обработки отдельных компонентов в программе определяется потоком данных.

FBD-программа (рис. 7.11) очень напоминает функциональную схему электронного устройства. Она строится из стандартных элементов библиотеки (например, ISaGRAF, КОНГРАФ и др.). Каждый ФБ имеет фиксированное количество входных точек связи и фиксированное количество выходных точек связи.

На ***рис.*** 7.11,7.12 приведены примеры FBD-программ в среде КОНГРАФ и ISaGRAF.

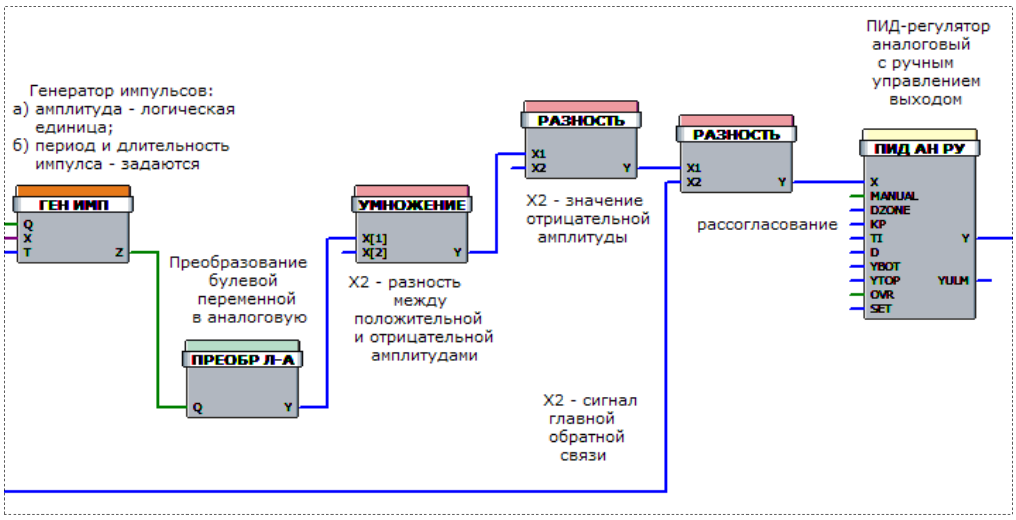


Рис. 7.11. Моделирование САР в среде КОНГРАФ; используются ФБ:
ГЕН ИМП – генератор последовательности импульсов; ПРЕОБР Л-А – преобразование логических переменных в аналоговые; УМНОЖЕНИЕ – умножение; РАЗНОСТЬ – вычитание; ПИД АН РУ – *аналоговый* ПИД-*закон регулирования*

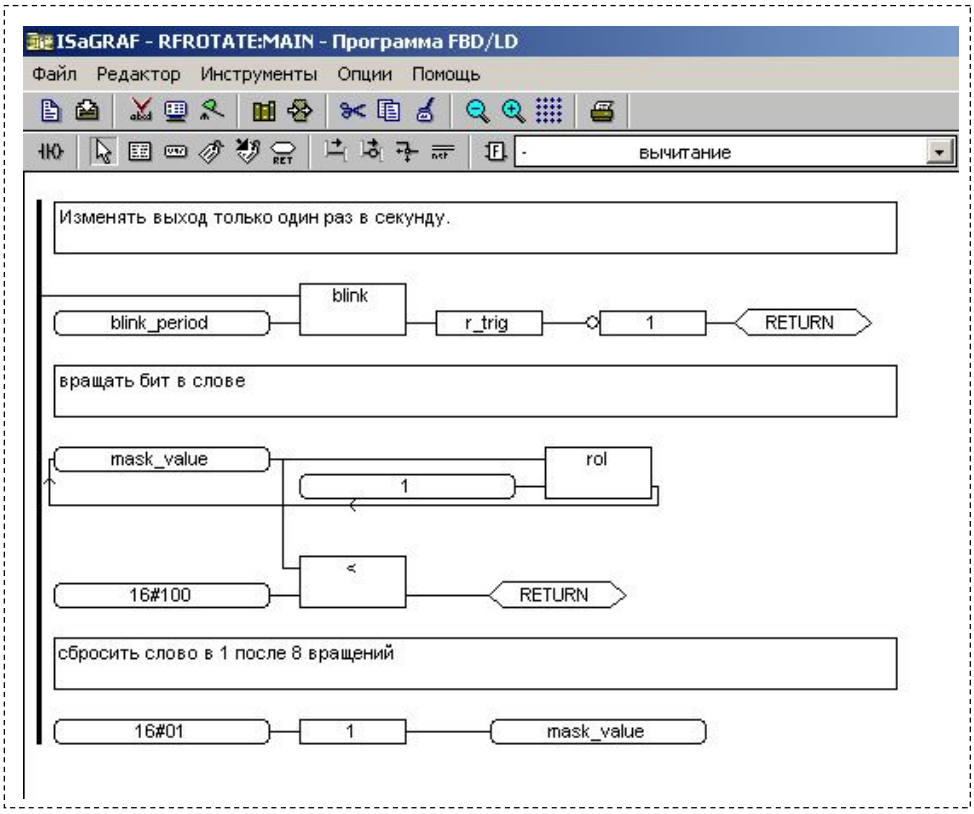


Рис. 7.12. Программа на языке FBD (ISaGRAF): blink – ФБ генератор импульсов; 1 – ФБ *присваивание*; r-trig – ФБ *обнаружение/определение переднего фронта*; RETURN – условное завершение программы; rol – ФБ *циклический сдвиг влево*; < – ФБ - сравнение "*меньше чем*"

FBD-программа описывает *функцию* между входными и выходными переменными. Эта функция представляется совокупностью элементарных ФБ. Тип каждой переменной должен быть тем же, что и тип соответствующего входа. Входом FBD-блока может быть константа, любая внутренняя, входная или выходная переменная.

Язык релейных диаграмм - **LD** (Ladder Diagram)

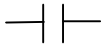
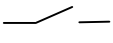
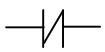
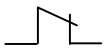
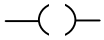
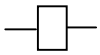
***Язык* LD** – это *графический* язык - применяется для описания логических выражений различного уровня сложности, графического представления булевых уравнений. Он содержит *контакты* (входные аргументы) и *катушки* (выходные переменные). Элементы организуются в *сеть* релейно-контактных схем. При необходимости можно реализовывать более сложную логику, используя, например элементы языка FBD.

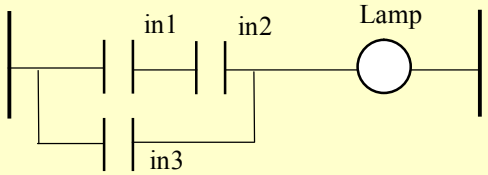
Каждому контакту ставится в соответствие логическая переменная, определяющая его состояние. Ее имя ставится над контактом и служит его названием. Если контакт замкнут, то переменная имеет значение *true*, если разомкнут – *false*. *Последовательное соединение* контактов или цепей соответствует логической операции И/AND, *параллельное* – ИЛИ/OR. Нормально *замкнутый* (*инверсный*) контакт равнозначен логической операции НЕ.

Релейно-контактные схемы и элементы

Релейная *схема* представляет собой 2 вертикальные шины питания, между которыми расположены горизонтальные *цепи* из контактов и катушек реле. Графические символы языка LD соответствуют элементам электрических цепей и имеют те же названия и обозначения (табл. 7.1).

Таблица 7.1.

LD	ЕСКД	Обозначение
		Нормально разомкнутый контакт
		Нормально замкнутый контакт
		Обмотка реле



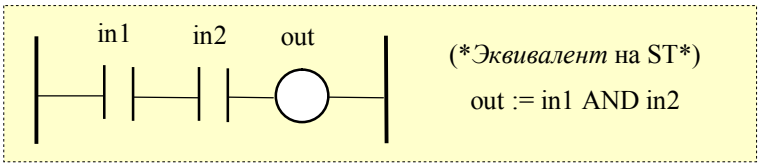
Эта схема эквивалентна выражению:

Lamp:= (In1 AND In2) OR In3;

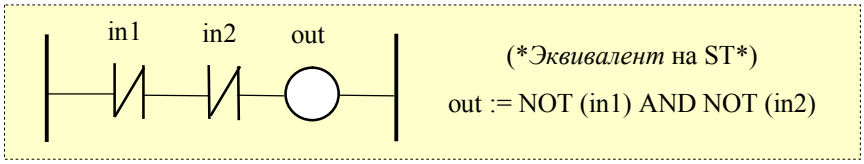
LD-программа выполняется *последовательно* слева направо и сверху вниз. В каждом рабочем цикле однократно выполняются все цепи, входящие в сеть. Любая переменная в рамках одной цепи всегда имеет одно и то же значение. Если даже реле в цепи изменит переменную, то новое значение поступит на контакты только в следующем цикле. Цепи, расположенные ниже, получат новое значение переменной сразу, а расположенные выше - только в следующем цикле. Строгий порядок выполнения цепей очень важен. Благодаря жесткому порядку выполнения LD-программы сохраняют устойчивость при наличии обратных связей.

Объекты языка LD

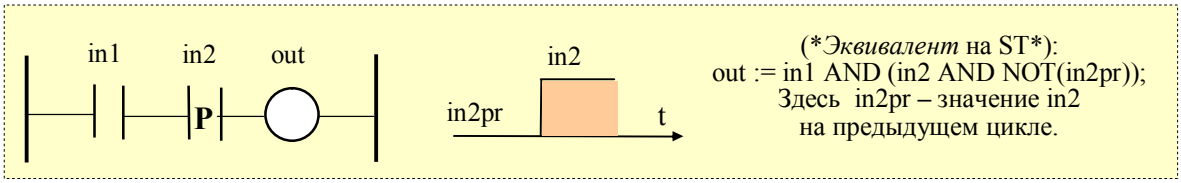
1. *Нормально разомкнутый контакт* (НРК).



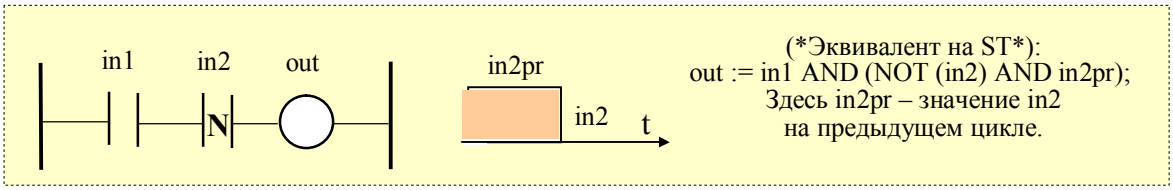
2. *Нормально замкнутый* (*инвертирующий*) *контакт* (НЗК).



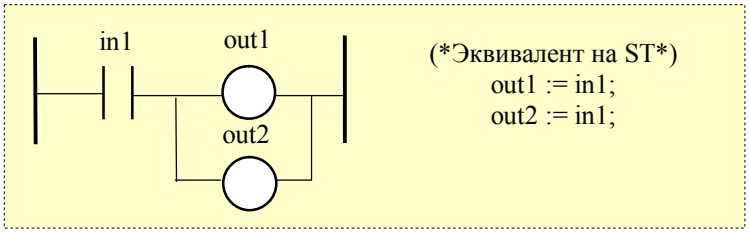
3. *Контакт, замыкаемый передним фронтом* (ЗПФ).



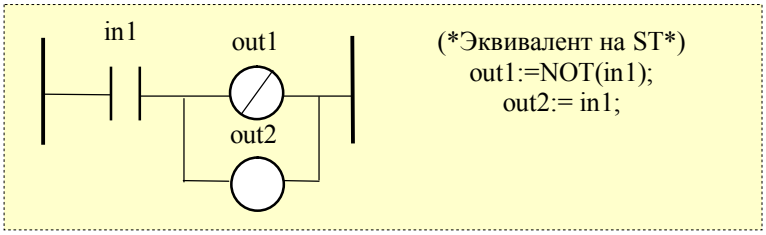
4. *Контакт, замыкаемый задним фронтом* (ЗЗФ).



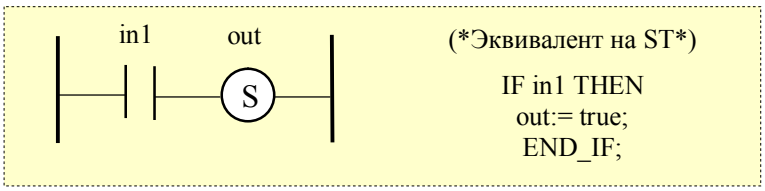
1. *Катушка* (*обычная*).



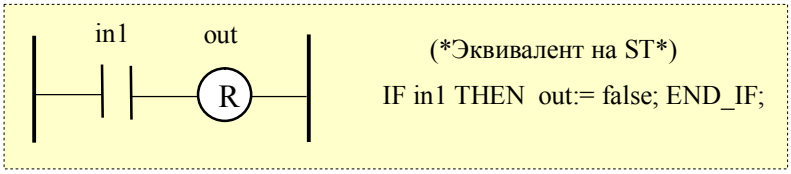
2. *Инвертирующая катушка*.



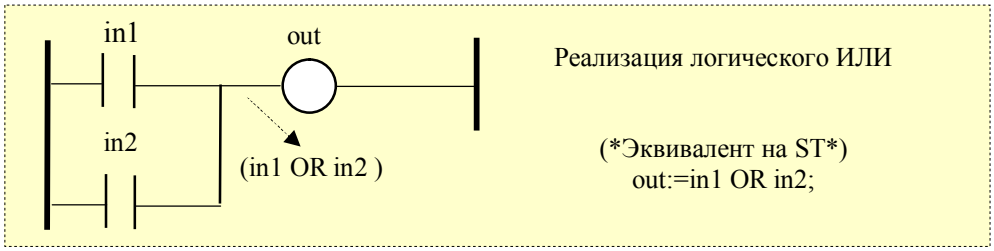
3. *Катушка* ***установки*** *значения переменной - SET-катушка*.



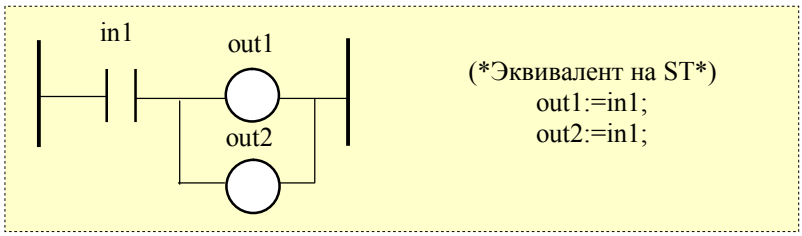
4. *Катушка* ***сброса*** *значения переменной - RESET-катушка*.



Левое множественное соединение



Правое множественное соединение



Для управления выполнением LD-программ, могут использоваться *метки* и условные/безусловные *переходы* к ним. Если линия соединения слева от длинного перехода имеет состояние TRUE, то выполнение программы продолжается с фрагмента, расположенного после соответствующей метки.

Используя LD-редактор, можно подключить функциональные блоки (ФБ) к логическим схемам. В общем случае это может быть оператор, функция или ФБ. Т.к. блоки не всегда имеют логические/*булевы* входы и/или логические выходы, то при введении блоков в LD-программы приводит к добавлению нескольких новых параметров (*входов*) EN, ENO в эти блоки.

На ***рис.*** 7.13 приведены LD-программы (ISaGRAF), реализующие *булевы функции*.

$f_1 = x_1 \wedge x_2$ - конъюнкция ($\wedge$, И, &);

$f_2 = x_3 \vee x_4$ - дизъюнкция ($\vee$, ИЛИ);

$f_3 = \overline{x_1}$ - отрицание (инверсия);

$f_4 = x_5 \oplus x_6$ - исключающее ИЛИ / сложение по модулю 2 ($\oplus$);

$f_5 = (x_7 \vee x_8 \vee x_9) \wedge \overline{x_{10}} \wedge (x_{11} \vee x_{12})$ - произвольная булева функция.

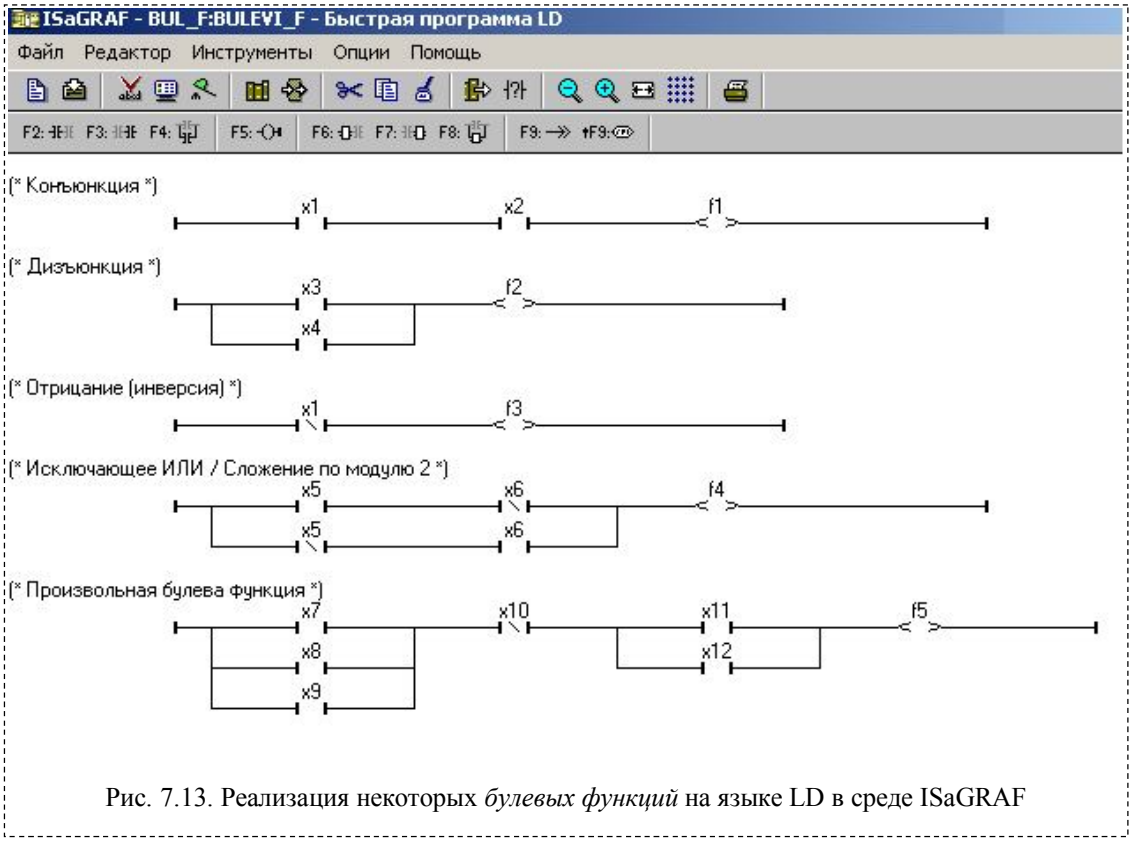


Рис. 7.13. Реализация некоторых *булевых функций* на языке LD в среде ISaGRAF

***Язык структурированного текста* – ST** (Structured Text)

Это *текстовый* язык *высокого уровня* с инструкциями и синтаксисом уровня адаптированного языка Паскаль. Он позволяет программировать сложные алгоритмы обработки данных – последовательности команд с использованием: переменных, вызовов функций и функциональных блоков (ФБ), операторов повторения и т.д., а также для описания действий внутри шагов и условий языка SFC. В основном используется в тех случаях, когда алгоритм трудно описать с помощью графических языков.

Логика функциональных блоков создается в C++ и не может быть изменена в ST редакторе.

Выражения

Основой ST-программы служат выражения. *Выражения* состоят из *операндов* (констант и переменных) и *операторов*.

Операторы - являются ”командами“ языка программирования *ST*. Они должны заканчиваться точкой с запятой. Одна строка может содержать несколько операторов (отделяемых точками с запятой).

Результат вычисления выражения присваивается переменной при помощи *оператора* присваивания “:=”. Каждое выражение обязательно заканчивается точкой с запятой “;”.

Выражение состоит из переменных, констант и функций, разделенных *операторами*, например: Var1 := 1+Var2 / ABS(Var2);

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

Имена, используемые в исходном коде (*идентификаторы переменных*, *константы*, *ключевые слова*) разделены *неактивными разделителями* (пробелами, символами окончания строки и табуляции) или *активными разделителями*, которые имеют заранее определенное значение (например, символ-разделитель “>” означает сравнение больше чем, а символ “+” операцию сложения и т.д.).

Неактивные разделители могут быть свободно введены между активными разделителями, константами и идентификаторами. В отличие от *неформатных* языков, таких как IL, конец строки может быть введен в любом месте программы.

Для улучшения читаемости программ нужно использовать неактивные разделители в соответствии со следующими правилами: не более одного оператора в строке; табуляцию для сдвига сложных операторов; комментарии.

В текст могут быть введены *комментарии*, которые должны начинаться символами “(*“ и заканчиваться ими же “)*“.

Несколько выражений можно записать в одну строку. Однако 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:

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

Тип всех операндов выражения должен быть одинаковым. Для изменения типов использовать функции преобразования типов: BOO, ANA, REAL, TMR и MSG.

Для того чтобы отделить подчасти выражения и явно определить приоритетность операций используются *скобки*. Когда в сложном выражении нет скобок, приоритетность ST-операторов задана неявно.

Например: 2 + 3 * 6 равно 2+18=20 - оператор * имеет высший приоритет;
(2+3) * 6 равно 5*6=30 - приоритет задается скобками.

Максимальное количество вложенных скобок - 8.

Вызовы функций и функциональных блоков

Стандартные ST-вызовы функций могут быть использованы для каждого из следующих объектов: Подпрограммы; Библиотечной функции и ФБ, написанных на IEC-языках; С-функции и ФБ; Функции преобразования типов.

Вызовы подпрограмм или функций предполагает использование следующих элементов:

Имя: имя вызываемой подпрограммы или библиотечной функции, написанной на IEC-языке или С.

Значение: вызывает ST-, IL-, LD- или FBD-подпрограммы или функции или С-функции и дает возвращаемое значение.

Синтаксис: <variable> := <subprog> (<par1>,...<parN>);

Операнды: За типом возвращаемого значения и параметрами вызова должен следовать интерфейс, определенный для подпрограммы.

Некоторые основные операторы языка ST: присваивание; вызов подпрограммы или функции; вызов функционального блока; операторы выбора/ветвления (IF-THEN-ELSE, CASE); операторы цикла (FOR, WHILE, REPEAT); управляющие операторы (RETURN, EXIT); операторы для управления программой, написанной на языке SFC.

На *рис.* 7.14 приведен пример ST-программы, разрабатываемой в соответствующем редакторе системы ISaGRAF.

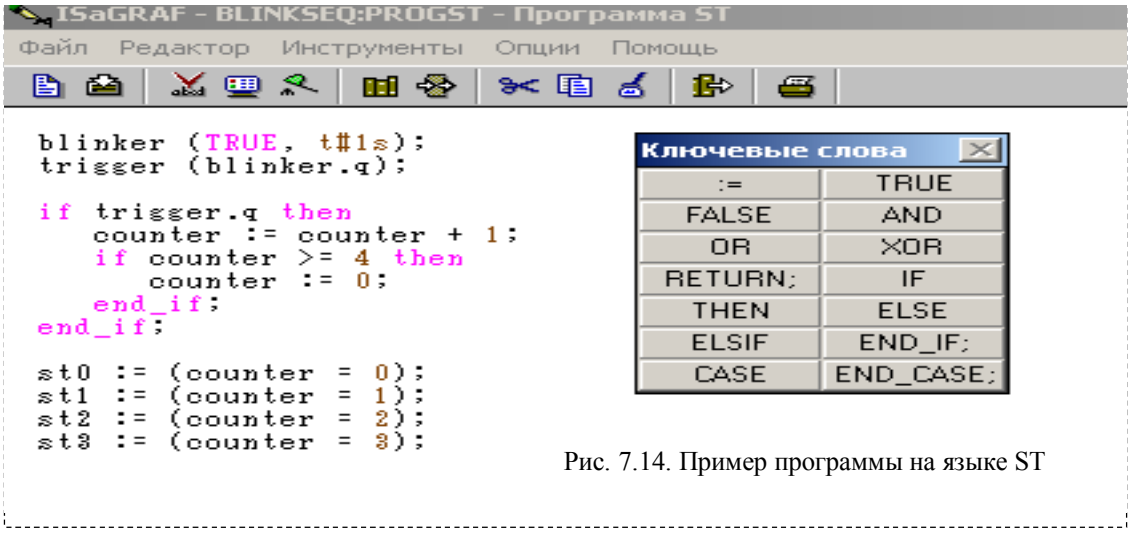


Рис. 7.14. Пример программы на языке ST

Язык инструкций/команд – **IL** (*Instruction List*)

IL – это *текстовый* язык *низкого уровня*, класса *ассемблера*, применяется для программирования булевых функций и эффективных, оптимизированных процедур. Это *одноадресный* язык, поэтому в большинстве его команд используется *неуказываемый явно* текущий результат. В микропроцессорах для этих целей используется внутренний регистр общего пользования, называемый *аккумулятором* (АК), поэтому далее для простоты будем использовать это обозначение.

Программа на языке IL представляет собой *список инструкций/команд*; каждая *команда* должна начинаться с новой строки и содержать *оператор*, заканчивающийся необязательным *модификатором* и, если необходимо, *операндом*.

Оператор – это символ арифметических или логических операций, или вызов функционального блока (ФБ). Он указывает операцию, которая должна быть выполнена над содержимым АК, в котором хранится текущий результат, представляющий собой неявно заданный *операнд*, и вторым *операндом*, явно указанным в этом операторе. Полученный результат выполнения оператора помещается снова в *АК*.

Команда – это один оператор плюс один или несколько операндов, разделенных запятыми; перед командой может стоять *метка* (необязательно), после которого ставится символ ":".

Комментарий - необязательный элемент, начинается и заканчивается символами "* и *" соответственно.

В таблице 7.2 приведен перечень операторов языка IL.

Таблицы 7.2

Оператор	*Модификатор*	*Операнд*	*Описание*
LD	N	Константа, переменная	Загружает значение операнда в аккумулятор (АК)
ST	N	Переменная	Присваивает значение аккумулятора операнду
S		Булева переменная	Устанавливает операнд в 1 (true), если (АК)=1 (true)
R		Булева переменная	Устанавливает операнд в 0 (false), если (АК)=0 (false)
AND	N, (	Булева	Логическое И
OR	N, (	Булева	Логическое ИЛИ
XOR	N, (	Булева	Логическое исключающее ИЛИ
ADD	(	Константа, переменная	Сложение
SUB	(	Константа, переменная	Вычитание
MUL	(	Константа, переменная	Умножение
DIV	(	Константа, переменная	Деление
GT	(	Константа, переменная	Проверить: >
GE	(	Константа, переменная	Проверить : >=
EQ	(	Константа, переменная	Проверить : =
NE	(	Константа, переменная	Проверить : <>
LE	(	Константа, переменная	Проверить : <=
LT	(	Константа, переменная	Проверить : <
JMP	C, N	Метка	Переход на метку
CAL	C, N	Экземпляр ФБ	Вызов ФБ
RET	C, N		Завершение программы, возврат из подпрограммы
)			Выполнить задержанную операцию

ФБ запускаются с помощью оператора CAL или посредством использования входов ФБ. Программа начинается с команды LD – загрузка операнда в аккумулятор.

Управление данными. Основными операторами для передачи данных являются команды ***LD*** и ***ST***: ***LD*** (LoaD) осуществляет *загрузку* в АК операнда; допустимый модификатор – **N;** ***ST*** (STore) осуществляет *сохранение* содержимого АК в операнде; допустимый модификатор – **N**.

Рассмотрим *пример* использования *операторов* **LD** и **ST**:

```
LD  10
ADD  25
ST  B
```

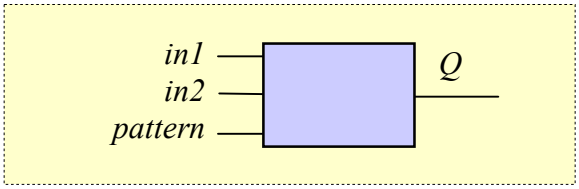
В АК загружается константа 10, затем прибавляется число 25 (оператор ADD), а результат присваивается переменной B. Теперь значение переменной B и содержимое АК равно 35. Любая *следующая* команда будет работать с содержимым АК, равным 35 (если это не команда LD).

Булевы операторы. *Оператор* **S** (Set) *устанавливает* в *true* указанный *операнд*, если *текущий* результат имеет булево значение *true*, и ничего не делает, когда содержимое АК равно *false*. *Оператор* **R** (Reset) *сбрасывает* указанную *булеву переменную* в *false*, если АК имеет значение *true*, и ничего не делает в противном случае. Команды S и R не имеют модификаторов.

Модификаторы операций: **N** – булево отрицание; **(** - отложенная операция; **C** – условная операция. "**N**" - булево отрицание операнда. "**(**" - "открывающая скобка" означает, что выполнение команды должно быть отложено до тех пор, пока не встретится оператор "закрывающая скобка". Это обусловлено тем, что существует только один регистр-аккумулятор. Поэтому для изменения порядка выполнения команд некоторые операции следует отложить. "**C**" - означает, что прикрепленная к нему команда выполняется только в том случае, когда в АК булево значение *true*. Может использоваться совместно с идентификатором N, для указания того, что команда будет выполняться, когда АК имеет булево значение *false*.

Пример: *разработка программы на различных языках* (*IL, ST, FBD*)

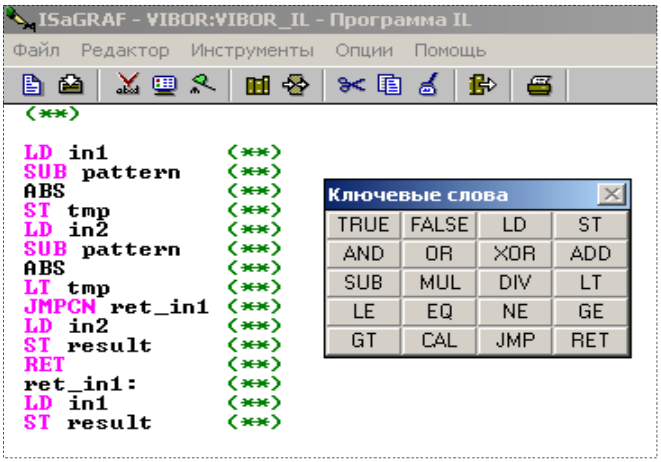
Разработать *программу*, осуществляющую *выбор одного из двух чисел* (*in1*, *in2*), *ближайшего к третьему* (*pattern*), и передачу его на выход Q.



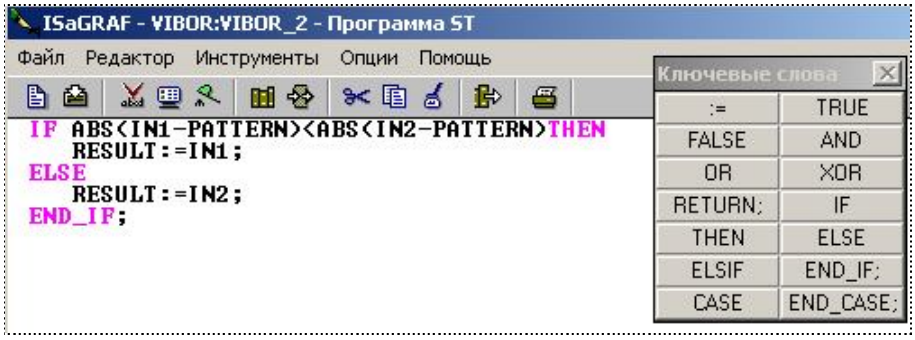
***Решение на языке* IL**

```
        LD  in1          (* загрузка в аккумулятор (АК) значения числа in1 *)
        SUB  pattern     (* вычисление разности  in1 - pattern *)
        ABS              (* определение абсолютного значения разности  *)
        ST  tmp          (* сохранение в tmp  абсолютного значения разности *)
        LD  in2          (* загрузка в аккумулятор значения числа in2 *)
        SUB  pattern     (* вычисление разности  in2 - pattern *)
        ABS              (* вычисление абсолютного значения разности *)
        LT  tmp          (* сравнение  (<) абсолютных значений двух разностей *)
        JMPCN ret_in1    (*условный переход на метку ret_in1, если (АК)=false *)
        LD  in2          (* загрузка в аккумулятор значения числа in2 *)
        GOTO ret_in2     (* безусловный переход на метку ret_in2 *)
ret_in1:  LD  in1        (* загрузка в аккумулятор значения числа in1 *)
ret_in2:  ST  result     (* сохранение в result  наиболее близкого к  pattern числа *)
```

Обозначено: АК – аккумулятор, (АК) – его содержимое; *tmp* – промежуточная переменная.



Решение на языке ST



Решение на языке FBD