

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

С.И. Акунович

СПЕЦИАЛИЗИРОВАННЫЕ ИНФОРМАЦИОННЫЕ СИСТЕМЫ

Рекомендовано

*учебно-методическим объединением Республики Беларусь по
образованию в области информатики и радиоэлектроники в качестве
учебно-методического пособия для студентов учреждений высшего обра-
зования по направлению специальности 1-40 05 01-03
«Информационные системы и технологии
(издательско-полиграфический комплекс)»*

Минск 2014

УДК 004.915 : 655(075.8)

ББК 32.97я73

А

Рецензенты:

кафедра информационных технологий автоматизированных систем БГУИР (кандидат технических наук, доцент,

О. В. Герман);

кандидат технических наук, доцент кафедры автоматизации производственных процессов и электротехники БГТУ

И. Ф. Кузьмицкий

Все права на данное издание защищены. Воспроизведение всей книги или ее части не может быть осуществлено без разрешения учреждения образования «Белорусский государственный технологический университет»

Акунович, С. И.

А Специализированные информационные системы: учеб.- метод. пособие для студентов по направлению специальности 1-40 05 01-03 «Информационные системы и технологии (издательско-полиграфический комплекс)»/ С. И. Акунович. – Минск: БГТУ, 2014. – 120 с.
ISBN 978 – 985 –434 –

Пособие посвящено методологии моделирования и анализа сложных систем логического управления. В нем описаны основы разработки моделей, включая разработанные автором в рамках создания приложения *ГИПЕРСИСТЕМА* средства автоматического преобразования символьных форм описания СЛУ в графически-епрограммы на языках визуального технологического программирования стандарта ИЕС 61131-3. Материал сопровождается рисунками и фрагментами листингов программ.

Пособие предназначено для студентов специальности «Информационные системы и технологии (издательско-полиграфический комплекс)», а также для студентов высших учебных заведений, изучающих программирование как основную дисциплину.

УДК 004.434(075.8)

ББК 32.97я73

ISBN 978 – 985 –434

–©УО «Белорусский государственный технологический университет», 2014

©Акунович С.И., 2014

©УО «Белорусский государственный технологический университет», 2014

Введение

В процессе изучения дисциплины «Специализированные информационные системы» студент должен освоить основы моделирования и анализа сложных систем логического управления (СЛУ), включая разработанные в рамках создания приложения *ГИПЕРСИСТЕМА*[2,3,4] средства автоматического преобразования символьных форм описания СЛУ в графические программы на языках визуального технологического программирования стандарта ИЕС 61131-3.

Системы логического управления находят широкое применение не только на производстве, но и на транспорте, в атомной энергетике, в сетях связи, в информационно-справочных системах, ракетно-космической отрасли, в компьютерах и бытовой аппаратуре и т.д. Проектирование систем логического управления представляет собой сложную проблему, решаемую с использованием богатого арсенала современной дискретной математики, в первую очередь алгебры логики.

В качестве ядра любой системы промышленной автоматизации используются программируемые логические контроллеры (ПЛК), а для специальных систем автоматизации используется огромный спектр микропроцессорных контроллеров (МПК). К каждому из них со стороны объекта автоматизации подключаются датчики и исполнительные органы. Через датчики в ПЛК (МПК) поступает информация о текущем состоянии объекта, а через выходные модули ПЛК (МПК) изменяется состояние управляемого объекта.

В настоящее время в качестве языков алгоритмизации в системах логического управления наиболее часто используются схемы алгоритмов и логические формулы, а в качестве языков программирования в зависимости от типа управляющего вычислительного устройства применяются три разновидности языков: алгоритмические языки высокого уровня, алгоритмические языки низкого уровня и специализированные языки стандарта ИЕС 61131-3.

Технической базой современных систем логического управления служит микроэлектроника. Она позволяет создавать мощные вычислительные комплексы, эффективные сети связи, сложные системы управления производственными процессами, содержащие тысячи и миллионы простейших логических элементов (или команд в управляющих программах).

Например, блок логического управления космического аппарата (КА) является сложной СЛУ, поведение которой на уровне технического задания (ТЗ) описывается в виде блок-схем алгоритмов (около 500) и логических функций (около 5000).

Компоненты СЛУ взаимодействуют с бортовым вычислительным комплексом, наземным комплексом управления, исполнительными механизмами КА и бортовой аппаратурой через внешние соединители (разъемы) (около 50), при этом общее число входов и выходов превышает 2000.

Наряду с логическим управлением в СЛУ используется временное управление с использованием десятков программных таймеров.

Общее количество переменных в такой СЛУ превышает 20000.

Ошибки в логическом описании подобной системы логического управления практически неизбежны.

Между тем специфика микроэлектронной технологии такова, что малейшая ошибка на начальных этапах проектирования, если только она не будет своевременно обнаружена и исправлена, может привести к необходимости полного повторения всего цикла проектирования.

Приложение *ГИПЕРСИСТЕМА* позволяет находить и устранять ошибки в сложных СЛУ средствами моделирования и анализа на самом раннем этапе их разработки - формировании логики управления в техническом задании.

В настоящем учебном пособии представлены не только наиболее современные средства моделирования и анализа СЛУ, разработанные в рамках создания приложения *ГИПЕРСИСТЕМА*, но и оригинальные методы, алгоритмы и программы для преобразования логических моделей в графические программы на языках стандарта IEC 61131-3, разработанные в рамках выполнения курсовых и дипломных проектов.

1. Инструментальная среда разработки встраиваемых приложений ISaGRAF

1.1. Назначение и принципы построения ISaGRAF

Международная электротехническая комиссия (МЭК - IEC) – это международный орган стандартизации, создающий базовые стандарты для последующей адаптации в национальных комитетах. Что касается стандартизации языков, используемых для программирования ПЛК, то эта проблема назрела давно. К концу 80-х десятков базовых концепций на практике был представлен более сотней вариаций. Их унификация сулила ощутимый экономический эффект. Для решения этой проблемы была создана рабочая группа, состоящая из представителей ведущих игроков на рынке автоматизации, которая начала работу.

Комплекс средств ISaGRAF компании ICS Triplex ISaGRAF[5] широко известен как инструмент разработки приложений для программируемых логических контроллеров на языках стандарта IEC 61131-3 и IEC 61499,

который позволяет создавать локальные или распределенные системы управления процессами. Основа технологии – среда разработки приложений ISaGRAFWorkbench и адаптируемая под различные аппаратно-программные платформы исполнительная система ISaGRAFRuntime (Target). В ISaGRAF поддерживаются все пять языков стандарта IEC 61131-3: IL (InstructionList – Список инструкций), ST (StructuredText – Структурированный текст), LD (LadderDiagram – Ступенчатая диаграмма), FBD (FunctionBlockDiagram – Диаграмма функциональных блоков), SFC (SequentialFunctionChart – Последовательная функциональная диаграмма) плюс языки FC (FlowChart, Поточковая диаграмма, Блок-схема) и ANSI C.

Основные особенности и достоинства программного пакета ISaGRAF:

- поддерживает все 5 языков программирования по стандарту IEC61131;
 - поддерживает функции и блоки, написанные на языке Си;
 - обладает разнообразными средствами обмена данными в режимах контроля, загрузки и отладки, используя RS232, RS485 и TCP/IP;
 - характеризуется простой интеграцией с HMI и SCADA программными пакетами посредством протоколов Modbus и OPC;
 - обеспечивает взаимодействие с другими устройствами автоматике, использующими Modbus протокол.
 - Для разработки и исполнения приложений в среде ISaGRAF необходимо наличие двух компонентов:
 - ISaGRAFWorkbench - функционально полная программная среда разработки;
 - аппаратный контроллер, который называется целевым контроллером (Target) и в котором будет исполняться созданное приложение.
- В настоящее время ISaGRAFWorkbench распространяется бесплатно.

Основной принцип работы системы ISaGRAF - синхронизация. Прикладная задача ISaGRAF работает строго по временным циклам, продолжительность которых задается при компиляции задачи.

Программы и процедуры прикладной задачи могут располагаться в трех секциях: Begin, Sequential, End. В начале временного цикла всегда целиком выполняется секция Begin, в конце вся секция End. Оставшееся в промежутке время выделяется для выполнения секции Sequential. Секции Begin и End отвечают за обновление переменных ввода-вывода. Такая схема гарантирует работу в рамках одного временного цикла только с одной копией переменных ввода/вывода.

Программное обеспечение целевой задачи строится вокруг ядра, исполняющего 5 языков стандарта IEC 61131-3 и обращающемуся к библиотеке плат ввода/вывода, функциям пользователя и системному интерфейсу.

Система программирования контроллеров ISaGRAF 6, основанная на фундаменте стандарта IEC 61131-3 и на инновационном стандарте IEC61499 (предназначен для унификации правил создания распределенных приложений и применения функциональных блоков в системах управления), позволяет создавать как специализированные так и универсальные системы управления на основе ПЛК.

Стандарт IEC61499 создавался как основа для создания распределенных приложений для контроллеров и развивал давно принятый инженерным сообществом стандарт IEC 61131-3, а также IEC61158 (Fieldbus). На рис. 1.1 показан пример системы управления, имеющей много устройств, соединенных вместе с помощью управляющей коммуникационной сети.

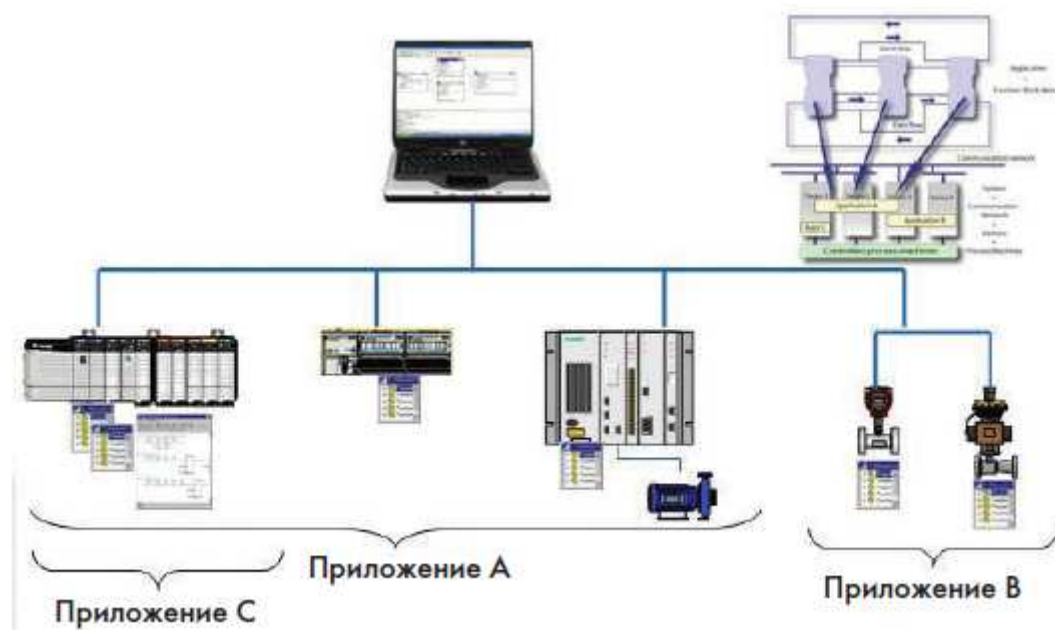


Рисунок 1.1. Пример системы управления с распределенными приложениями

Стандарт IEC61499 определяет распределенную модель как разбиение различных частей промышленного процесса автоматизации и сложной системы управления на модули, называемые функциональными блоками. Эти функциональные блоки могут распределяться и взаимодействовать по коммуникационной сети через множество контроллеров. Приложение становится распределенным путем размещения экземпляров функциональных

блоков на различных ресурсах в одном или более устройствах. Ключевой особенностью функциональных блоков IEC61499 является управление ими с помощью внешних событий, а не только с помощью входных данных.

Стандарт IEC61850 является самой современной разработкой в области коммуникационных технологий для систем управления в энергетике. Он значительно облегчает интеграцию в единую энергетическую систему устройств различных производителей и разных поколений, позволяет сделать это с наименьшими трудовыми и финансовыми затратами. Применяя IEC61850, можно реализовать все функции управления и автоматизации на подстанциях. Многие ученые находят ряд близких концептуальных идей в IEC61850 и IEC61499 и поэтому предлагают использовать инструментальные средства, поддерживающие IEC61499, для реализации подходов, предлагаемых в IEC61850.

Приложения в ISaGRAF 6 состоят из виртуальных машин, работающих на различных аппаратных платформах, называемых исполнительными узлами (targetnodes). Процесс разработки заключается в создании проекта, состоящего из устройств (devices), представляющих собой узлы с одним или несколькими экземплярами ресурсов. Проекты могут разрабатываться, используя различные языки программирования, включая языки стандарта IEC 61131-3. После этапа разработки ресурсы компилируются в TIS-код («targetindependentcode») или в программу на языке Си.

1.2. Инженерно-ориентированные языки программирования

Языки программирования стандарта IEC 61131-3 являются инженерно-ориентированными и не требуют наличия навыков работы с языками высокого уровня. В качестве основных концепций инструментальной среды ISaGRAF можно отметить следующие:

- Архитектура проекта представлена в виде иерархического дерева программ, каждая из которых описывается на одном из языков;
- Программы разделены на логические секции согласно цикличности исполнения.

В зависимости от типа решаемой задачи проектировщик пользователь выбирает тот или иной язык описания программ, составляющих проект. Например, для реализации логических задач последовательного циклического управления (алгоритмов управления) наиболее подходит графический язык SFC. Логические функции представляются и моделируются (вычисляются) в наглядной графической форме на языке LD. Контуры аналогового регулирования могут быть успешно описаны средствами языков LD и FBD. А текстовые языки IL и ST позволяют разработчику решить другие задачи, требующие более развитого математического аппарата.

В силу того, что общепринятого подхода к программированию ПЛК не существовало (и не существует до сих пор), было принято компромиссное решение – включить в стандарт (IEC 61131-3) наиболее широко используемые языки, назначение которых представлено ниже.

SFC (SequentialFunctionChart) – графический язык, используемый для описания алгоритма в виде набора связанных пар: шаг (step) и переход (transition). Шаг представляет собой набор операций над переменными. Переход – набор логических условных выражений, определяющий передачу управления к следующей паре шаг - переход. По внешнему виду описание на языке SFC напоминает хорошо известные логические блок-схемы алгоритмов и имеет возможность распараллеливания алгоритма.

LD (LadderDiagram) – графический язык, стандартизованный вариант класса языков релейно-контактных схем для описания логических выражений. Из-за своих ограниченных возможностей язык дополнен разнообразными средствами: таймерами, счетчиками и т.п.

При переходе на ПЛК язык обладает вполне объяснимыми преимуществами, т.к. снимает психологические проблемы переучивания персонала.

Типовой пример использования – реализация аварийных блокировок системы, включающая преимущественно логические сигналы. Вполне приемлем в случаях, когда задачей предполагаются частые коррекции несложного логического алгоритма неквалифицированным (с точки зрения программирования) персоналом (ремонтники, механики и т.п.).

FBD (FunctionalBlockDiagram) – графический язык, по своей сути похожий на LD. Вместо реле в этом языке используются функциональные блоки. Алгоритм работы некоторого устройства, выраженный средствами этого языка, напоминает функциональную схему электронного устройства: элементы типа “логическое И”, “логическое ИЛИ” и т.п., соединенные линиями. Язык FBD обладает характерной для метафорических языков легкостью начального изучения. Предоставляет достаточно естественную возможность работы с аналоговыми переменными и минимальные средства структуризации (новые функциональные блоки можно компоновать, используя уже существующие). Языку присущ логический параллелизм. Средства синхронизации достаточно естественны для языка.

Типовой пример использования – алгоритмы регулирования, обработка аналоговых сигналов. В качестве пользователей предполагаются специалисты в области автоматического регулирования с привлечением квалифицированных системных программистов в сложных случаях.

ST (StructuredText) – текстовый высокоуровневый язык общего назначения, по синтаксису ориентированный на Паскаль. Обычно используется совместно с SFC. SFC+ST – это, пожалуй, наиболее мощное и универсаль-

ное средство из состава языков МЭК 61131-3. Графика языка SFC облегчает изучение языка. Программирование операций с аналоговыми и логическими переменными достаточно комфортно за счет использования текстового языка ST. Управление потоком команд не вызывает проблем. Существенное преимущество подхода – событийность, естественным образом поддерживаемая через механизм “шаг-переход”. Отладка программ может быть облегчена визуальной трассировкой потока управления. По сравнению с LD и FBD подход SFC+ST проигрывает в удобстве программирования параллелизма алгоритма и, соответственно, более подходит для программирования линейных алгоритмических последовательностей.

Типовой пример использования – совмещенные алгоритмы логического и аналогового управления. В качестве пользователей предполагаются программирующие специалисты, совмещающие знание алгоритмических языков программирования и специфики автоматизируемого технологического процесса.

BISaGRAF 6 реализована поддержкой нового графического языка SAMA (Scientific Apparatus Makers-Manufactures Association). Язык SAMA представляет собой специальный вид функциональных диаграмм управления, широко применяемых, например, в области энергоснабжения. Эти диаграммы используются для описания и документирования стратегий управления объектами, позволяют легко представлять простые вычислительные функции, такие как сумматор, верхний или нижний ограничитель и блоки PID-регулирования. В ISaGRAF 6 язык SAMA реализован на базе ранее описанного языка FBD.

Таким образом, один проект может содержать несколько различных программ, реализованных на различных языках, наиболее удобных для представления конкретных алгоритмов, которые будут скомпилированы в один исполняемый программный модуль.

Кроме того, у разработчиков всегда есть возможность подключить к проекту свои готовые программные модули и функциональные блоки, написанные на языке Си. Все языки оперируют локальными и глобальными переменными булевого, аналогового, временного и строкового типов.

Программная среда ISaGRAF содержит графический отладчик и имитатор системы исполнения приложения на ПК.

Возможна также отладка в целевом контроллере назначения, при этом наблюдение за ходом процесса отладки ведется через списки переменных.

Возможен режим наблюдения за исполнением программы через Internet, например, используя подключение к контроллеру модема.

Программа ISaGRAF поддерживает непрерывный и пошаговый режимы отладки с установками контрольных точек и трассировкой переменных.

При внимательном знакомстве с этим программным пакетом разработчик убеждается в том, что в нем нет ничего лишнего, все операции производятся быстро и интуитивно понятны. С помощью пакета ISaGRAF и моделей контроллеров, оснащенных сетевым адаптером Ethernet, пользователь может создать реально функционирующий проект WEB-автоматизации.

В этом случае, используя модемное соединение, возможна удаленная отладка и загрузка проектов в целевой контроллер назначения через Internet.

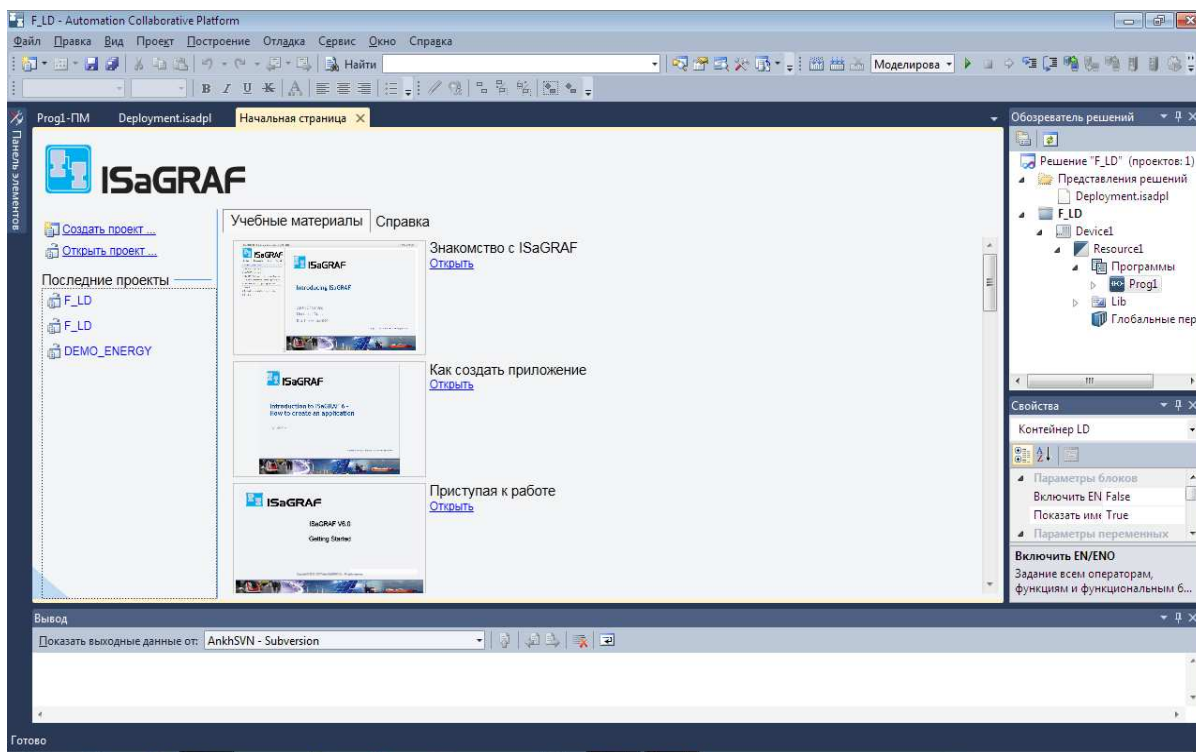
1.3. Создание проектов в приложении ISaGRAF

Для создания проекта (решения) в приложении ISaGRAF необходимо выполнить ряд этапов.

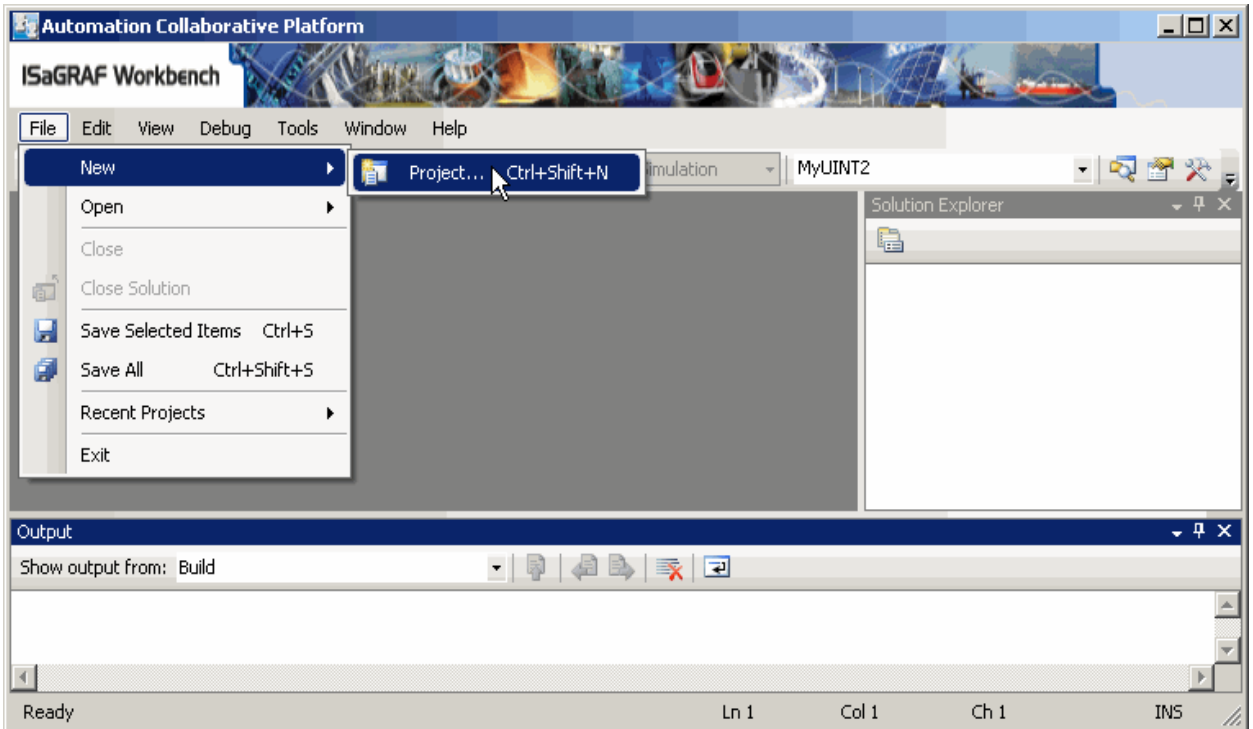
1.3.1. Запуск приложения

Запустите приложение ISaGRAF.

а) Главное окно приложения ISaGRAF имеет вид:

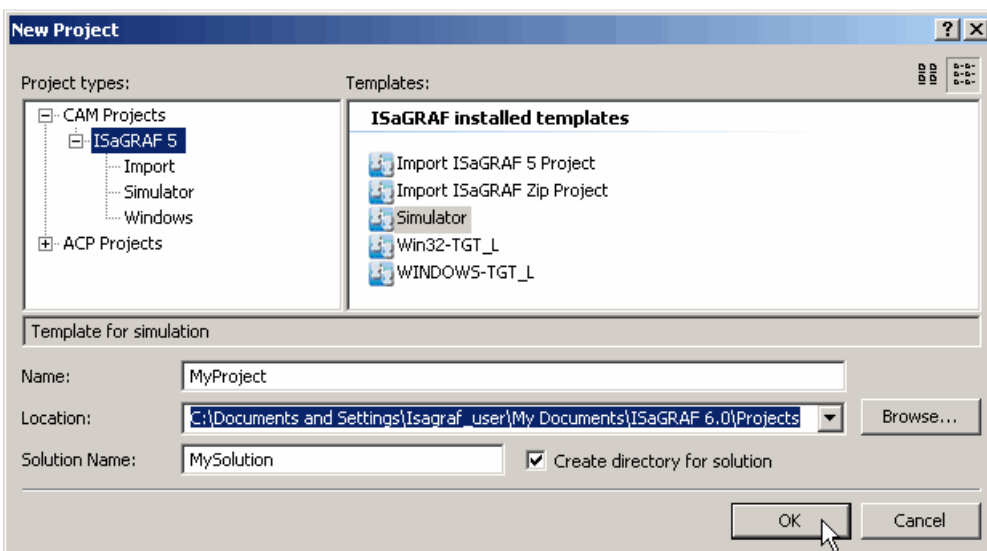


б) Из меню File (Файл), выберите **New**, затем **Project**.



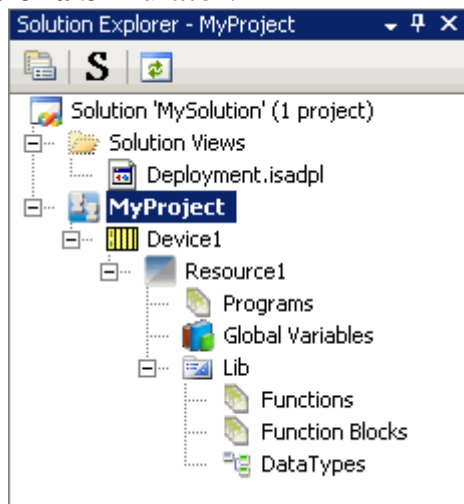
с) В окне NewProject (Новый проект), выберите тип проекта **ISaGRAF5** и шаблон **Simulator** (Симулятор), затем определите имя и место на диске для проекта. Кроме того, определите имя для решения и выберите *Createdirectoryfornewsolution* (Создать каталог для нового решения).

Проекты являются частью существующих или новых решений, решение может содержать множество проектов.



1.3.2. Задание имен устройству и ресурсу

В Solution Explorer (Обозреватель решений), раскройте узел проекта и дайте имя устройству (Device) и ресурсу (Resource), созданному как часть шаблона Simulator.

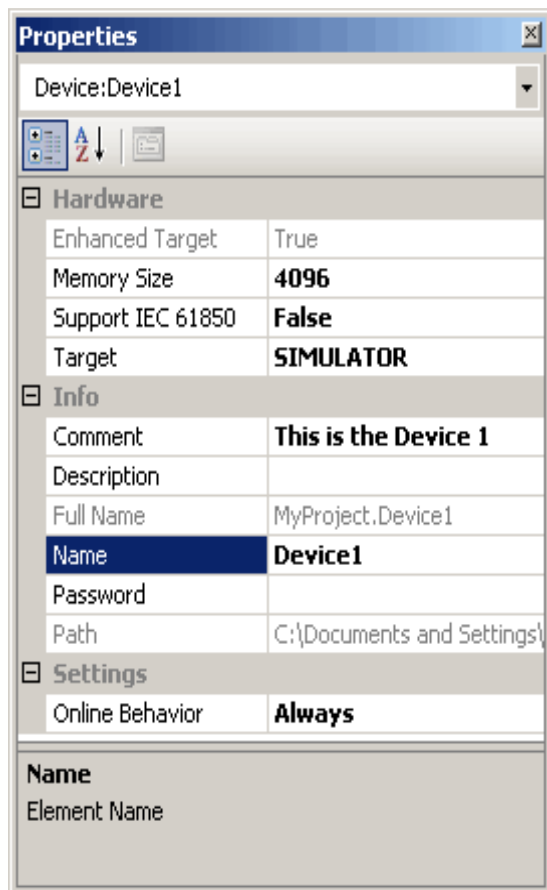


1.3.3. Определение свойств для устройств и ресурсов.

Определите свойства для устройств и ресурсов.

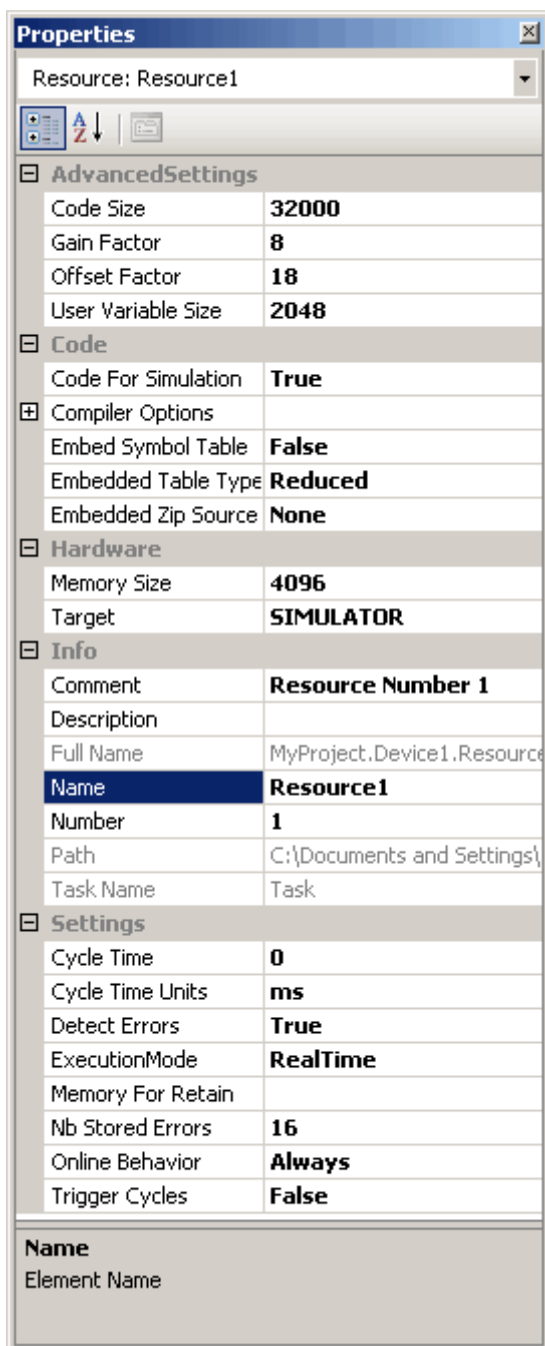
a) В Solution Explorer (Обозреватель решений) выберите устройство, затем в меню View (Вид) выберите **Properties Window (Окно свойств)** или нажмите F4.

b) В Properties window (Окно свойств) определите имя, комментарий, размер памяти (memorysize), и тип целевой системы (targettype).



c) В SolutionExplorer (Обозреватель решений) выберите ресурс (resource), затем в меню View (Вид) выберите **Propertieswindow (Окно свойств)** или нажмите F4.

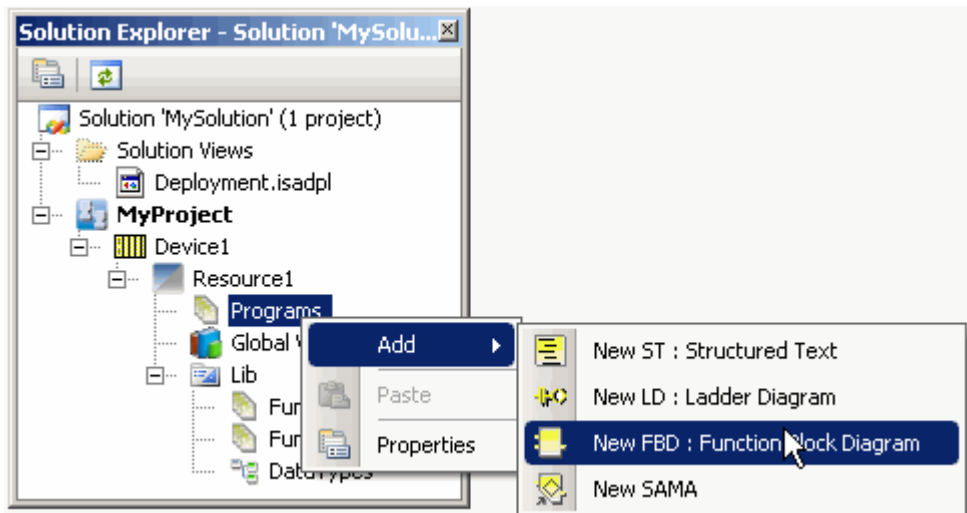
d) Настройте режим выполнения (executionmode), размер памяти (memorysize), имя (name), число сохраняемых ошибок (number (nb) ofstorederrors), номер ресурса (resourcenumber), комментарий (comment), описание (description) и тип целевой системы (targettype).



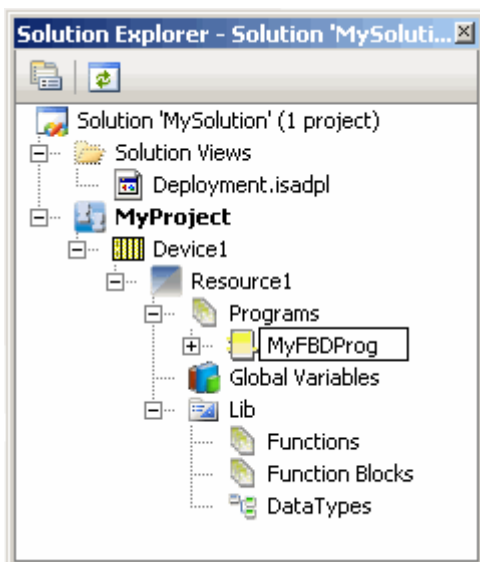
1.3.4. Добавление программы

В Solution Explorer (Обозреватель решений) добавьте программу и определите её имя.

а) Кликните правой кнопкой на узле Programs (Программы), выберите **Add (Добавить)**, затем выберите необходимый язык программирования.



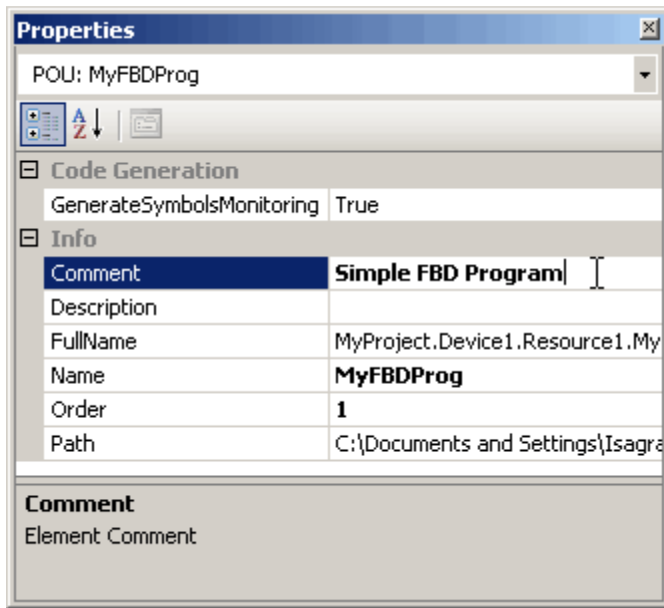
б) Кликните правой кнопкой на узле Programs (Программы), выберите **Rename (Переименовать)**, задайте имя.



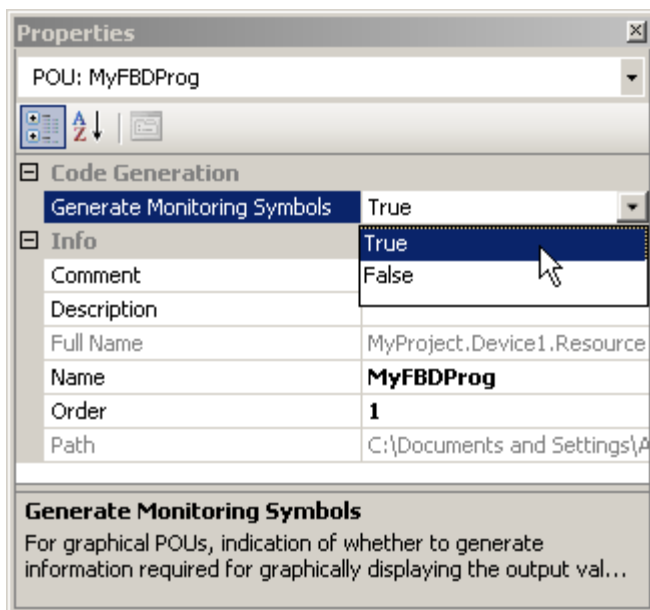
1.3.5.Задание свойств программы.

В Propertieswindow (Окно свойств) задайте свойства программы.

а) Для *Comment (Комментарий)* введите комментарий.



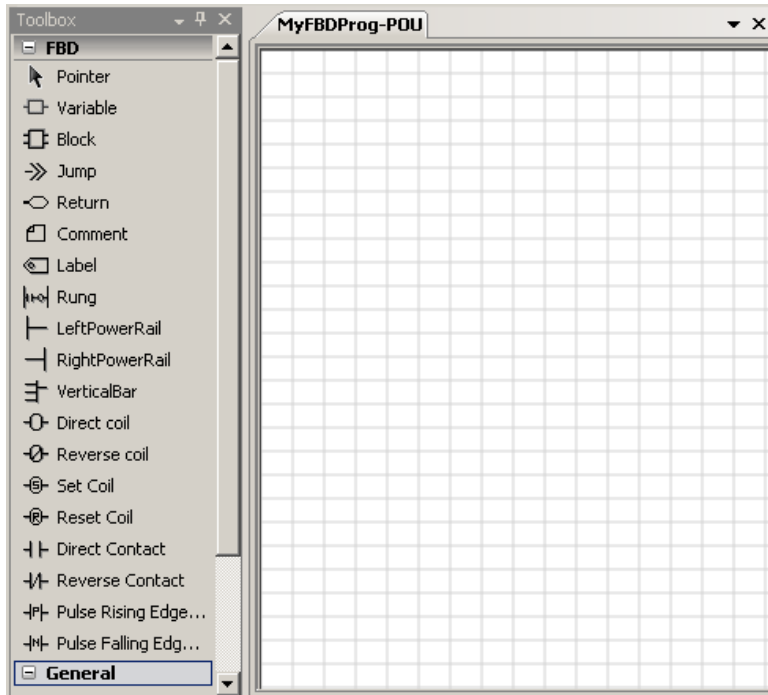
б) Для *GenerateSymbolsMonitoring* (Генерировать символы мониторинга), раскройте список и выберите **True**.



1.3.6. Добавка содержимого программы.

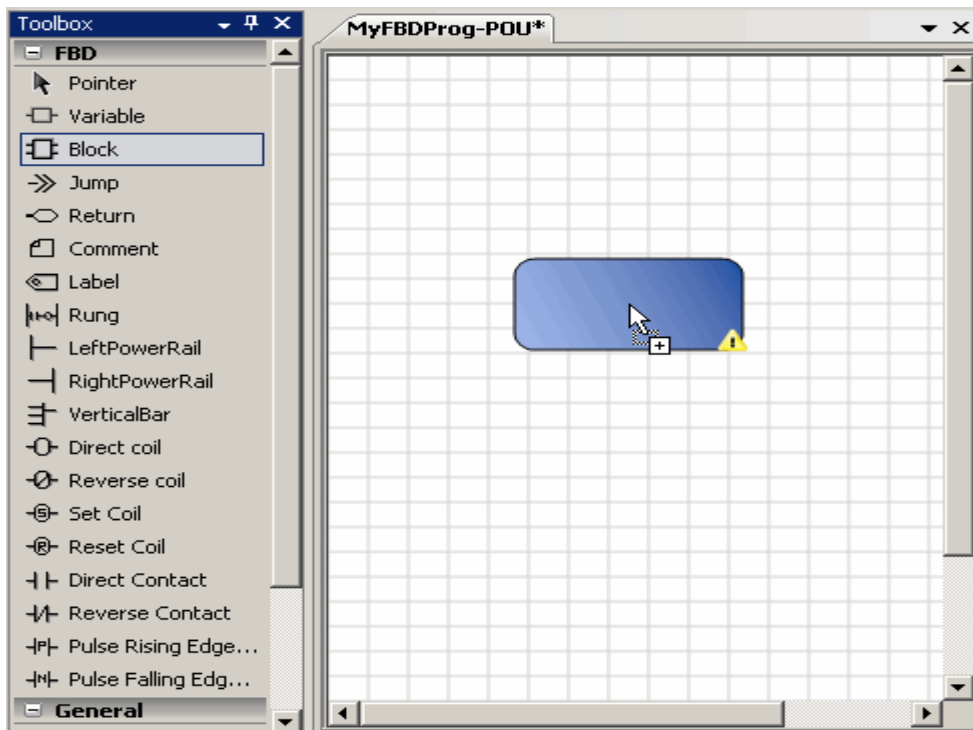
На рабочем пространстве добавьте содержимое программы.

а) В Solution Explorer (Обозреватель решений) кликните дважды на узле программы, затем в меню View (Вид) выберите **Toolbox (Панель элементов)**.



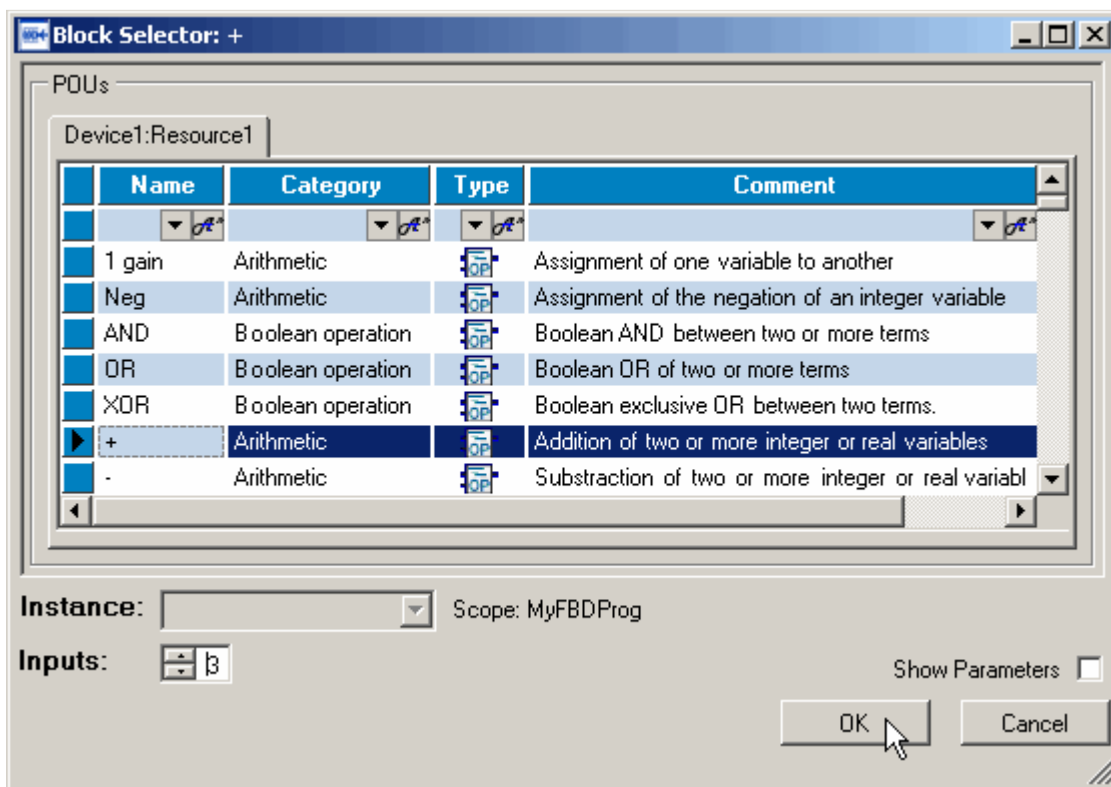
б) Добавьте блок на рабочее пространство.

и) С Toolbox (Панель элементов), перетащите элемент **Block** (Блок) на рабочее пространство.



Отобразится BlockSelector (Окно выбора блока).

ii) В *POUs-списке* выберите необходимый POU, определите instance (экземпляр) и количество входов, затем нажмите **ОК**.



Блок отобразится на рабочем пространстве.

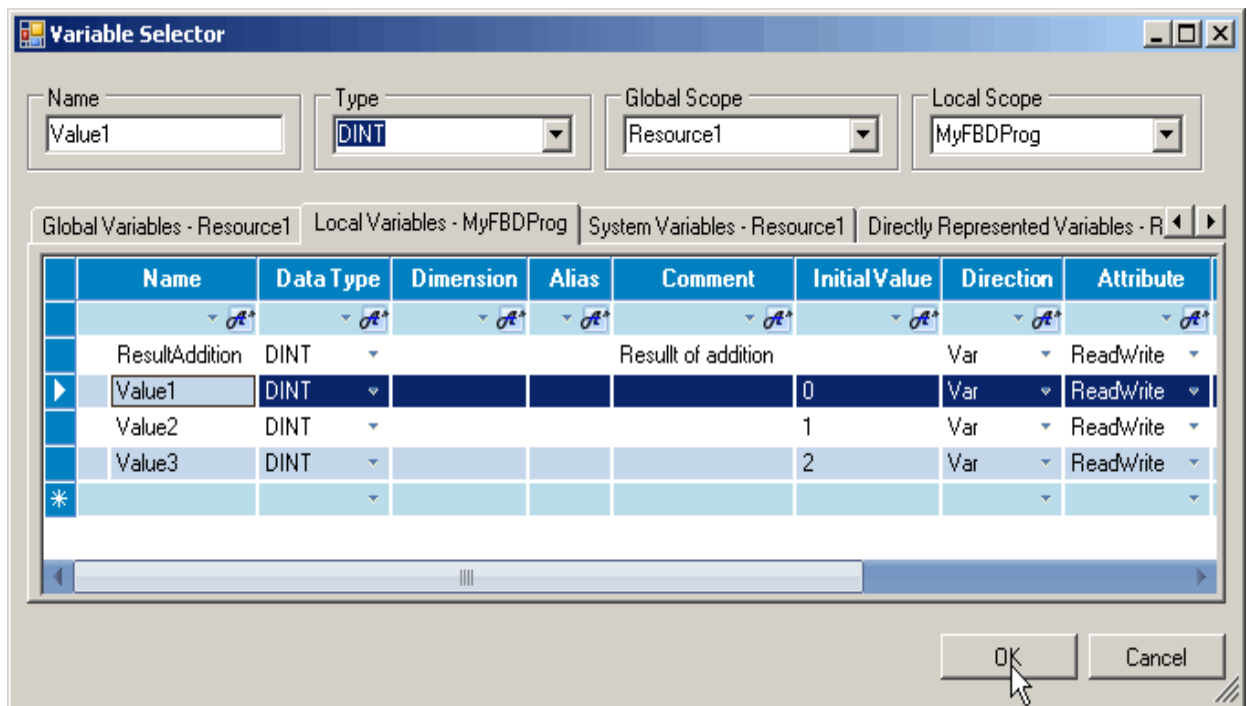
iii) Добавьте больше блоков на рабочее пространство, повторяя шаг б).

с) Добавьте переменные на рабочее пространство.

i) С Toolbox (Панели элементов), перетащите элемент variable (вставить переменную) на рабочее пространство.

Отобразится VariableSelector, содержащий пустые списки *Localvariables* (Локальные переменные) и *Global variables* (Глобальные переменные).

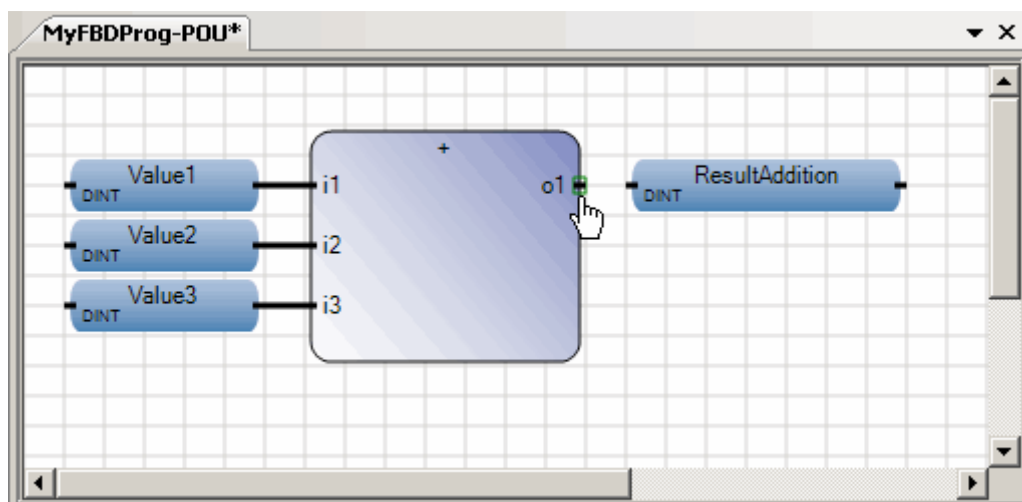
ii) В списке *LocalVariable* (Локальные переменные), введите необходимую информацию в ячейки, выберите переменную, затем нажмите **ОК**.



Переменная отобразится на рабочем пространстве.

iv) Чтобы добавить больше переменных на рабочее пространство, повторите шаг б с).

d) Перетащите связь от выхода ко входу, как показано ниже.

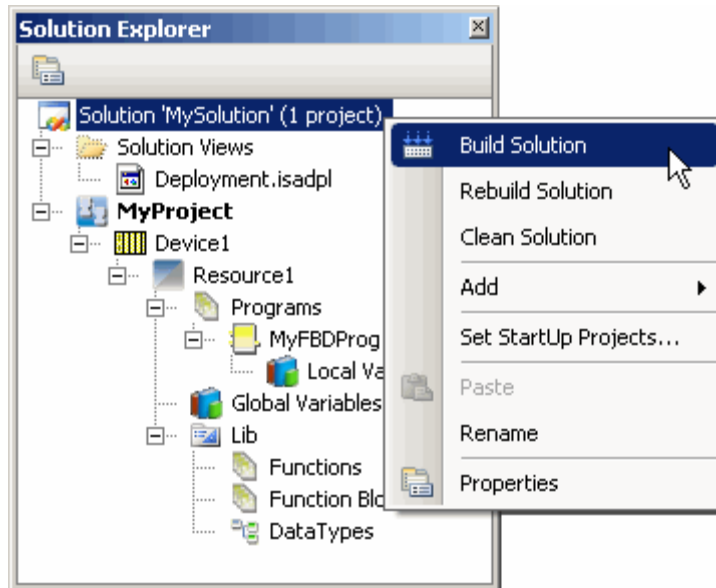


1.1.7. Сборка решения

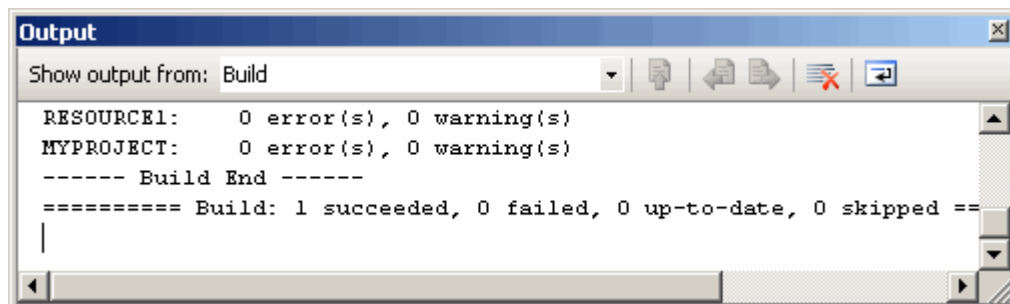
Соберите решение, используя **solutionexplorer**.

Просмотрите полученные ошибки, сообщения и предупреждения (если они есть).

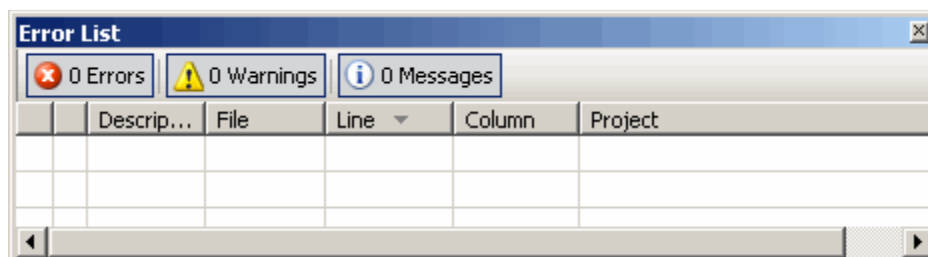
а) Кликните правой кнопкой на узле решения (solution) и выберите **Build Solution (Построить решение)**.



б) Чтобы посмотреть информацию о построении решения, выберите **Output (Вывод)** в меню View (Вид).



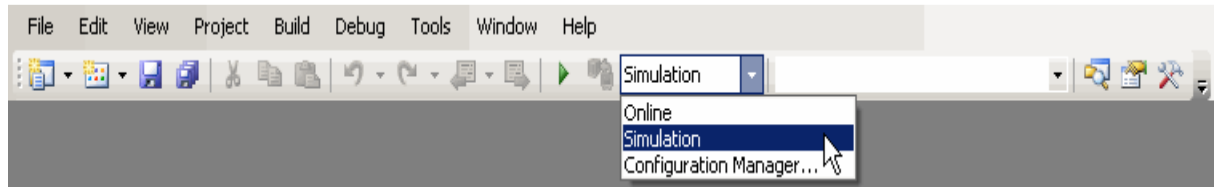
в) Чтобы посмотреть ошибки, предупреждения и сообщения, сгенерированные в процессе построения, выберите **ErrorList (Список ошибок)** в меню View (Вид).



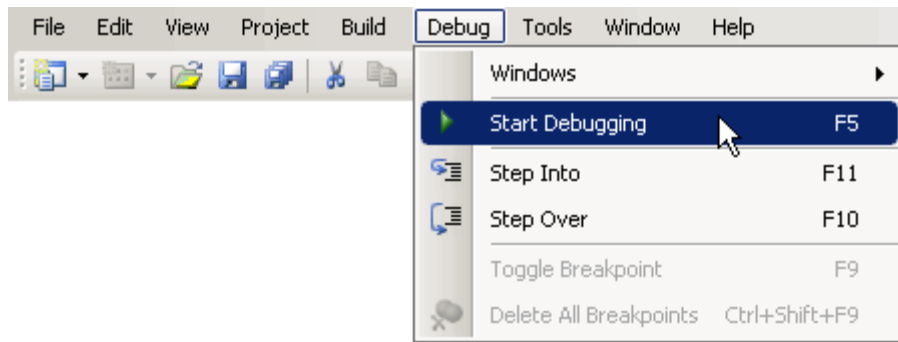
1.3.8. Запуск процесса отладки

Запустите процесс отладки, затем просмотрите программы и словарь.

a) На стандартной панели инструментов, в списке SolutionConfiguration (Конфигурация решения), выберите **Simulation (Моделирование)**.

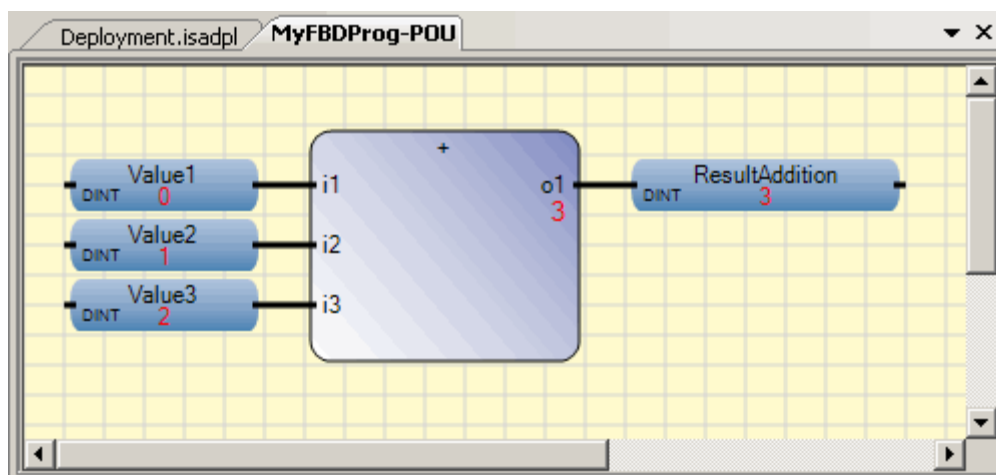


b) В меню Debug (Отладка) выберите **StartDebugging (Начать отладку)** или нажмите **F5**.



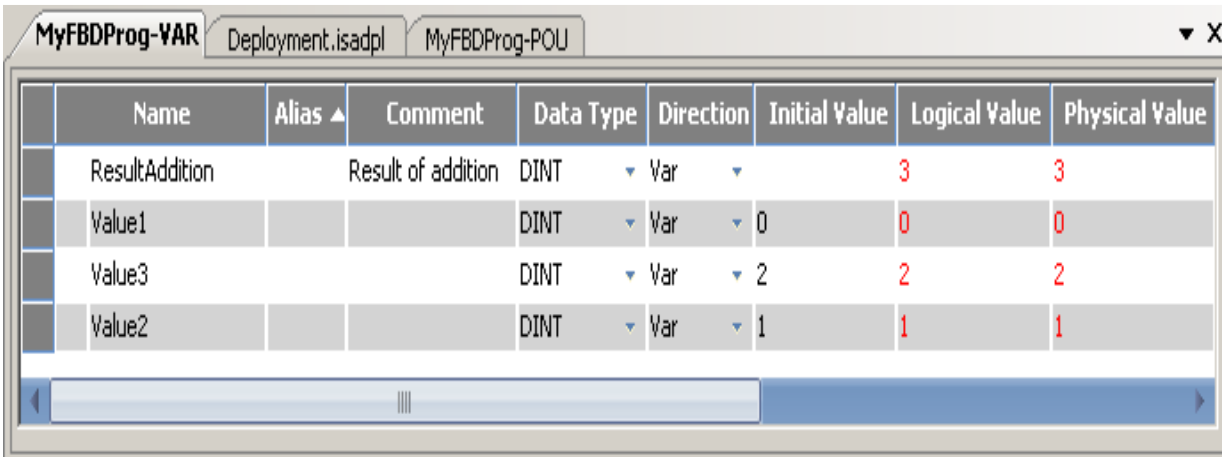
c) В SolutionExplorer (Обозреватель решений) просмотрите программу путем двойного щелчка на узле программы.

Отладочная информация отображается красным цветом.



d) В SolutionExplorer (Обозреватель решений) просмотрите словарь переменных, щёлкнув дважды на узле словаря.

Логические и физические значения переменных отображаются красным цветом.



Name	Alias	Comment	Data Type	Direction	Initial Value	Logical Value	Physical Value
ResultAddition		Result of addition	DINT	Var		3	3
Value1			DINT	Var	0	0	0
Value3			DINT	Var	2	2	2
Value2			DINT	Var	1	1	1

1.3.9. Остановка процесса отладки

Чтобы остановить процесс отладки, выберите **StopDebugging (Остановить отладку)** в меню Debug (Отладка).

2. Разработка проектов СЛУ на языке LD

2.1. Описание языка LD

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

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(NOT).

Графические представления элементов языка LD и их текстовые аналоги в среде разработки IsaGraf 6 приведены на рис. 2.1 – 2.16.

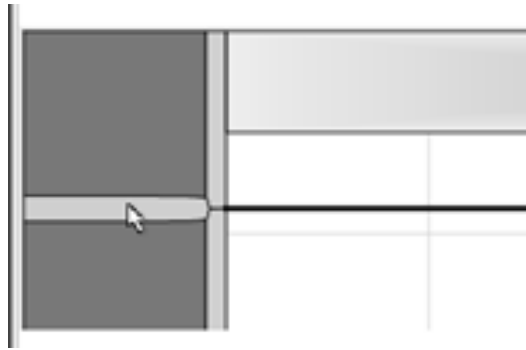


Рис. 2.1. Цепи

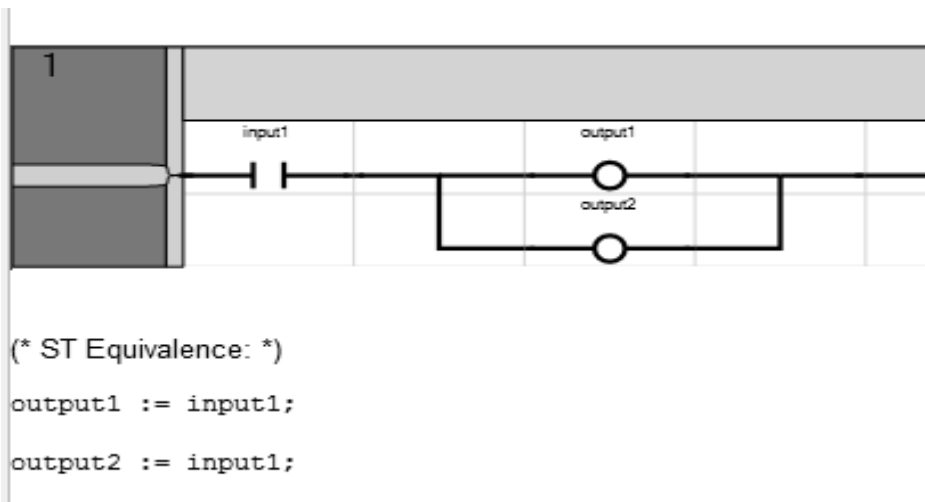


Рис. 2.2. Прямая обмотка

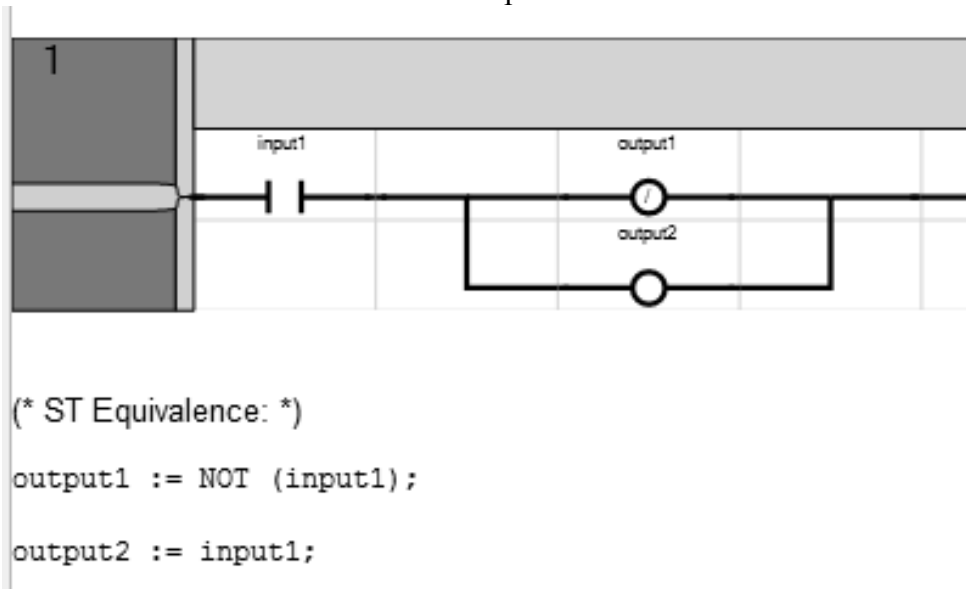


Рис. 2.3. Инверсная обмотка

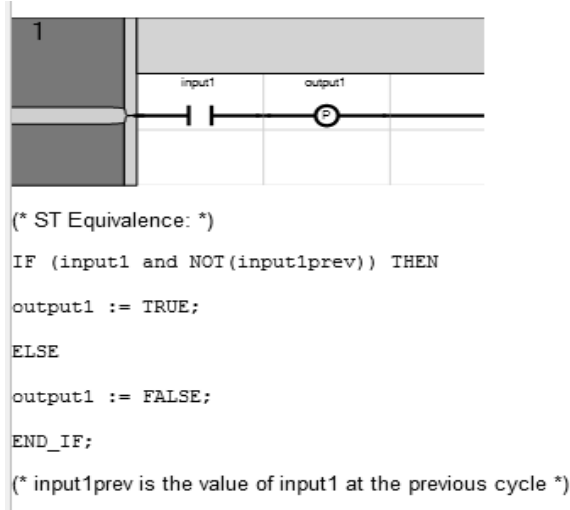


Рис. 2.4. Импульсная обмотка (Передний фронт)

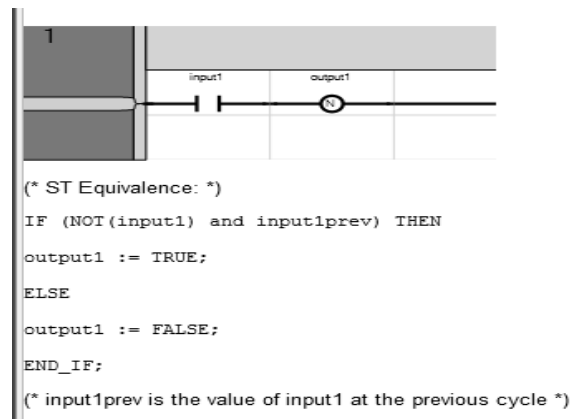


Рис. 2.5. Импульсная обмотка (Задний фронт)

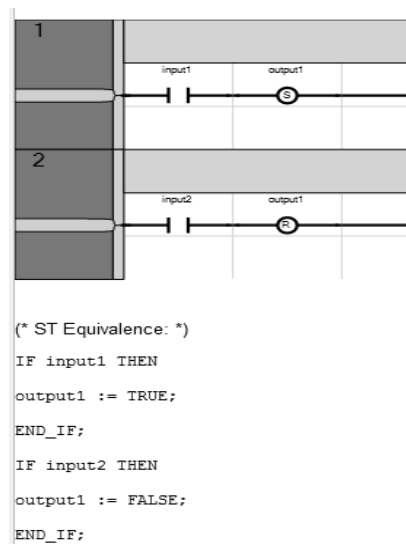


Рис. 14. Включающая обмотка (Set) и Выключающая обмотка (Reset)

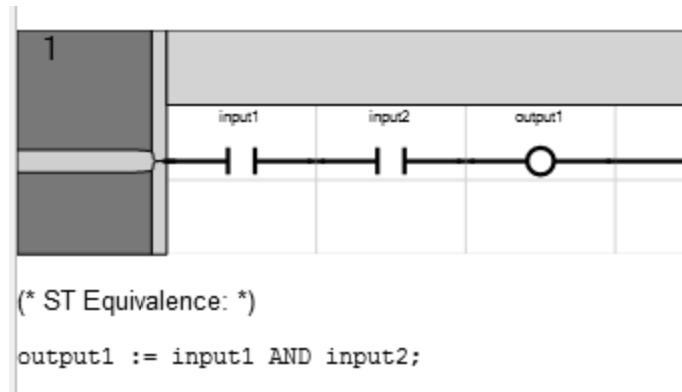


Рис. 2.6. Прямой контакт

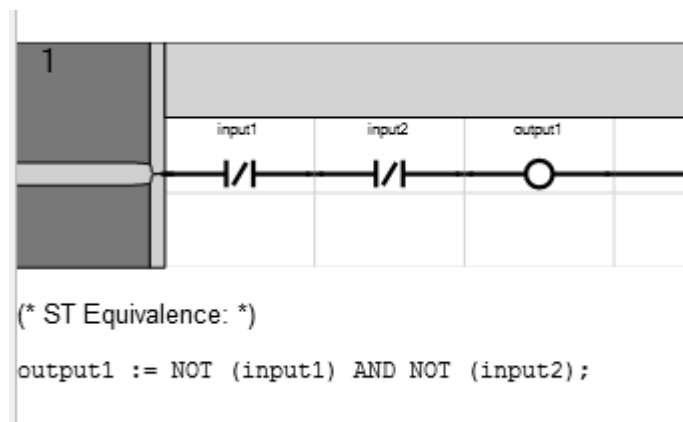


Рис. 2.7. Инверсный контакт

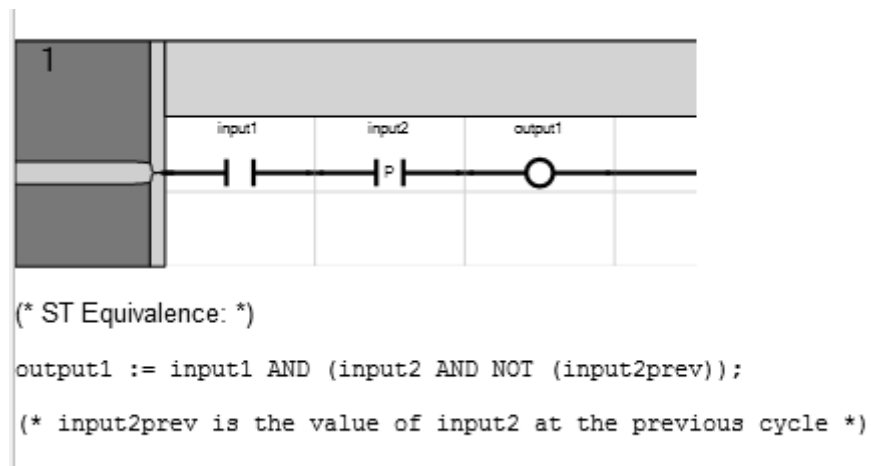
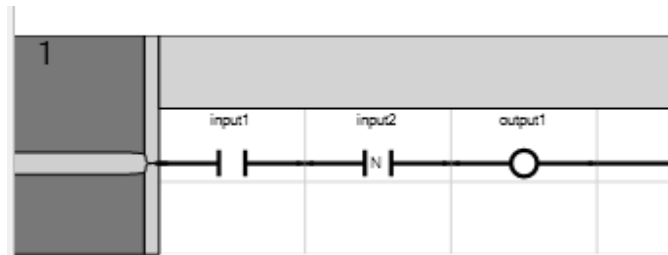


Рис. 2.8. Импульсный контакт (Передний фронт)



(* ST Equivalence: *)

```
output1 := input1 AND (NOT (input2) AND input2prev);
```

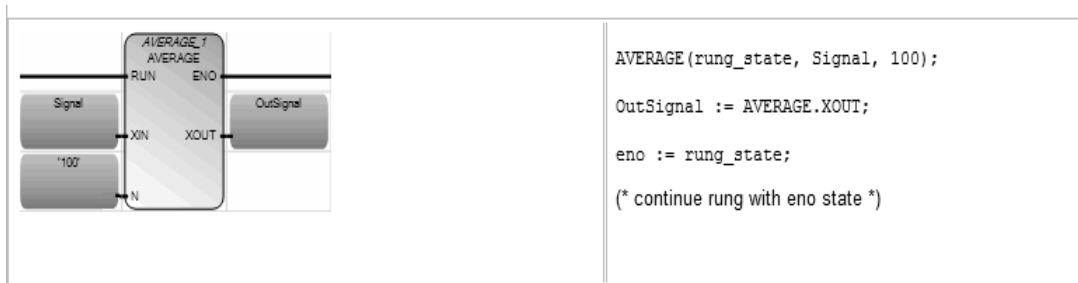
(* input2prev is the value of input2 at the previous cycle *)

Рис. 2.9. Импульсный контакт (Задний фронт)



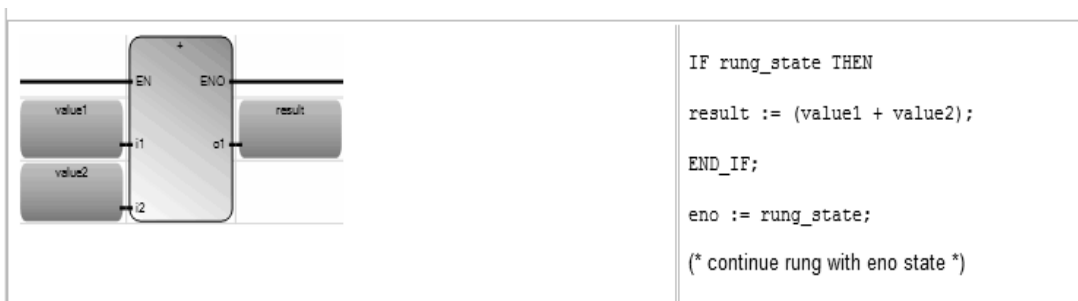
```
IF rung_state THEN
q := (value1 > value 2);
ELSE
q := FALSE;
END_IF;
(* continue rung with q state *)
```

Рис. 2.10. Блоки. EN вход



```
AVERAGE(rung_state, Signal, 100);
OutSignal := AVERAGE.XOUT;
eno := rung_state;
(* continue rung with eno state *)
```

Рис. 2.11. Блоки. ENO выход



```
IF rung_state THEN
result := (value1 + value2);
END_IF;
eno := rung_state;
(* continue rung with eno state *)
```

Рис. 2.12. Блоки. EN и ENO параметры

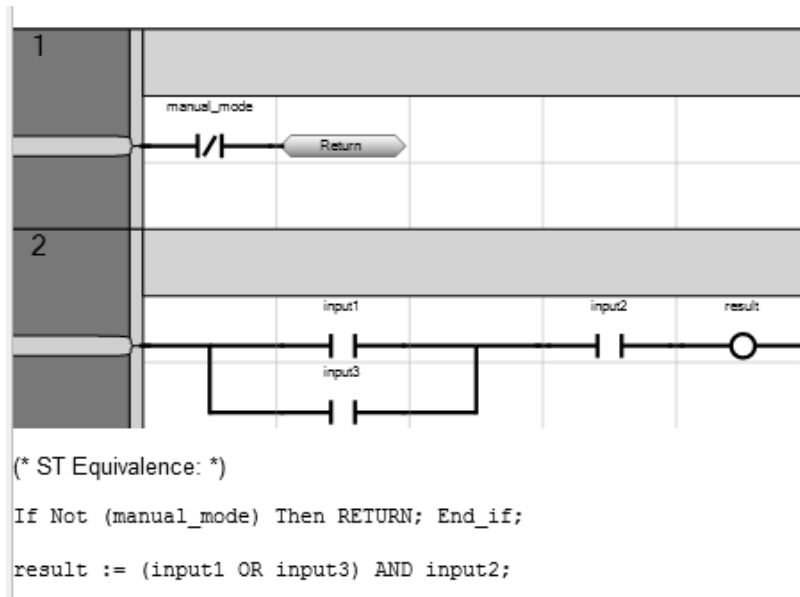


Рис. 2.13. Возвраты

Если линия подключения (слева от символа Return) имеет состояние TRUE, программа завершается – никакая дальнейшая часть диаграммы не выполняется.

Никакое подключение не может быть произведено справа от символа RETURN.

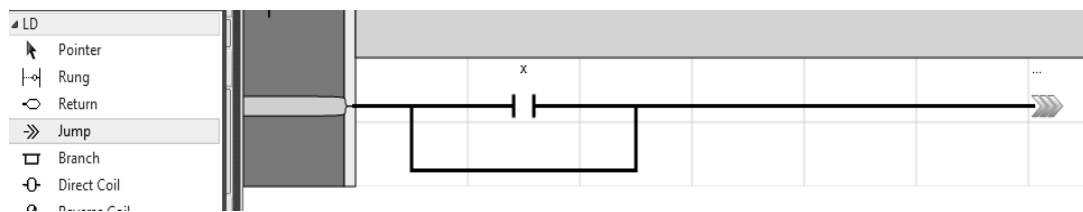


Рис. 2.14. Переходы и метки



Рис. 2.15. Разветвления

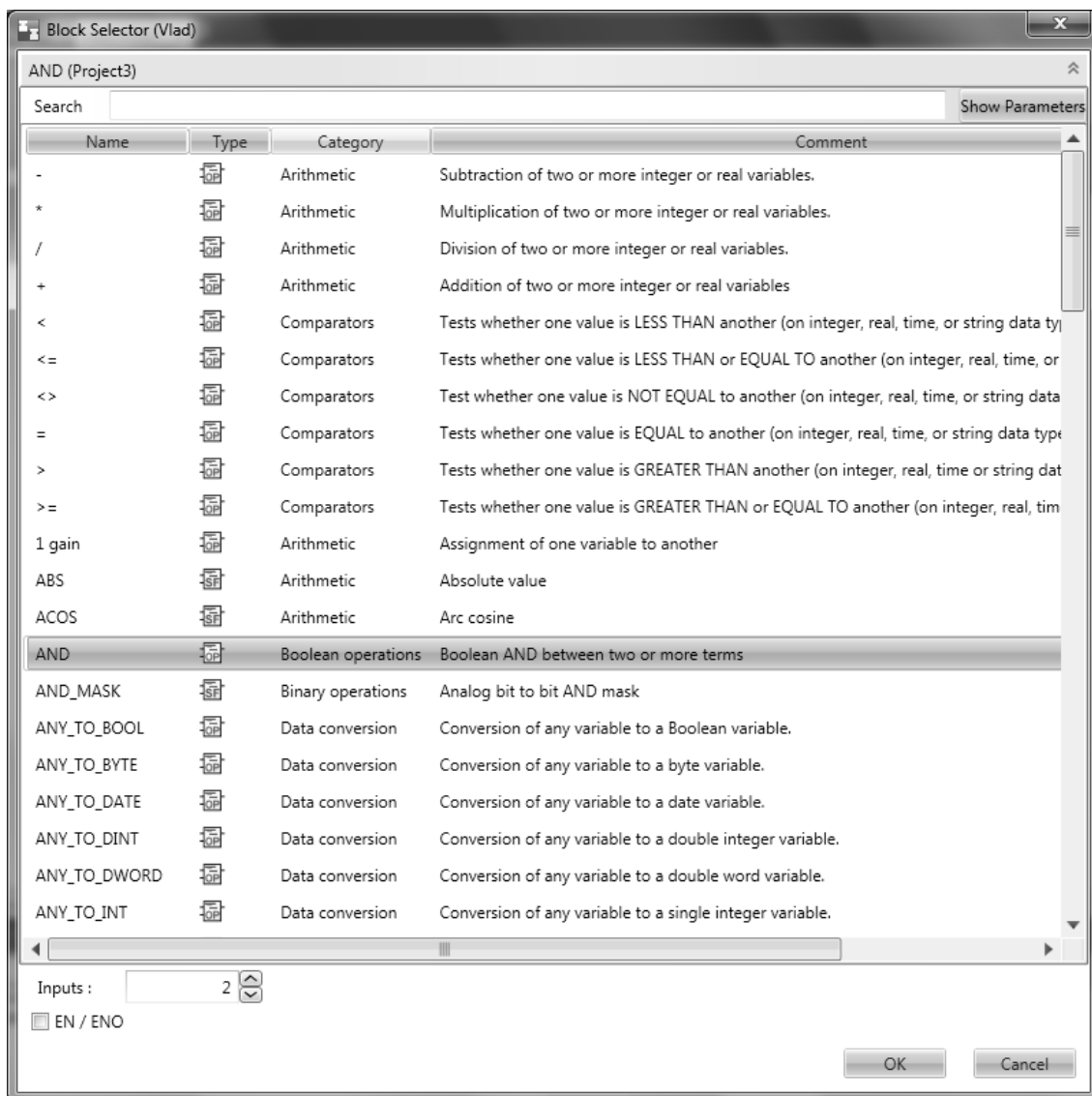


Рис. 2.16. Функции и функциональные блоки

2.2. Разработка программ на языке LD

2.2.1. Исходные данные

Разработать программу на языке LD для моделирования логических функций, представленных в таблице Excel[4] и зависящих от входного контакта X012_28. Выявим ячейки, зависящие от ячейки X012_28 (рис. 2.17)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Входные контакты		Технологические команды		Функции памяти		Выходы		Выходные контакты		Силовые контакты		Сухие контакты		Электронные ключи	
2	Имя	Сост	Имя	Сост	Имя	Сост	Имя	Сост	Имя	Сост	Имя	Сост	Имя	Сост	Имя	Сост
3	X011_15	0	ХВКЛТ	0	ZPBM	0	УП1П1_2	0			X011_1	0	X012_1	0		
4	X011_34	0	ХОТКЛТ	0	ZPKO	0	УП1П1_3	0			X011_3	0	X012_3	0		
5	X011_36	0	ХРБЛКО	0	ZBLVKL	0	УП2П1_2	0			X011_5	0	X012_5	0		
6	X011_17	0	ХРБЛВ	0	ZPOBIBV	0	УП2П1_3	0			X011_7	0	X012_7	0		
7	X012_19	0	ХБЛКО	0	ZBIBVK	0	УП3П1_2	0			X011_9	0	X012_13	0		
8	X012_9	0	ХБЛВМ	0	ZBKIS	0	УП3П1_3	0			X011_11	0	X012_17	0		
9	X012_22	0	ХИСХ	0	ZMTKCEP	0	УВПРД	0			X011_38	0	X014_9	0		
10	X012_28	0	ХИОН	0	ZSTRV	0	УМ1П1_3	0			X011_24	0	X014_18	0		
11	X012_32	0	ХРБЛВК	0	ZPACO	1	УМ3П1_2	0			X011_30	0	X017_38	0		
12	X014_4	0	ХВНКУ	0	ZONBU	0	УБЛВКЛ	0			X015_1	0	X020_4	0		
13	X014_5	0	ХОНКУ	0	ZON	0	УВБИВК	0			X015_3	0	X020_6	0		
14	X016_10	0					УМТКСЭП	0			X018_18	0	X022_28	0		
15	X016_30	0					УРАСО	0			X021_22	0	X022_24	0		
16	X020_30	0					УБИВКП	0								
17							УРОБИВКТМ	0								
18							УВБИВКТМ	0								

Рис. 2.17. Моделирование логических функций

2.2.2. Структура проекта

Создадим проект типа Simulator, в котором будет располагаться 2 программы типа LadderDiagram (рис. 2.18).

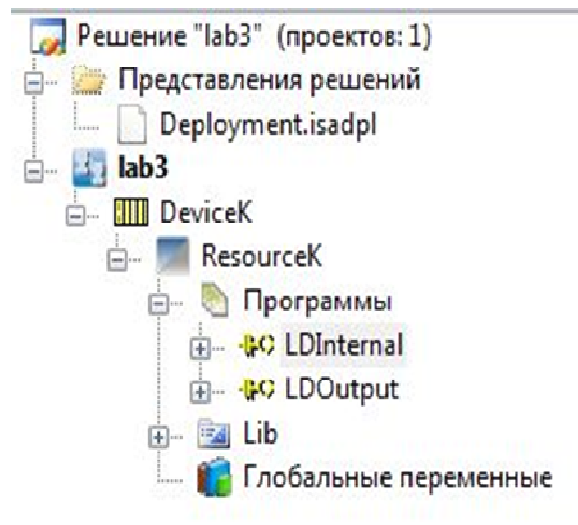


Рис. 2.18. – Структура проекта.

2.2.3. Разработка программ

В программе LDInternal(рис. 2.19) отображены внутренние функции. Функции памяти представлены двумя витками: S – включение, R – выключение.

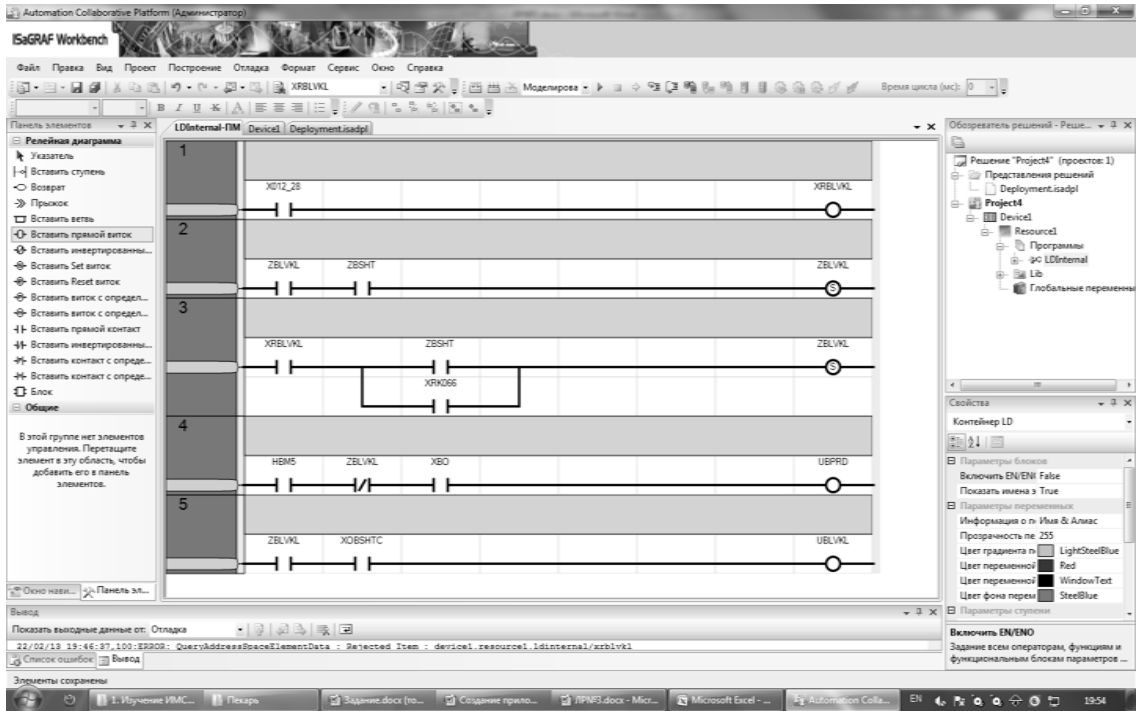


Рис. 2.19. Программа LDInternal

Программа LDOOutput(рис. 2.20) отображает выходные функции:

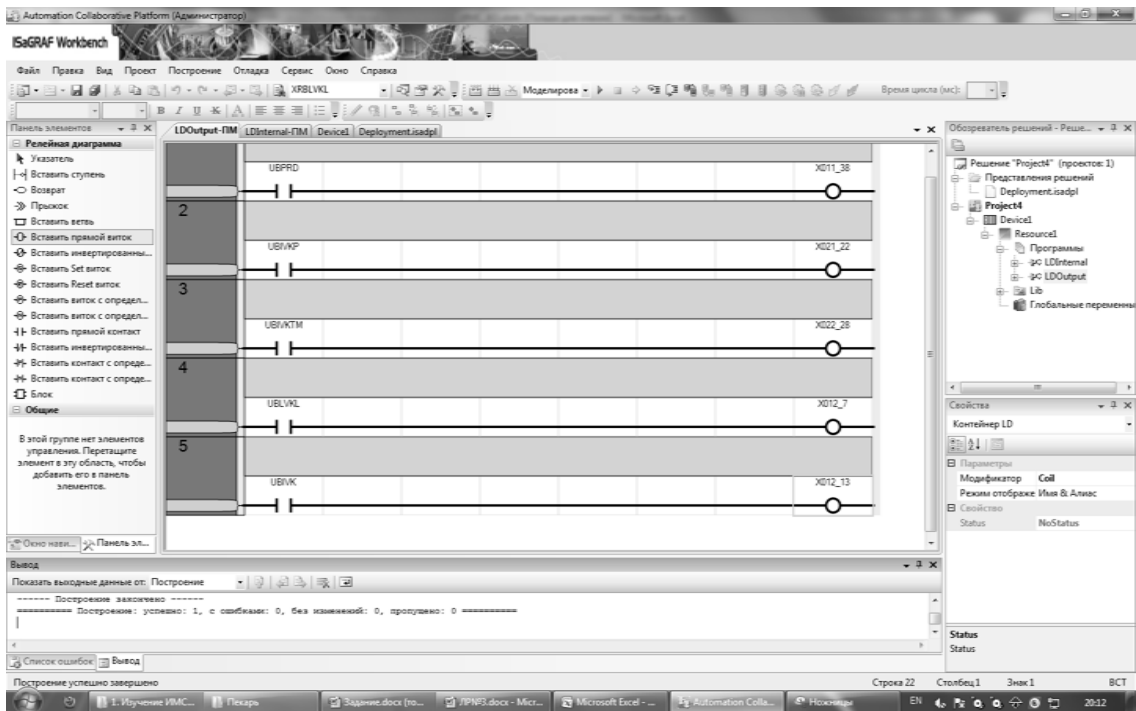


Рис. 2.20. Подпрограмма LDOOutput

2.2.4. Монтрование переменных

Окно устройства ввода-вывода отображает добавленные устройства, на нем также можно добавить новые устройства (рис. 2.21).

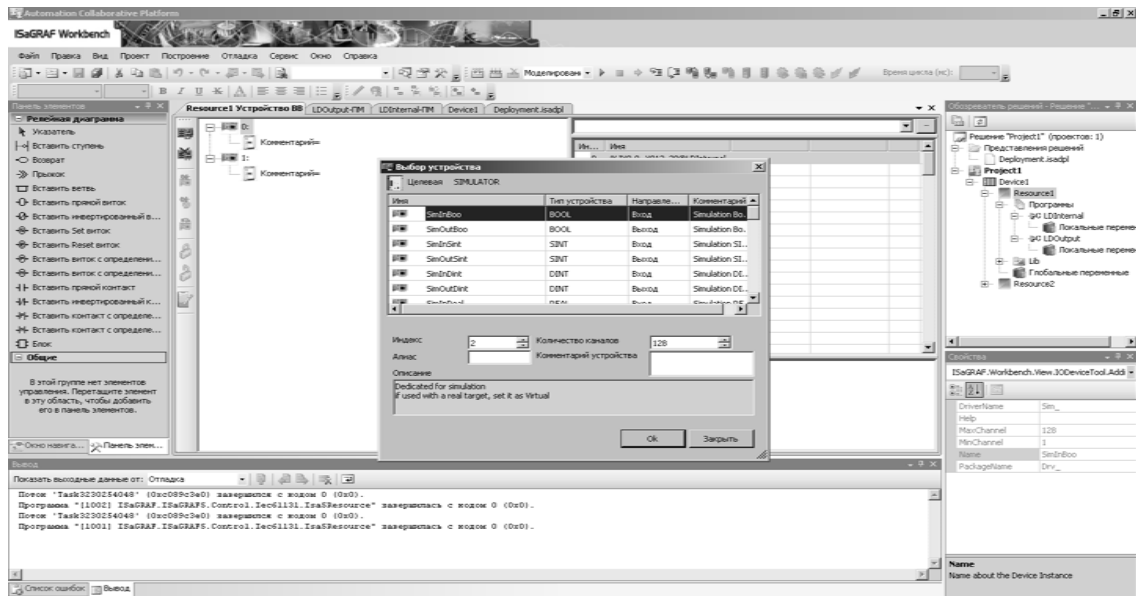


Рис. 2.21. Добавление устройства

На рисунке 2.22 отображены добавленные виртуальные устройства

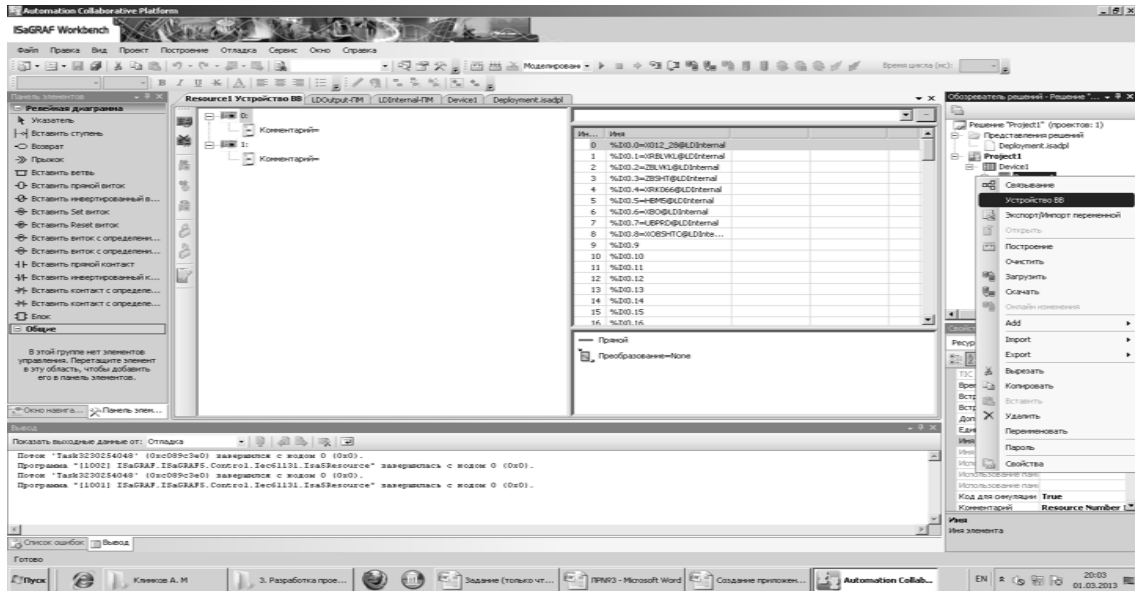


Рис. 2.22. Добавленные устройства

В области справа от таблицы с устройствами отображаются переменные, смонтированные на входы (рис. 2.23).

Инд...	Имя			
0	%IX0.0=X012_28@LDInternal			
1	%IX0.1=XRBLVKL@LDInternal			
2	%IX0.2=ZBLVKL@LDInternal			
3	%IX0.3=ZBSHT@LDInternal			
4	%IX0.4=XRK066@LDInternal			
5	%IX0.5=HBM5@LDInternal			
6	%IX0.6=XB0@LDInternal			
7	%IX0.7=UBPRD@LDInternal			
8	%IX0.8=XOBSHTC@LDInte...			
9	%IX0.9			
10	%IX0.10			
11	%IX0.11			
12	%IX0.12			
13	%IX0.13			
14	%IX0.14			
15	%IX0.15			
16	%IX0.16			

Рис. 2.23. Переменные, смонтированные на входы

Монтирование осуществляется двойным кликом на строку входа и выбором переменной во всплывающем окне (рис. 2.24).

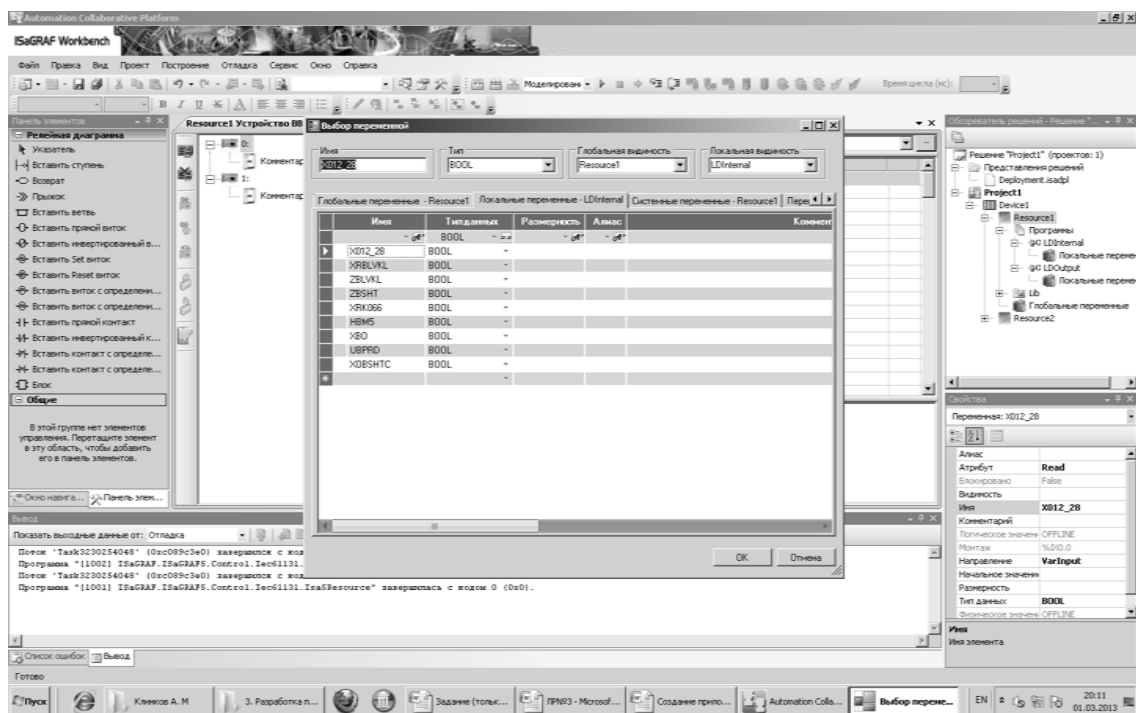


Рис. 2.24. Монтирование переменной

После монтирования задается направление каждой переменной (рис. 2.25).

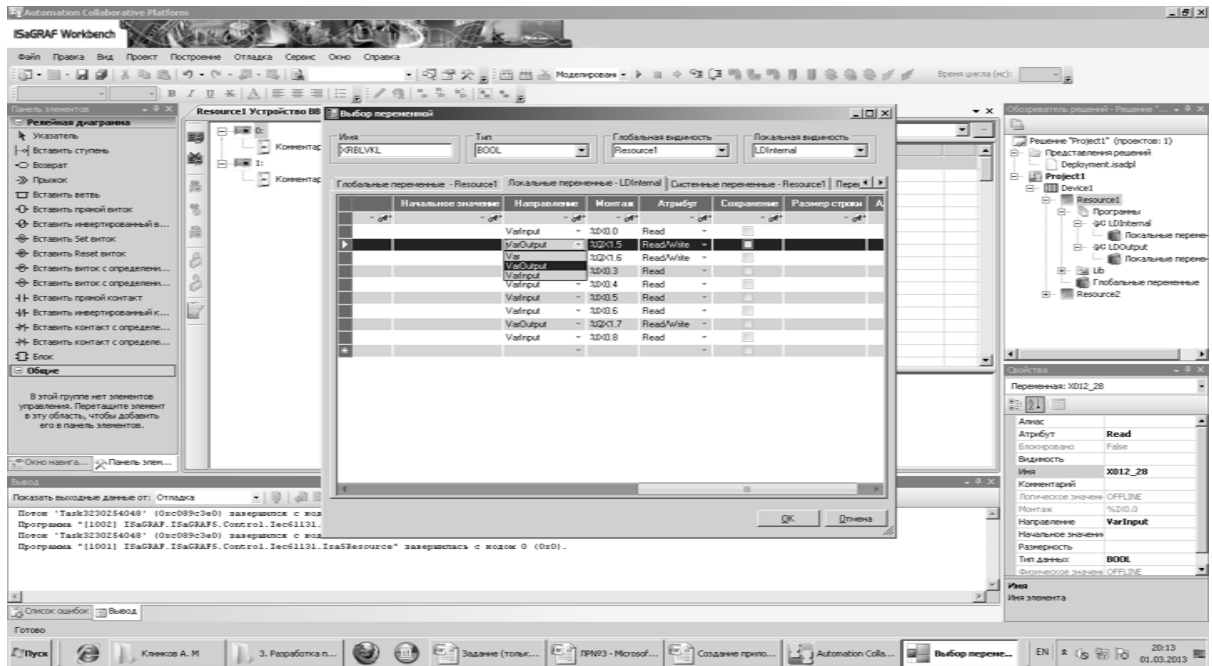


Рис. 2.25 Задание направления переменной

2.2.5. Отладка программ

По завершении действий по монтированию производится отладка, во время которой на входы подаются значения из окна Устройств (рис. 2.26) или в программе LD (рис. 2.27-2.28).

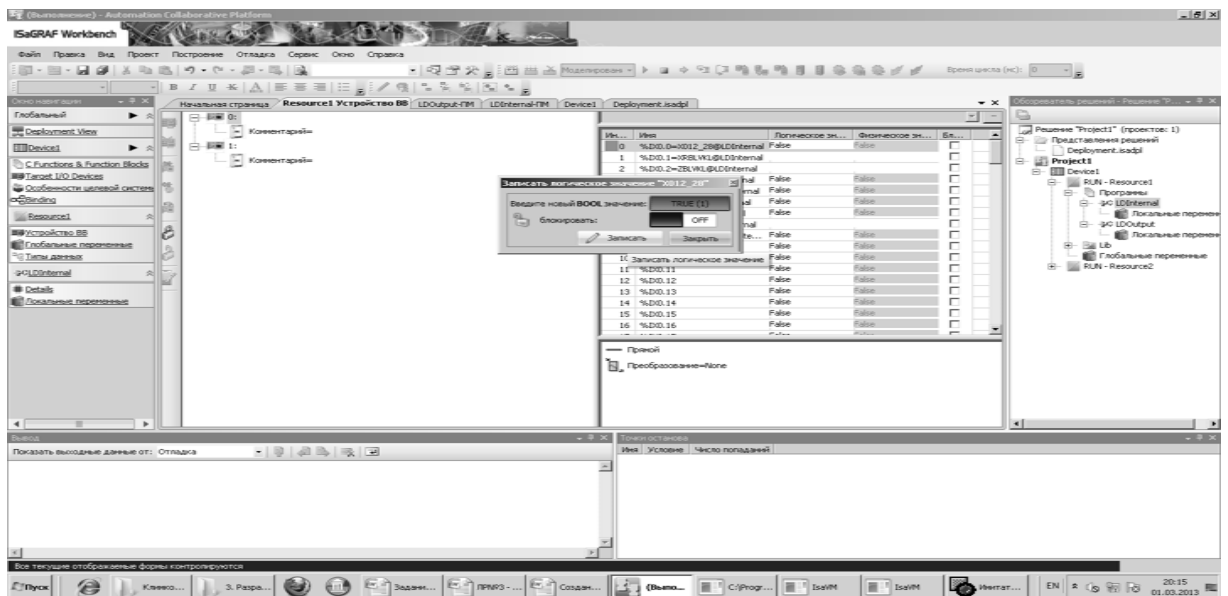


Рис. 2.26. Запись значения переменной через устройство

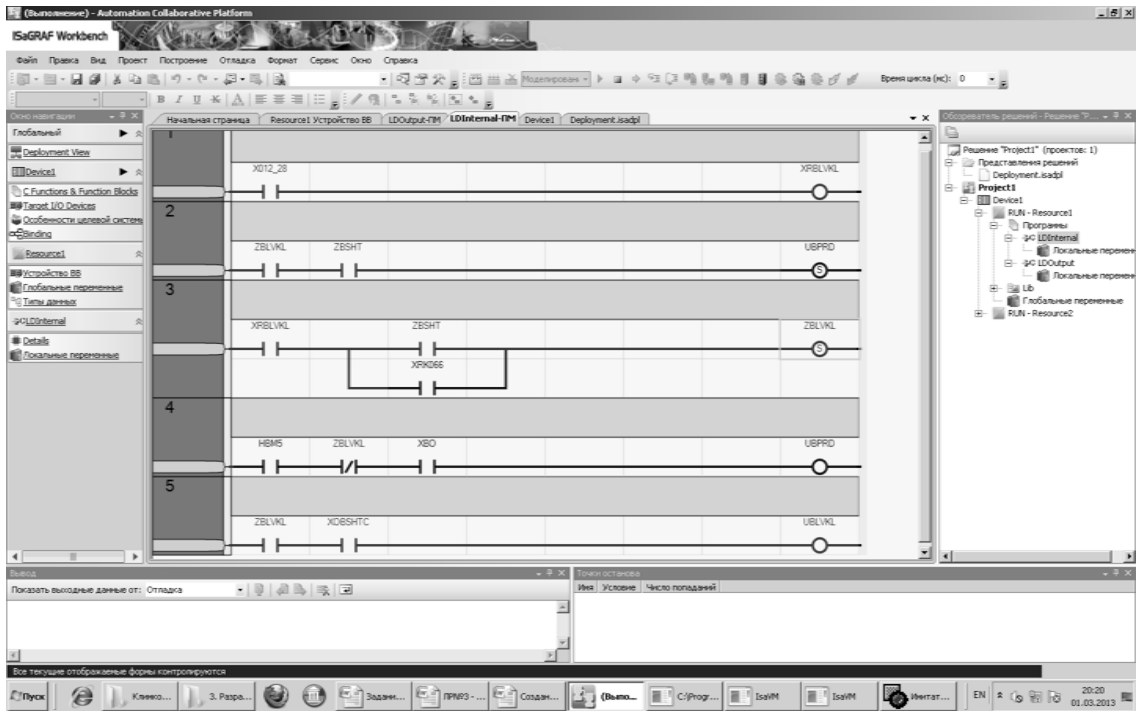


Рис. 2.27. Результат ввода значений переменных через устройство вода на диаграмме LDInternal

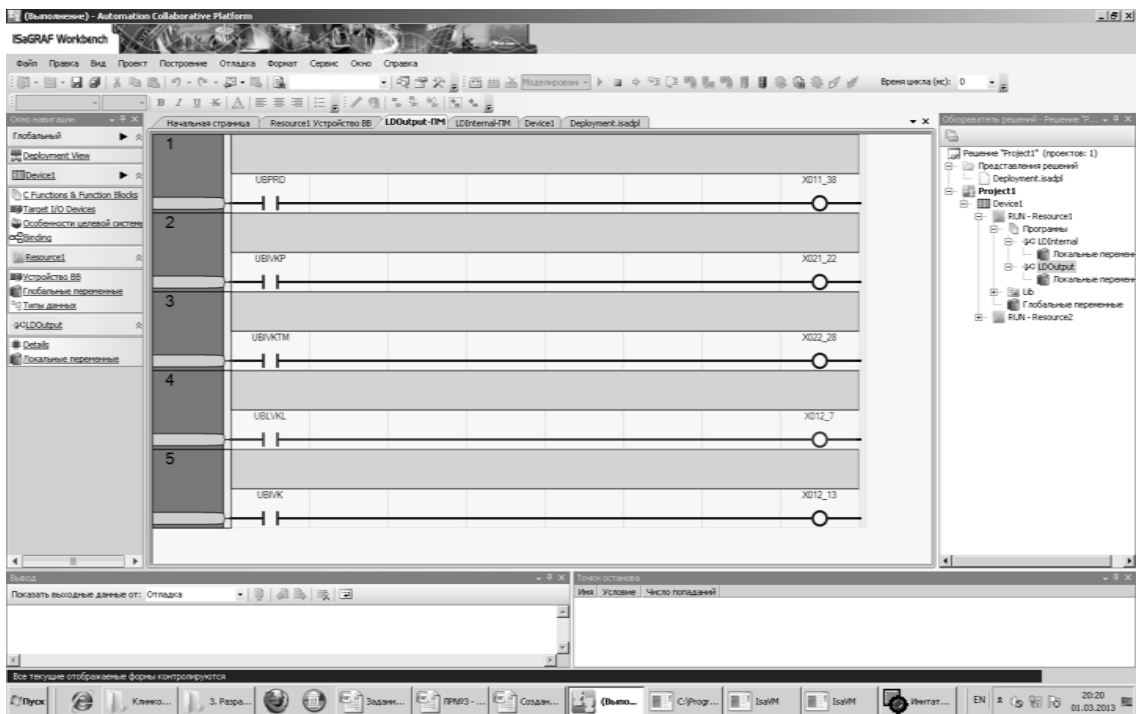


Рис. 2.28. Результат ввода значений переменных через устройство вода на диаграмме LDOOutput

1.3. Автоматизация программирования логических формул на языке LD

Логические формулы, представленные в ТЗ, имеют краткую (компактную) символьную форму записи. Однако эта форма практически не пригодна для анализа и поиска неизбежных ошибок в сложных СЛУ. Кроме этого, разработка программ на языке LD по логическим формулам в среде ISaGRAF является трудоемким процессом, подверженным появлению дополнительных ошибок. В связи с этим в приложении *ГИПЕРСИСТЕМА* разработаны программные средства для верификации и валидации логических формул и их последующего автоматического преобразования на язык LD.

2.3.1. Верификация и валидация логических формул

Верификация логических формул, представленных в ТЗ, выполняется программой, главное окно которой представлено на рис. 2.29.

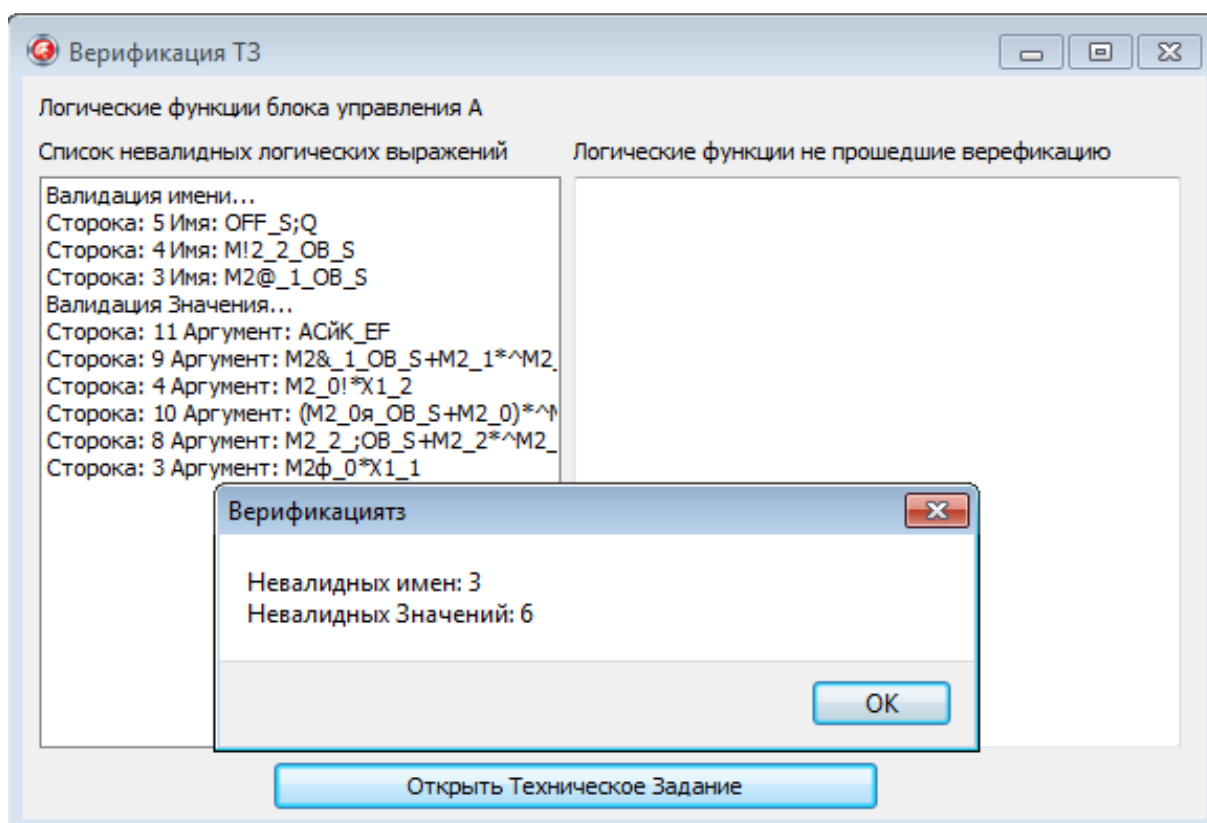


Рис. 2.29. Главное окно программы Верификация ТЗ

Как показано на рисунке 2.30, в исходное техническое задание были внесены некоторые изменения для имитации синтаксических ошибок.

Логическая функция
$M2_0_OB_S = X0_0 * \neg X0_1 * X0_2 * X0_3 * X0_5 * \neg Y5_0 * \neg X1_0$
$M2_1_OB_S = M2_0 * X1_1$
$M2_2_OB_S = M2_0 * X1_2$
$OFF_SQ = \neg X0_0$
$INI\%T_SQ = X1_6$
$ACSK_EF = X1_7$
$M2_2 = M2_2_OB_S + M2_2 * \neg M2_2_S1_R$

Рисунок 2.30 – Фрагмент технического задания с внесенными изменениями

На рисунке 2.31 приведен пример вывода результата работы программы верификации.

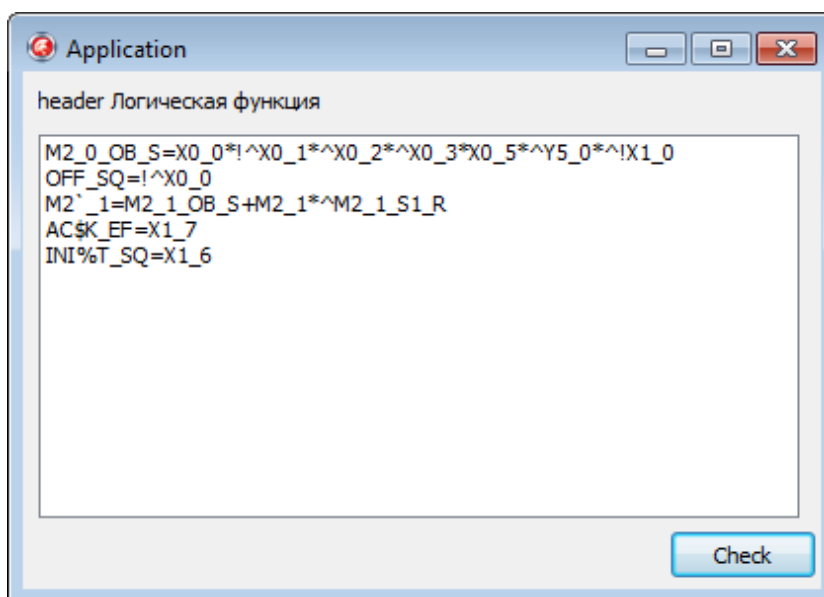


Рисунок 2.31 – Результат выполнения верификации логических формул

Как показано на рисунке 2.32, в исходное техническое задание были дополнительно внесены некоторые изменения для имитации ошибок в именах логических функций.

Логическая функция
$M2_0_OB_S = X0_0 * X0_1 * X0_2 * X0_3 * X0_5 * Y5_0 * X1_0$
$M2_1_OB_S = M2_0 * X1_1$
$M2_2_OB_S = M2_0 * X1_2$
$1OFF_SQ = X0_0$
$фЫINIT_SQ = X1_6$
$_ACK_EF = X1_7$
$M2_2 = M2_2_OB_S + M2_2 * M2_2_S1_R$
$Mк2\`_1 = M2_1_OB_S + M2_1 * M2_1_S1_R$

Рисунок 2.32 – Фрагмент технического задания с внесенными изменениями

На рисунке 2.33 приведен пример вывода результата работы программы верификации имен логических функций.

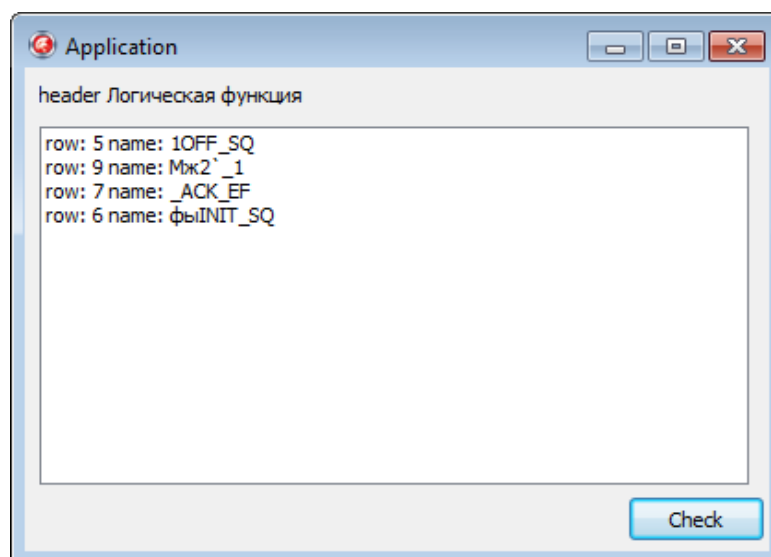


Рисунок 2.33 – Результат выполнения верификации имен логических функций

Модуль валидации предназначен для поиска дубликатов имен логических функций и поиска имен существующих аргументов логических функций.

Для поиска дубликатов имен были продублированы имена нескольких функций (рисунок 2.34).

ACK = ACK_EF
NT1 = M2_2 *X0_2 *X1_0*(S1+NT1)
NT2 = X0_4 *X0_5 *(S2+NT2)
NT3 = X0_5 **^ M2_1 *(S3+NT3)
NT3 = X0_5 *M2_1 *(S3+NT4)
NT5 = X0_4 *(S4+NT5)

Рисунок 2.34 – Фрагмент технического задания с дубликатами имен функций

На рисунке 2.35 приведен пример результата работы модуля валидации имен функций.

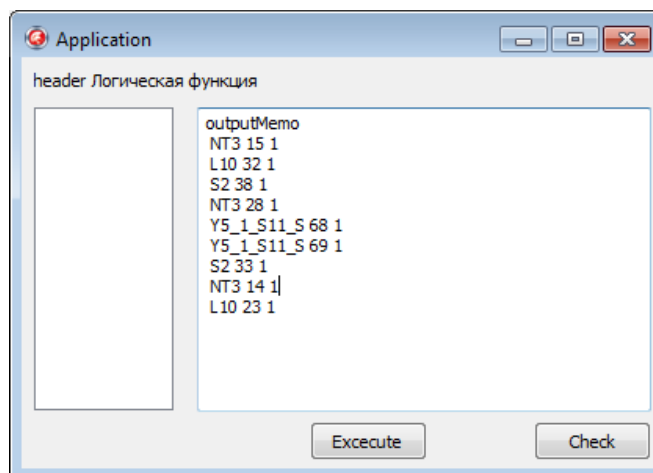


Рисунок 2.35– Результат выполнения валидации имен логических функций

Для поиска несуществующих имен аргументов были переписаны имена нескольких функций на уже существующие (рисунок 2.36).

Логическая функция
M2_0_OB_S=X0_0*^X0_1 *^X0_2*^X0_3 *X0_5*^Y5_0 *^X1_0
M2@_1_OB_S= M2ф_0*X1_1
M12_2_OB_S= M2_0!*X1_2
OFF_SaQ=^X0_0
INIT_SQ=X1_6+ OFF_SQ
ACK_EF=X1_7
M2_2 = M2_2 ;OB_S + M2_2 *^ M2_2_S1_R
M2_1 = M2&_1_OB_S + M2_1 *^ M2_1_S1_R
M2_0 = (M2_0я_OB_S + M2_0) *^ M2_0_S1_R
ACKb = AСяK_EF
NT1 = M2_2 *X0_2 *X1_0*(S1+NT1) -ACK

Рисунок 2.36 – Фрагмент технического задания, с несуществующими именами аргументов

На рисунке 2.37 приведен пример результата работы валидации имен аргументов.

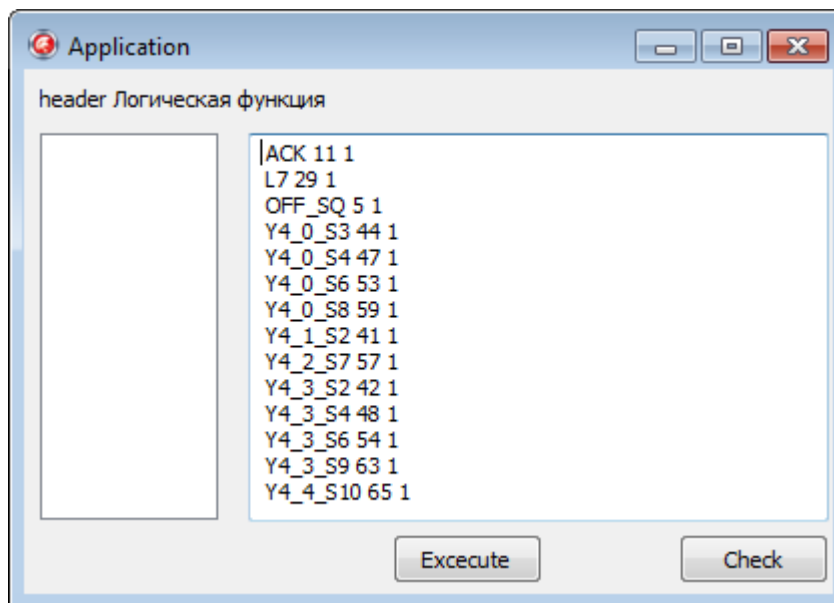


Рисунок 2.37 – Результат выполнения валидации аргументов логических функций

В процессе работы средств верификации и валидации формируется общая таблица с данными по всем логическим формулам (Рис. 2.38).

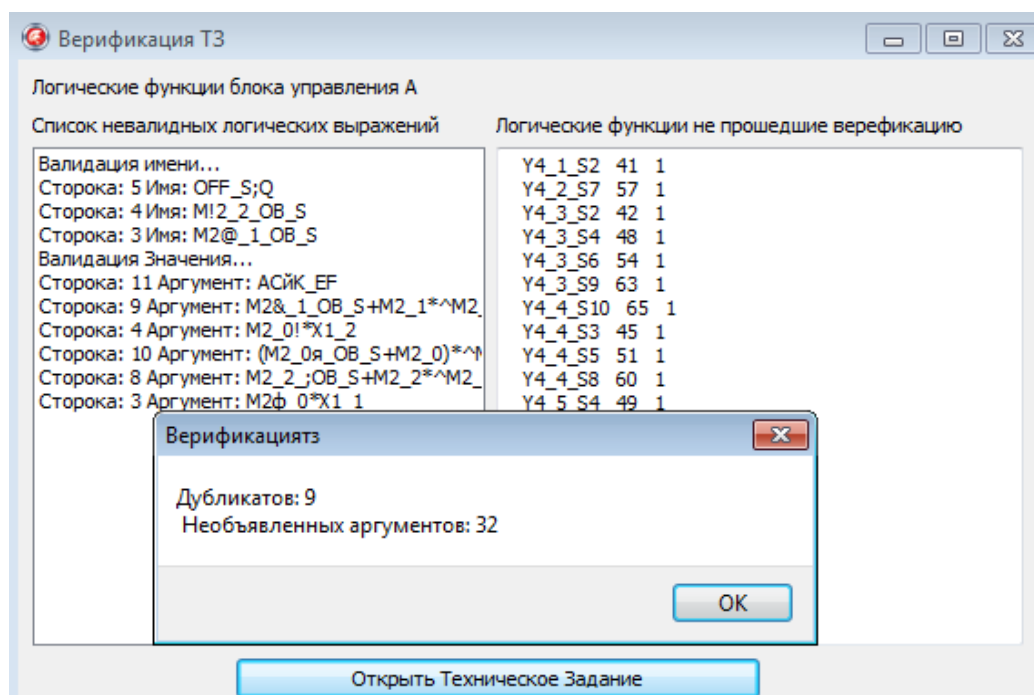


Рисунок 2.38 –Общий итог верификации и валидации

Для выполнения задач верификации и валидации используется легковесная встраиваемая реляционная база данных – SQLite.

Слово «встраиваемый» означает, что SQLite не использует парадигму клиент-сервер, то есть движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а предоставляет библиотеку, с которой программа компонуется и движок становится составной частью программы. Таким образом, в качестве протокола обмена используются вызовы функций (API) библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных в единственном стандартном файле на том компьютере, на котором исполняется программа.

По завершении работы средств верификации и валидации формируется текстовый файл, содержащий более подробный отчет об ошибках (рисунок 2.39).

```
Документ: D:\Diploma\Техническое задание.docx
-----
Заголовок:
*****
Заголовок: логические функции блока управления а
Невалидная операция: строка -- 14 столбец -- 1
***
Невалидный символ: функция -- M2_0=(M2_0Я_OB_S+M2_0)*^M2_0_S1_R строка -- 10 столбец -- 1
Номер: 11 Символ: Я
Невалидный символ: функция -- АСК=АСЙК_EF строка -- 11 столбец -- 1
Номер: 7 Символ: Й
-----
Поиск дубликатов...
Имя Столбец
NT3 -- 15 -- 1
L10 -- 32 -- 1
S2 -- 38 -- 1
NT3 -- 28 -- 1
S2 -- 33 -- 1
NT3 -- 14 -- 1
L10 -- 23 -- 1
Поиск необъявленных аргументов...
Имя Столбец
АСК -- 11 -- 1
АСК_EF -- 7 -- 1
Y4_5_S6 -- 55 -- 1
Y4_6_S9 -- 62 -- 1
Y5_1_S11_S -- 69 -- 1
Y5_1_S2_R -- 39 -- 1
*****
Заголовок: алгоритмы формирования а
```

Рисунок 2.39 – Файл-отчет

2.3.2. Преобразование логических формул в LD-диаграммы

Принцип работы программы *Функции_LD* заключается в преобразовании логических формул, записанных в текстовом виде, в файл .isaxml, который среда ISaGRAF отображает в графической форме (Рис. 2.40).

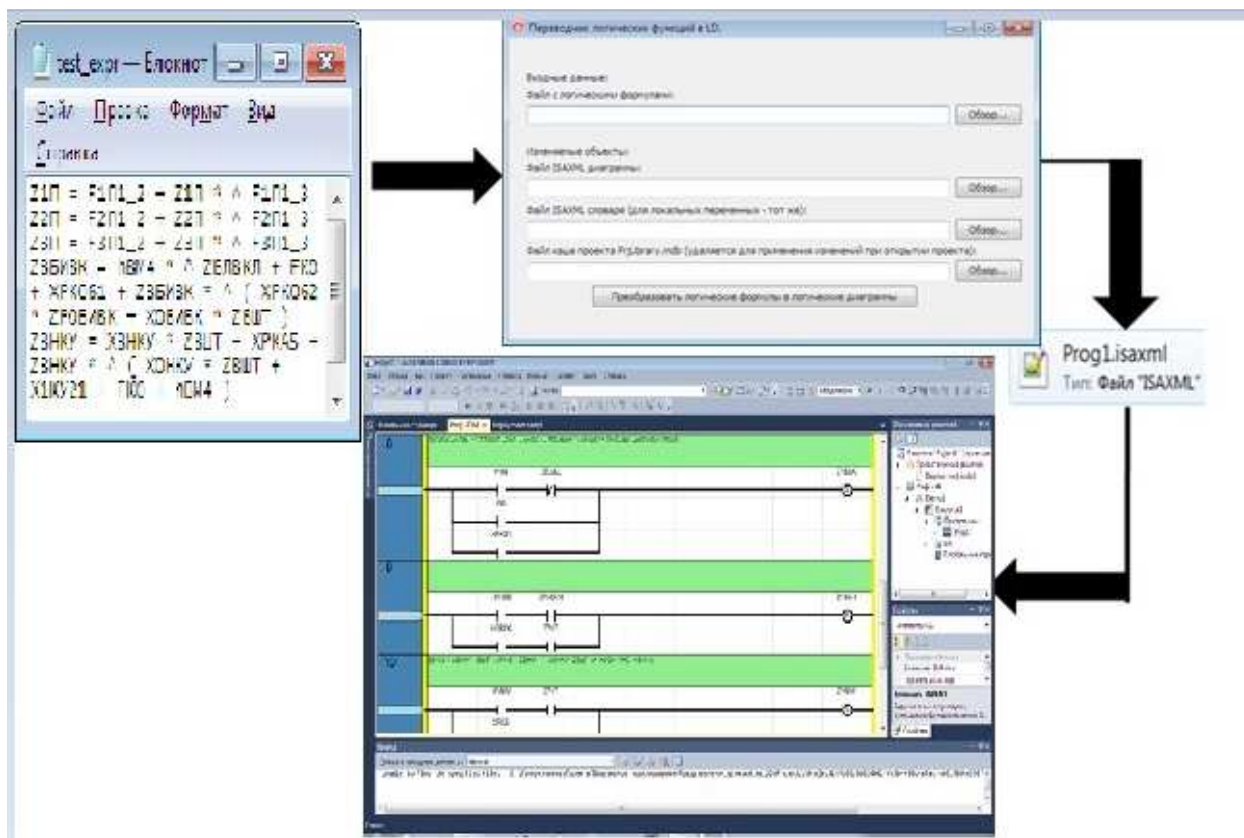


Рисунок 2.40 Принцип работы программы *Функции_LD*

Язык логических диаграмм LD позволяет представить логические формулы в наглядной графической форме и исполняемом виде.

В среде разработки ISaGRAF диаграмма LD сохраняется в файле с расширением .isaxml.

При запуске ISaGRAF создается шаблон файла Prog1.isaxml (Рис. 2.41):

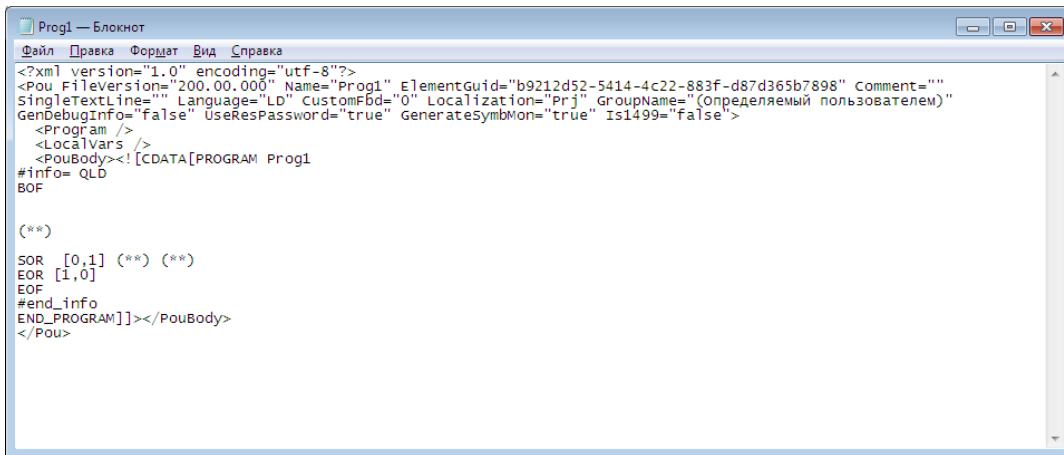


Рисунок 2.41 Шаблон файла Prog1.isaxml

В среде разработки ISaGRAF шаблону соответствует пустая диаграмма LD (Рис. 2.42).

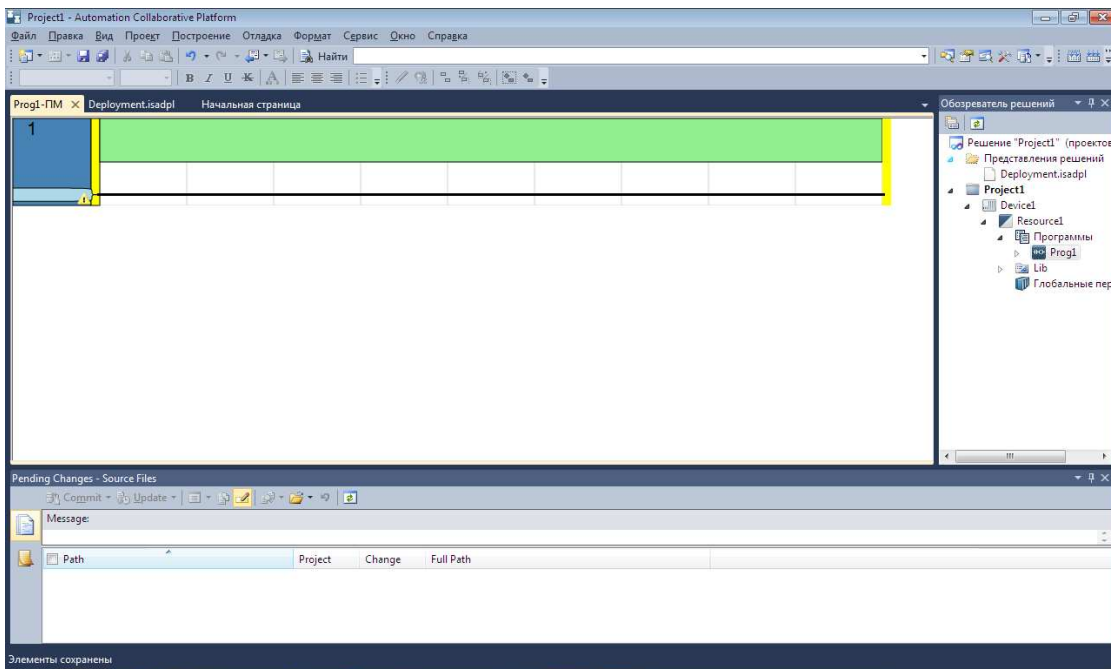


Рисунок 2.42 Пустая диаграмма LD

Программа *Функции_LD* заполняет этот шаблон в соответствии с заданной системой логических формул.

Рассмотрим подробнее структуру файла Prog1.isaxml.

Файл Prog1.isaxml имеет структуру XML документа и начинается тегом объявления версии языка, на которой написан документ, и кодировки документа:

```
<?xmlversion="1.0" encoding="utf-8"?>
```

В следующем теге `Pou` определены атрибуты версии файла (`FileVersion`), имени проекта (`Name`), комментария (`Comment`), языка программы проекта (`Language`), генерации отладочной информации (`GenDebugInfo`) и различные другие атрибуты.

Следующий Тег `<LocalVars>` представляет собой словарь переменных проекта.

Каждая переменная описывается рядом параметров через атрибуты в теге `<Variable />`:

- имя переменной (`Name`),
- тип данных (`DataType`),
- начальное значение (`InitialValue`),
- комментарий (`Comment`),
- адрес (`Address`),
- направление (`Kind`),
- алиас (`Alias`),
- права доступа (`AccessRights`),
- размер строки (`StringSize`)
- другие атрибуты, которые можно задать в редакторе переменных среды ISaGRAF.

Далее идет тег `<PouBody>` и в нем секция CDATA:

```
<PouBody><![CDATA[
```

В этой секции и размещается структура логических формул на языке LD, которая отображается при открытии проекта в ISaGRAF.

Исходный файл с описанием логических формул создается в приложении *ГИПЕРСИСТЕМА* программой *Ввод данных ТЗ* на этапе *Ввод данных модели* как обычный текстовый файл с Функции.txt в папке **Предметные конструкции модели**.

Окно программы Функции_LD имеет вид (Рис. 2.43):

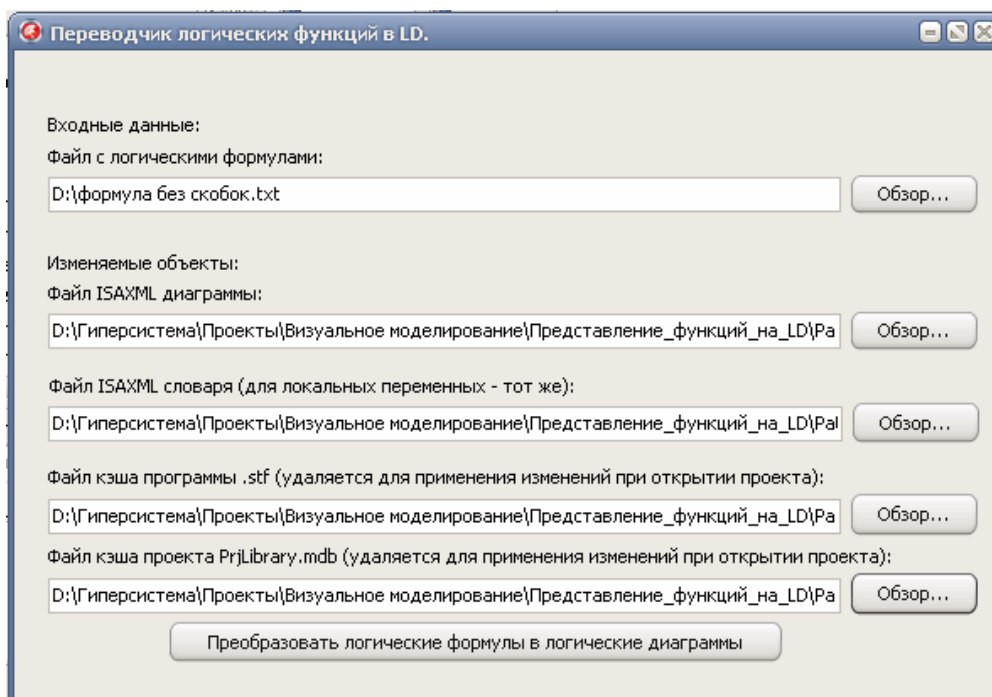


Рисунок 2.43 Окно программы Функции_LD

Для формулы с вложенными скобками $Q1 = (Q2 + Q3) * ((Q7 + ^ (Q7 + ^ Q8) * (Q9 + Q10)) * (Q9 + Q10) + Q5) + ^ Q6$ после открытия проекта в IsaGRAF результат преобразования имеет вид (рисунок 2.44).

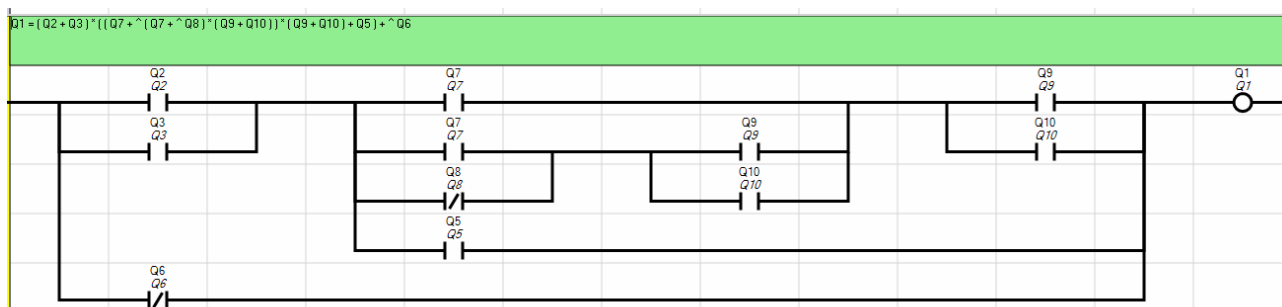


Рисунок 2.44. Результат преобразования на язык LD

2. Разработка проектов СЛУ на языке FBD

В настоящее время наблюдается стремительный рост потребности в современных средствах автоматизации производства. Задачи автоматизации с успехом решаются с помощью так называемых программируемых логических контроллеров (ПЛК). Одним из распространённых графических языков, используемых при работе с ПЛК, является язык диаграмм функциональных блоков (Functionalblockdiagram, сокращенно FBD).

К достоинствам функциональных схем при их использовании в качестве языка алгоритмизации относятся традиционность и однозначность описания, в том числе и параллельных процессов.

Необходимо отметить, что чтение функциональных схем представляет собой вычисления по их отдельным цепям с целью определения значений выходных переменных при различных наборах входных переменных. В этой ситуации при наличии даже сравнительно небольшого числа входов по функциональной схеме весьма трудно определить, какие воздействия влияют на тот или иной переход в ней. В схемах этого класса достаточно сложно составить целостное представление о поведении даже сравнительно небольшого фрагмента схемы, содержащего триггера и обратные связи. Так, например, при трех взаимосвязанных триггерах в схеме, непосредственно по ней (без вычислений) весьма трудно определить, какое число состояний эта схема реализует, так как с помощью указанного числа триггеров может быть закодировано от трех до восьми состояний.

При этом необходимо отметить, что использование в качестве тестов соотношений «вход-выход», обеспечивающих полноту проверки для схем без памяти, не решает проблему определения всех функциональных возможностей для схем с памятью, так как в этом случае необходимо проверять также и правильность порядка изменений выходных переменных. Однако, именно соотношения «вход-выход» и применяются при создании методик проверки на функционирование большинства систем логического управления. Это, как отмечалось выше, не обеспечивает качественной их проверки, так как такие соотношения не позволяют анализировать все имеющиеся в схеме переходы между состояниями. Более того, число состояний и переходов в схемах обычно не известно, так как они часто строятся эвристически без использования понятия «состояние».

Функциональные схемы при их применении в качестве языка программирования обладают всеми достоинствами декларативных языков функционального программирования, "основным из которых является функциональность (прозрачность по ссылкам). При этом каждое выражение определяет единственную величину, а все ссылки на нее эквивалентны самой этой величине, и тот факт, что на выражение можно ссылаться из другой части программы, никак не влияет на величину рассматриваемого выражения. Это свойство определяет различие между математическими функциями и функциями, которые можно написать на процедурных языках программирования таких, например, как Паскаль, позволяющих функциям ссылаться на глобальные данные и применять "разрушающее" присваивание. Такое присваивание может привести к побочным эффектам, например к изменению значения функции при повторном ее вызове даже без

изменения значений аргументов. Это приводит к тому, что функцию трудно использовать, так как для того, чтобы определить, какая величина получится при ее вычислении, необходимо рассмотреть текущую величину глобальных данных, что, в свою очередь, требует изучения предыстории вычислений для определения того, что порождает эту величину в каждый момент времени”.

При определенных условиях в системах булевых формул, по которым функциональные схемы могут строиться, даже для автоматов с памятью удастся обеспечить и другое достоинство декларативных языков – независимость результатов от порядка вычисления формул.

3.1. Описание языка FBD

Язык функциональных блок-схем - FBD (FunctionBlockDiagram)

Язык программирования FBD (FunctionBlockDiagram) является составной частью стандарта IEC-61131 и так же входит в стандарт IEC-61499.

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.

Объектами языка FBD являются:

- элементарные функции и элементарные функциональные блоки (ФБ), логика работы (программа) которых написана на языке С и не может быть изменена в редакторе FBD;
- функции и ФБ *пользователя*, которые 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, либо непосредственно линиями связи – *графическими связями*.

3.1.1. Главные элементы FBD

Перечислим главные элементы языка FBD с короткими графическими иллюстрациями (рис. 3.1 – 3.17).

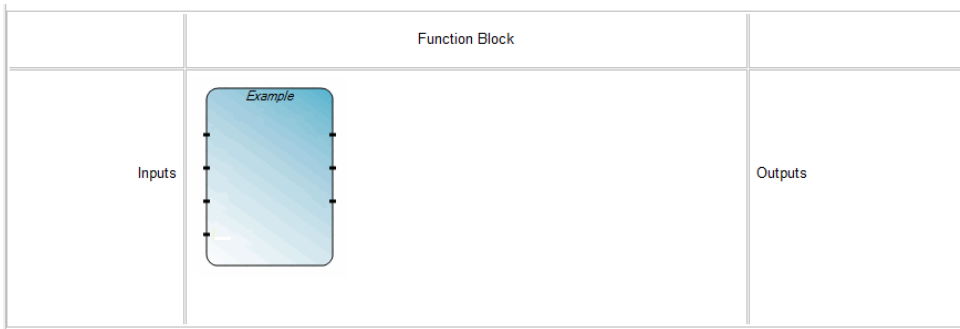


Рис. 3.1. Функциональный блок

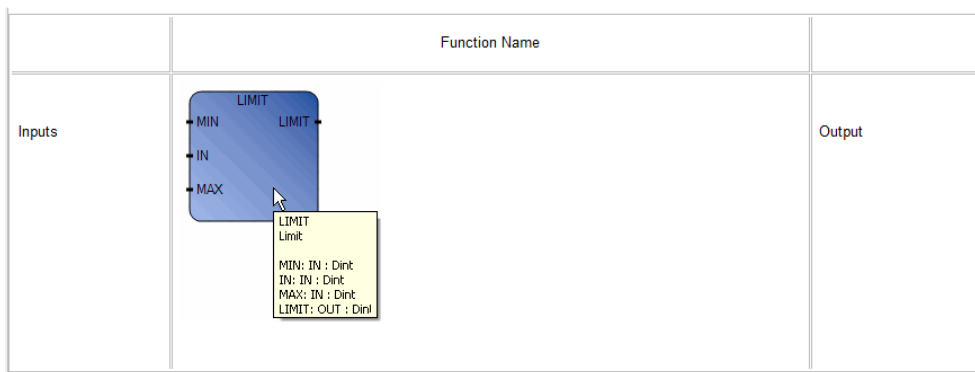


Рис. 3.1. Функция

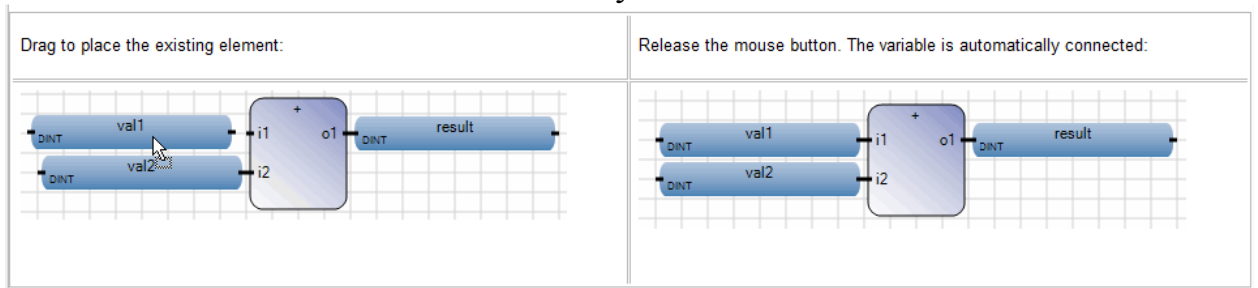


Рис. 3.2. Переменные

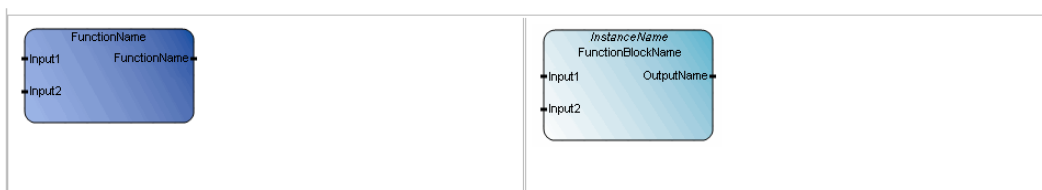


Рис. 3.3. Вызов функций и блоков

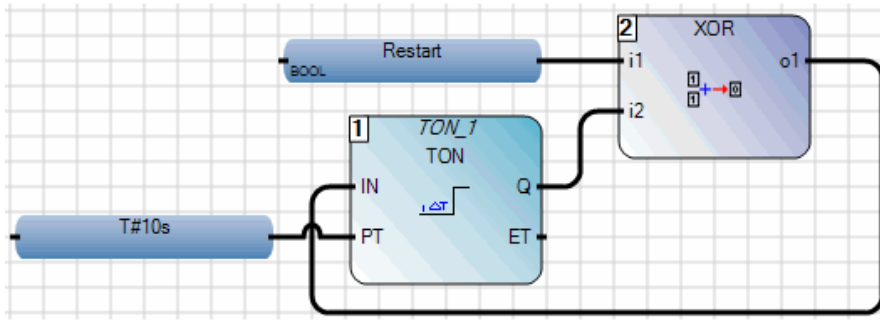


Рис. 3.4. Связи блоков

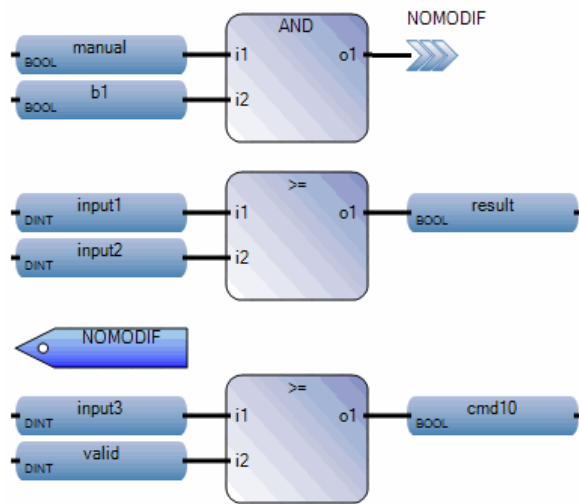


Рис. 3.5. Переходы

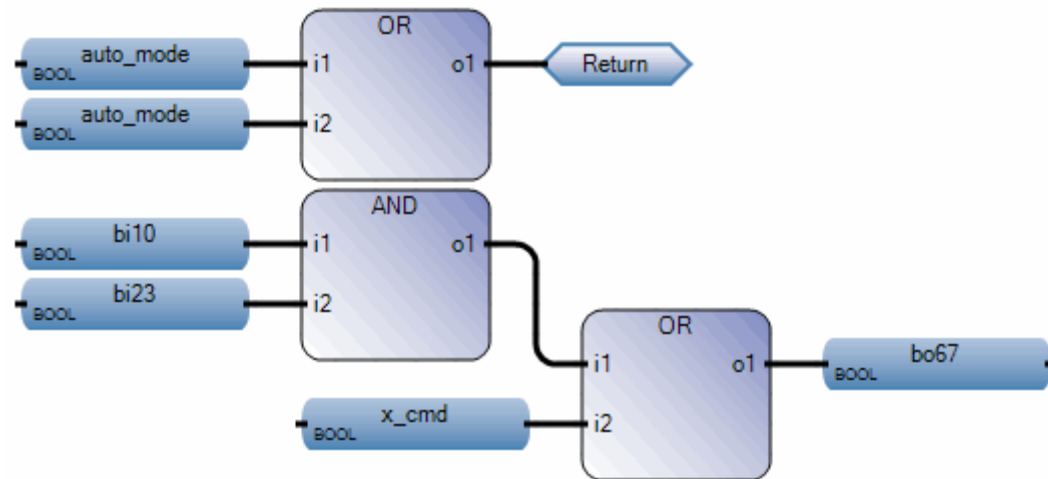


Рис. 3.6. Возвраты

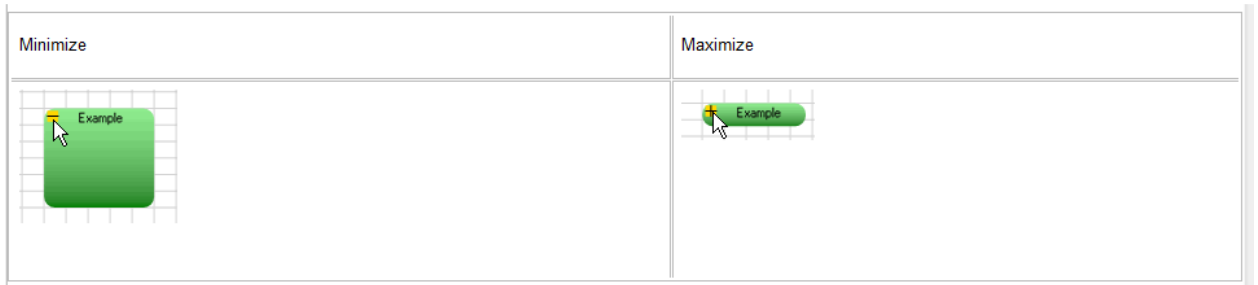


Рис. 3.7. Комментарии

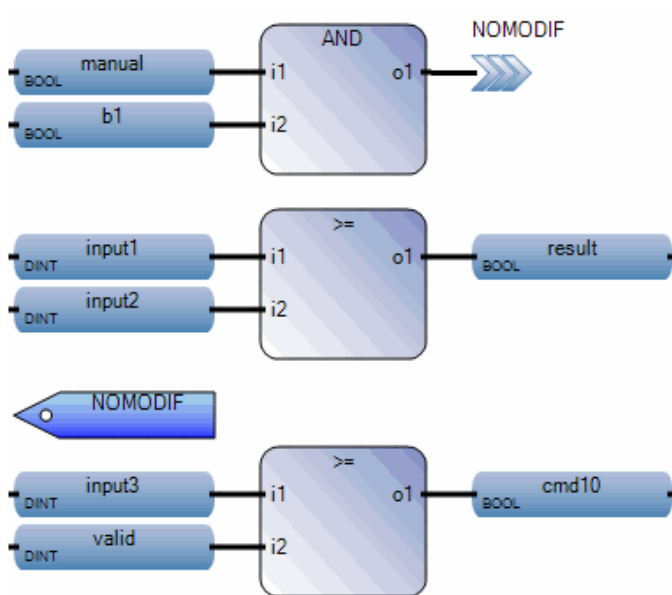


Рис. 3.8. Метки

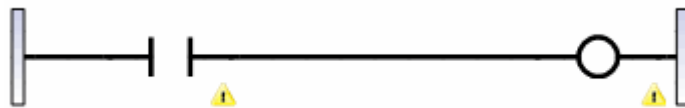


Рис. 3.9. Цепь

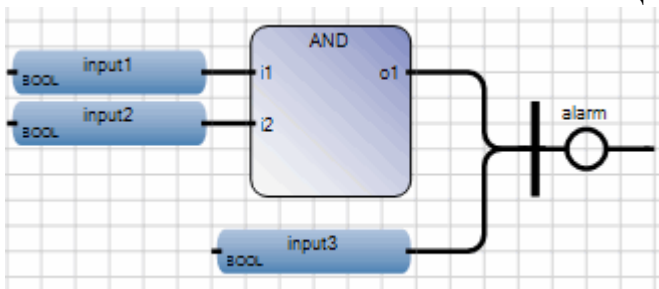


Рис. 3.10. Вертикальная шина

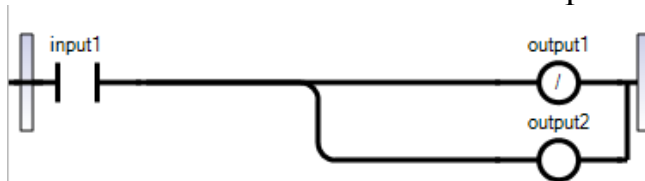


(* ST Equivalence: *)

output1 := input1;

output2 := input1;

Рис. 3.11. Прямая обмотка

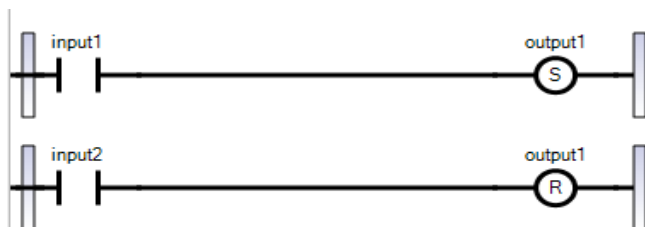


(* ST Equivalence: *)

output1 := NOT (input1);

output2 := input1;

Рис. 3.12. Инверсная обмотка



(* ST Equivalence: *)

IF input1 THEN

output1 := TRUE;

END_IF;

IF input2 THEN

output1 := FALSE;

END_IF;

Рис. 3.13. Включающая обмотка (Set) и Выключающая обмотка (Reset)

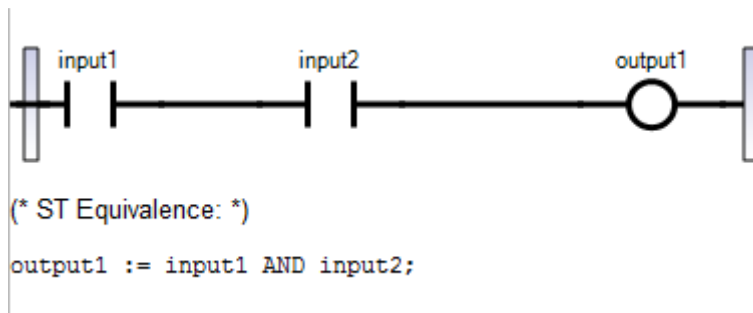


Рис. 3.14. Прямой контакт

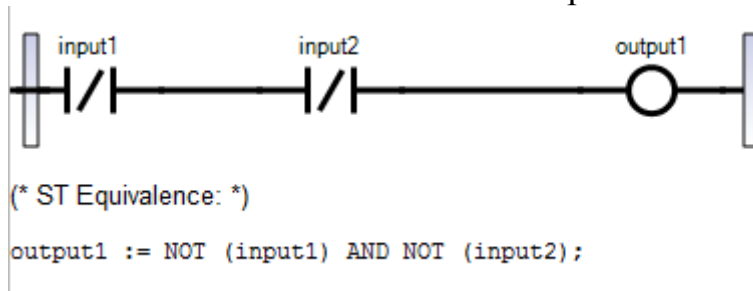


Рис. 3.15. Инверсный контакт

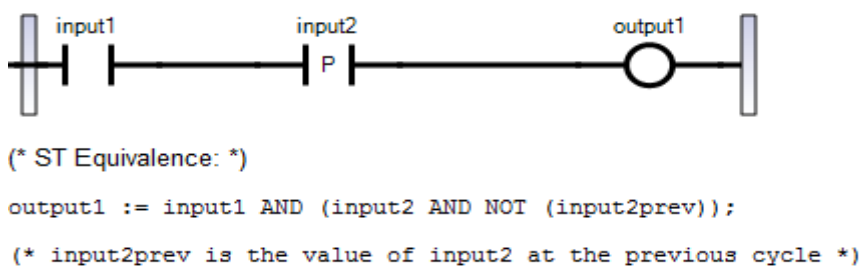


Рис. 3.16. Импульсный контакт (Передний фронт)

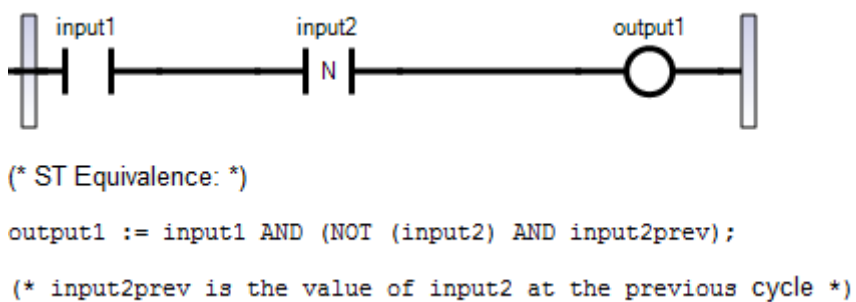


Рис. 3.17. Импульсный контакт (Задний фронт)

3.2. Разработка программ на языке FBD

Рассмотрим процесс разработки FBD – программы на простейшем примере логического элемента AND. Для создания проекта (решения) в

приложении ISaGRAF необходимо выполнить ряд этапов, представленных на рис 3.18 -3.28.

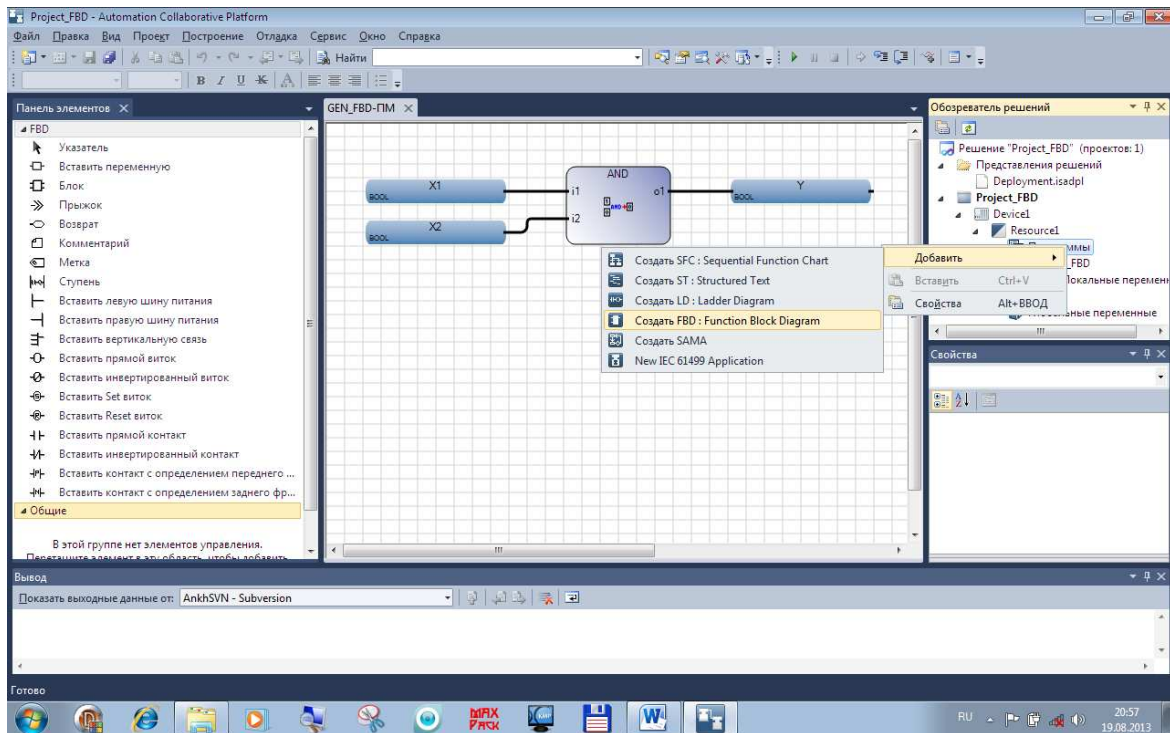


Рис 3.18. Создание программы с помощью панели элементов

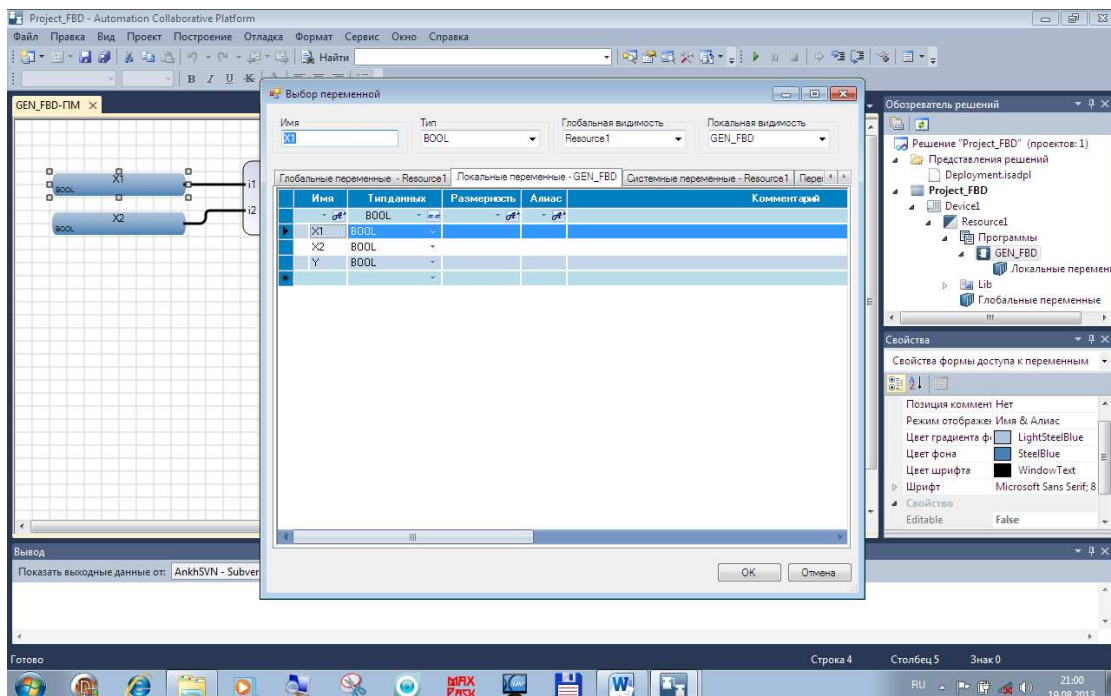


Рис 3.19. Создание словаря локальных переменных

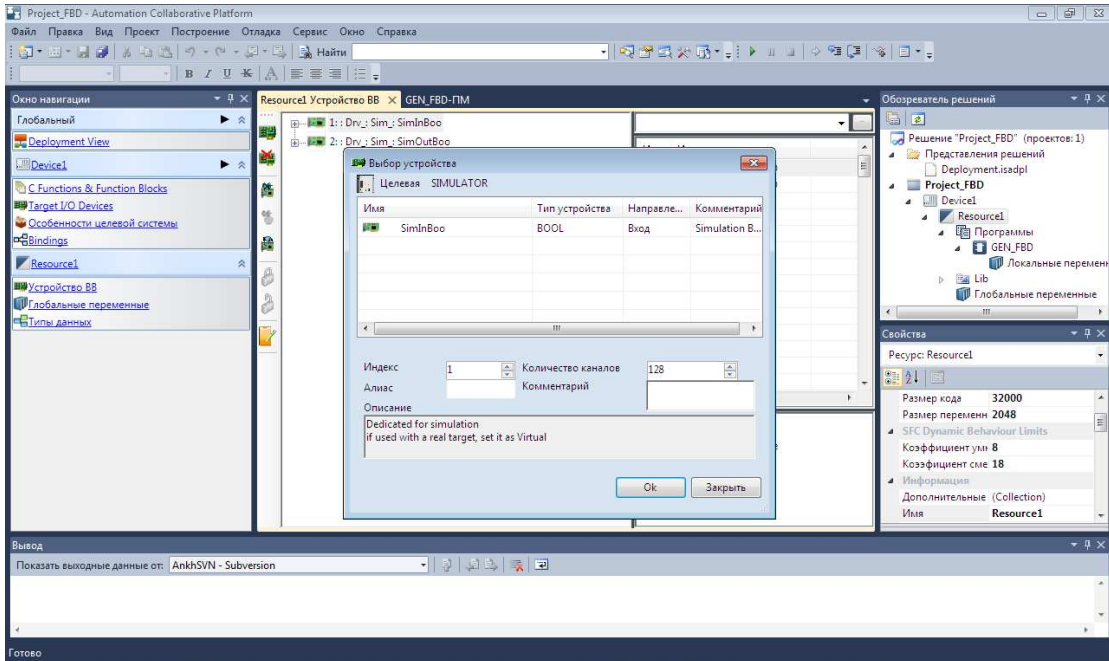


Рис 3.20. Создание виртуальных устройств ввода – вывода

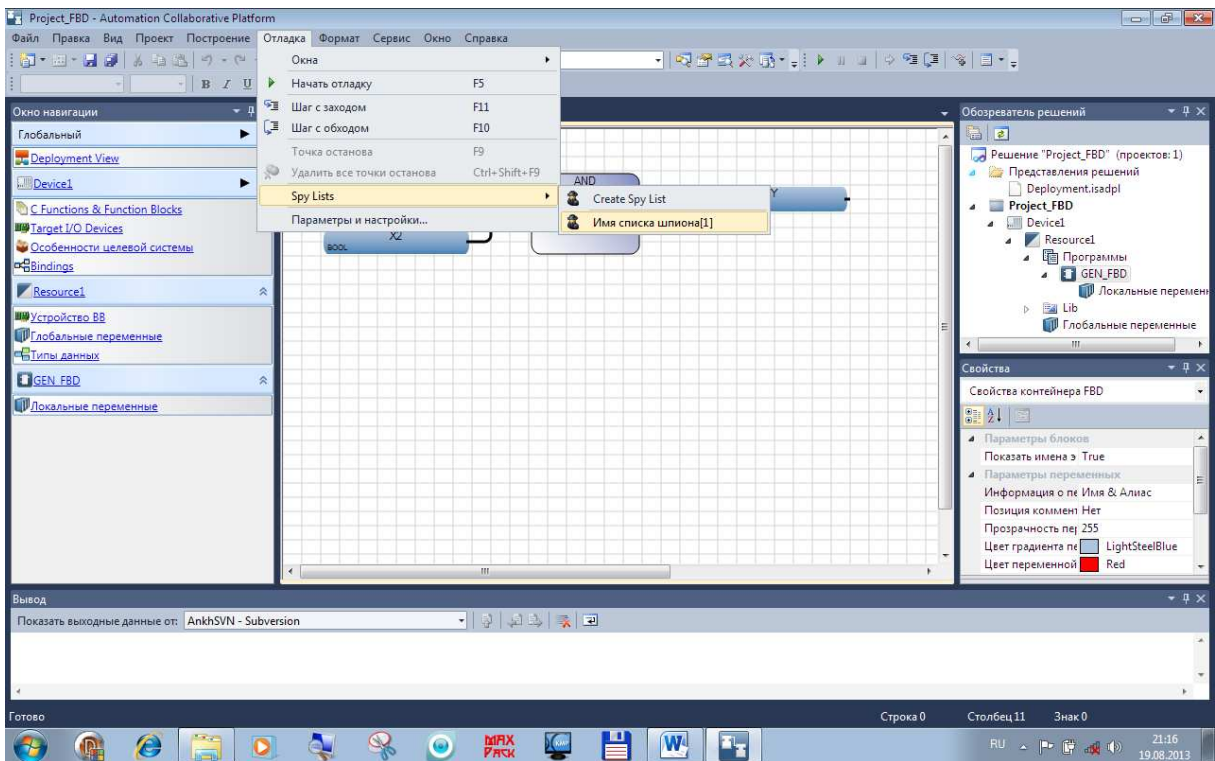


Рис 3.21. Создание списка наблюдения (шпиона)

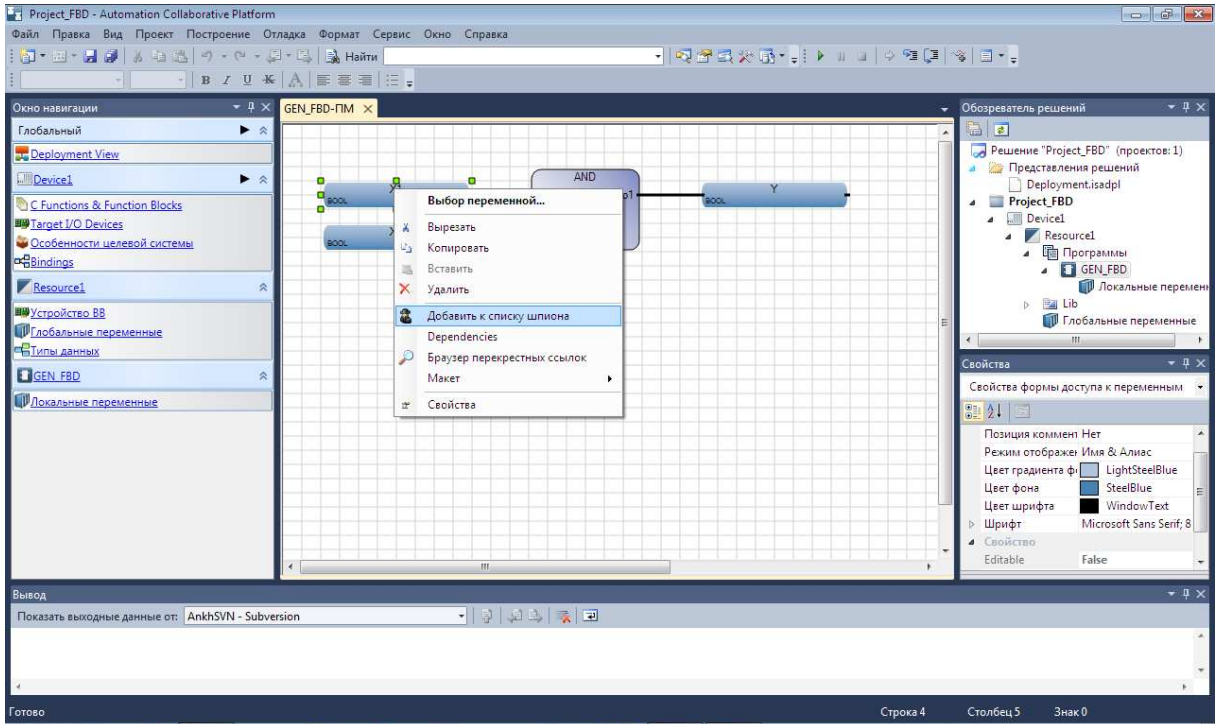


Рис 3.22. Добавление переменных в список наблюдения

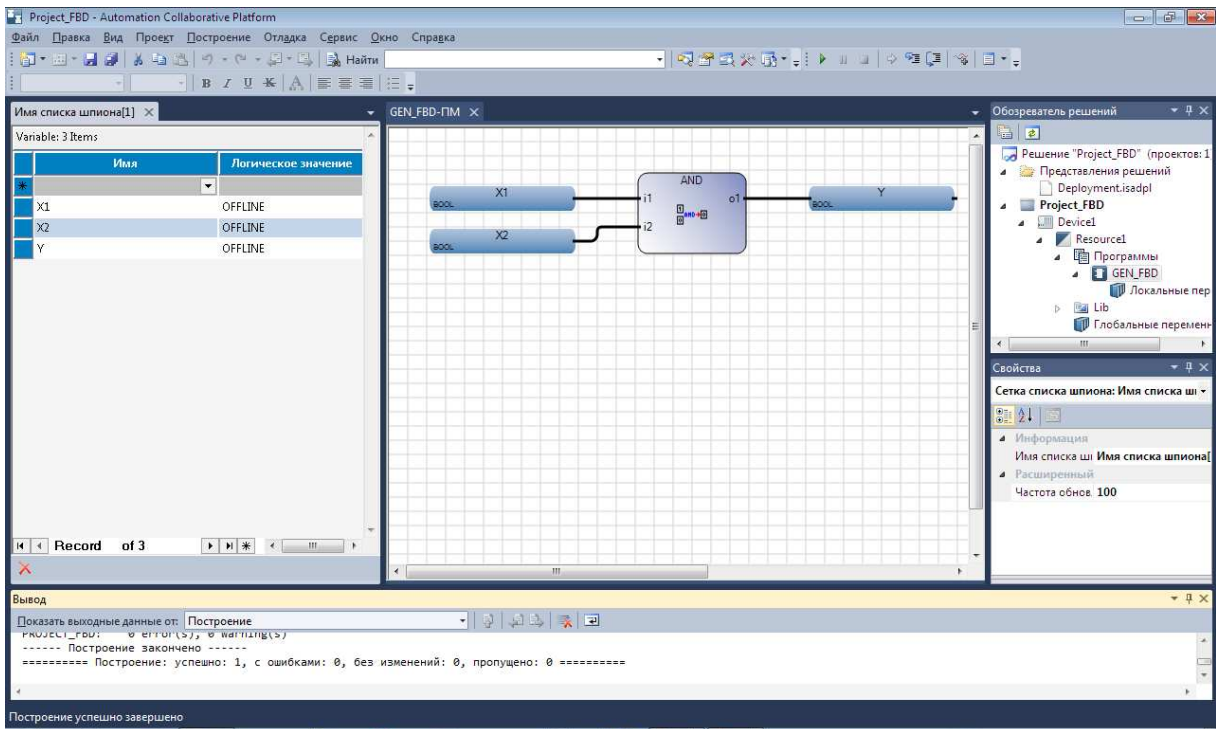


Рис 3.23. Построение решения (компиляция программы)

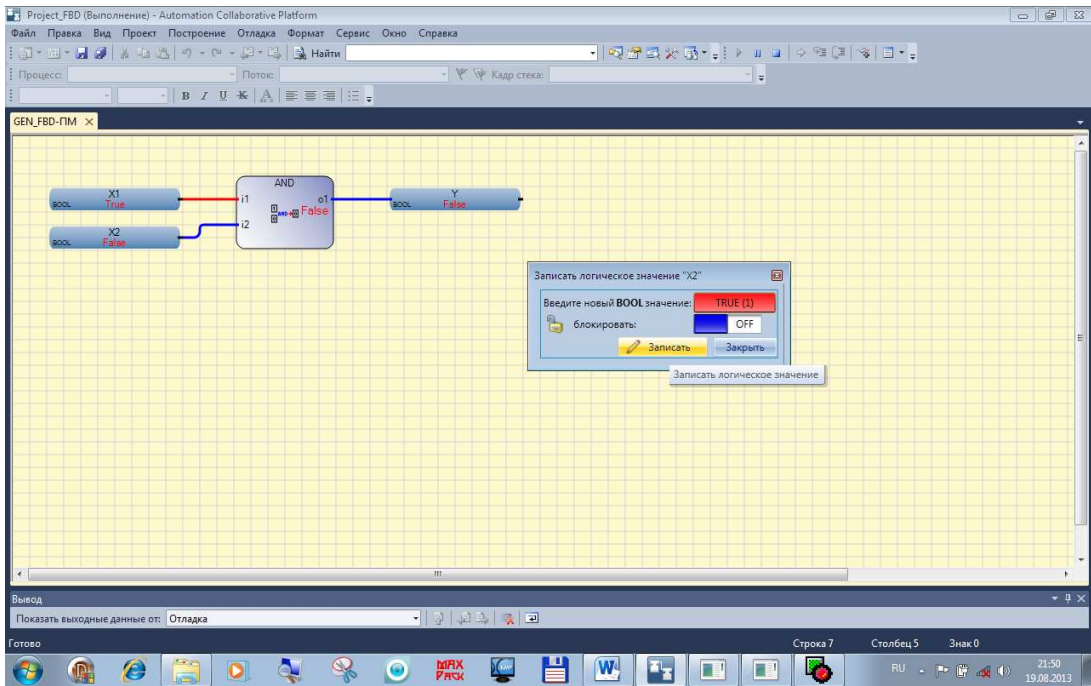


Рис 3.24. Моделирование с подачей входов в FBD – программе

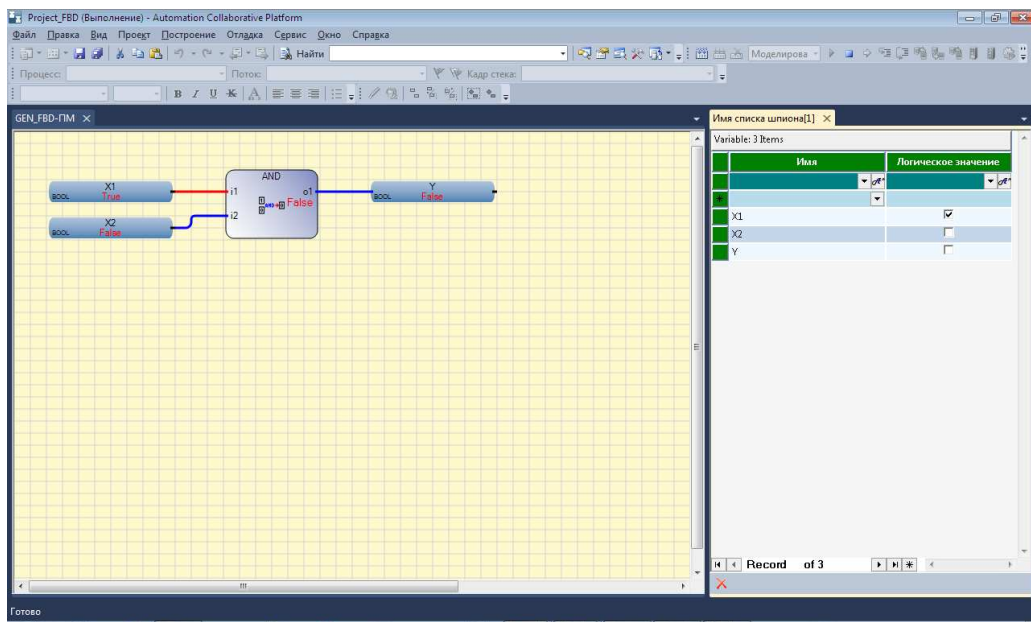


Рис 3.25. Подача входа X1 в окне наблюдения

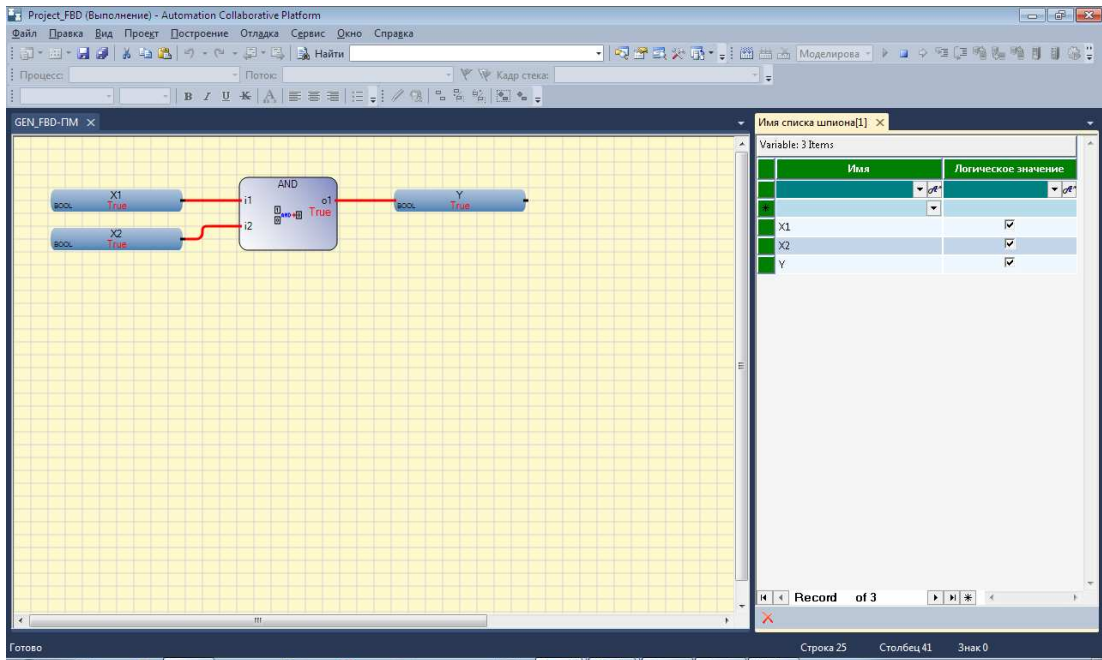


Рис 3.26. Подача входа X2 в окне наблюдения

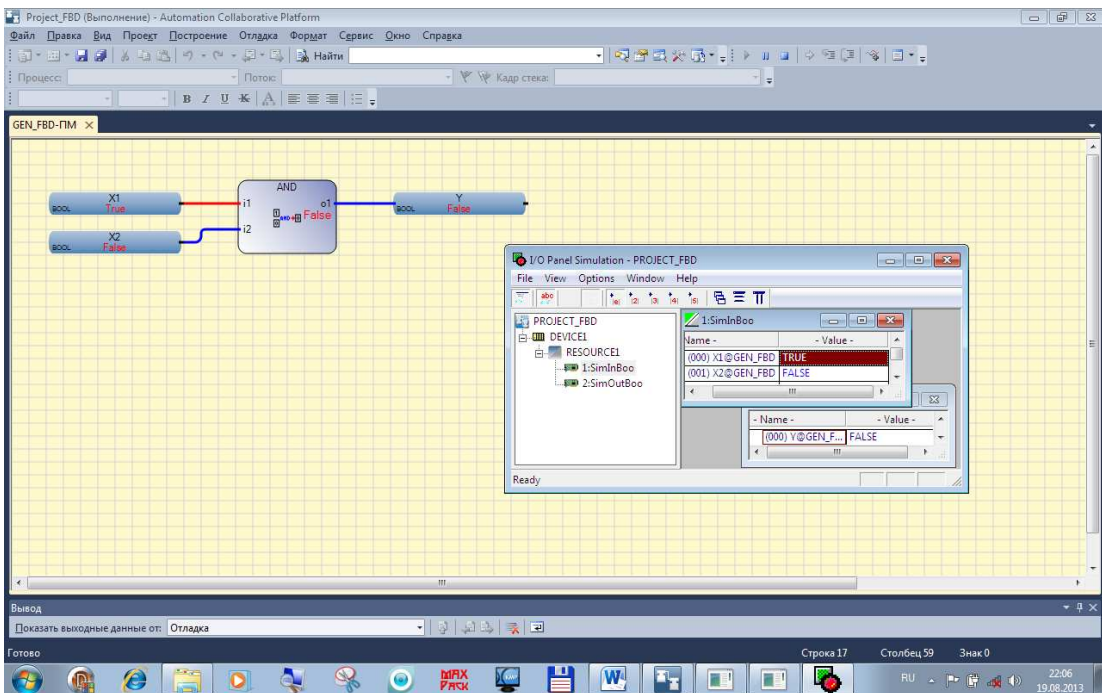


Рис 3.27. Подача входа X1 с виртуальной панели ввода

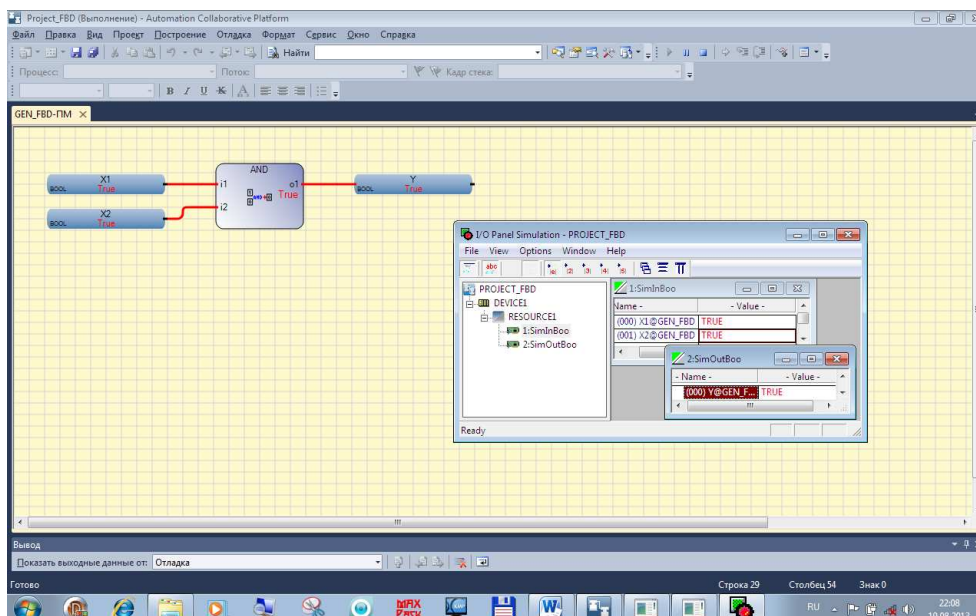


Рис 3.28. Подача входа X2 с виртуальной панели ввода

2.3. Автоматизация программирования логических формул на языке FBD

Логические формулы, представленные в ТЗ, имеют краткую (компактную) символьную форму записи. Однако эта форма практически не пригодна для анализа и поиска неизбежных ошибок в сложных СЛУ. Кроме этого, разработка программ на языке FBD по логическим формулам в среде ISaGRAF является трудоемким процессом, подверженным появлению дополнительных ошибок. В связи с этим в приложении ГИПЕРСИСТЕМА разработаны программные средства для верификации и валидации логических формул и их последующего автоматического преобразования на язык LD (см. 2.3) и FBD.

Для автоматического преобразования логических формул на язык FBD требуется их предварительное преобразование из операторной формы (на базе логических операторов AND, OR, NOT) представления в функциональную (на базе логических функций AND, OR, NOT).

Например, формула $Y = X1 \text{ or } X2 \text{ andnot } X3$ преобразуется в $Y = \text{or}(X1, \text{and}(X2, \text{not}(X3)))$.

Для сложных скобочных формул такое преобразование является очень сложной задачей.

3.3.1. Преобразование логических формул из операторной формы в функциональную

Проблему преобразования логических формул из операторных в функциональные будет решать путем использования дерева синтаксиче-

ского разбора, состоящего из узлов. Это дерево можно представить программным кодом (рис 3.29):

```
public class Node
{
    public string symbol;           // и(*), или(+)
    public bool not;               // не(^)
    public Node left, right;
}

public class Tree
{
    public Node root;
    public string value;
}
```

Рис. 3.29. Класс дерева и узлов дерева

Реализация данного алгоритма обеспечивает преобразование скобочных логических формул неограниченной глубины вложенности.

Для этого создана функция, которая подсчитывает количество вложенных скобок в исходном выражении, запоминает их позиции и сопоставляет позициям открывающихся скобок позиции закрывающихся скобок (рис.3.30)

```
public SortedList<int, int> brackets(string str)
{
    // индексы всех открывающихся скобок / закрывающихся скобок
    List<int> input = new List<int>();
    List<int> output = new List<int>();
    for (int i = 0; i < str.Count(); i++)
    {
        if (str[i] == '(') input.Add(i);
        if (str[i] == ')') output.Add(i);
    }

    // сопоставление каждой открывающейся скобки и закрывающейся скобки
    SortedList<int, int> list = new SortedList<int, int>();
    for (int i = 0; i < output.Count(); i++)
    {
        int max = 0;
        for (int j = 0; j < input.Count(); j++)
        {
            if (input[j] < output[i] && input[j] > max)
                max = input[j];
        }

        input.RemoveAt(input.IndexOf(max));
        list.Add(max, output[i]);
    }

    return list;
}
```

Рис. 3.30. Функция сопоставления индексов открывающихся скобок закрывающимся скобкам

После нахождения позиций всех открывающихся и закрывающихся скобок, выполняется проход по циклу всех закрывающихся скобок и поиск среди открывающихся скобок индексов, которые меньше индекса закрывающейся скобки, но являются наибольшими среди других индексов открывающихся скобок.

Сведения о сопоставлении позиций открывающихся и закрывающихся скобок сохраняются в сортированном списке list.

Функция вывода дерева похожа на функцию построения дерева и выглядит следующим образом (рис. 3.31):

```
public string view = "";
public void drawTree(Node node)
{
    if (node.left == null && node.right == null)
    {
        if (node.not == true) view += " HE (" + node.symbol + ")";
        else view += node.symbol;
    }
    if (node.left != null)
    {
        if (node.left.left == null && node.left.right == null)
            if (node.not == true) view += " HE ("; // 2 вариант
        if (node.symbol == "***") view += " И (";
        else if (node.symbol == "+") view += " ИЛИ (";
        if (node.left.left != null || node.left.right != null)
            if (node.not == true) view += " HE ("; // 1 вариант
        drawTree(node.left);
    }
    if (node.right != null)
    {
        if (node.left.left != null || node.left.right != null)
            if (node.not == true) view += ")"; // 1 вариант
        if (node.symbol == "***") view += ";";
        else if (node.symbol == "+") view += ";";
        drawTree(node.right);
        if (node.symbol == "***") view += ")";
        else if (node.symbol == "+") view += ")";

        if (node.left.left == null && node.left.right == null)
            if (node.not == true) view += ")"; // 2 вариант
    }
}
```

Рис. 3.31. Функция вывода дерева

Прохождение дерева происходит рекурсивно.

Для начала выводится текущий символ, далее идет проверка, есть ли у узла левая ветка или правая ветка. Если есть, то функция вызывает сама

себя рекурсивно, но входным параметром, является уже левая или правая ветка дерева.

На рис. 3.32 представлен фрагмент файла, содержащего исходные логические формулы в операторной форме, а на рис.3.33 эти формулы показаны в преобразованной функциональной форме.

```
Функции — Блокнот
Файл  Правка  Формат  Вид  Справка
ZVIP1M = XPKA0 + FKO + ZVIP1M * ^ ( XPKA1 + XSK1 * ^ Z3PVI1 + XOSPМ * ZVSHТ )
ZBIP2M = XPKA1 + XSK1 * ^ Z3PVI1 + ZBIP2M * ^ ( XPKA0 + XOЦПМ * ZBШТ + FKO )
Z3PVI1 = XPK40 + Z3PVI1 * ^ ( XPK60 + FKO )
Z3PYU = XPK41 + Z3PYU * ^ ( XPK63 + FKO )
ZCK1 = XSK1 + ZCK1 * ^ X5KY12
ZCK2 = XSK2 + ZCK2 * ^ X5KY12
ZBLHTЦ = X5KY11 + XOЦПМ * ZBШТ + ZBLHTЦ * ^ X5KY10
ZMYU1 = XPK44 + FKO + ZMYU1 * ^ ( XPK45 + XSK2 * ^ Z3PYU )
ZMYU2 = XPK45 + XSK2 * ^ Z3PYU + ZMYU2 * ^ ( XPK44 + FKO )
ZMKO1 = XPK70 + FKO + ZMKO1 * ^ ( XPK72 + X5KY06 )
ZMKO2 = XPK72 + X5KY06 + ZMKO2 * ^ ( XPK70 + FKO )
ZMPЭK1 = FKO + X5KY08 + ZMPЭK1 * ^ ( XPK71 + X5KY0A )
ZMPЭK2 = XPK71 + X5KY0C + ZMPЭK2 * ^ ( FKO + X5KY0B )
ZMCK1 = X5KY0D + ZMCK1 * ^ ( XPK73 + X5KY0E )
ZMCK2 = XPK73 + X5KY0E + ZMCK2 * ^ X5KY0D
ZMPIT1 = X5KY07 + ZMPIT1 * ^ ( XPK81 + X5KY09 )
ZMPIT2 = XPK81 + X5KY09 + ZMPIT2 * ^ X5KY07
ZMPHK1 = XPK64 + FKO + ZMPHK1 * ^ ( XPK65 + X5KY00 + X5KY01 + XOЦПМ * ZBШТ )
ZMPHK2 = XPK65 + X5KY00 + ZMPHK2 * ^ ( XPK64 + FKO + X5KY05 + XOЦПМ * ZBШТ )
SCTT1 = SCTT18 * ZЦПM1
SCTT2 = SCTT18 * ZЦПM2
ZCTT1 = SCTT1 + ZCTT1 * ^ XPK84
ZCTT2 = SCTT2 + ZCTT2 * ^ XPK84
ZBLCTT = XPK54 + ZBLCTT * ^ ( XPK55 + FKO )
FCPY = XCPY
ZTCY2 = FCPY + ZTCY2 * ^ SCY2
ZCTCY2 = SCY2 + ZCTCY2 * ^ XPK84
ZCPY = FCPY + ZCPY * ^ ( SCPY + X5KY21 + X5KY22 )
ZCTCPY = SCPY + ZCTCPY * ^ XPK84
ZBLCPY = XPK74 + ZBLCPY * ^ ( XPK75 + FKO )
SCTT = SCTT18 * ^ ZBLCTT + SCPY * ^ ZBLCPY
ZCTCPY = SCPC + ZCTCPY * ^ XPK84
FKO = ( XKO13 + XKO24 ) * ZPKO + XPK92
ZPBM = XРБЛBM * ZBШТ + ZPBM * ^ ( XБЛBM * ZBШТ + FKO + X1KY27 )
```

Рис. 3.32. Исходный файл с формулами в операторной форме

```
Функции_out — Блокнот
Файл  Правка  Формат  Вид  Справка
ZVIP1M = ИЛИ (XPKA0; ИЛИ (FKO; И (ZVIP1M; НЕ ( ИЛИ (XPKA1; И (XSK1; ИЛИ ( НЕ (Z3PVI1; И (XOSPМ; ZVSHТ))))))))
ZBIP2M = ИЛИ (XPKA1; И (XSK1; ИЛИ ( НЕ (Z3PVI1); И (ZBIP2M; НЕ ( ИЛИ (XPKA0; И (XOЦПМ; ИЛИ (ZBШТ; FKO))))))))
Z3PVI1 = ИЛИ (XPK40; И (Z3PVI1; НЕ ( ИЛИ (XPK60; FKO))))
Z3PYU = ИЛИ (XPK41; И (Z3PYU; НЕ ( ИЛИ (XPK63; FKO))))
ZCK1 = ИЛИ (XSK1; И (ZCK1; НЕ (X5KY12)))
ZCK2 = ИЛИ (XSK2; И (ZCK2; НЕ (X5KY12)))
ZBLHTЦ = ИЛИ (X5KY11; И (XOЦПМ; ИЛИ (ZBШТ; И (ZBLHTЦ; НЕ (X5KY10))))))
ZMYU1 = ИЛИ (XPK44; ИЛИ (FKO; И (ZMYU1; НЕ ( ИЛИ (XPK45; И (XSK2; НЕ (Z3PYU))))))
ZMYU2 = ИЛИ (XPK45; И (XSK2; ИЛИ ( НЕ (Z3PYU); И (ZMYU2; НЕ ( ИЛИ (XPK44; FKO))))))
ZMKO1 = ИЛИ (XPK70; ИЛИ (FKO; И (ZMKO1; НЕ ( ИЛИ (XPK72; X5KY06))))))
ZMKO2 = ИЛИ (XPK72; ИЛИ (X5KY06; И (ZMKO2; НЕ ( ИЛИ (XPK70; FKO))))))
ZMPЭK1 = ИЛИ (FKO; ИЛИ (X5KY08; И (ZMPЭK1; НЕ ( ИЛИ (XPK71; X5KY0A))))))
ZMPЭK2 = ИЛИ (XPK71; ИЛИ (X5KY0C; И (ZMPЭK2; НЕ ( ИЛИ (FKO; X5KY0B))))))
ZMCK1 = ИЛИ (X5KY0D; И (ZMCK1; НЕ ( ИЛИ (XPK73; X5KY0E))))
ZMCK2 = ИЛИ (XPK73; ИЛИ (X5KY0E; И (ZMCK2; НЕ (X5KY0D))))
ZMPIT1 = ИЛИ (X5KY07; И (ZMPIT1; НЕ ( ИЛИ (XPK81; X5KY09))))
ZMPIT2 = ИЛИ (XPK81; ИЛИ (X5KY09; И (ZMPIT2; НЕ (X5KY07))))
ZMPHK1 = ИЛИ (XPK64; ИЛИ (FKO; И (ZMPHK1; НЕ ( ИЛИ (XPK65; ИЛИ (X5KY00; ИЛИ (X5KY01; И (XOЦПМ; ZBШТ))))))))
ZMPHK2 = ИЛИ (XPK65; ИЛИ (X5KY00; И (ZMPHK2; НЕ ( ИЛИ (XPK64; ИЛИ (FKO; ИЛИ (X5KY05; И (XOЦПМ; ZBШТ))))))))
SCTT1 = И (SCTT18; ZЦПM1)
SCTT2 = И (SCTT18; ZЦПM2)
ZCTT1 = ИЛИ (SCTT1; И (ZCTT1; НЕ (XPK84)))
ZCTT2 = ИЛИ (SCTT2; И (ZCTT2; НЕ (XPK84)))
ZBLCTT = ИЛИ (XPK54; И (ZBLCTT; НЕ ( ИЛИ (XPK55; FKO))))
FCPY = XCPY
ZTCY2 = ИЛИ (FCPY; И (ZTCY2; НЕ (SCY2)))
ZCTCY2 = ИЛИ (SCY2; И (ZCTCY2; НЕ (XPK84)))
ZCPY = ИЛИ (FCPY; И (ZCPY; НЕ ( ИЛИ (SCPY; ИЛИ (X5KY21; X5KY22))))))
ZCTCPY = ИЛИ (SCPY; И (ZCTCPY; НЕ (XPK84)))
ZBLCPY = ИЛИ (XPK74; И (ZBLCPY; НЕ ( ИЛИ (XPK75; FKO))))
SCTT = И (SCTT18; ИЛИ ( НЕ (ZBLCTT); И (SCPY; НЕ (ZBLCPY))))
ZCTCPY = ИЛИ (SCPC; И (ZCTCPY; НЕ (XPK84)))
FKO = И ( ИЛИ (XKO13; XKO24); ИЛИ (ZPKO; XPK92))
ZPBM = XРБЛBM * ZBШТ + ZPBM * ^ ( XБЛBM * ZBШТ + FKO + X1KY27 )
```

Рис. 3.33. Преобразованный файл с формулами в функциональной форме

3.3.2. Преобразование логических формул в FBD-программы

Файл с функциональными формулами преобразуется в файл .isaxml, который программа ISaGRAF 6 интерпретирует в виде графической исполняемой программы на языке FBD (рис. 3.34).

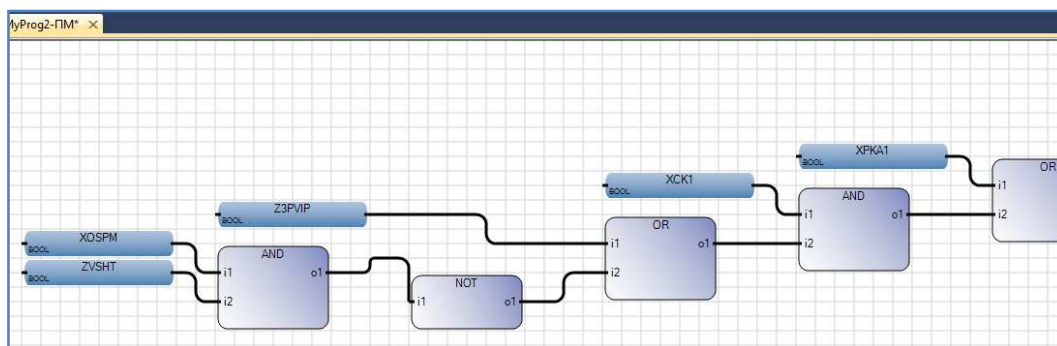


Рис. 3.34. Программа на языке FBD для формулы ZVIP1M

4. Разработка проектов СЛУ на языке ST

4.1. Описание языка ST

Язык структурированного текста – ST (StructuredText) – 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.

Основой ST-программы служат выражения, которые состоят из операндов (констант и переменных) и операций.

Операторы – являются «командами» языка программирования ST. Они должны заканчиваться точкой с запятой. Одна строка может содержать несколько операторов (отделяемых точками с запятой).

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, сим-

вол-разделитель «>» означает сравнение больше чем, а символ «+» операцию сложения и т.д.).

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

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

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

Рассмотрим операторы языка ST.

Вызов функций из st

Имя: имя вызываемой функции, написанной на языках IEC 61131 или «C»

Назначение: вызывает ST, IL, LD или FBD Функции или «C» функцию и получает возвращаемое значение

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

Операнды: тип возвращаемого значения и параметров вызова должен соответствовать интерфейсу, определенному для функции.

Возвращаемое значение: значение, возвращаемое функцией.

Вызовы функций могут быть использованы в любом выражении.

Вызов функциональных блоков из ST

Имя: имя экземпляра функционального блока

Назначение: вызывает функциональный блок из стандартной библиотеки или библиотеки пользователя и обращается к его возвращаемым параметрам.

Синтаксис: (* вызвать функциональный блок *)

<blockname>(<p1>, <p2> ...);

(* получить возвращаемые параметры *)

<result>:= <blockname>. <ret_param1>;

...

<result>:= <blockname>. <ret_paramN>;

Операнды: параметры являются выражениями, которые соответствуют типу параметров, специфицированных для этого ФБ

Возвращаемое значение: см. синтаксис получения возвращаемых параметров

Оператор присваивания

Имя: :=

Назначение: Результат вычисления выражения присваивается переменной.

Синтаксис: <variable> := <any_expression> ;

Операнды: переменная должна быть **Internal** или **Output** и выражение должно иметь тот же тип.

Выражение может быть вызовом функции

Оператор IF-THEN-ELSIF-ELSE

Имя: IF ... THEN ... ELSIF ... THEN ... ELSE ... END_IF

Назначение: выполняет один из нескольких списков предложений ST. Выбор осуществляется в соответствии со значением булевского выражения

Синтаксис: IF <Boolean_expression> THEN

<statement> ;

<statement> ;

...

ELSIF <Boolean_expression> THEN

<statement> ;

<statement> ;

...

ELSE

<statement> ;

<statement> ;

...

END_IF;

Предложения ELSE и ELSIF являются необязательными. Если предложение ELSE опущено и условие равно FALSE, то никакие инструкции не выполняются. Предложение ELSIF может использоваться многократно. Предложение ELSE, если используется, должно появляться только один раз в конце последовательности 'IF, ELSIF ...'.

Оператор CASE

Имя: CASE ... OF ... ELSE ... END_CASE

Назначение: выполняет один из нескольких списков предложений ST.

Выбор осуществляется в соответствии с целочисленным выражением

Синтаксис: CASE <integer_expression> OF

<value> : <statements> ;

<value> , <value> : <statements> ;

...

ELSE

<statements> ;

END_CASE;

Значениями Case должны быть целые константные выражения. Несколько значений, разделенных запятыми, могут предшествовать одному и

тому же списку предложений. Предложение ELSE является необязательным.

Оператор FOR

Имя: FOR ... TO ... BY ... DO ... END_FOR

Назначение: выполняет ограниченное число итераций, используя целую

индексную переменную

Синтаксис: FOR <index> := <mini> TO <maxi> BY <step> DO

<statement> ;

<statement> ;

END_FOR;

Операнды: index: внутренняя целая переменная, увеличивающаяся в каждом цикле итерации;

mini: начальное значение для индекса (перед первой итерацией);

maxi: максимально допустимое значение для индекса;

step: приращение индекса в каждом цикле итерации

Предложение [BY step] является необязательным. Если оно не приведено, то шаг приращения равен 1.

Предупреждение: поскольку ядро является **синхронной** системой, входные переменные не обновляются в течение итераций FOR.

Оператор WHILE

Имя: WHILE ... DO ... END_WHILE

Назначение: итерационная структура для группы ST предложений.

Условие "продолжения" оценивается ПЕРЕД любой итерацией

Синтаксис: WHILE <Boolean_expression> DO

<statement> ;

<statement> ;

...

END_WHILE

Предупреждение: поскольку ядро является **синхронной** системой, входные переменные не обновляются в течение итераций WHILE. Изменение состояния входной переменной не может быть использовано для описания условия предложения WHILE.

Оператор REPEAT

Имя: REPEAT ... UNTIL ... END_REPEAT

Назначение: итерационная структура для группы ST предложений. Условие "продолжения" оценивается ПОСЛЕ любой итерации

Синтаксис: REPEAT

<statement> ;

<statement> ;

...

UNTIL <Boolean_condition>

END_REPEAT ;

Предупреждение: поскольку ядро является синхронной системой, входные переменные не обновляются в течение итераций REPEAT. Изменение состояния входной переменной не может быть использовано для описания условия предложения REPEAT.

Оператор EXIT

Имя: EXIT

Назначение: выход из итерационных предложений FOR, WHILE или REPEAT

Синтаксис: EXIT

EXIT обычно используется в предложении IF внутри блока FOR, WHILE или REPEAT.

Оператор RETURN

Имя: RETURN

Назначение: прекращает выполнение текущей программы

Синтаксис: RETURN;

Операнды: (нет).

В блоке действий SFC предложение RETURN обозначает конец выполнения только данного блока.

Рассмотрим операторы, используемые для управления программой, написанной на языке SFC.

GSTART

Имя: GSTART

Назначение: запускает дочернюю программу SFC, устанавливая маркер на все её начальные шаги

Синтаксис: GSTART (<child_program>);

Операнды: указываемая SFC программа должна быть дочерней по отношению к программе, в которой написано предложение

Возвращаемое значение: (нет).

Потомки дочерней программы предложением GSTART автоматически не запускаются.

GKILL

Имя: GKILL

Назначение: убивает дочернюю SFC программу, удаляя маркеры, существующие на её шагах

Синтаксис: GKILL(<child_program>);

Операнды: указываемая SFC программа должна быть дочерней по отношению к программе, в которой написано предложение

Возвращаемое значение: (нет).

GFREEZE

Имя: GFREEZE

Назначение: замораживает дочерний модуль SFC (программу или функциональный блок); приостанавливает его выполнение. Приостановленный ПМ SFC может затем быть перезапущен с использованием предложения GRST

Синтаксис: GFREEZE (<child_program>);

Операнды: указываемая SFC программа должна быть дочерней по отношению к программе, в которой написано предложение

Возвращаемое значение:(нет).

Потомки дочерней программы замораживаются автоматически совместно с указанной программой.

GRST

Имя: GRST

Назначение: перезапускает дочернюю программу, замороженную предложением GFREEZE: все маркеры, удаленные GFREEZE, восстанавливаются

Синтаксис: GRST (<child_program>);

Операнды: указываемая SFC программа должна быть дочерней по отношению к программе, в которой написано предложение

Возвращаемое значение:(нет).

Потомки дочерней программы предложением GRST перезапускаются автоматически.

GSTATUS

Имя: GSTATUS

Назначение: возвращает текущее состояние SFC программы

Синтаксис: <var> := GSTATUS (<child_program>);

Операнды: указываемая SFC программа должна быть дочерней по отношению к программе, в которой написано предложение

Возвращаемое значение:

0 = программа неактивна (убита)

1 = программа активна (запущена)

2 = программа заморожена.

4.2. Разработка программ на языке ST

Структура разрабатываемого проекта приведена на рис. 4.1.

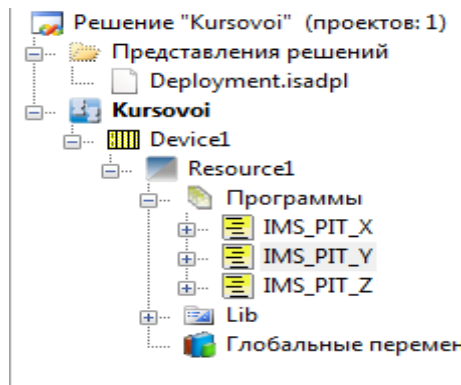


Рис. 4.1. Структура проекта

1.2.1. Словарь проекта

Словарь проекта (рис.4.2) представляет собой набор глобальных переменных.

В ISAGRAF переменные могут иметь одно из следующих направлений:

- Var. Внутренняя переменная, обновляемая программами.
- VarInput. Переменная, подключенная к устройству ввода (регенерируется системой).
- VarOutput. Переменная, подключенная к устройству вывода.

Имя	Тип данных	Алиас	Направление	Атрибут
X063_12	BOOL		VarInput	Read
X063_1	BOOL		VarInput	Read
X000_1	BOOL		VarInput	Read
X000_12	BOOL		VarInput	Read
X067_10	BOOL		VarInput	Read
XBPIT	BOOL		Var	Read/Write
XBORIT	BOOL		Var	Read/Write
XBO	BOOL		Var	Read/Write
XB	BOOL		Var	Read/Write
XTM	BOOL		Var	Read/Write
ZPZV1	BOOL		Var	Read/Write
ZPZV2	BOOL		Var	Read/Write
ZPZV3	BOOL		Var	Read/Write
ZPOZ1	BOOL		Var	Read/Write
ZPOZ2	BOOL		Var	Read/Write
ZPOC1	BOOL		Var	Read/Write
ZPOC2	BOOL		Var	Read/Write
ZGK1	BOOL		Var	Read/Write
ZGK2	BOOL		Var	Read/Write
ZGK3	BOOL		Var	Read/Write

Рис. 4.2. Словарь проекта

1.2.2. Разработка программ

В данном проекте все переменные являются глобальными переменными булевого типа. Каждая переменная имеет имя, заданное по принципу транслитерации переменных из исходной таблицы Excel[4].

Программа IMS_PIT_X, представленная ниже, содержит коды, определяющие значения переменных «Шины питания», «Пороговые элементы».

(*Шины питания*)

(*ХБПИТ=Х063_12*)

ХВРПТ :=Х063_12;

(*ХБОПИТ=Х063_1*)

ХВОРПТ := Х063_1;

(*Пороговые элементы*)

(*Рднс1 = 1*)

Rdns1 := TRUE;

(*Рднс2 = 1*)

Rdns2 := TRUE;

Программа IMS_PIT_Y содержит коды (фрагменты), определяющие значения переменных «Разряды слов», «Выходы», «Силовые контакты», «Сухие контакты», «Электронные ключи».

(*Разряды слов*)

(*СД18_00=ЗПОЗ1*)

CD18_00 := ZPOZ1;

(*СД18_01=ЗПОЗ2*)

CD18_01 := ZPOZ2;

(*СД18_02=ЗГК1*)

CD18_02 := ZGK1;

(*Выходы*)

(*УПЗВ1П=ЗПЗВ1*ХБОПИТ*)

УРЗВ1Р := ЗРЗВ1 AND ХВОРПТ;

(*УПЗВ1=ХБПИТ*)

УРЗВ1 := ХВРПТ;

(*УПЗВ2П=ЗПЗВ2*ХБОПИТ*)

УРЗВ2Р := ЗРЗВ2 AND ХВОРПТ;

(*Силовые контакты*)

(*Х056_1=УПЗВ1П*)

Х056_1 := УРЗВ1Р;

(*Х056_33=УПЗВ1*)

Х056_33 := УРЗВ1;

(*Х056_5=УПЗВ2П*)

Х056_5 := УРЗВ2Р;

(*Сухие контакты*)

(*Х067_1=УРДНС_ТМ*)

Х067_1 := УРДНС_ТМ;

(*Х067_3=УДНС1_ТМ*)

Х067_3 := УДНС1_ТМ;

(*Электронные ключи*)

(*Х067_26=УРУБИС_ТМ*)

Х067_26 := УРУБИС_ТМ;

(*Х067_27=УРЗБИС_ТМ*)

Х067_27 := УРЗБИС_ТМ;

Программа IMS_PIT_Z содержит коды, определяющие значения переменных «Функции памяти».

(*Функции памяти*)

(*ЗПЗВ1=ХЗКУ00+ЗПЗВ1*^ХЗКУ02*)

ЗРЗВ1 :=ХЗКУ00 OR ЗРЗВ1 AND NOT

(*ЗГК2=ХЗКУ0F+ЗГК2*^ХЗКУ12*)

ЗГК2 :=ХЗКУ0F OR ЗГК2 AND NOT ХЗКУ12;

(*ЗГК3=ХЗКУ10+ЗГК3*^ХЗКУ13*)

```

XZKU02;
(*ZP3B2=X3KY01+ZP3B2*^X3KY03*)
ZPV2 :=XZKU01 OR ZPV2 AND NOT
XZKU03;

```

```
ZGK3 :=XZKU10 OR ZGK3 AND NOT XZKU13;
```

1.2.3. Монтирование переменных

После написания программы на языке ST мы должны смонтировать переменные. Ниже приведены примеры смонтированных входных и выходных переменных (рис. 4.3 – 4.4). Переменные монтируем по двум типам: входные булевские переменные и выходные булевские переменные.

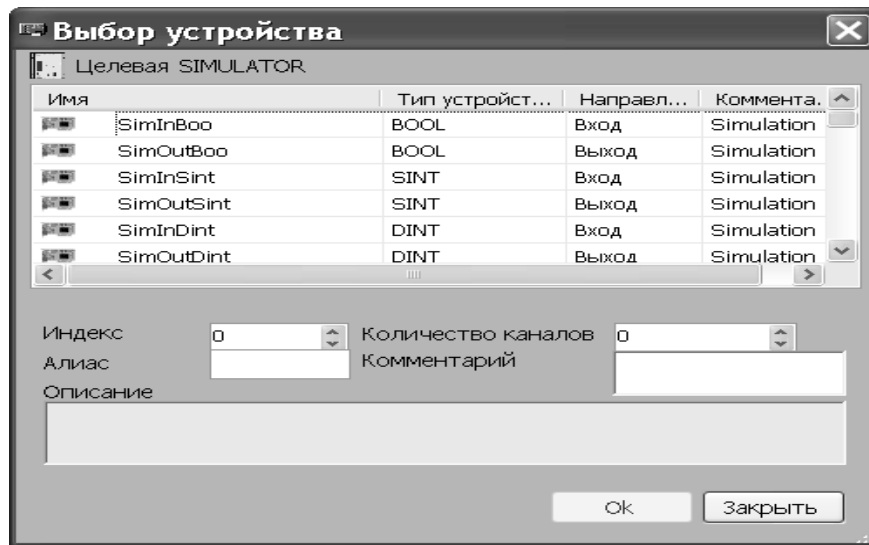


Рис. 4.3. Вид экрана выбора устройства

Ин...	Имя
0	%IX0.0=X063_12
1	%IX0.1=X063_1
2	%IX0.2=X000_1
3	%IX0.3=X000_12
4	%IX0.4=X067_10
5	%IX0.5=XZKU00
6	%IX0.6=XZKU02
7	%IX0.7=XZKU01
8	%IX0.8=XZKU03
9	%IX0.9=XZKU1C
10	%IX0.10=XZKU1D
11	%IX0.11=XZKU0A
12	%IX0.12=XZKU0C
13	%IX0.13=XZKU0V
14	%IX0.14=XZKU0D
15	%IX0.15=XZKU04
16	%IX0.16=XZKU06
17	%IX0.17=XZKU05
18	%IX0.18=XZKU07
19	%IX0.19=XZKU0E
20	%IX0.20=XZKU11
21	%IX0.21=XZKU0F

Ин...	Имя
0	%QX1.0=X056_1
1	%QX1.1=X056_33
2	%QX1.2=X056_5
3	%QX1.3=X056_37
4	%QX1.4=X056_9
5	%QX1.5=X056_11
6	%QX1.6=X056_13
7	%QX1.7=X056_15
8	%QX1.8=X056_22
9	%QX1.9=X056_24
10	%QX1.10=X056_29
11	%QX1.11=X056_31
12	%QX1.12=X057_1
13	%QX1.13=X057_3
14	%QX1.14=X057_13
15	%QX1.15=X057_5
16	%QX1.16=X057_9
17	%QX1.17=X057_18
18	%QX1.18=X057_22
19	%QX1.19=X057_25
20	%QX1.20=X057_29
21	%OX1.21=X057_31

Рис. 4.4. Смонтированные входные и выходные булевы переменные

4.3. Отладка программST

На рисунках 4.5 – 4.6 приведены примеры результатов выполнения кода программы IMS_PIT.

Имя	Значение
(000) x063_12	TRUE
(001) x063_1	FALSE
(002) x000_1	FALSE
(003) x000_12	FALSE
(004) x067_10	FALSE
(005) xZKU00	FALSE
(006) xZKU02	FALSE
(007) xZKU01	FALSE
(008) xZKU03	FALSE
(009) xZKU1C	FALSE
(010) xZKU1D	FALSE
(011) xZKU0A	FALSE
(012) xZKU0C	FALSE
(013) xZKU0V	FALSE
(014) xZKU0D	FALSE
(015) xZKU04	FALSE
(016) xZKU06	FALSE
(017) xZKU05	FALSE
(018) xZKU07	FALSE
(019) xZKU0E	FALSE
(020) xZKU11	FALSE
(021) xZKU0F	FALSE
(022) xZKU12	FALSE
(023) xZKU10	FALSE
(024) xZKU13	FALSE
(025) xZKU18	FALSE
(026) xZKU1V	FALSE

Имя	Значение
(000) x056_1	FALSE
(001) x056_33	TRUE
(002) x056_5	FALSE
(003) x056_37	TRUE
(004) x056_9	FALSE
(005) x056_11	TRUE
(006) x056_13	FALSE
(007) x056_15	TRUE
(008) x056_22	FALSE
(009) x056_24	TRUE
(010) x056_29	FALSE
(011) x056_31	TRUE
(012) x057_1	FALSE
(013) x057_3	FALSE
(014) x057_13	TRUE
(015) x057_5	FALSE
(016) x057_9	TRUE
(017) x057_18	TRUE
(018) x057_22	TRUE
(019) x057_25	TRUE
(020) x057_29	FALSE
(021) x057_31	FALSE
(022) x057_33	FALSE
(023) x057_35	FALSE
(024) x057_37	FALSE
(025) x057_39	FALSE
(026) x060_1	FALSE

Рис. 4.5. Пример результата выполнения сценария программы IMS_PIT

Имя	Значение
(000) x063_12	TRUE
(001) x063_1	TRUE
(002) x000_1	FALSE
(003) x000_12	FALSE
(004) x067_10	FALSE
(005) xZKU00	FALSE
(006) xZKU02	FALSE
(007) xZKU01	FALSE
(008) xZKU03	FALSE
(009) xZKU1C	FALSE
(010) xZKU1D	FALSE
(011) xZKU0A	FALSE
(012) xZKU0C	FALSE
(013) xZKU0V	FALSE
(014) xZKU0D	FALSE
(015) xZKU04	FALSE
(016) xZKU06	FALSE
(017) xZKU05	FALSE
(018) xZKU07	FALSE
(019) xZKU0E	FALSE
(020) xZKU11	FALSE
(021) xZKU0F	FALSE
(022) xZKU12	FALSE
(023) xZKU10	FALSE
(024) xZKU13	FALSE

Имя	Значение
(000) x056_1	FALSE
(001) x056_33	TRUE
(002) x056_5	TRUE
(003) x056_37	TRUE
(004) x056_9	FALSE
(005) x056_11	TRUE
(006) x056_13	TRUE
(007) x056_15	TRUE
(008) x056_22	FALSE
(009) x056_24	TRUE
(010) x056_29	FALSE
(011) x056_31	TRUE
(012) x057_1	FALSE
(013) x057_3	FALSE
(014) x057_13	TRUE
(015) x057_5	FALSE
(016) x057_9	TRUE
(017) x057_18	TRUE
(018) x057_22	TRUE
(019) x057_25	TRUE
(020) x057_29	FALSE
(021) x057_31	FALSE
(022) x057_33	TRUE
(023) x057_35	TRUE
(024) x057_37	TRUE

Рис. 4.6. Пример результата выполнения сценария программы IMS_PIT

Изменяя значения входных переменных, наблюдаем и оцениваем изменения выходных переменных.

5. Разработка проектов СЛУ на языке SFC

5.1. Описание языка SFC

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.

Логические структуры, связанные с шагом, обрабатываются до тех пор, пока не произойдет событие, предписывающее перейти к обработке другого шага. Каждому переходу сопоставлено логическое условие, а шагу – совокупность действий.

SFC-программа – это графически представленная совокупность шагов и переходов, соединенных направленными связями. Основное правило при построении схем: шаги не могут следовать подряд; переходы тоже не могут следовать подряд.

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;

уровень 2 – программирование действий внутри шага или условий, присоединённых к переходу, на языке ST или IL.

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

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.

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.

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 схождения 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.

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.

Элементы языка SFC в системе ISaGRAF 6 представлены ниже на рис. 5.1 – 5.15.

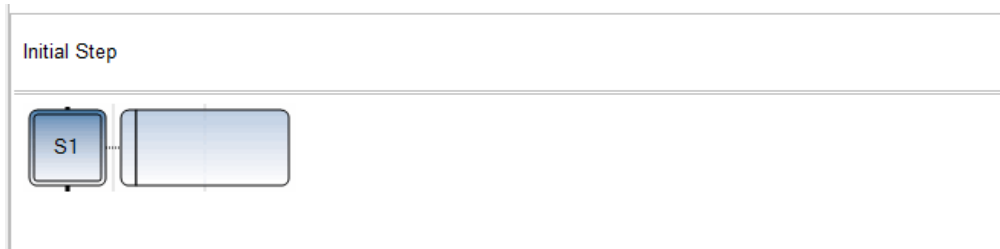


Рис. 5.1. Начальный шаг



Рис. 5.2. Шаг



Рис. 5.3. Активный и неактивный шаг

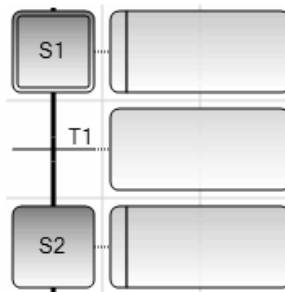


Рис. 5.4. Переход

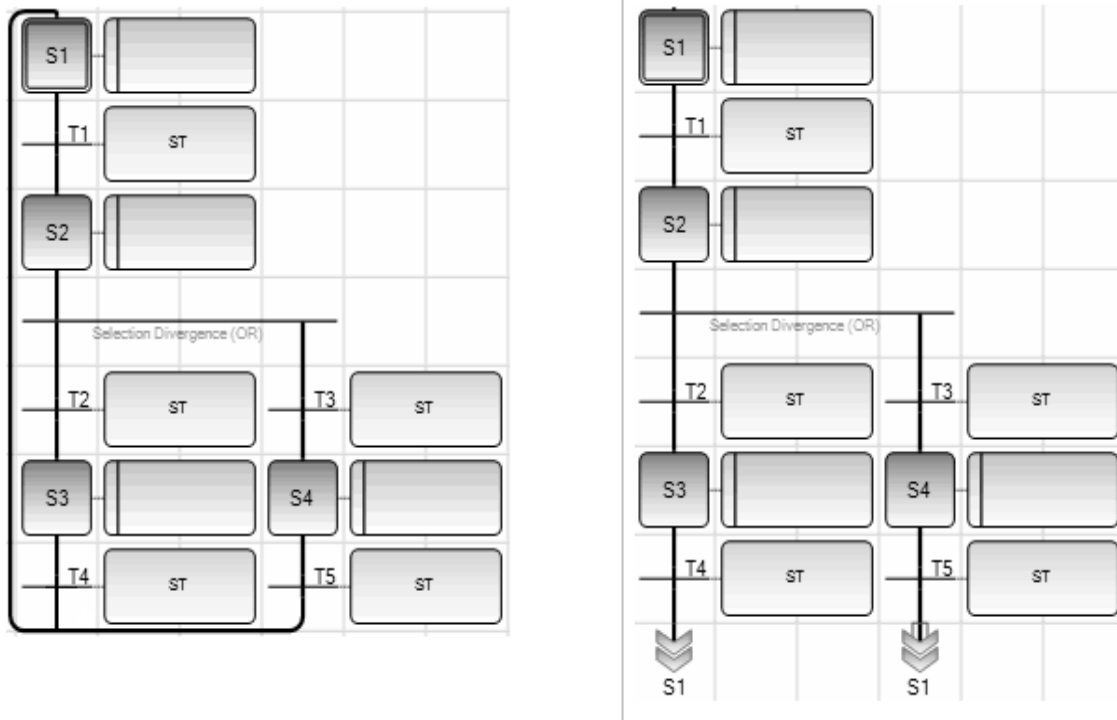


Рис. 5.5. Длинный переход

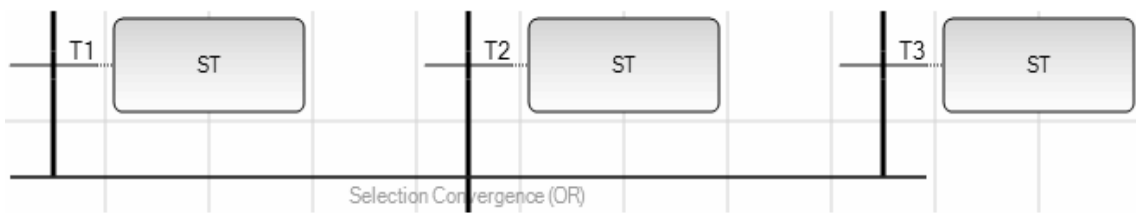
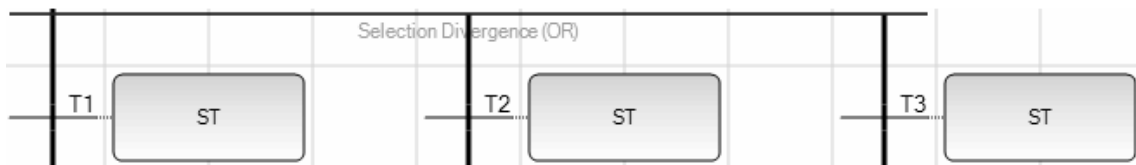


Рис. 5.6. Альтернативное схождение



(* SFC Program with selection divergence and convergence *)

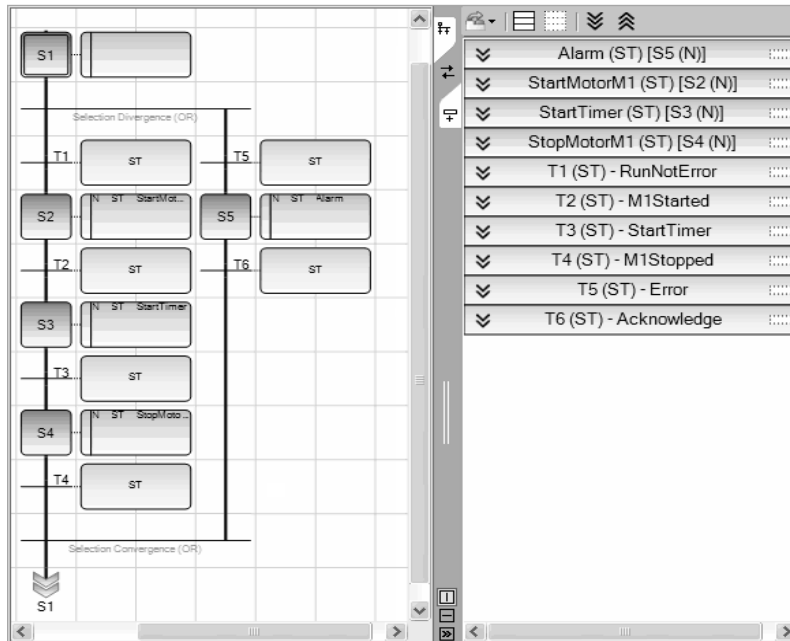
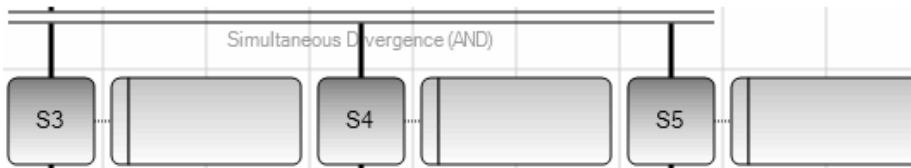


Рис. 5.7. Альтернативное расхождение



(* SFC program with simultaneous divergence and convergence *)

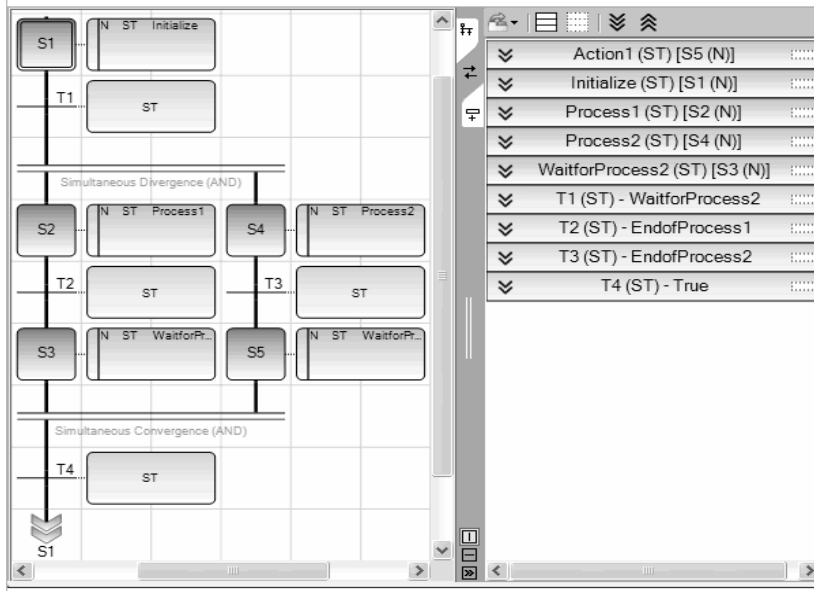


Рис. 5.8. Параллельное расхождение

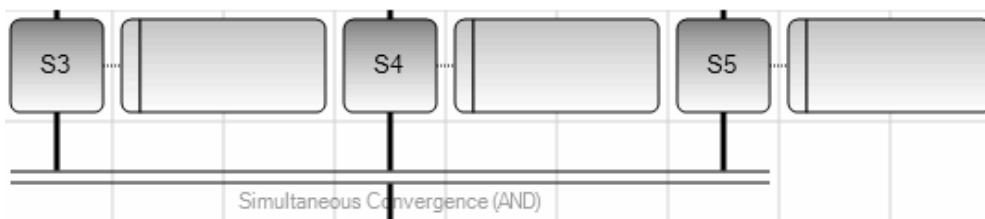


Рис. 5.9. Параллельное схождение

5.2. Действия внутри шагов

Уровень 2 шага SFC представляет собой детальное описание **действий**, выполняемых в течение **активности шага**.

Это описание делается с использованием **текстовых свойств языка SFC** и других языков, таких как Структурированный Текст (**ST**) или язык Релейных Диаграмм (**LD**).

Основные типы действий:

- булевские действия со спецификаторами: 'Set', 'Reset' или 'Non-Stored';
- список команд, программируемых на ST, LD или IL, со спецификатором 'Pulse' или 'Non-Stored';
- SFC действия (управление дочерними SFC программами) со спецификатором 'Set', 'Reset' или 'Non-Stored'.

Несколько действий (одного и того же или разных типов) могут быть описаны в одном шаге.

Специальным средством, позволяющим использовать любой другой язык, является вызов функций и функциональных блоков (написанных на ST, IL, LD, FBD).

5.2.1. Булевские действия

Булевские действия присваивают значение логической переменной при активизации шага. Логические переменные могут быть выходными или внутренними. Им присваивается значение каждый раз, когда шаг становится активным или перестает быть активным. Синтаксис основных логических действий:

Логические переменные должны быть выходными (OUTPUT) или внутренними (MEMORY). Следующая SFC программа ведет себя таким образом:

Non a BooleanVariable присваивает переменной сигнал активности шага.

S on a BooleanVariable присваивает переменной значение TRUE, когда сигналактивности шага становится TRUE

R on a BooleanVariable присваивает переменной значение FALSE, когда сигналактивности шага становится TRUE

Имя переменной (S10.X - это активность шага S10)

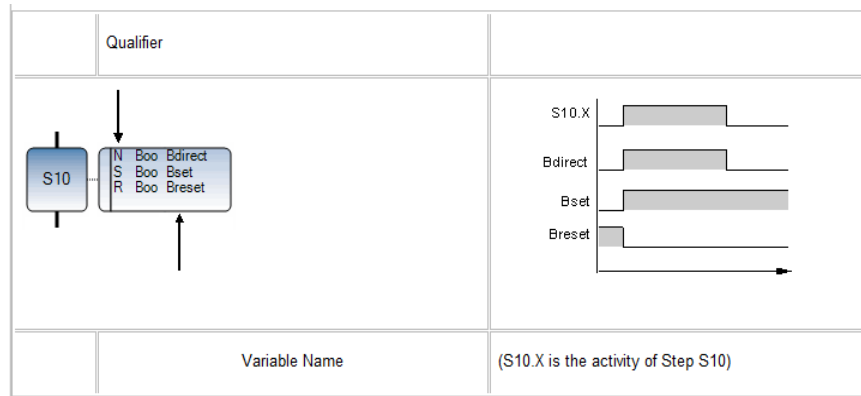
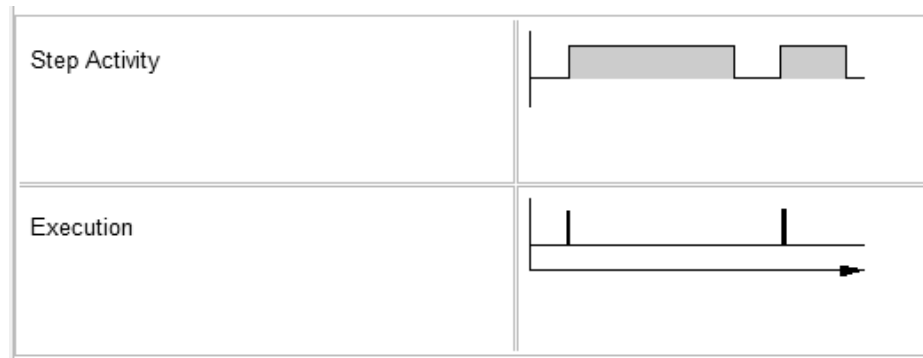


Рис. 5.10. Булевские действия

5.2.2. Импульсные действия

Импульсные действия - это список инструкций, которые выполняются только **однажды при активизации шага: спецификатор P1**, или **только однажды при деактивизации шага: спецификатор P0**. Инструкции пишутся в соответствии с синтаксисом языков ST, IL или LD.

Ниже показан результат импульсного действия со спецификатором P1:



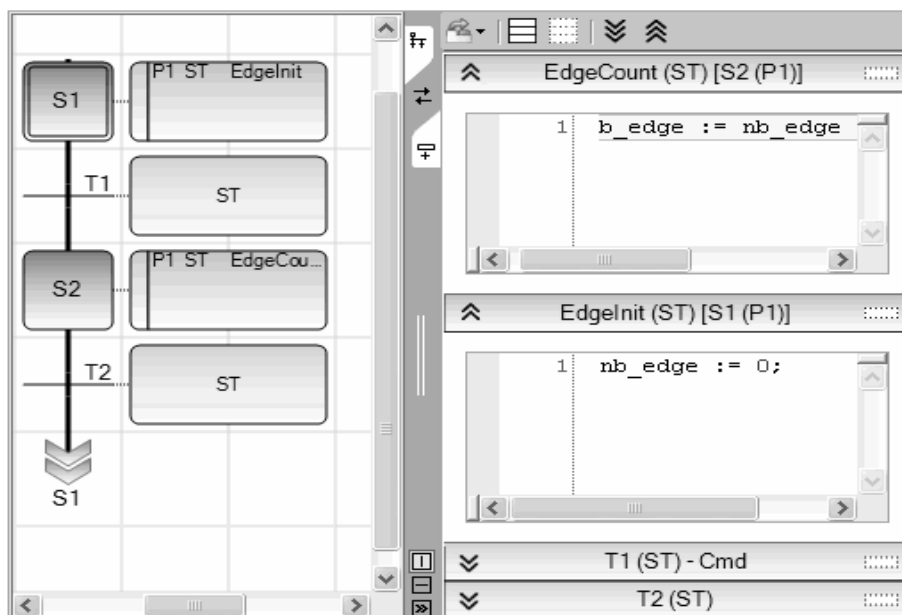
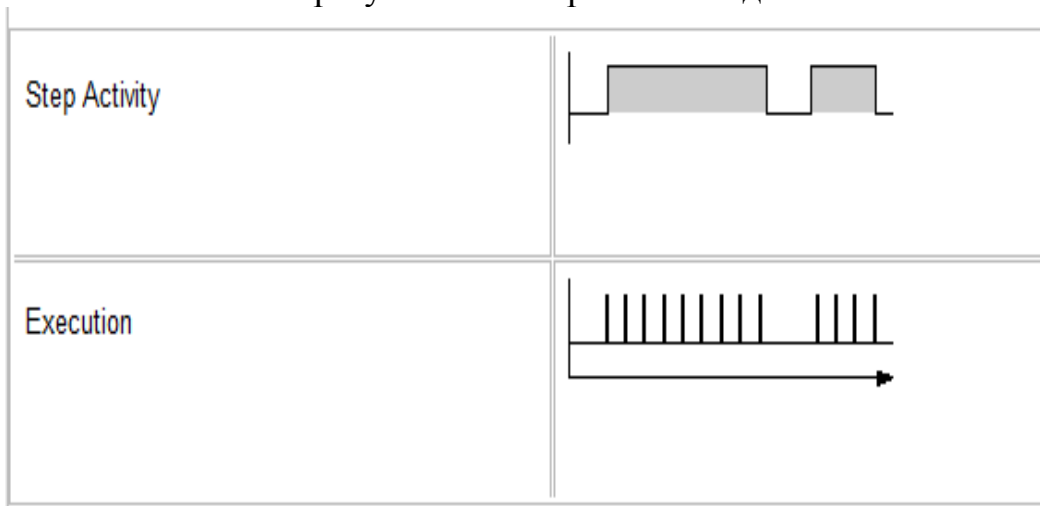


Рис. 5.11. Импульсные действия

5.2.3. Несохраняемые действия

Несохраняемое действие (нормальное) - это список инструкций ST, IL или LD, которые выполняются в **каждом** цикле в течении всего периода **активности** шага. Инструкции пишутся в соответствии с синтаксисом используемого языка. Несохраняемое действие обозначается спецификатором 'N'.

Ниже показан результат несохраняемого действия:



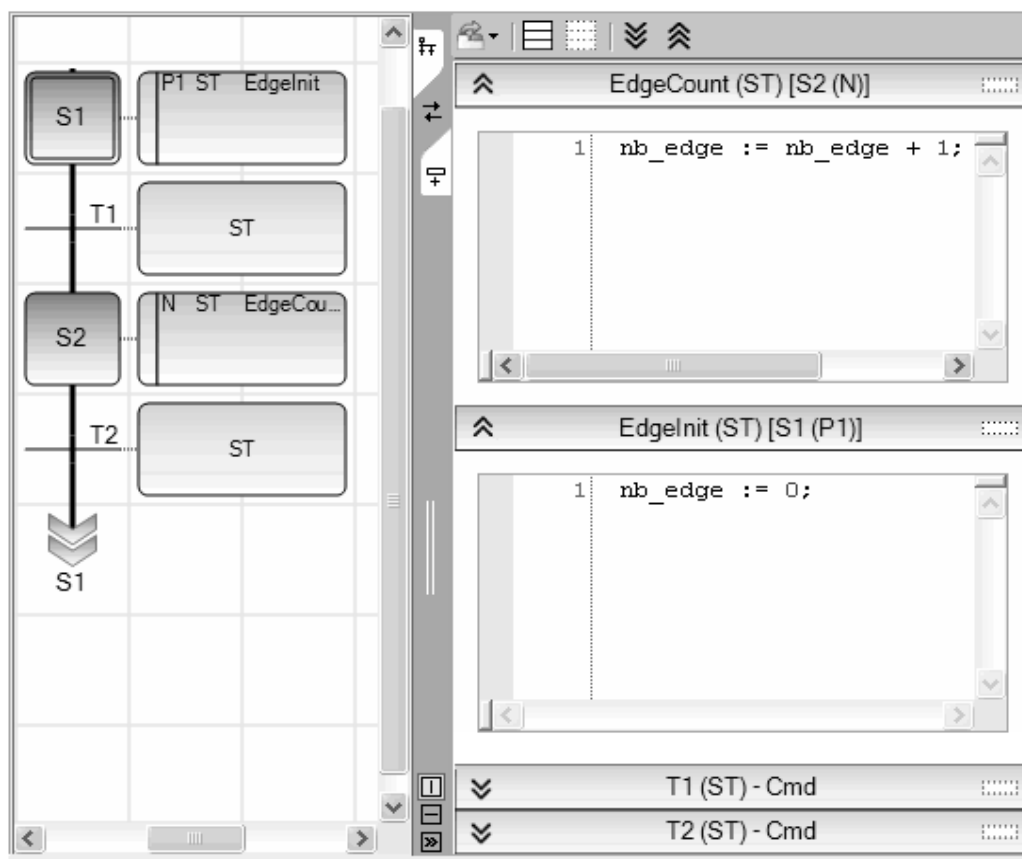


Рис. 5.12. Не сохраняемые действия

5.2.4. Действия SFC

SFC действие - это дочерняя последовательность SFC, стартуемая и убиваемая в соответствии с изменением сигнала активности шага. SFC действие может иметь спецификатор **N** (несохраняемое), **S** (установка), или **R** (сброс). Ниже приведен синтаксис основных SFC действий:

SFC последовательность, определенная как действие, должна быть дочерней SFC программой текущей редактируемой программы. Использование спецификаторов **S** (установка) или **R** (сброс) для SFC действий имеет тот же самый эффект, что и использование операторов **GSTART** и **GKILL** в импульсном действии на языке **ST**.

Спецификатор Имя действия

N on a child запустить дочернюю последовательность, когда шаг становится активным и убить её, когда шаг становится пассивным.

S on a child запустить дочернюю последовательность, когда шаг становится активным и ничего не делать, когда шаг становится пассивным.

R on a child убить дочернюю последовательность, когда шаг становится активным и ничего не делать, когда шаг становится пассивным.

5.3. Программирование переходов

(* SFC Program with ST programming for Transitions *)



Рис. 5.13. Программирование условий переходов на ST

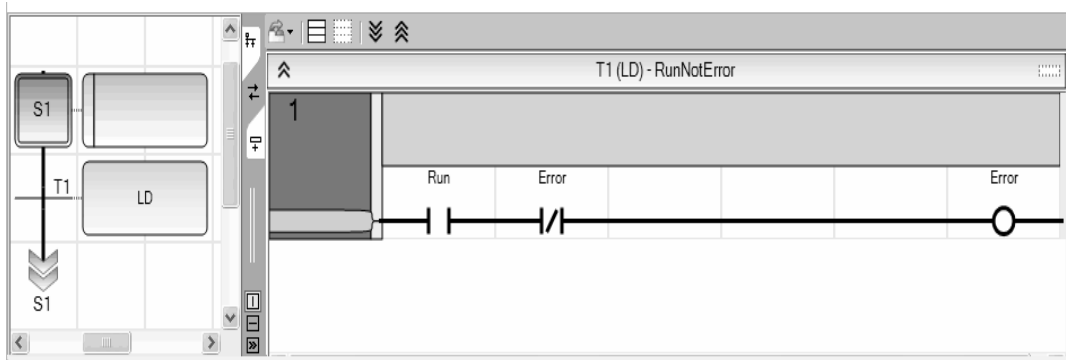


Рис. 5.14. Программирование условий переходов на LD

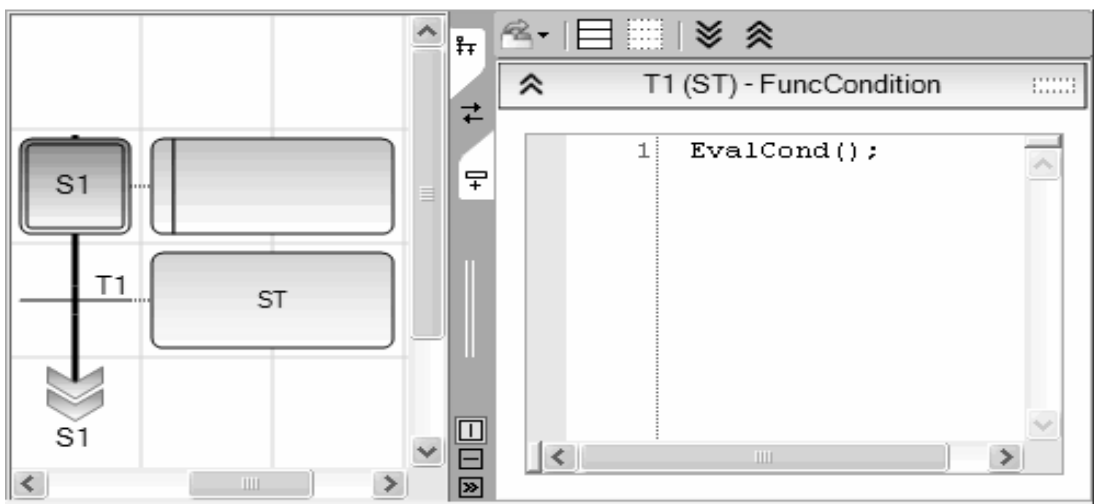


Рис. 5.15. Вызов функций из переходов

5.4. Разработка программ на языке SFC

Процесс разработки программ на языке SFC рассмотрим на примере проекта термообработки детали.

В данном примере имеются основные операции, необходимые для создания, построения и моделирования проекта управления процессом термической обработки детали. Зона обработки детали условно разделена на 4 секции (фиксированные положения термопары, на каждом из которых непрерывно измеряется средняя температура). Значения температуры в программе SFC задаются случайными величинами.

В зависимости от значения температуры выполняется включение (нагревание секции) или выключение (охлаждение секции) газовых горелок.

Разработка проекта состоит из двух этапов.

На *1-м этапе* создаем проект **Project1**, основную (**Main1**) и дочернюю (**Temp**) SFC программы (рис. 5.16).

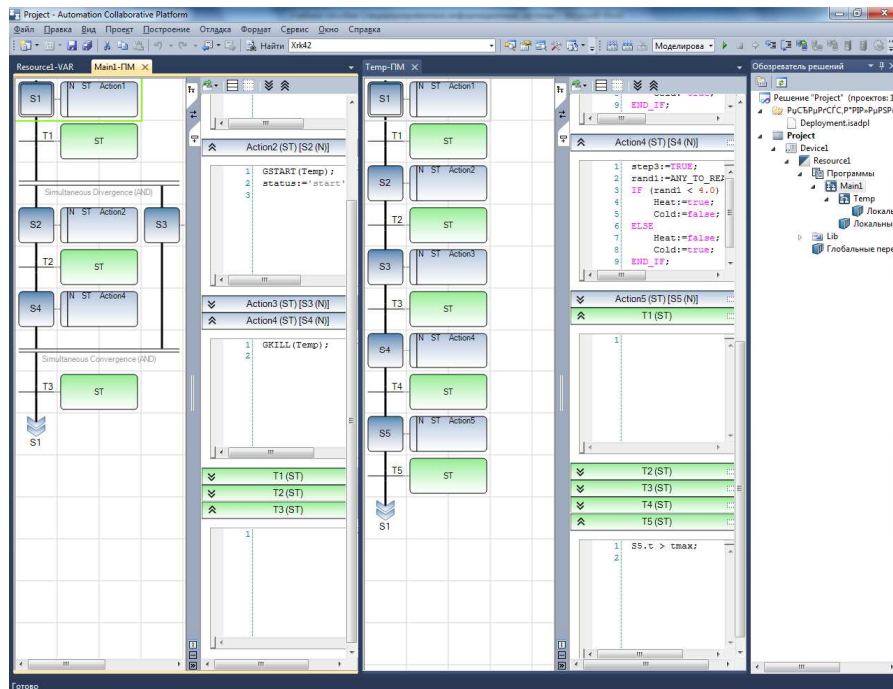


Рис. 5.16. Общая структура Project1

На *2-м этапе* решения задачи объявляем переменные (рис. 5.17) и записываем тексты программ.

Булевские:

bstart – команда старт или стоп электропривода термопары;

gas_on – команда включения или выключения газовых горелок;

step1, step2, step3, step4 – фиксированные положения термопары вдоль заготовки;

Heat – включение нагревателя секции;
 Cold – включение контура охлаждения секции.

Целые и действительные:

timeprog – целая (входное значение времени);
 rand1 – действительная (число секций).
 nbcycle – действительная(начальное значение 0).

Таймерные:

tmax – внутренняя (начальное значение t#100ms).

Сообщения:

status – выходная(основной статус).

Имя	Тип данных	Размерность	Алиас	Коммент	Начальное значение	Направление	Монтаж	Атрибут
bstart	BOOL					VarInput	%IX0.0	Read
gas_on	BOOL					VarInput	%IX0.1	Read
step1	BOOL					VarOutput	%QX1.0	Read/Write
step2	BOOL					VarOutput	%QX1.1	Read/Write
step3	BOOL					VarOutput	%QX1.2	Read/Write
step4	BOOL					VarOutput	%QX1.3	Read/Write
Heat	BOOL					VarOutput	%QX1.5	Read/Write
Cold	BOOL					VarOutput	%QX1.4	Read/Write
timeprog	INT					VarInput	%IW5.0	Read
rand1	REAL					VarOutput	%QR6.1	Read/Write
nbcycle	REAL				0.0	VarOutput	%QR6.0	Read/Write
tmax	TIME				t#0s	Var		Read/Write
status	STRING					VarOutput	%QS4.0	Read/Write

Рис. 5.17. Объявление переменных в словареProject1

Рассмотрим программирование **Main1** на 2-м уровне (язык ST) (рис. 5.18).

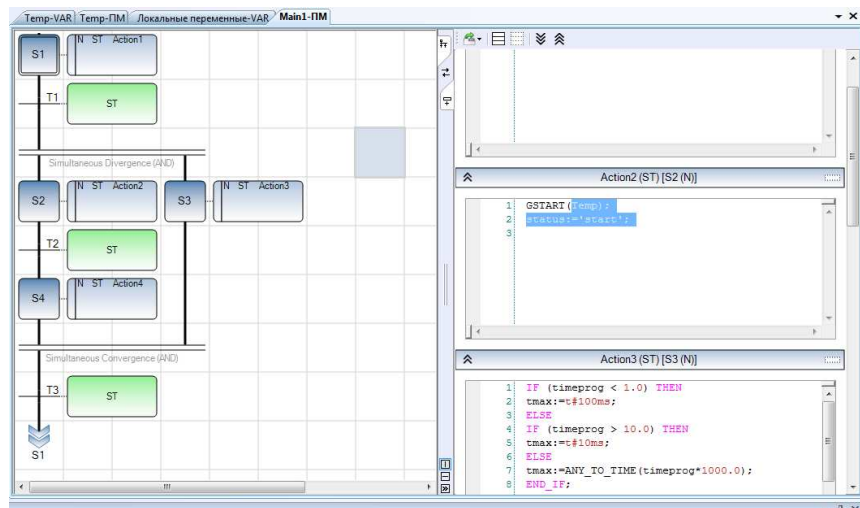


Рис. 5.18. 2-й уровень программы **Main1**

На 1-м шаге две входные булевы переменные устанавливаются в false, а переменная – сообщение status в момент активации шага принимает значение stop:

```
status:='stop';  
bstart:=false;  
gas_on:=false;
```

Условие перехода 1 к следующим шагам – это совместное выполнение команд bstart (включение электропривода термопары) и gas_on (подача газа на горелки):

```
bstart&gas_on;
```

Далее следует параллельное выполнение операций (двойная дивергенция). В левой ветви программы на шаге 2 вызывается дочерняя программа Temp и переменной status присваивается значение start:

```
GSTART(Temp);  
status:='start';
```

Условием перехода 2 к шагу 3 является перевод входных булевых переменных в состояние false:

```
not(bstart) & not(gas_on);
```

На 3-м шаге правой ветви программы для входного значения времени timeprog, устанавливаемого пользователем, определяется внутренняя переменная tmax – период времени между двумя соседними положениями термопары над деталью:

```
if (timeprog<1) then  
tmax:=t#100ms;  
else  
if (timeprog>10) then  
tmax:=t#10ms;  
else  
tmax:=ANY_TO_TIME(timeprog*1000.0);;  
end_if;  
end_if;
```

На 4-м шаге программы производится остановка дочерней программы:

```
GKILL(Temp);
```

Далее рассмотрим программирование Temp на 2-м уровне (язык ST) (рис. 5.19).

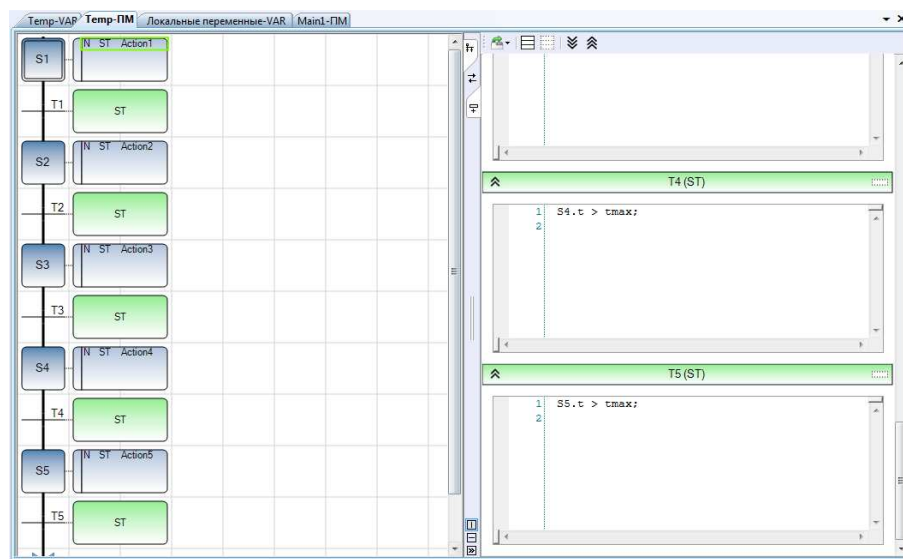


Рис. 5.19. 2-й уровень программы **Temp**

На *1-м шаге* выходным переменным положения термомпары присваиваются значения false, а выходной переменной количества итераций nbcycle – начальное значение 0:

```
step1(R);
step2(R);
step3(R);
step4(R);
nbcycle:=0.0;
```

После перехода на *2-й шаг* входная переменная step1 устанавливается в состояние true, переменная nbcycle получает единичное приращение, производится генерация случайной температуры в условной шкале 0...7 и с помощью оператора ветвления IF-THEN-ELSE включается нагрев или охлаждение 1-й секции детали:

```
step1:=TRUE;
rand1:=ANY_TO_REAL(rand(8));
IF (rand1 < 4.0) THEN
    Heat:=true;
    Cold:=false;
ELSE
    Heat:=false;
    Cold:=true;
END_IF;
```

На *3-м шаге* программы входная переменная step2 устанавливается в состояние true, производится генерация случайной температуры в ус-

ловной шкале 0...7 и с помощью оператора ветвления IF-THEN-ELSE включается нагрев или охлаждение 2-й секции детали:

```
step2:=TRUE;  
rand1:=ANY_TO_REAL(rand(8));  
IF (rand1 < 4.0) THEN  
    Heat:=true;  
    Cold:=false;  
ELSE  
    Heat:=false;  
    Cold:=true;  
END_IF;
```

Затем после задержки на время t_{max} осуществляется переход к 4-му шагу:

```
step3:=TRUE;  
rand1:=ANY_TO_REAL(rand(8));  
IF (rand1 < 4.0) THEN  
    Heat:=true;  
    Cold:=false;  
ELSE  
    Heat:=false;  
    Cold:=true;  
END_IF;
```

На 5-м шаге программы выполняются аналогичные действия:

```
step4:=TRUE;  
rand1:=ANY_TO_REAL(rand(8));  
IF (rand1 < 4.0) THEN  
    Heat:=true;  
    Cold:=false;  
ELSE  
    Heat:=false;  
    Cold:=true;  
END_IF;
```

Условия перехода к последующим шагам имеют одинаковый вид:

```
S2.t > tmax;  
S3.t > tmax;  
S4.t > tmax;  
S5.t > tmax;
```

После завершения программирования выполняем монтирование виртуальных плат (рис. 5.20 – 5.24).

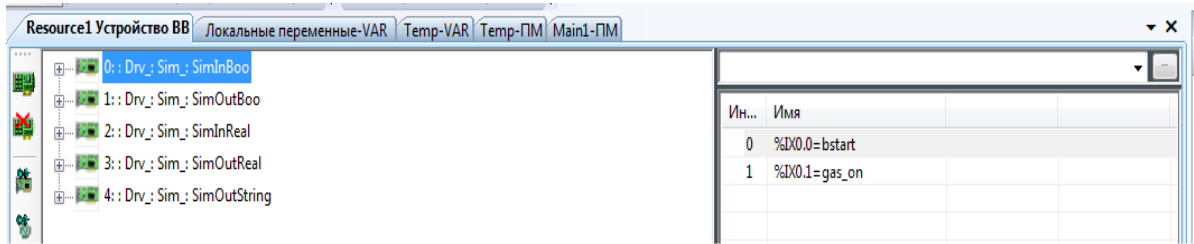


Рис. 5.20. Устройство ввода для булевых команд *bstart* и *gas_on*

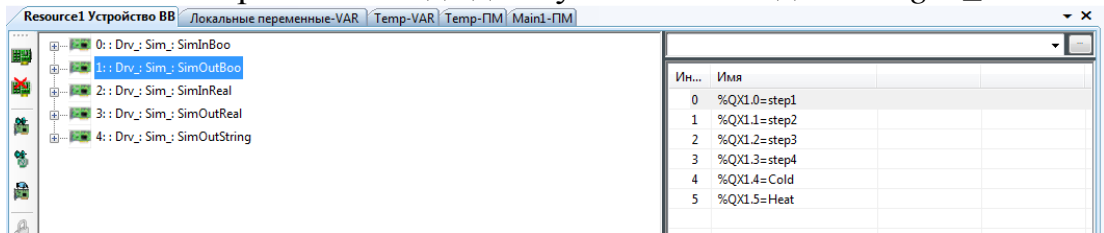


Рис. 5.21. Устройство вывода для булевых переменных *step1*, *step2*, *step3*, *step4*, *Cold*, *Heat*

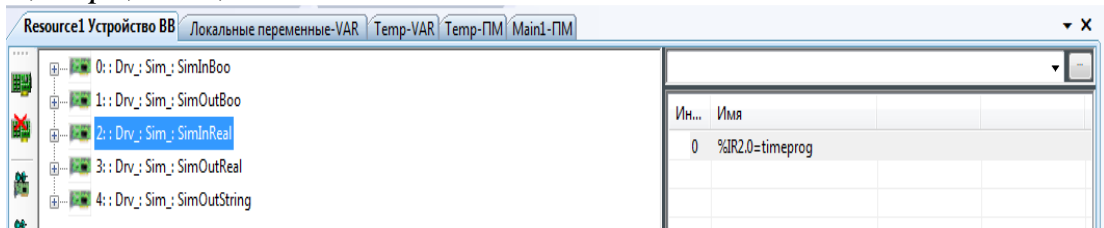


Рис. 5.22. Устройство ввода для действительной переменной *timeprog*

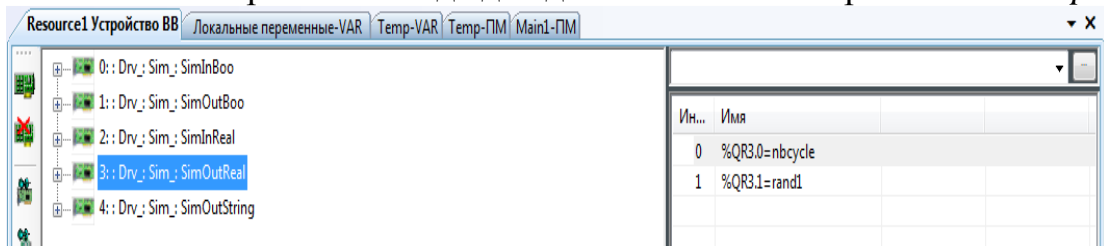


Рис. 5.23. Устройство вывода для действительных переменных *nbcycle* и *rand1*



Рис. 5.24. Устройство вывода для статусной переменной *status*

Выполняем моделирование, установив переменные bstart и gas_on в true (рис. 5.25 - 5.26).

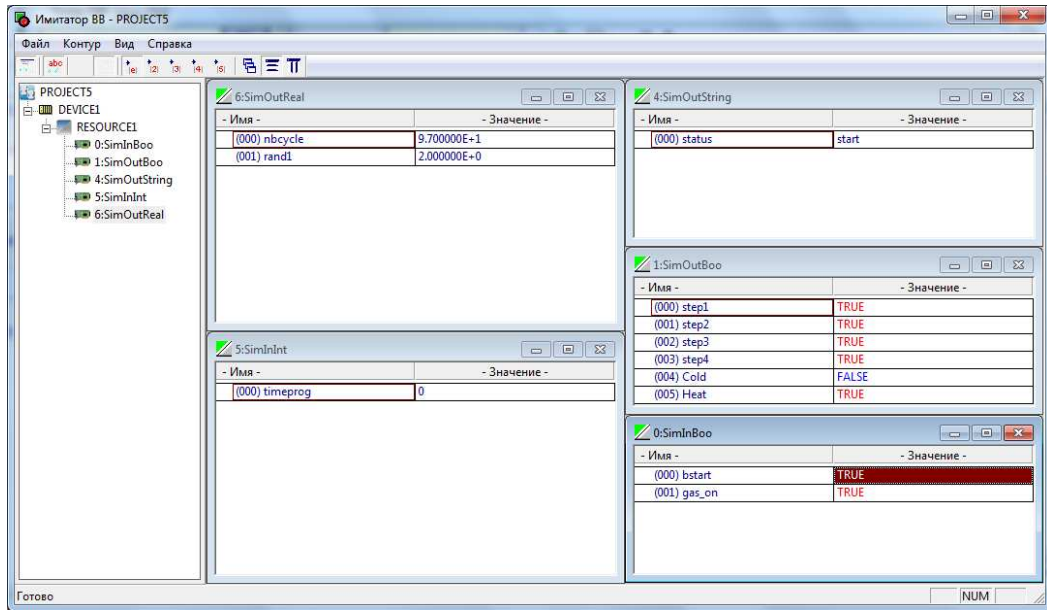


Рис. 5.25. Отображение процесса моделирования на виртуальных платах

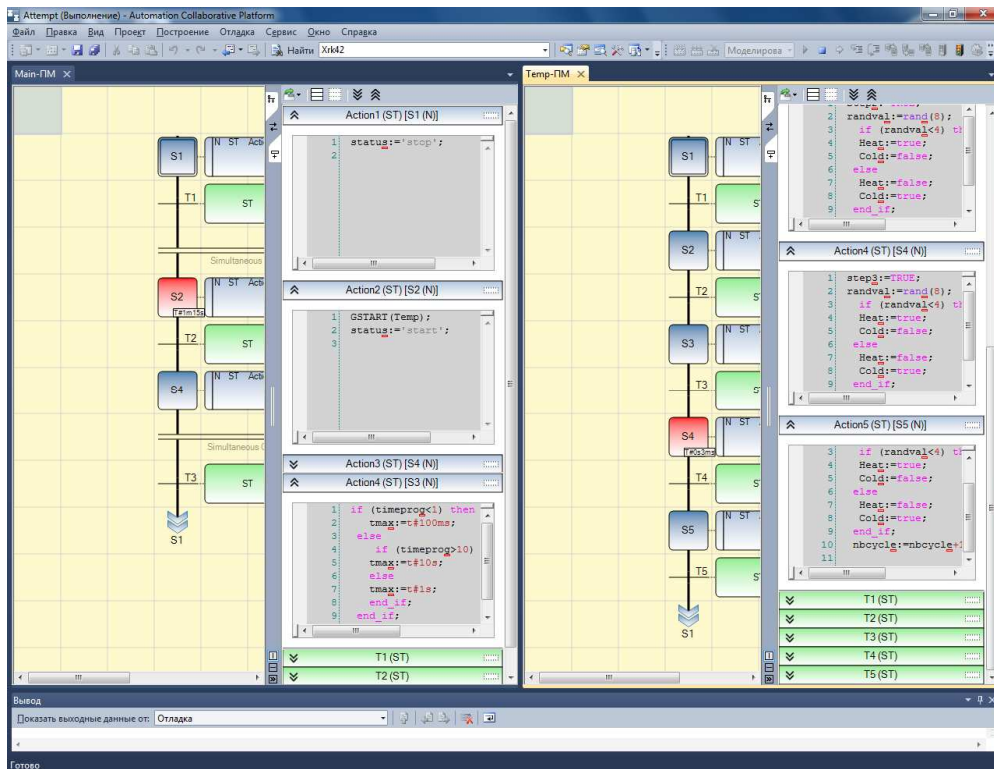


Рис. 5.26. Отображение процесса моделирования в программе SFC

5.3. Автоматизация программирования алгоритмов управления на языке SFC

5.3.1. Формальное описание СЛУ

Более двадцати лет назад с появлением дешевых микроконтроллеров началось их успешное внедрение в качестве базового элемента систем управления, и в настоящее время автоматизация исследовательских и промышленных комплексов уже не представляется возможным без использования цифровой техники. Цифровые системы управления обеспечивают проведение высокоточных научных экспериментов в области физики, химии, биологии, при исследовании космоса, определяют уровни обороноспособности страны и ее промышленного развития.

Классическими подходами к описанию задания на проектирование системы логического управления являются: таблицы автоматного графа, системы секвенций, логические схемы алгоритмов или логические схемы программ, а также описание на каком-либо алгоритмическом языке программирования. Возможны и другие формальные описания. Такое многообразие свидетельствует о том, что невозможно выбрать один единственный, пригодный для любого случая способ формализованного описания проектируемого устройства управления. Выбор того или иного способа во многих случаях определяется теми задачами, которые решаются при формализации, а также методом реализации. На алгоритмическом этапе проектирования устройства управления, когда создается описание алгоритма управления, прежде всего должны быть решены проблемы полноты и непротиворечивости задания.

Задание алгоритма функционирования на языке программирования высокого уровня требует от технолога-заказчика знания программирования, что, как правило, не выполняется. Преимуществом же такой записи является то, что при включении в контур управления вычислительной машины или какого-либо вычислительного устройства (микропроцессорного программируемого контроллера, однокристальной микроЭВМ и др.) возможно простое получение готовой программы для ее работы, так как преобразование с языка программирования в коды машины выполняется стандартной программой-транслятором.

Применение простого алгоритмического языка высокого уровня, удовлетворяющего требованиям структурного проектирования программ, позволяет самому технологу-заказчику формировать задание на проектирование устройства управления достаточно эффективно и быстро.

Структурное проектирование, как и программирование, стало необходимым при создании больших работоспособных проектов, в которые могут быть легко внесены добавления и исправления. Однако такое проектирование не только определяет свойство конечного продукта, но включает также соответствующую методологию, т. е. особенности мыслительного процесса, управляющего проектированием, для получения структурного решения. Методология, которой следует придерживаться в рамках структурного проектирования, заключается в пошаговой детализации. Пошаговая детализация представляет собой простой процесс, предполагающий первоначальное выражение логики в терминах гипотетического языка «очень высокого уровня» с последующей детализацией каждого предложения в терминах языка более низкого уровня, до тех пор пока наконец не будет достигнут уровень используемого языка.

Таким языком «очень высокого уровня» для формального описания логики сложных СЛУ и является рассматриваемый далее язык Геракл.

В приложении ГИПЕРСИСТЕМА реализовано алгоритмически эквивалентное преобразование программ на языке Геракл в программы на языке Delphi (допускается C), а также в программы на языках LD, FBDиSFC стандарта ИЕС 61131-3, что кардинально упростило задачу построения исполняемых моделей для компьютерного моделирования и анализа сложных СЛУ и их реализации на различных платформах.

5.3.2. Краткое описание языка Геракл

5.3.2.1. Цели разработки

При разработке языка Геракл ставятся следующие основные цели:

- Язык Геракл должен обеспечивать формальное описание сложных СЛУ на уровне разработки технических требований, не являясь языком программирования МПК, как например язык Рефлекс [7];
- Язык должен базироваться на основных принципах структурного программирования, объектно-ориентированного анализа и проектирования [8], которые стали необходимыми при создании больших работоспособных проектов;
- Язык должен быть адекватен решаемым задачам при организации общей структуры сложных систем логического управления на уровне ТЗ и практическим способам и формам представления этой структуры;
- Язык должен быть прост, удобен в использовании предметными специалистами и обеспечивать единую методику для создания на его базе исполняемых моделей на языках высокого уровня и языках технологического программирования для ПЛК и МПК;

- Формальное описание логики управления на языке Геракл должно выполняться в текстовом виде;
- Синтаксис языка должен представляться в русскоязычном варианте;
- Язык должен обеспечивать возможность алгоритмически эквивалентного преобразования программ на языке Геракл в наглядные графические представления в приложении MSVisio, а также в исполняемые графические программы на языках LD, FBD и SFC стандарта IEC 61131-3;
- Язык должен поддерживать основные концепции стандартов IEEE 830 и IEEE 1233, описывающих те характеристики, которые должны иметь технические требования (ТЗ):
 - Анализ требований, целью которого является обнаружение и устранение противоречий и неоднозначностей в требованиях, их уточнение и систематизация;
 - Описание требований. В результате этой деятельности требования должны быть оформлены в виде структурированного набора документов и моделей, который может систематически анализироваться, оцениваться с разных позиций и в итоге должен быть утвержден как официальная формулировка требований к системе;
 - Валидация требований, которая решает задачу оценки понятности сформулированных требований и их характеристик, в первую очередь, непротиворечивости и полноты, а также соответствия корпоративным стандартам на техническую документацию.

5.3.2.2. Структура описания СЛУ

Формальное описание СЛУ на языке Геракл имеет следующую концептуальную структуру:

СЛУ – описание СЛУ;

Команды- раздел описания, содержащий внешние воздействия на СЛУ;

Входы- раздел описания, содержащий воздействия датчиков исполнительных механизмов на СЛУ, а также внутренние воздействия на входные *Соединители*.

Формулы - раздел описания, содержащий структуру СЛУ в виде логические формул;

Супервизоры - раздел описания, содержащий формулы запрещенных ситуаций в СЛУ;

Алгоритмы- раздел описания, содержащий поведение СЛУ в виде алгоритмов;

Запуск_алгоритмов- раздел описания, содержащий условия запуска алгоритмов;

Таймеры- раздел описания, содержащий временные зависимости СЛУ и исполнительных механизмов;

Соединители- раздел описания, содержащий разъемные соединения частей СЛУ;

Функциональные_блоки – раздел описания, содержащий описания СЛУ, хранящиеся в депозитории приложения *ГИПЕРСИСТЕМА*;

Функции- раздел описания, содержащий описание функций, вызываемых из *Формул*, *Алгоритмов* и *Таймеров* и хранящихся в депозитории приложения *ГИПЕРСИСТЕМА*.

Основные принципы структурного проектирования и программирования наиболее полно реализованы в языке Delphi. Предлагаемый язык Геракл соответствует этим принципам и поддерживает нисходящее структурное описание СЛУ (иерархическое описание по принципу «сверху - вниз»).

На рис. 5.27 вышеописанная концептуальная структура описания СЛУ на языке Геракл представлена в виде программы на языке Delphi.

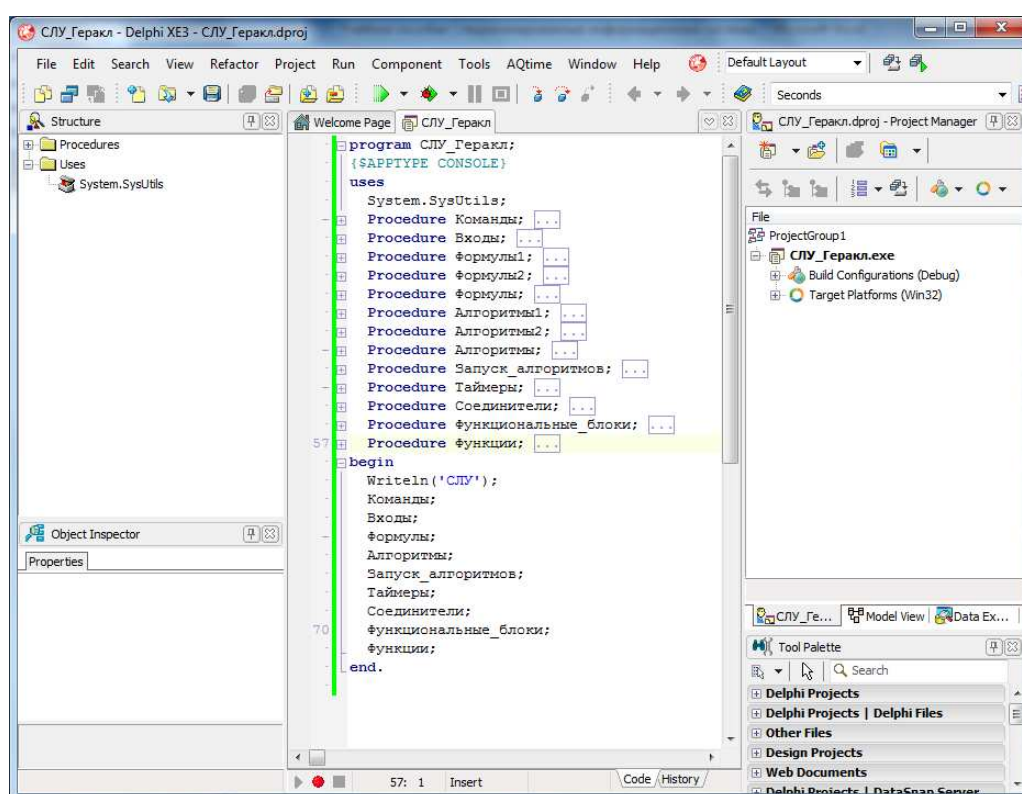


Рис. 5.27. Концептуальная структура описания СЛУ

Принципиально важно, что уже на этом макроуровне описание СЛУ может компилироваться и выполняться (рис. 5.28).

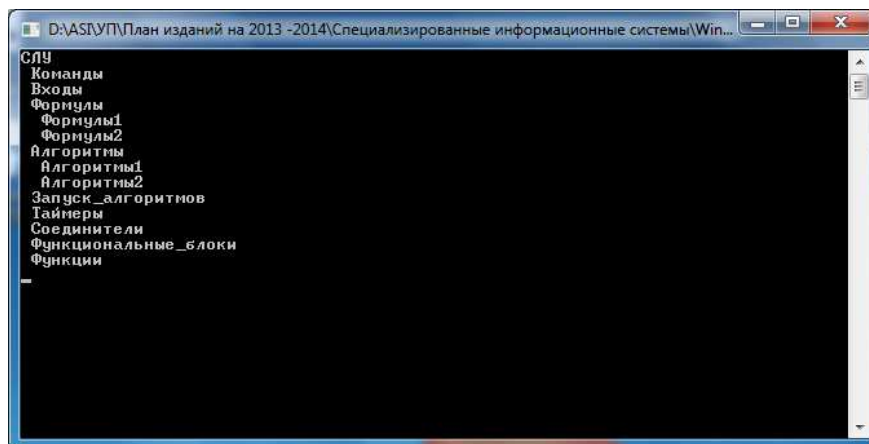


Рис. 5.28.Результат выполнения концептуального описания СЛУ

5.3.2.3. Синтаксис языка

Синтаксис— это раздел описания формального языка или языка программирования, содержащий вид, форму и структуру конструкций (без учета их значения или практической применимости).Задать синтаксис языка возможно, перечислив описание его конструкций с помощью метаязыка, например, с помощью форм Бэкуса-Наура (БНФ).

Язык Гераклне является строгим подмножеством языка Delphi, однако основные синтаксические элементы целесообразно определить через соответствующие элементы Delphi, используя БНФ, которые являются теоретически адекватным и практически применимым средством формализации синтаксиса языков программирования.

Основным обозначением, принятым в БНФ, является определяющий символ “::=“ , который отделяет определяемуюконструкцию от составляющих ее ранее определенныхбазовых конструкций.

Определяемая конструкция записывается слева от символа “::=“ в угловых скобках “<” и “>”.Другие обозначения покажем на примере.

В качестве примера приведем описание с помощью БНФ понятия*выражение*.

```

<выражение> ::= [ - ] <слагаемое> { <знак + или - > <слагаемое> }
<знак + или - > ::= + | -
<слагаемое> ::= <терм> { <знак * или / > <терм> }
<знак * или / > ::= * | /
<терм> ::= <число> | ( <выражение> )

```

Рассмотрим также в качестве примера используемые в языке Геракл понятия *Перечисляемый тип* и *Интервальный тип*.

Перечисляемый тип определяет упорядоченное множество значений путем перечисления идентификаторов, выражающих эти значения как постоянные.

<Перечисляемый тип> := ТУРЕ< идентификатор типа> = (<идентификатор>[,<идентификатор>,...]);

Интервальный тип определяет некоторое подмножество значений, которые может принимать данная переменная, задавая наименьшее и наибольшее значения порядкового типа.

< Интервальный тип> := ТУРЕ< идентификатор типа>=< константа>.
<константа>;

Основные синтаксические элементы языка Геракл, используемые в вышеописанных разделах описания СЛУ, можно представить через синтаксические конструкции Delphi (выделены жирным шрифтом) в виде:

<ИМЯ>::= **Идентификатор;**

<ТИП>::= **Порядковый тип;**

<ПРЕФИКС>::= **Идентификатор;**

<СУФФИКС>::= **Идентификатор;**

<КОМ>::= **Комментарий;**

<МЕТКАРАЗДЕЛА>::= **Идентификатор;**

<ОПЕРАЦИЯ>::= **Операция отношения / Булева операция / In;**

<ФОРМУЛА>::= **Логическое выражение;**

<АЛГОРИТМ>::= **Процедура;**

<ДЕЙСТВИЕ>::= **Оператор присваивания;**

<ПЕРЕХОД>::= **Оператор Goto;**

<МЕТКА>::= **Целое число;**

<ДЛЯ>::= **Оператор With;**

<ЕСЛИ>::= **Оператор IF;**

<ВЫБОР>::= **Оператор Case;**

<ВСЕ> := **Execute(Именованные потоки);**

<ИМЯ>(<СПИСОК ПАРАМЕТРОВ>) ::= **Вызов функции;**

<СПИСОК ПАРАМЕТРОВ>::= **Список параметров;**

<ВКЛЮЧИТЬ><ИМЯ>[(<СПИСОК ИМЕНОВАННЫХ ПАРАМЕТРОВ>)]:= **Директива Include;**

<СПИСОК ИМЕНОВАННЫХ ПАРАМЕТРОВ>:= **Список именованных параметров;**

5.3.2.4. Пример описания СЛУ

Формальное описание на языке Геракл простой системы логического управления рассмотрим на примере СЛУ технологической машины для мойки автомобилей [1].

СЛУ Машина_мойки_автомобилей

Команды

X0_0

КОМ Готовность мойки

X1_0

КОМ Включение мойки

X1_1

КОМ Предварительная мойка

X1_2

КОМ Главная мойка

X1_6

КОМ Инициализация

X1_7

КОМ Подтверждение аварии

Входы

КОМ Формируются автоматически

Формулы

$Z40_0 = X0_0 * X0_1 * X0_2 * X0_3 * X0_5$

$Z40_2 = X0_1 * X0_2 * X0_3$

$Y5_1 = X0_1 * X0_2 * X0_3$

$Y5_0 = Z40_3$

$Z40_3 = (X1_0 * (Z40_0 + X1_7) + Z40_3) * (E0_4 + E0_5) * (Z2_2 + Y4_4) * Z40_2 * X1_6 * X0_0$

$Y4_1 = Z40_3 * Z2_0 * Z2_2$

$Z2_0 = ((Y4_1 * X0_4) + Z2_0) * (Z2_2 + X1_6)$

$Y4_3 = Z40_3 * (Y4_1 * Z2_0) + (Y4_0 * Y4_5) + (Y4_6 * Z2_2)$

$Y4_0 = Z40_3 * Z2_0 * (Z2_1 * T2) + Z2_3 + Z2_4$

$T2 = (Y4_0 + T2) * Z2_1$

$Y4_4 = Z40_3 * ((Y4_0 * Z2_1) + Z2_2) * E0_5$

$Z2_1 = ((T2 * Y4_4 * X0_5) + Z2_1) * (Z2_2 + X1_6)$

$Z2_3 = ((X1_1 * Z2_1) + Z2_3) * (Z2_2 + X1_6)$

$Y4_5 = Z40_3 * (Z2_3 + Z2_4) * E0_5$

$Z2_4 = ((X1_1 * Z2_1 * Z2_3) + X0_5 + Z2_4) * (Z2_2 + X1_6)$

$Y4_2 = Z40_3 * Z2_4 * E0_4$

$T3 = Y4_2$

$Y4_6 = Z40_3 * (E0_5 + Y4_6) * Z2_2$

$$Z2_2 = ((Y4_6 * Y4_3 * X0_4) + Z2_2) * ^{(X05 + X1_6)}$$

Супервизоры

Авария = X0_1 * X0_5

Алгоритмы

Нач: Мост_вперед

1:

Запуск_таймера(TY4_3_X0_4, 100)

Кон: Мост_вперед

Нач: Мост_назад

1:

Запуск_таймера(TY4_4_X0_5, 100)

Кон: Мост_назад

Нач: Въезд_автомобиля

1:

Запуск_таймера(TY5_1_X0_1, 1000)

Запуск_таймера(TY5_1_X0_2, 2000)

Кон: Въезд_автомобиля

Нач: Выезд_автомобиля

1:

Запуск_таймера(TY5_0_X0_3, 100)

Кон: Выезд_автомобиля

Запуск_алгоритмов

Мост_вперед = _Y4_3

Мост_назад = _Y4_4

Въезд_автомобиля = _Y5_1

Выезд_автомобиля = _Y5_0

Таймеры

TY4_3_X0_4(100, X0_4)

TY4_4_X0_5(100, X0_5)

TY5_1_X0_1(10000, X0_1)

TY5_1_X0_2(20000, X0_2)

TY5_0_X0_3(100, X0_3)

T1_Z40_1(100, T1)

T2_Y4_0(30000, T2)

T3_Y4_2(30000, T3)

Соединители

ДЛЯ XX0

5X0_5

КОМ Мост в исходном

ДЛЯ XY4

0Y4_0
КОМ Включение насоса воды
1Y4_1
КОМ Включение насоса моющих средств
2Y4_2
КОМ Включение насоса обмыва снизу
3Y4_3
КОМ Мост вперед
4Y4_4
Мост назад
5Y4_5
КОМ Включение щеток
6Y4_6
КОМ Включение горячего воздуха
ДЛЯ XY5
0 Y5_0
КОМ Включение светофора на выезде
1 Y5_1
КОМ Включение светофора на въезде

5.3.3. Преобразование алгоритмов управления в SFC-программы

Язык последовательных функциональных схем SFC позволяет представить алгоритмы управления в наглядной графической форме и исполняемом виде.

Принцип работы программы *GeraklTo SFC Converter* заключается в преобразовании алгоритма, записанного в текстовом виде на языке Геракл, в файл .isaxml (на базе шаблона Prog1.isaxml), который среда ISaGRAF отображает в графической форме (рис. 5.29).

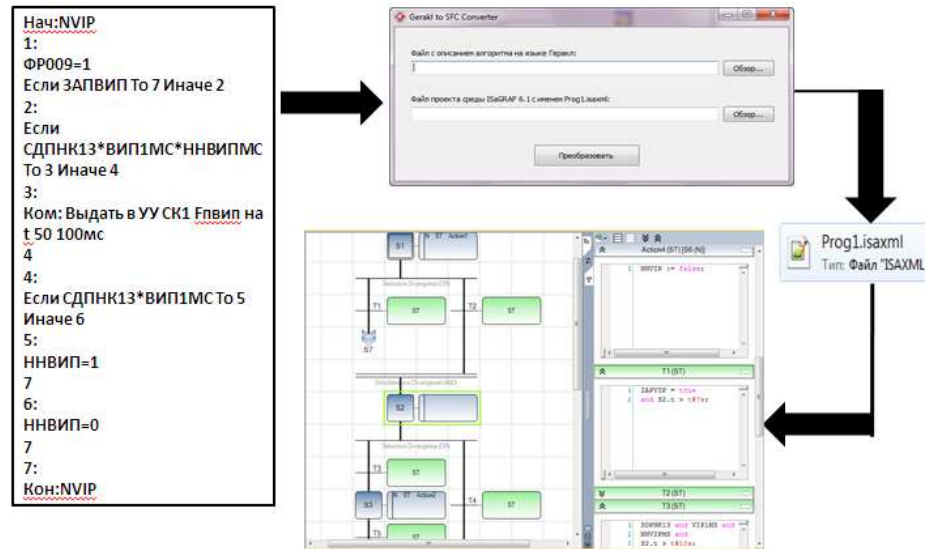


Рис. 5.29. Принцип работы программы *GeraklTo SFC Converter*

Шаблонный файл Prog1.isaxml хранится в папке проекта среды ISaGRAF и содержит в себе:

- координаты связей между шагами и переходами, OR и AND-дивергенций и конвергенций, а также координаты прыжков (длинных переходов);
- конструкции описания шагов;
- описания действий;
- описания переходов;
- словарь переменных проекта;
- некоторые другие элементы файла.

Алгоритм на языке Геракл записывается разработчиком или создается программой *Алгоритмы_Visio_Геракл* на этапе *Ввод данных модели* приложения ГИПЕРСИСТЕМА и хранится как обычный текстовый файл *Алгоритмы.txt* в папке **Предметные конструкции модели**.

Окно программы Geraklto SFC Converter (рис. 5.30) содержит два текстовых поля, две текстовых надписи и три кнопки.

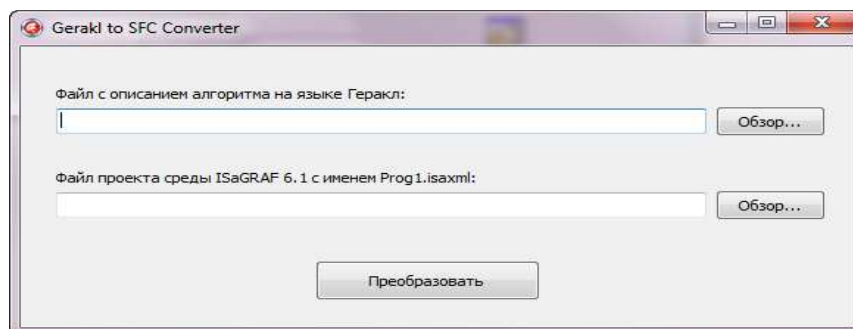


Рис. 5.30. Окно программы Gerakl to SFC Converter

В первом текстовом поле указывается путь к файлу с описанием алгоритма на языке Геракл. Этот файл выбирается в диалоговом окне открытия файла, которое вызывается при нажатии на первую кнопку «Обзор...». Аналогичным образом второе текстовое поле хранит путь к результирующему файлу SFC программы проекта среды ISaGRAF.

Кнопка «Преобразовать» запускает алгоритм преобразования блок-схемы алгоритма на языке Геракл в графическую блок-схему алгоритма на языке SFC в формате проекта среды ISaGRAF.

Открываем полученную программу в среде ISaGRAF (рис. 5.31).

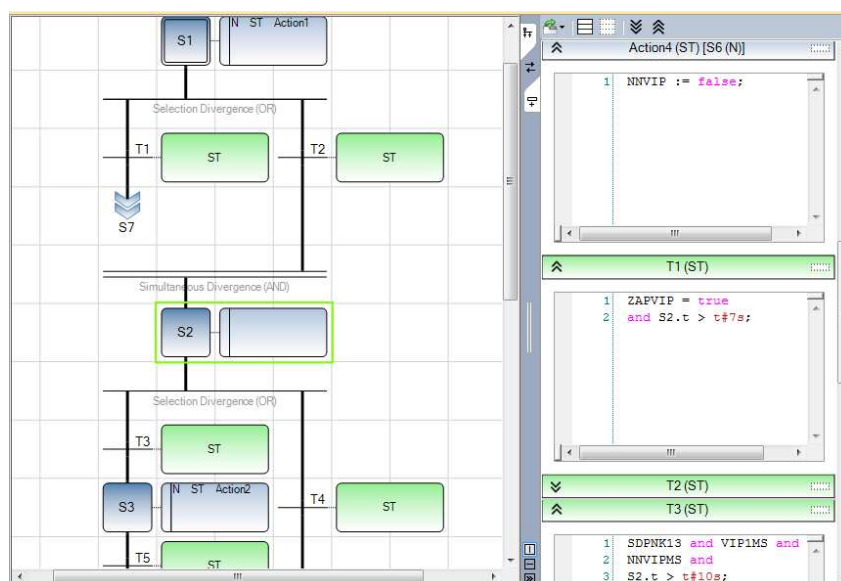


Рис. 5.31. Программная SFC, полученная из алгоритма на языке Геракл

Можно также открыть словарь программы (рис. 5.32), щелкнув дважды по элементу «Локальные переменные» в раскрывающемся списке программы.

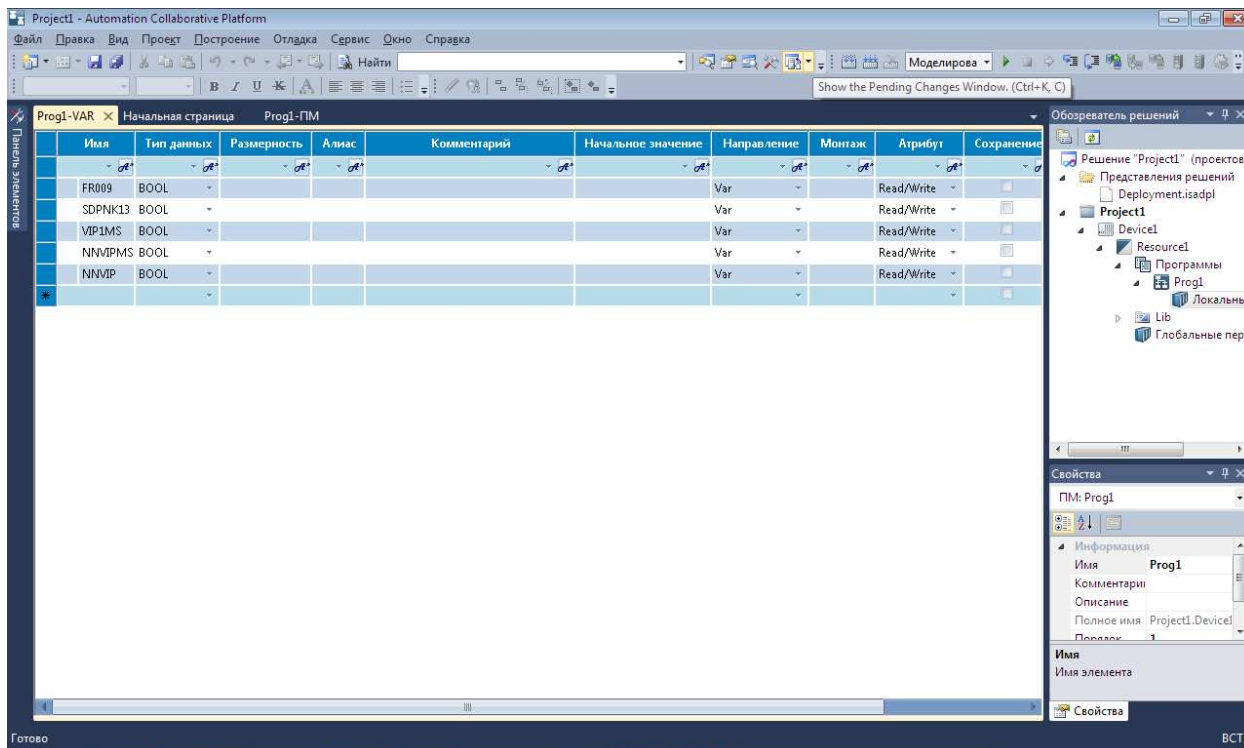


Рис. 5.31. Словарь программы на языке SFC

6. Разработка человеко-машинного интерфейса

6.1. Визуализация процессов управления

Визуализация техпроцесса — способ отображения информации о состоянии технологического оборудования и параметрах технологического процесса на мониторе компьютера или операторской панели в системе автоматического управления в промышленности, предусматривающий также графические способы управления техпроцессом. Система визуализации должна учитывать требования, предъявляемые к человеко-машинному интерфейсу.

Визуализация техпроцесса реализуется в ряде экранов или окон, которые могут представлять собой иерархическую систему. В основе системы отображения лежит мнемосхема техпроцесса, статическое изображение в визуальном простом и интуитивно понятной форме показывающей элементы оборудования, возможные, обрабатываемые материалы и продукцию, и их взаимодействие, порядок обработки. Статическая мнемосхема оживляется -анимируется, отображая реальное состояние оборудования.

6.2. Создание объектов HMI в плагине ISaVIEW

BISaGRAF 6 имеет плагин ISaVIEW, который обеспечивает пользователя простыми, но в то же время мощными интегрированными средствами человеко-машинного интерфейса (HMI). Страницы ISaVIEW встраиваются в структуру проекта автоматизации. Работа пользователя поддерживается с помощью настраиваемых шаблонов и готовых к применению наборов объектов. Вид анимации может легко графически и программно модифицироваться. Доступны средства проектирования и on-line режимы, причем это не требует перекомпиляции проекта ISaGRAF.

ISaVIEW позволяет пользователю быстро создавать объекты с определенным видом эффектов анимации, такими, как действие, изменение цвета, перемещение, вращение, изменение размера, текст, видимость. В качестве графических объектов в ISaVIEW могут быть использованы такие примитивы, как дуга, стрелка, эллипс, прямоугольник, растровый рисунок, кнопка, слайдер и др.

6.2.1. Графические объекты ISaVIEW

На рис. 6.1 – 6.2 показаны панель элементов ISaVIEW, окно определения их свойств, а также словарь переменных, через которые можно управлять этими свойствами из программ ISaGRAF.

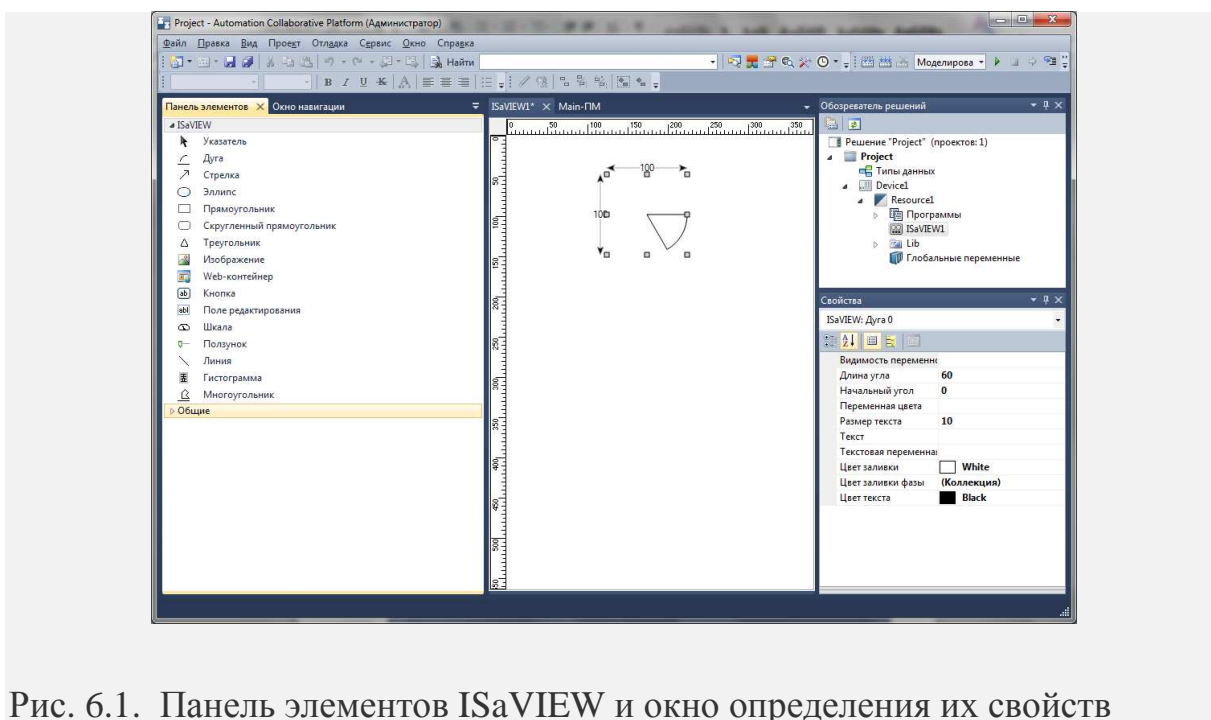


Рис. 6.1. Панель элементов ISaVIEW и окно определения их свойств

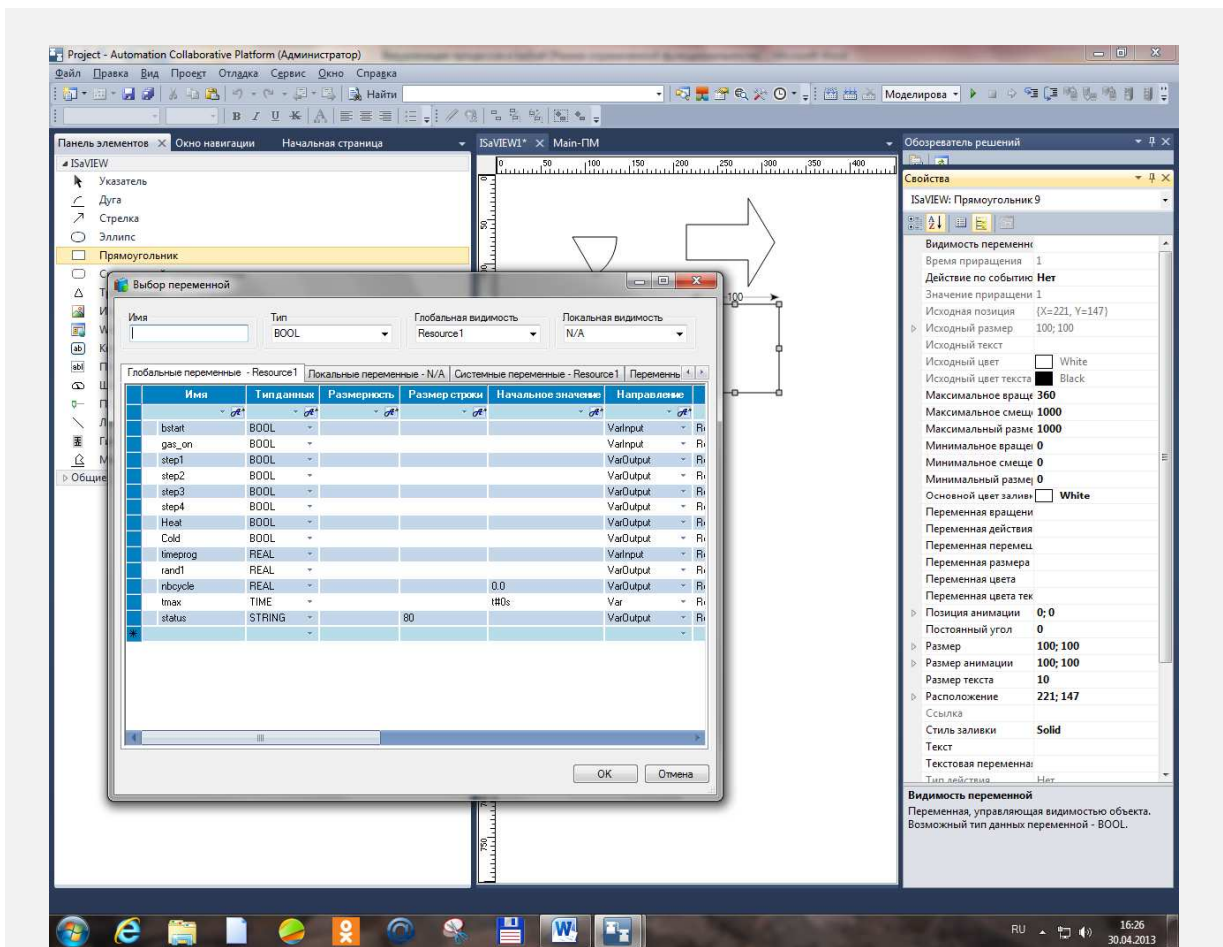


Рис. 6.2 Расширенное окно определения свойств и словарь переменных

6.2.3. Управление свойствами графических объектов

6.2.3.1. Изменение цвета

Поместим на экран (*Screen*) элемент “эллипс” (*Ellipse*) (рис. 6.3.)

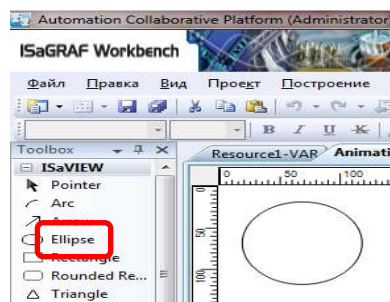


Рис. 6.3 Элемент *Ellipse*

Затем перейдем на вкладку расширенных свойств (*Extended properties*) объекта. Здесь нас будут интересовать два свойства: **Color Variable** – переменная, которая определяет состояние цвета в режиме анимации, а также **Fill Color Phase** – список цветов, применяемых к объекту в некотором состоянии в режиме анимации(рис. 6.4.).

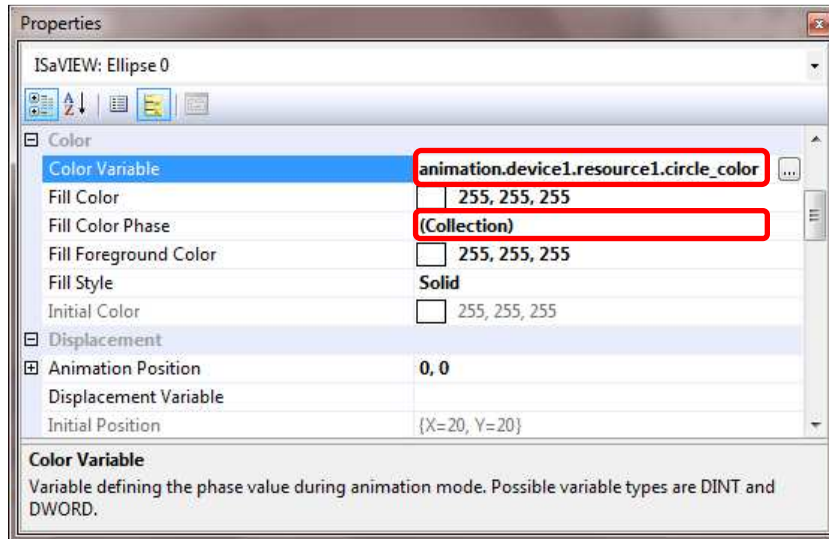


Рис. 6.4. Свойства для определения цвета

Зададим переменную в свойстве **Color Variable**, а также состояние объекта(рис. 6.5).

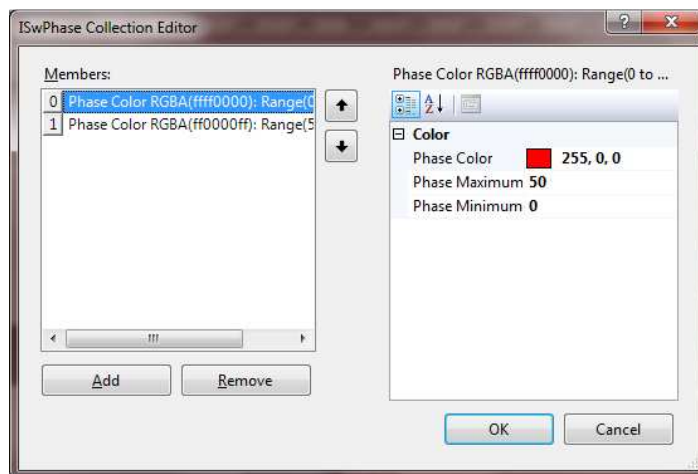


Рис. 6.5. Свойства для определения состояния объекта

Параметры **Phase Minimum** и **Phase Maximum** определяют интервал для переменной, указанной в свойстве объекта **Color Variable**. Т.е. если

переменная **Color Variable** принимает значение в интервале от 0 до 50, то объект будет покрашен в красный цвет.

Теперь достаточно каким либо образом (в диалоге или программно) изменять значение переменной, указанной в свойстве **Color Variable**.

Рассмотрим программное управление цветом эллипса. Добавим две кнопки на экран и зададим для них действие по нажатию(рис. 6.6).

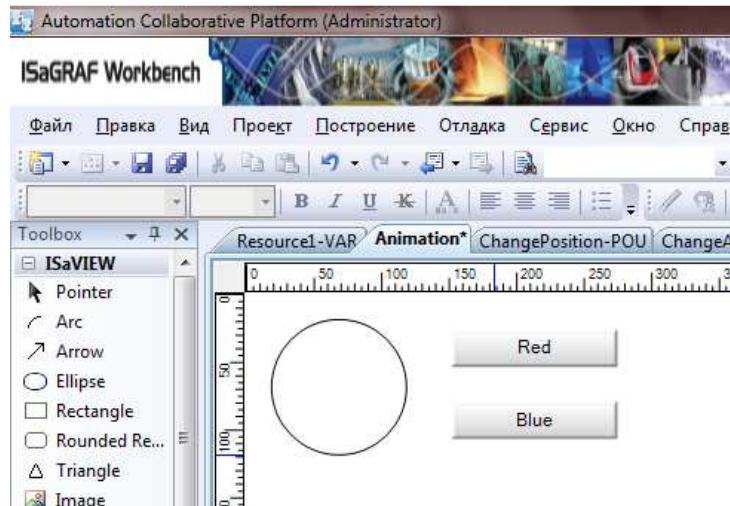


Рис. 6.6. – Кнопки управления цветом

Перейдем в расширенные свойства(рис. 6.7).

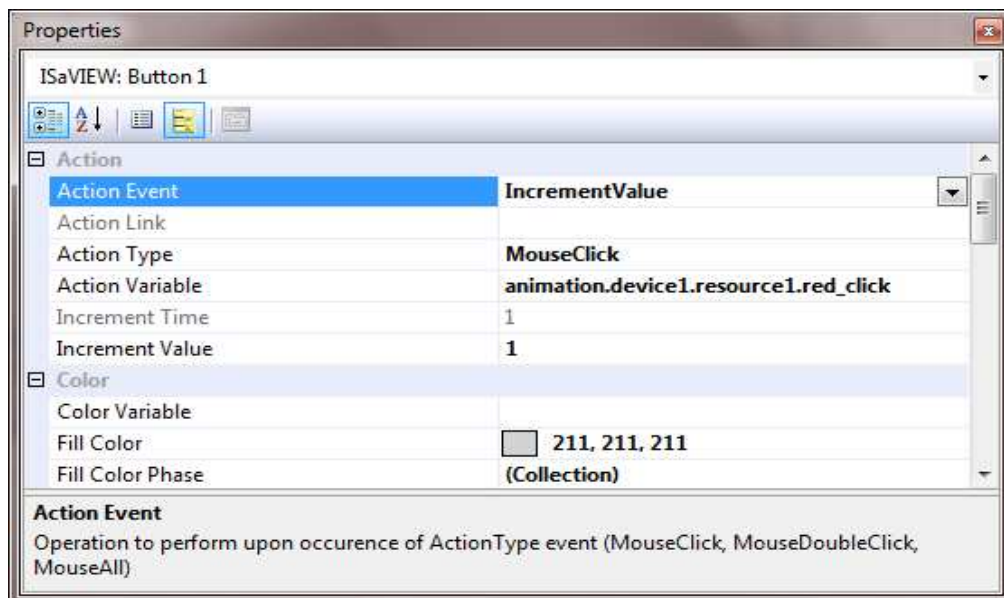


Рис. 6.7. – Расширенные свойства кнопки управления цветом

Action Type – определяет тип события, по которому будет выполняться действие.

Action Event – определяет тип действия, выполняемого при возникновении указанного события.

Action Variable – определяет переменную, над которой будет производиться заданное действие.

Increment Value – определяет значение инкремента.

Таким образом, мы добавили две кнопки, по клику мыши на которые будет увеличиваться значение переменных `red_click` и `blue_click` на единицу.

Осталось связать изменение значение переменной `circle_color` с событием клика мыши на кнопку. Для этого используем программу на языке ST(рис. 6.8).

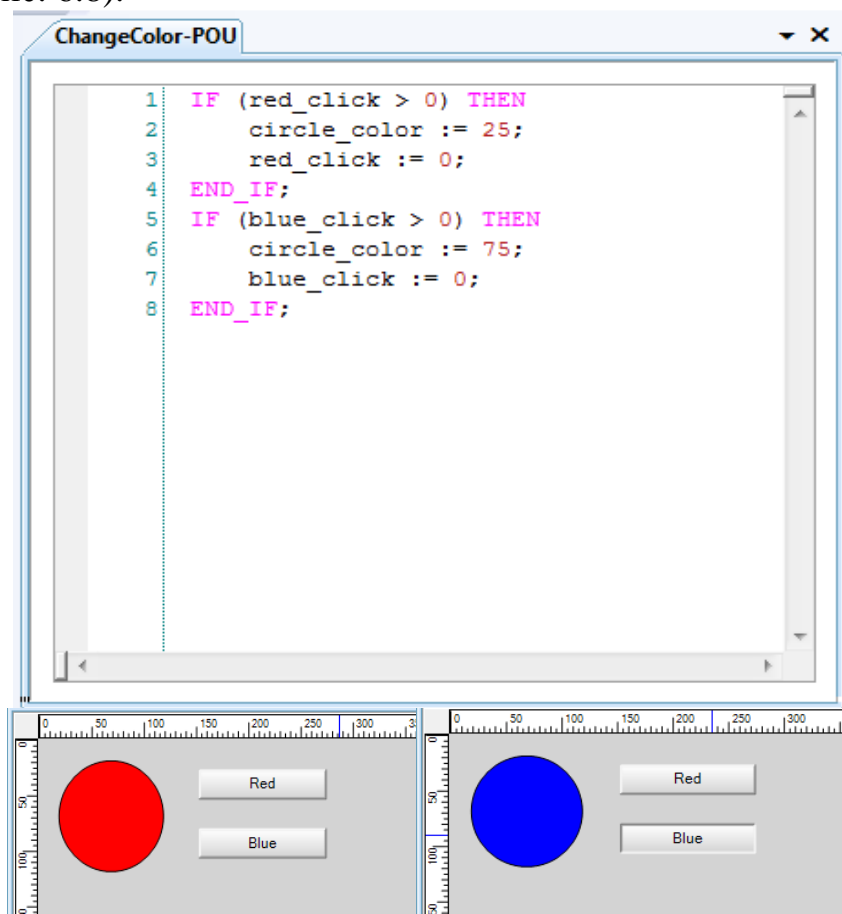


Рис. 6.8. Программное изменение переменной `circle_color`

6.3.3.2. Изменение размера объекта

Добавим на экран большой серый прямоугольник, внутри него маленький зеленый прямоугольник и красную линию.

Изменим размер зеленого прямоугольника, для этого перейдем на вкладку расширенных свойств(рис. 6.9).

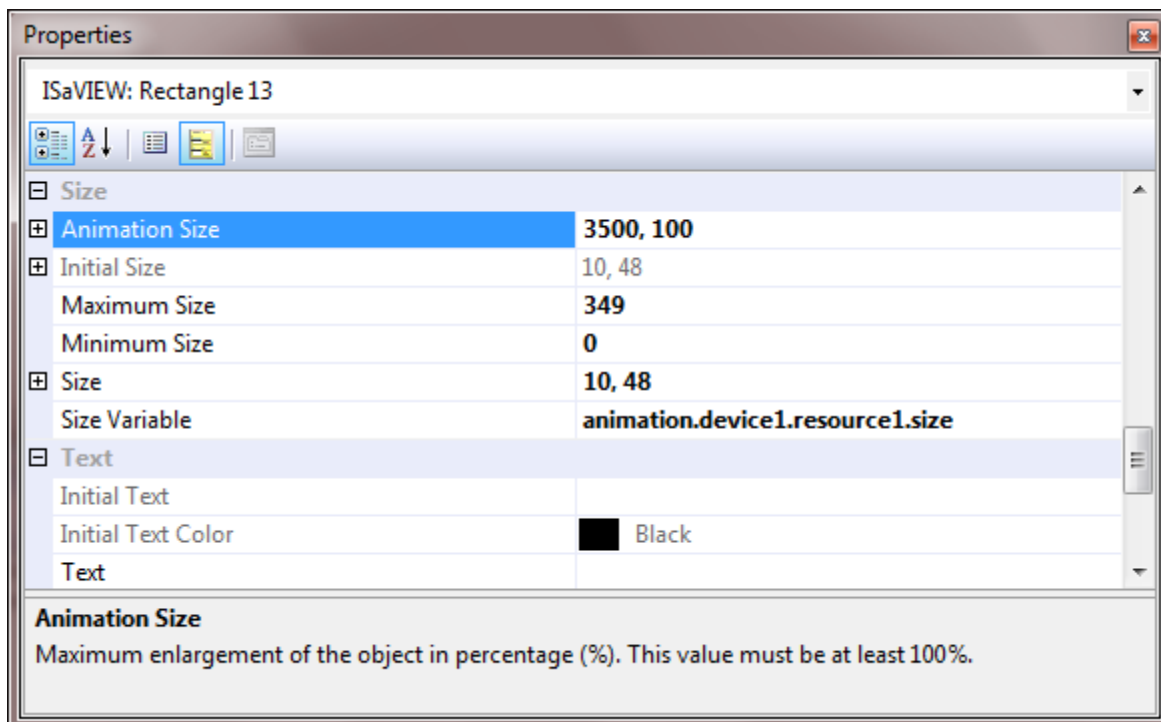


Рис. 6.9. Расширенные свойства для изменения размера

AnimationSize – определяет максимальное увеличение размера в процентах (100 % - без увеличения). Таким образом, мы указали, что прямоугольник может быть увеличен в 35 раз по горизонтали.

SizeVariable – указывает переменную, определяющую размер объекта.

MinimumSize и **MaximumSize**- задают интервал для переменной, указанной в свойстве **SizeVariable**. Таким образом, увеличению объекта в 35 раз соответствует значение переменной size равное 349, т.е. единичное приращение переменной увеличивает объект на 10 %.

Напишем программу на языке ST, которая изменяет значение переменной size и соответственно размер объекта(рис. 6.10).

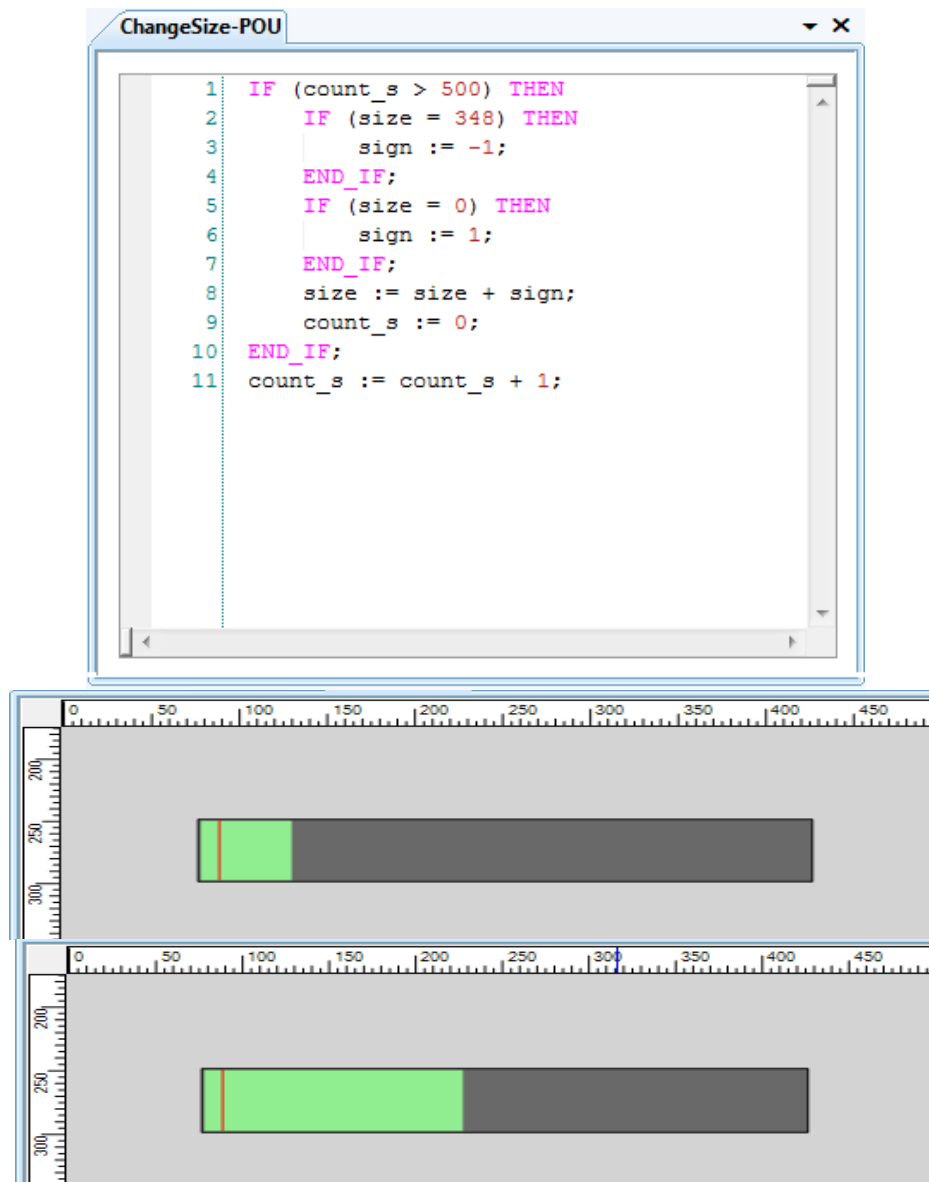


Рис. 6.10. Программное изменение переменной *size*

6.3.3.4. Вращение объектов

Для анимации поворота объекта необходимо указать следующие свойства:

CenterofRotation – задает координаты точки, относительно которой будет выполняться поворот (координаты задаются относительно верхнего левого угла границы объекта).

RotationVariable – задает переменную, которая определяет угол поворота объекта.

Далее необходимо написать программу, изменяющую угол поворота объекта (рис. 6.11).

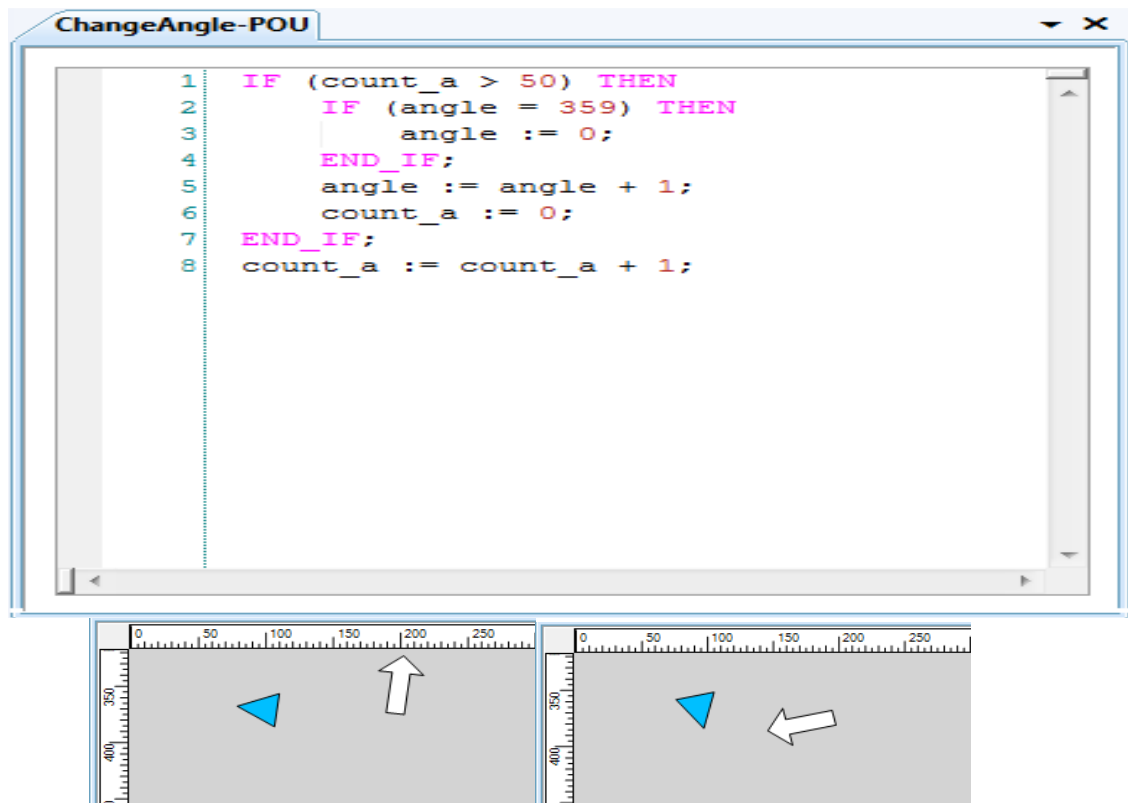


Рис. 6.11. Изменение угла поворота объекта

7. Комплексное визуальное моделирование технологической машины

Комплексное визуальное моделирование включает в себя этапы разработки человеко-машинного интерфейса (НМИ), создания визуальной программы системы логического управления (СЛУ) и выполнения процесса моделирования СЛУ в комплексе с НМИ.

7.1. НМИ технологической машины

В качестве примера рассмотрим демонстрационный пример *DemoUsinage*, построенный вышеописанными средствами создания объектов НМИ в плагине ISaVIEW (рис. 7.1).

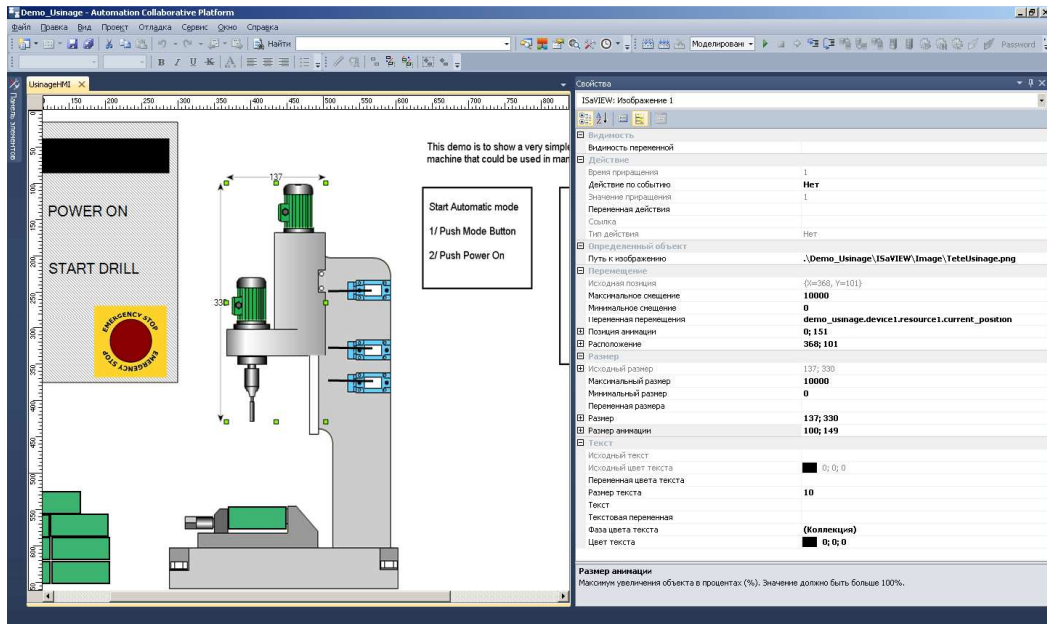


Рис. 7.1. Человечно-машинный интерфейс ТМ

7.2. Программирование СЛУТМ

Программирование СЛУ выполнено на языке SFC (главная программа *Usinage* и дочерняя программа *UsinageMove*) и ST (программа *Sensors*) (рис. 7.2).

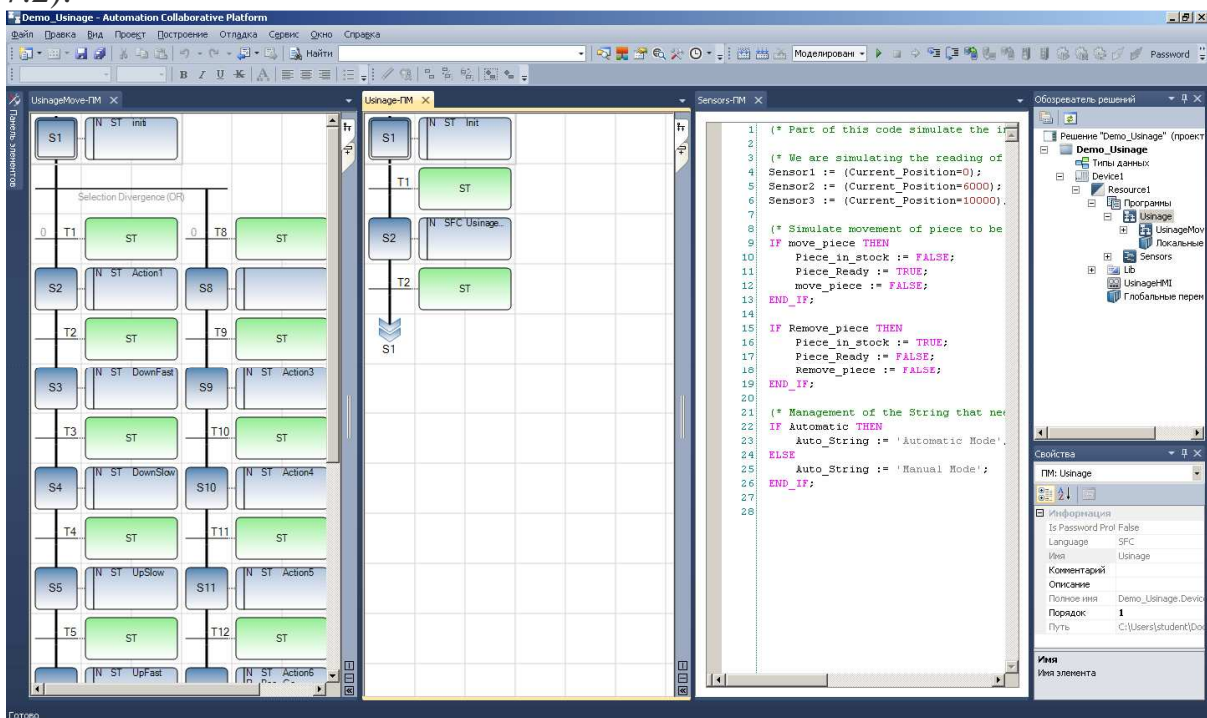


Рис. 7.2. Программирование СЛУ выполнено на языке SFC и ST

Рассмотрим основные части этих программ.

Переключение сенсоров осуществляется с помощью операторов ST.

```
Sensor1 := (Current_Position=0);  
Sensor2 := (Current_Position=6000);  
Sensor3 := (Current_Position=10000);  
Sensor1 := (Current_Position=0);  
Sensor2 := (Current_Position=6000);  
Sensor3 := (Current_Position=10000);  
IF Automatic THEN  
  Auto_String := 'Automatic Mode';  
ELSE  
  Auto_String := 'Manual Mode';  
END_IF;
```

Перемещение деталей для сверления осуществляется с помощью переменной Piece_Ready, при установке в true которой деталь перемещается в зажим.

```
IF move_piece THEN  
  Piece_in_stock := FALSE;  
  Piece_Ready := TRUE;  
  move_piece := FALSE;  
END_IF;
```

Перемещение рабочего органа ТМ осуществляется с помощью изменения значения переменной Current_Position.

```
Current_Position := Current_Position + 1;  
Current_Position := Current_Position - 1;
```

Изменение скорости перемещения головки осуществляется с помощью методов DownFast, UpFast и DownSlow, UpSlow.

```
Current_Position := Current_Position + 3;  
Current_Position := Current_Position + 1;
```

Остановка ТМ осуществляется сбросом переменной Current_Position и установкой переменной Start в false.

7.3. Структурный анализ СЛУ ТМ

Для выполнения структурного анализа СЛУ создаем браузер перекрестных ссылки применяем его для переменной Current_Position (рис. 7.3). У этой переменной 34 ссылки (одна из них отображена в программах синим цветом).

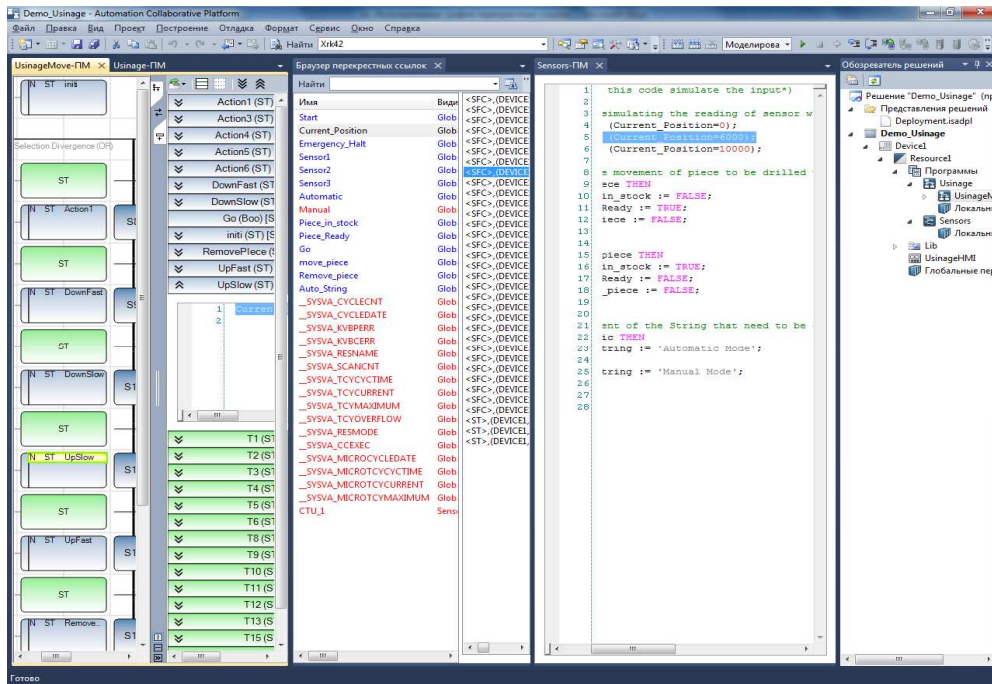


Рис. 7.3. Браузер перекрестных ссылок

Создаем дерево зависимостей для переменной *Current_Position* (рис. 7.4).

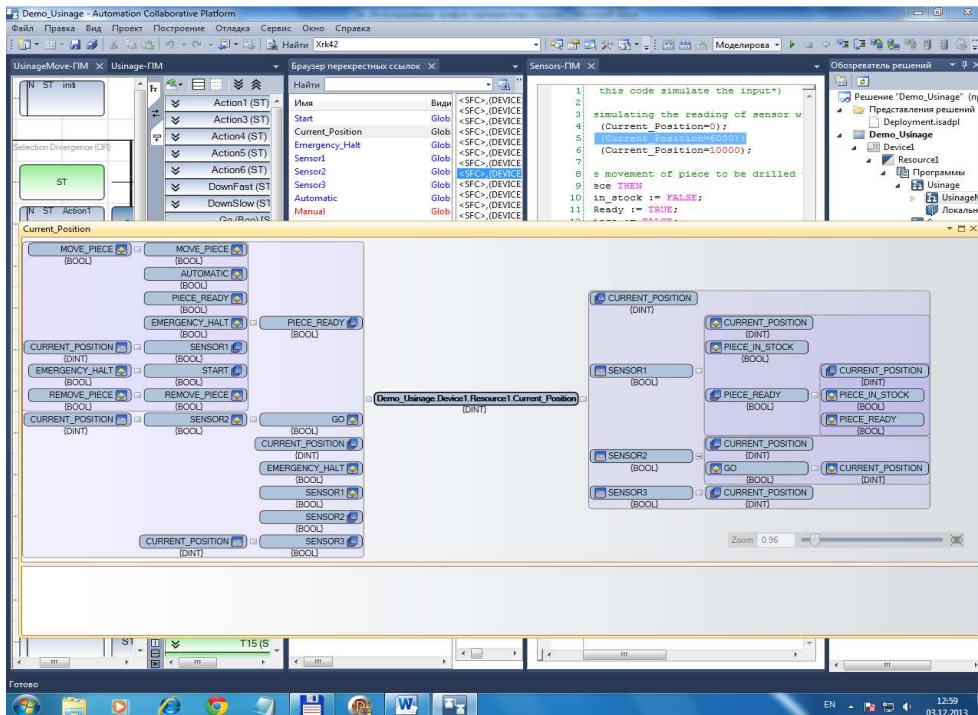


Рис. 7.4. Дерево зависимостей для переменной *Current_Position*

Выполним поиск переменной *Piece_Ready* по дереву зависимостей. Эта переменная имеет пиктограмму **контейнер** и влияет на 3 другие переменные (рис. 7.5).

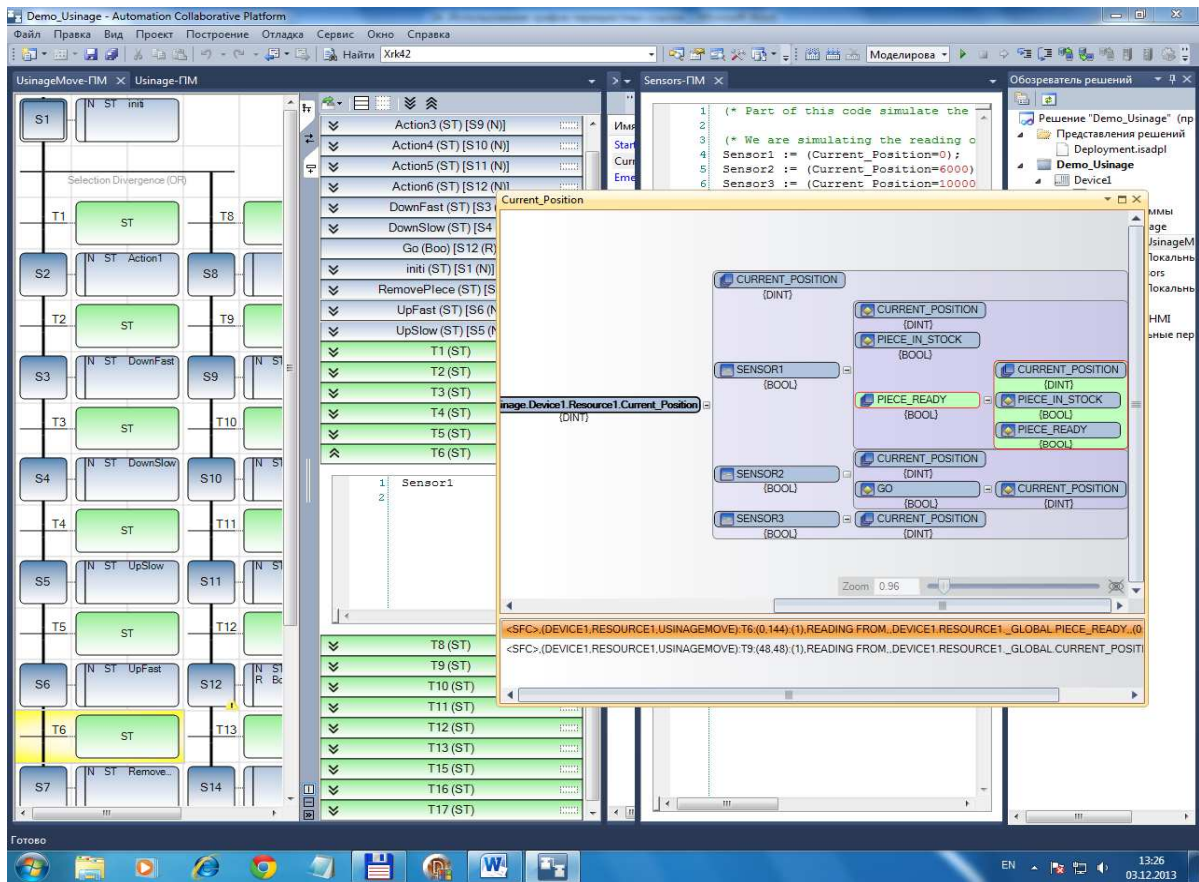


Рис. 7.5. Поддерево зависимостей для переменной *Piece_Ready*

7.4. Визуальное моделированиеТМ в ручном режиме

Подаем питание на СЛУ кнопкой POWERON на панели НМІ, в результате чего программа SFCпереходит на шаг S8 (рис. 7.6).

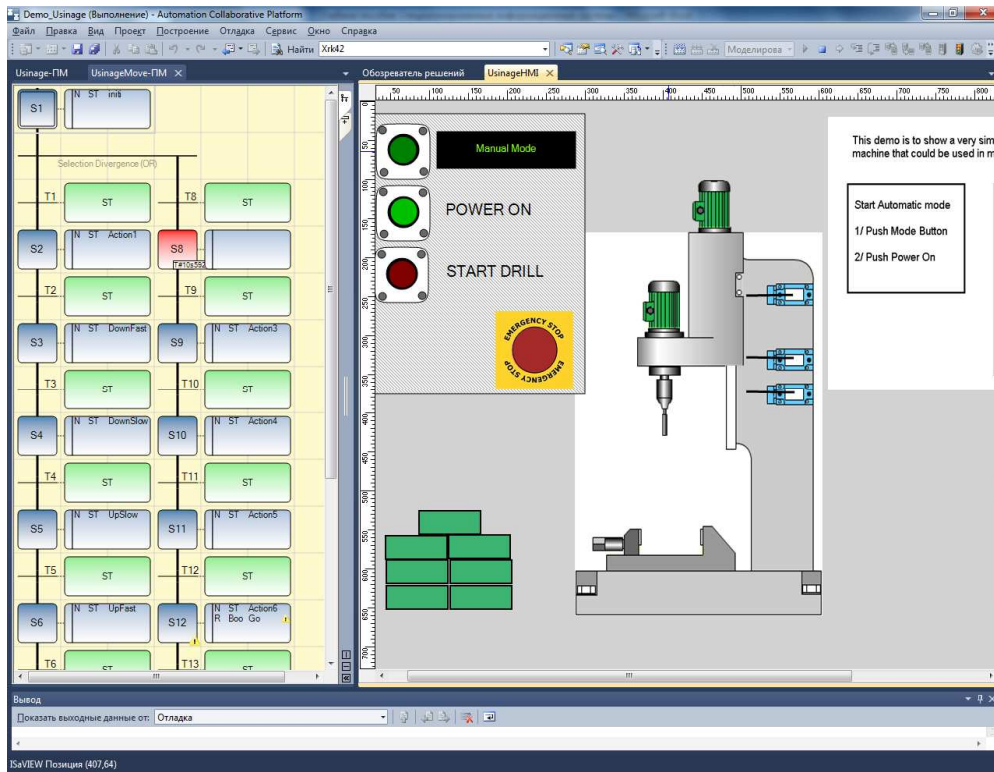


Рис. 7.6. Переход программы SFC на шаг S8

Двойным щелчком мыши переносим деталь в позицию обработки (рис. 7.7).

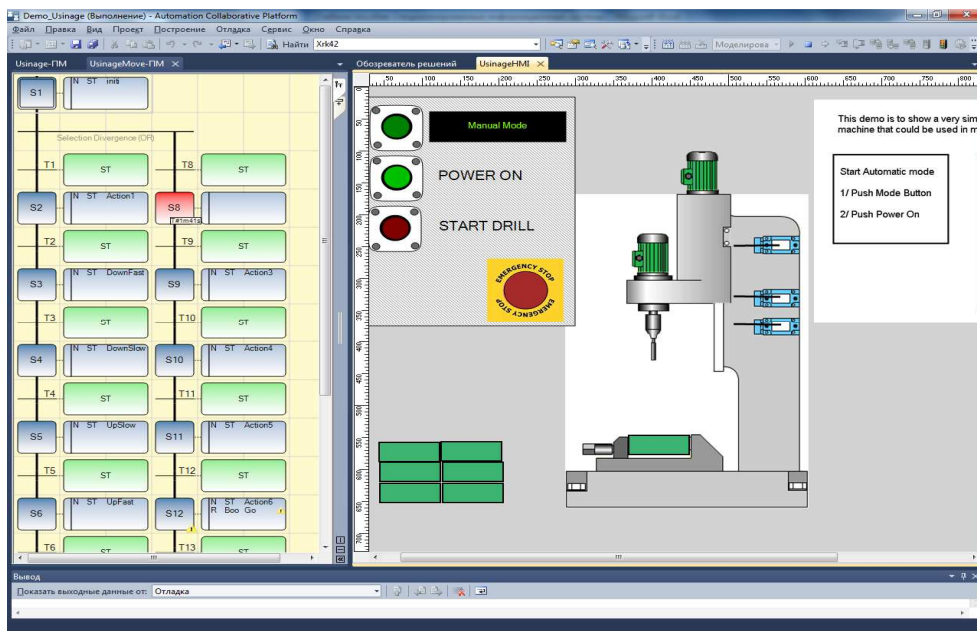


Рис. 7.7. Перенос детали в позицию обработки

Включаем процесс сверления кнопкой STARTDRILL, в результате чего программа SFC переходит на шаг S9, S10 ..., а на панели HMI отображается перемещение сверлильной головки.

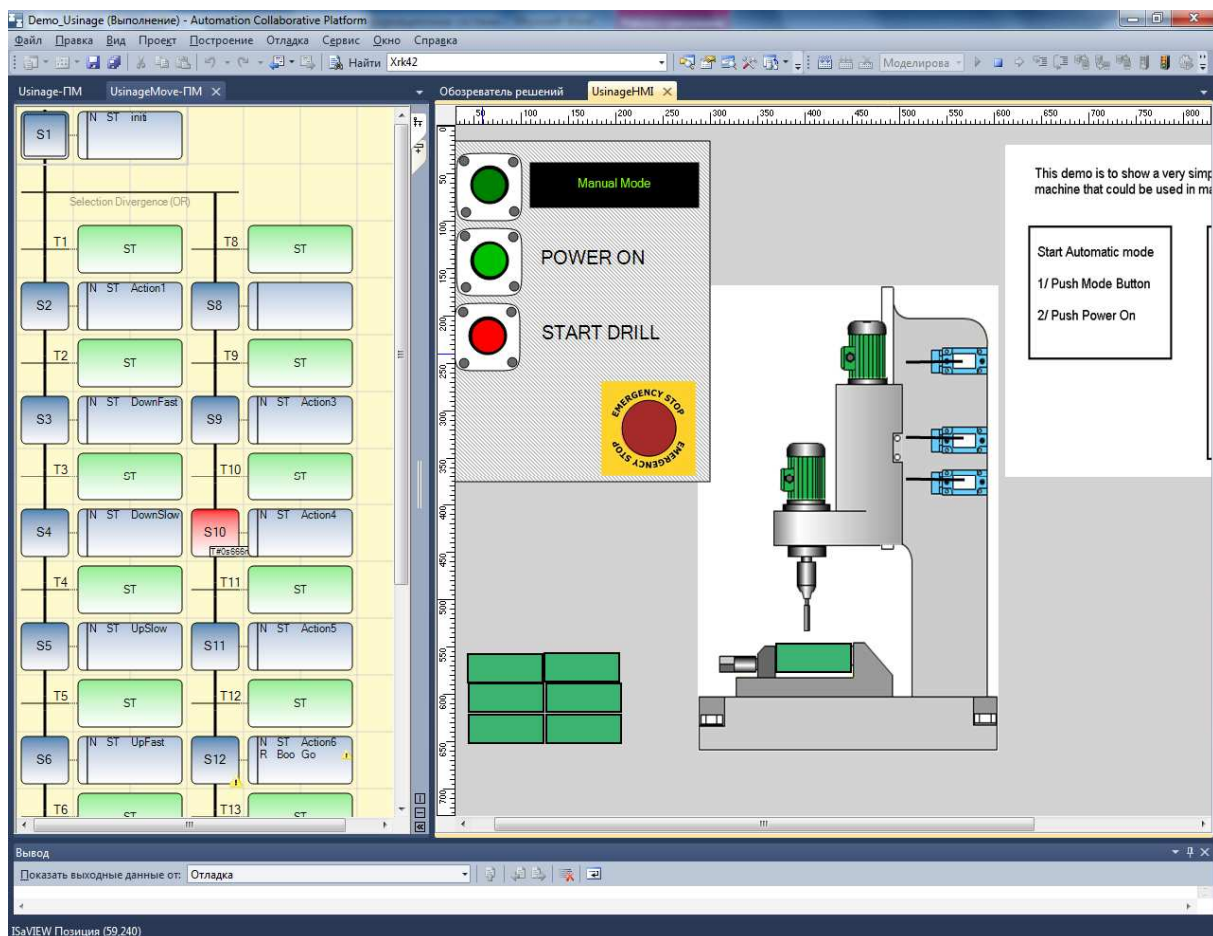


Рис. 7.8. Переход на шаг S10 и перемещение сверлильной головки

Литература

1. Акунович, С. И. Дискретные системы логического управления технологических машин / С. И. Акунович, А. А. Гончаров, Ю. Н. Петренко. – Минск.: Юнипак, 2006. – 334 с.
2. Акунович С.И., Брусенцова Т. П., Гончаров А. А. Реализация метода композиционного синтеза моделей логического управления в приложении «ГИПЕРСИСТЕМА» // Труды БГТУ. – 2013. - №6: Физ.-мат. науки и информатика. – С. 80–82.
3. Акунович, С. И. Об использовании визуального программирования при изучении алгоритмов управления дискретных устройств / С. И. Акунович // Международная научно-практическая конференция «Информатизация образовательных процессов: автоматизация управления, технологии, дистанционное обучение», Минск, 19 – 20 апреля 2001. / МГВРК. – Минск, 2001. – С. 152–162.
4. Акунович С.И. Визуальное моделирование систем логического управления технологическим оборудованием и процессами // Труды БГТУ. – 2011. - №6: Физ.-мат. науки и информатика. – С. 112–116.
5. ISaGRAF 5.0. Среда разработки проекта. Руководство пользователя. Октябрь 2006 г.
6. Осипов, Д. Delphi XE2. / Д. Осипов. – СПб.: БХВ–Петербург, 2012 – 892 с.
7. Зюбин, В. Е. Язык «рефлекс» – диалект си для программируемых логических контроллеров // Шестая международная научно-практическая конференция «Средства и системы автоматизации» CSAF'06 // Томск, 1-3 ноября 2005.
8. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд. / Г. Буч и др. – М.: ООО «И.Д. Вильямс», 2008. – 718 с.

Оглавление

Введение.....	3
1. Инструментальная среда разработки встраиваемых приложений ISaGRAF	4
1.1. Назначение и принципы построения ISaGRAF.....	4
1.2. Инженерно-ориентированные языки программирования	7
1.3. Создание проектов в приложении ISaGRAF	10
1.3.1. Запуск приложения	10
1.3.2. Задание имен устройству и ресурсу.....	12
1.3.3. Определение свойств для устройств и ресурсов.	12
1.3.4. Добавление программы.....	14
1.3.5. Задание свойств программы.	15
1.3.6. Добавка содержимого программы.	16
1.3.7. Сборка решения	19
1.3.8. Запуск процесса отладки.....	21
1.3.9. Остановка процесса отладки	22
2. Разработка проектов СЛЮ на языке LD	22
2.1. Описание языка LD.....	22
2.2. Разработка программ на языке LD	28
2.2.1. Исходные данные.....	28
2.2.2. Структура проекта	29
2.2.3. Разработка программ.....	29
2.2.4. Монтирование переменных	31
2.2.5. Отладка программ.....	33
1.3. Автоматизация программирования логических формул на языке LD 35	
2.3.1. Верификация и валидация логических формул.....	35
2.3.2. Преобразование логических формул в LD-диаграммы	41
2. Разработка проектов СЛЮ на языке FBD	44
3.1. Описание языка FBD	46
3.1.1. Главные элементы FBD.....	46

3.2. Разработка программ на языке FBD.....	51
2.3. Автоматизация программирования логических формул на языке FBD	57
3.3.1. Преобразование логических формул из операторной формы в функциональную.....	57
3.3.2. Преобразование логических формул в FBD-программы.....	61
4. Разработка проектов СЛЮ на языке ST.....	61
4.1. Описание языка ST.....	61
4.2. Разработка программ на языке ST.....	66
1.2.1. Словарь проекта	67
1.2.2. Разработка программ	68
1.2.3. Монтирование переменных.....	69
4.3. Отладка программ ST	70
5. Разработка проектов СЛЮ на языке SFC	71
5.1. Описание языка SFC	71
5.2. Действия внутри шагов	76
5.2.1. Булевские действия.....	76
5.2.2. Импульсные действия	77
5.2.3. Не сохраняемые действия	78
5.2.4. Действия SFC	79
5.3. Программирование переходов.....	80
5.4. Разработка программ на языке SFC	81
5.3. Автоматизация программирования алгоритмов управления на языке SFC.....	88
5.3.1. Формальное описание СЛЮ	88
5.3.2. Краткое описание языка Геракл.....	89
5.3.3. Преобразование алгоритмов управления в SFC-программы	96
6. Разработка человеко-машинного интерфейса.....	99
6.1. Визуализация процессов управления.....	99
6.2. Создание объектов НМІ в плагине ISaVIEW.....	100
6.2.1. Графические объекты ISaVIEW	100

6.2.3. Управление свойствами графических объектов.....	101
7. Комплексное визуальное моделирование технологической машины....	107
7.1. HMI технологической машины	107
7.2. Программирование СЛУ ТМ	108
7.3. Структурный анализ СЛУ ТМ	109
7.4. Визуальное моделирование ТМ в ручном режиме.....	111

СПЕЦИАЛИЗИРОВАННЫЕ ИНФОРМАЦИОННЫЕ СИСТЕМЫ

Составитель: **Акунович** Станислав Иванович

Редактор *ЧЧЧЧЧЧ*

Компьютерная верстка *ЧЧЧЧЧЧ*

Учреждение образования

«Белорусский государственный технологический университет».

220006. Минск, Свердлова, 13а.

ЛИ № 02330,0549423 от 08.04.2009.