Федеральное государственное бюджетное учреждение науки
ИНСТИТУТ ПРОБЛЕМ УПРАВЛЕНИЯ
им. В.А. ТРАПЕЗНИКОВА
Российской академии наук

А.А. Рощин

Расчет Динамических Систем (РДС)

Руководство для программистов

Москва 2016

УДК 658.512.011.56:004.42

Редактор: заведующий лабораторией Института проблем управления им. Трапезникова РАН, профессор М.Х. Дорри.

Работа выполнена в рамках проектов, выполняемых Учреждением Российской академии наук Институтом проблем управления им. В.А. Трапезникова РАН:

- проект 3121: "Разработка теории и алгоритмов синтеза управления объектами морской техники и методологии создания исследовательских стендов", (Перечень комплексных проектов, разрабатываемых в 2010-2012 гг.)
- проект 400-09: "Методы анализа и синтеза информационно-управляющих систем с интеллектуальным интерфейсом" (Программа фундаментальных исследований № 15, 2009-2011 гг.).

Рассматривается общая структура инструментального комплекса РДС (Расчет Динамических систем) Описаны принципы и внутренние механизмы построения модулей. Многие параграфы дополнительно снабжены действующими программами, иллюстрирующими работоспособность излагаемых принципов построения.

Руководство ориентировано на программистов, знакомых с написанием программ на языках высокого уровня (в частности, на языках C и C++) и имеющих опыт работы с инструментальным комплексом РДС.

Работа выходит за рамки простого описания – она дает некоторую методологию разработки крупных программных продуктов, пример преодоления встречающихся трудностей и методов их устранения.

Целью издания руководства является желание сделать разработанный программный продукт абсолютно открытым и способствовать его дальнейшему развитию силами желающих воспользоваться инструментальным комплексом РДС в своей работе.

Оглавление

Предисловие редактора

Актуальность создания специализированной инструментальной среды для представления состояния и поведения технических объектов обусловлена необходимостью сокращения сроков и стоимости создания объектов, повышения качества их разработки и безопасности эксплуатации. Одним из наиболее эффективных путей решения этой проблемы является разработка и использование на всех этапах жизненного цикла объектов широкомасштабных компьютерных моделирующих стендов, обеспечивающих возможность наглядного представления поведения и состояния, как образцов в целом, так и их отдельных систем.

Моделирующие стенды, существующие на крупных предприятиях, построены, как правило, по аппаратно-программному принципу и представляют собой сложную систему технических средств. Они соединены в локальную сеть, ориентированы на определенные объекты и, как правило, создаются в уникальном исполнении для определенных объектов техники и трудно поддаются модернизации. Создание таких стендов традиционными способами является не только процессом трудоемким и дорогостоящим, но еще требует привлечения большого количества высококвалифицированных специалистов. Для преодоления этих недостатков целесообразно использовать специализированные инструментальные среды, которые позволяют создавать модели объектов любой сложности, осуществлять их оперативную перестройку в сжатые сроки силами специалистов разработчиков.

В нашей стране очень мало ценятся работы по созданию программных продуктов. Эта область науки, которая во всем мире занимает передовые позиции, в нашей стране сильно отстала. Такие работы трудно поддаются строгому описанию. Хороший программный продукт напоминает невидимое архитектурное сооружение. Оценить свойства программного продукта можно, только используя его, иными словами, войдя в это невидимое сооружение. Обо всех находках и оригинальных решениях очень трудно рассказывать - о них можно судить только после решения конкретных задач и проблем.

Исследования, проведенные в Институте проблем управления им. В.А. Трапезникова РАН в 2000-2010 гг. позволили решить ряд теоретических вопросов, связанных с расчетом динамических систем на вычислительных машинах, обосновать структурно-иерархическое построение инструментальных средств автоматизации для динамического анализа и расчета систем управлении. Были предложены процедуры и алгоритмы исследования устойчивости, моделирования, оптимизации параметров и т.п.

Теоретические работы легли в основу разработки ряда программных комплексов для анализа и синтеза систем управления. [1, 2] Последний из них – РДС (Расчет Динамических Систем) [3, 4, 5], по многим характеристикам превзошел известные отечественные и зарубежные аналоги.

Инструментальный комплекс РДС обладает такими возможностями как:

1. Возможность наглядного отображения процессов с использованием графиков, легкость создания мультипликации пользователем.
2. Использование в структурных схемах как стандартных (библиотечных) блоков, так и блоков с автоматически компилируемыми моделями, написанными на синтаксисе языков высокого уровня.
3. Возможность задания логики работы блоков систем.
4. Наличие видимых и невидимых слоев.
5. Удобные средства взаимодействия подсистем по шинам.
6. Групповое изменение характеристик блоков.
7. Возможность организации многоуровневого взаимодействия блоков между собой и с системой.
8. Возможность соединения нескольких компьютеров в общую сеть.

9. Возможность исследования процессов в разных временных масштабах и осуществления связи с реальным объектом.

Инструментальный комплекс РДС уже прошел апробацию, с его помощью были созданы исследовательские стенды для ряда организаций. Он использовался при разработке стенда для построения систем управления энергетическими установками. На его основе был построен настольный полномасштабный стенд для исследования ряда режимов движения и определения эффективности алгоритмов управления атомными подводными лодками. Комплекс также использовался при разработке информационного обеспечения для повышения эффективности управления системами жизнеобеспечения муниципальных образований. На основе РДС созданы учебные пособия, которые используются в учебном процессе ряда ВУЗов страны.

При создании РДС мне посчастливилось тесно взаимодействовать с А.А. Рощиным, с которым мы обсуждали архитектуру инструментального комплекса, построение подсистем программного обеспечения, свойства необходимые для удобного взаимодействия пользователя с разрабатываемым интерфейсом системы и т.п.

Рощин, на мой взгляд, является выдающимся программистом. Он обладает уникальной способностью не только правильно запрограммировать алгоритм программы, но и предусмотреть многие возможные ошибки, которые могут возникнуть в результате неправильных действий пользователя или в процессе работы системы и вызывающие непредсказуемый останов расчета. Само построение программ он подчинил определенным правилам, способствующим повышению качества их исполнения. Именно он написал все программные модули комплекса РДС, и поэтому только в его силах было описать принципы и внутренние механизмы построения модулей. Этот труд воплотился в представляемом руководстве для программистов, позволяющем развивать и совершенствовать созданный инструментальный комплекс. Однако, работа выходит за рамки простого описания – она дает некоторую методологию разработки крупных программных продуктов, пример преодоления встречающихся трудностей и методов их устранения.

Текст руководства хорошо структурирован, все функции и модули, описанные в параграфах, были А.А. Рощиным дополнительно проверены и откорректированы. Многие параграфы снабжены действующими программами, иллюстрирующими работоспособность излагаемых принципов построения. Тем не менее, как мне кажется, текст руководства написан сухо и трудно читается. Он требует предварительного знакомства и опыта работы с программным комплексом РДС. Не всегда очевидна необходимость и полезность вводимых конструкций и игнорируется объяснение того, как они помогают или упрощают написание программ. Хотелось бы, чтобы прилагаемые к параграфам действующие программы не только подтверждали их работоспособность, но и давали исследователю возможность провести ряд экспериментов (описание которых можно было бы прочитать в комментариях к программе). Такие примеры позволили бы лучше понять особенности и целесообразность вводимых понятий. Это сделало бы изложение более живым и увлекательным. Однако, и без того, достаточно большой объем текста не позволяет осуществить указанные пожелания в предлагаемом руководстве.

Целью издания руководства является желание сделать программный продукт абсолютно открытым и способствовать его дальнейшему развитию силами желающих воспользоваться РДС в своей работе.

Комплекс РДС позволяет:

- создавать универсальные и гибкие исследовательские стенды, легко перестраиваемые под различные типы объектов и различные пульты управления;
- предоставлять исследователям и операторам виртуальную среду разработки, наглядно демонстрирующую последствия воздействия тех или иных возмущений и сбоев в аппаратуре;

- упростить процесс подготовки программных модулей для введения их в комплексный стенд;
- строить модели стендов для обучения операторов с имитацией различных аварийных ситуаций и использовать его в качестве тренажера-прототипа;
- существенно сократить время проектирования автоматизированных систем управления;
- разрабатывать и проверять алгоритмическое и программное обеспечение для современных систем управления объектами, работая с реальными устройствами в режиме полунатурного моделирования;
- применять современные методы теории управления к разработке и совершенствованию систем управления.

Инструментальный программный комплекс РДС несомненно займет достойное место среди других программных продуктов, облегчающих труд проектировщиков и исследователей современных объектов управления.

Введение

Это руководство ориентировано на программистов, знакомых с написанием программ на языках высокого уровня (в частности, на языках C и C++) и имеющих опыт работы с инструментальным комплексом РДС. Предварительное знакомство с комплексом можно получить по опубликованным статьям [3, 4, 5], методическим указаниям [6] и видеоклипам, которые можно найти в сети Интернет [7]. Подразумевается, что читатель умеет работать с РДС как пользователь: создавать схему и добавлять в нее библиотечные блоки, соединять блоки связями, задавать блокам векторные картинки, работать со слоями и т.п. Кроме того, читатель должен иметь представление о работе с функциями Windows API [8, 9], а также с графической библиотекой OpenGL [10, 11] (она используется в единственном примере в §2.10.4). Руководство состоит из четырех глав и нескольких приложений. Изложение построено таким образом, что примеры в тексте часто базируются на примерах, описанных в предыдущих параграфах, поэтому при изучении руководства рекомендуется либо читать его последовательно, либо обращать внимание на встречающиеся в тексте ссылки и продолжать чтение только после изучения материала по ссылке.

В главе 1 рассматривается общая структура РДС и описываются возможности, доступные моделям блоков. Независимо от того, какие действия и реакции читатель собирается закладывать в разрабатываемые модели, ему следует прочесть эту главу для того, чтобы разобраться в режимах РДС, в устройстве блоков и способах передачи данных между ними и т.п. Кроме того, в этой главе кратко описываются некоторые проблемы, которые могут возникнуть при неправильном написании моделей блоков, а также способы их устранения.

Глава 2 целиком посвящена написанию моделей блоков, это самая обширная и сложная глава руководства. В каждом ее параграфе рассматривается то или иное действие, которое может выполнить блок, используемые при этом сервисные функции РДС и конкретные примеры моделей блоков, выполняющих это действие.

В главе 3 рассматриваются способы управления РДС из других приложений Windows при помощи библиотеки RdsCtrl.dll, позволяющие программистам включать РДС в состав сложных программных комплексов. Для понимания этого материала читатель должен иметь представление о написании моделей блоков, поскольку, во многих случаях, именно модели блоков откликаются на команды от приложений и выполняют какие-либо сложные действия.

Глава 4 посвящена созданию модулей автоматической компиляции, позволяющих существенно упростить создание моделей блоков для пользователей, не являющихся программистами. Модули автоматической компиляции более сложны, чем модели блоков, поэтому эта часть руководства рекомендуется для чтения только квалифицированным программистам, умеющим настраивать компиляторы и вызывать их из командной строки. Примеры в этой главе не связаны с примерами в главе 2, но написание модуля автоматической компиляции модели блока требует понимания принципов работы этой модели, поэтому чтение главы 2 перед изучением главы 4 настоятельно рекомендуется. Материал главы 3 в описаниях модулей автоматической компиляции не используется.

Приложения к руководству представляют собой справочник по используемым в РДС описаниям, функциям и константам. В приложении А перечислены все сервисные функции и структуры РДС, обеспечивающие функционирование моделей блоков, а также события, на которые реагируют эти модели. В приложении Б описываются функции и структуры библиотеки RdsCtrl.dll. В приложении В описываются параметры командной строки РДС. Приложения не требуется читать последовательно, каждый их раздел содержит подробную информацию по конкретной функции, структуре или событию, а также ссылки на связанные разделы, если они есть.

Описанные в этом руководстве сервисные функции и структуры входят в состав РДС начиная с версии 1.0.264 – в более ранних версиях часть из них может отсутствовать.

Глава 1. Устройство РДС

В этой главе рассматривается основные принципы работы РДС: модули, из которых состоит приложение, внутреннее устройство блоков схемы, режимы работы, способы взаимодействия блоков с пользователем. Вводятся основные термины, которые будут использованы в следующих главах.

§1.1. Общая структура РДС

В этом параграфе кратко описывается структура РДС как приложения Windows: перечислены основные папки и файлы приложения. Они будут описаны более подробно в следующих главах.

РДС состоит из основной программы (“rds.exe”) и набора библиотек в папке “Dll\”, обеспечивающих работу различных блоков в схемах. Основная программа позволяет редактировать схемы, обеспечивает вызовы внешних модулей, отвечающих за работу конкретных блоков, а также содержит большую библиотеку *сервисных функций*, облегчающих взаимодействие блоков с основной программой и между собой (здесь и далее курсивом выделены термины, используемые в диалогах РДС).

Кроме “rds.exe” и папки “Dll\” в состав РДС могут входить:

- Библиотека RdsCtrl.dll, позволяющая управлять основной программой из других приложений;
- Папки “Panel\” и “Library\” – содержимое панели стандартных блоков и библиотеки стандартных блоков соответственно;
- Папка “Template\” – набор шаблонов схем;
- Папка “Models\” – набор автоматически компилируемых моделей, используемых в разных схемах;
- Папка “Include\” – файлы заголовков для самостоятельного написания моделей блоков на языке C/C++;
- Папка “Doc\” – папка с документацией и файлами справки РДС.

РДС – однодокументное приложение, то есть в основную программу можно загрузить только одну схему. Если необходимо работать с несколькими схемами одновременно, следует запустить несколько экземпляров программы. Основная программа “rds.exe” содержит и среду разработки, и большую библиотеку функций, которые используются моделями блоков для взаимодействия с РДС и между собой. Среда разработки не может быть отделена от этой библиотеки – отдельной версии РДС, позволяющей только запускать расчет в заранее созданных схемах, не существует.

В РДС встроены интерфейсы, позволяющие запускать основную программу из других приложений и управлять ее действиями: загружать схемы, запускать расчет, обмениваться информацией с блоками. Эти интерфейсы также позволяют управляющему приложению реагировать на события, произошедшие в РДС (например, на действия пользователя) и получать сообщения от блоков. РДС не является COM-сервером, все эти интерфейсы реализованы на низком уровне при помощи Windows API. Их использование может оказаться сложным для программиста, не имевшего ранее дела с передачей информации между процессами при помощи сообщений, общей памяти и каналов передачи данных (pipes). Чтобы облегчить программистам жизнь, в состав РДС включена библиотека RdsCtrl.dll, содержащая функции более высокого уровня, обеспечивающие более удобный обмен данными с РДС. Управляющее приложение может загрузить эту библиотеку и, вызывая ее функции, управлять РДС, передавать произвольные данные блокам, а также реагировать на события (для такой реакции приложение может зарегистрировать свои функции обратного вызова, которые будут вызываться при наступлении событий). Кроме того, если управляющее приложение работает с окнами, оно, используя RdsCtrl.dll, может настроить РДС так, что схемы будут изображаться и работать внутри окна управляющей программы. При этом пользователь может даже не догадываться, что эти схемы на самом

деле обслуживаются РДС. Библиотека RdsCtrl.dll и ее использование подробно рассмотрено в главе 3 и приложении Б.

Панель и библиотека стандартных блоков в РДС организованы как папки и файлы внутри папок “Panel\” и “Library\”. Каждая вкладка панели блоков представляет собой папку внутри “Panel\”, имя этой папки является названием вкладки. Подразделы библиотеки блоков тоже представляют собой папки внутри “Library\”, однако, если в панели блоков внутри папок-вкладок не может быть других папок (на панели могут находиться только блоки), в библиотеке может быть сколько угодно вложенных подразделов. С каждым блоком библиотеки и панели связано несколько файлов с одинаковым названием и разными расширениями:

- “.blk” или “.rds” – собственно описание библиотечного блока в стандартном формате РДС. Для простых блоков используется расширение “.blk”, для подсистем, сохраненных как единый блок (см. §1.2) – расширение “.rds”.
- “.bmp”, “.ico”, “.i16”, “.i32”, “.b16”, “.b32” – растровые изображения, используемые в качестве значка блока на вкладке панели или в библиотеке. В файлах “.bmp”, “.b16” и “.b32” находятся изображения в стандартном растровом формате Windows (bitmap): в “.b16” – 16x16 точек, в “.b32” – 32x32, в “.bmp” – любого размера. В файлах “.ico”, “.i16” и “.i32” содержатся иконки в стандартном формате Windows – произвольного размера, 16x16 и 32x32 точки соответственно. Значки 16x16 используются на панели блоков и в библиотеке в режимах “мелкие значки” и “список”, 32x32 – только в библиотеке и только в режиме “крупные значки”. Если РДС не может найти в этих файлах значок нужного размера, этот значок будет автоматически сгенерирован путем сжатия или растягивания найденного значка, однако, качество изображения при этом, как правило, ухудшается. Левая нижняя точка растрового изображения всегда считается прозрачной, поэтому при его выводе все точки того же цвета будут заменены на цвет фона. Формат иконки включает в себя маску прозрачности, поэтому цвет левой нижней точки в ней не анализируется. Для блока в библиотеке или на панели может быть задан любой набор из указанных графических файлов, РДС будет искать среди них подходящий. При поиске мелкого значка файлы просматриваются в следующем порядке: “.i16” → “.b16” → “.ico” → “.bmp”, при поиске крупного – “.i32” → “.b32” → “.ico” → “.bmp”. Таким образом, если, например, в папке есть файл с расширением “.i16”, РДС не будет обращаться к файлам “.b16”, “.ico” и “.bmp” при поиске мелкого значка. Подразделы библиотеки блоков тоже могут иметь такие файлы со значками – в этом случае, имя файла должно совпадать с именем папки подраздела.
- “.txt” – текст всплывающей подсказки блока, которая выводится при наведении курсора на его изображение на вкладке панели блоков или в окне библиотеки. Этот файл может отсутствовать, тогда у блока не будет всплывающей подсказки.

При добавлении в схему блока из библиотеки или с панели блоков этому блоку дается имя, соответствующее имени файла, в котором находится его описание, с добавлением порядкового номера. Например, если блок хранится в файле “MyBlock.blk”, первый добавленный в подсистему блок получит имя “MyBlock1”, второй – “MyBlock2”, и т.д.

В библиотеке блоки и подразделы всегда отсортированы в алфавитном порядке: блоки – по имени файла, а подразделы – по имени папки. Порядок вкладок на панели блоков и блоков на каждой вкладке задается пользователем, поэтому, кроме указанных выше файлов, в папке “Panel\” и каждой ее внутренней папке находится файл “order.lst”, в котором в текстовом виде описан порядок следования блоков на вкладке или вкладок на панели – по одному имени блока (вкладки) на каждой строке.

В папке “Include\” находятся файлы заголовков для написания моделей блоков и других программ, взаимодействующих с РДС, на языке C или C++. В других языках программирования эти файлы нельзя использовать напрямую, но описания типов и констант,

находящиеся в них, могут помочь в создании таких же описаний на другом языке. Обычно в этой папке находятся следующие файлы:

- “RdsDef.h” – описания типов и констант, необходимые как для написания функций моделей блоков, так и для прочих модулей и программ, работающих с РДС. Эти описания будут постоянно использоваться в этом тексте.
- “RdsFunc.h” – макросы для быстрого получения доступа ко всем сервисным функциям РДС, то есть функциям, экспортированным из “rds.exe”. Функции моделей блоков можно писать и без использования этого файла, но с ним это делать гораздо удобнее – он автоматизирует получение указателей на все сервисные функции. Включение этого файла в программу модели подробно рассмотрено в §2.2.
- “RdsComp.h” – описания типов и констант, необходимые для создания новых модулей автоматической компиляции моделей блоков. Для написания самих моделей блоков этот файл не требуется. Создание пользовательских модулей автокомпиляции рассмотрено в главе 4.
- “RdsCtrl.h” – описания типов, констант и функций, используемых для работы с библиотекой RdsCtrl.dll (ее использование подробно описано в главе 3), которая позволяет управлять РДС из другого приложения.
- “RdsRem.h” – описания интерфейсов дистанционного управления, которыми пользуется библиотека RdsCtrl.dll при взаимодействии с РДС. Этот файл обычно не нужен для написания программ, работающих с РДС, он может потребоваться программисту, решившему написать свою библиотеку вместо RdsCtrl.dll. Рассмотрение интерфейсов, описанных в этом файле, выходит за рамки данного руководства.
- “CommonAC.hpp” – макросы для стандартных модулей автоматической компиляции моделей блоков. Эти макросы используются модулями, включенными в состав библиотеки “Common.dll”. Файл “CommonAC.hpp” не нужно включать в программы вручную – модули автокомпиляции включают его автоматически, а для написания моделей без использования автокомпиляции он не нужен.
- “CommonBl.h” – типы и имена функций блоков и динамических переменных из стандартной библиотеки “Common.dll”. Этот файл имеет смысл включать в программы моделей, которые будут взаимодействовать с блоками из этой библиотеки.
- Другие файлы заголовков, которые не используются программистом непосредственно. Они подключаются автоматически при включении в текст программы перечисленных выше файлов.

Все эти файлы, кроме “CommonAC.hpp”, содержат описания в синтаксисе “простого” C, то есть без использования классов и других дополнений, появившихся в C++. Таким образом, файлы заголовков можно использовать при написании моделей как на C++, так и на C.

§1.2. Блоки РДС и их типы

Описываются типы блоков, из которых состоит схема РДС, и особенности каждого из этих типов.

Схема в РДС состоит из набора *блоков*, соединенных друг с другом. Блоку может ставиться в соответствие функция-*модель* во внешней библиотеке (DLL), определяющая его реакцию на различные события и действия пользователя (при вызове модели ей передается одна из констант `RDS_BFM_*`, соответствующая произошедшему событию). У большинства блоков есть набор *статических переменных*, то есть переменных, которые, как правило, создаются вместе с блоком, и не изменяют свою структуру в процессе его работы. Они служат для хранения данных и передачи их между блоками. Статическая переменная может быть входом блока, выходом или внутренней. Выход одного блока может соединяться со входами нескольких других при помощи *связи*, которая изображается как линия (возможно, разветвленная) со стрелками в местах соединения со входами блоков. Каждая связь может быть подключена только к одному выходу блока и к произвольному числу входов. К одному

входу блока может быть подключено несколько связей, в этом случае на вход будут переданы данные той связи, которая сработала последней. Связь не может реагировать на какие-либо события или изменять передаваемые данные, вся обработка должна осуществляться в блоках, к которым подключена эта связь. Кроме обычных связей в схеме могут присутствовать *шины* – группы связей, изображаемых одной линией (как правило, жирной). Шина состоит из независимых друг от друга *каналов* передачи данных, к которым снаружи могут подключаться обычные связи (одна ко входу канала, произвольное число к выходу).

Все блоки в РДС принадлежат к одному из пяти типов:

- *Простой блок*. Это – самый распространенный тип блока (рис. 1). Его поведение определяется только его параметрами и моделью в DLL. Статические переменные простого блока чаще всего задаются при его создании и более не изменяются, поэтому внутри функции модели к ним можно обращаться непосредственно по адресам. Структура этих переменных может быть любой, но первые две переменных любого простого блока всегда жестко зафиксированы: первая из них является входом, управляющим запуском модели блока, а вторая – выходом, сигнализирующим о срабатывании модели и разрешающим передачу данных по связям. Особенности работы с этими переменными будут описаны в §1.3.

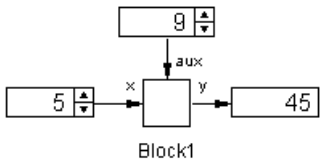


Рис. 1. Простой блок с подключенными связями

- *Подсистема* – блок, содержащий внутри себя другие блоки и связи (рис. 2). Каждая подсистема отображается в отдельном окне. Статические переменные подсистемы появляются и исчезают при редактировании ее содержимого, поэтому обращение к ним из функции модели нежелательно. Схема всегда содержит хотя бы одну подсистему – *корневую* (главную). В корневой подсистеме содержатся все остальные блоки схемы.
- *Внешний вход* – блок, служащий для передачи данных внутрь подсистемы. Он имеет единственную статическую переменную-выход, при этом ей всегда соответствует статическая переменная-вход того же типа в *родительской* подсистеме (то есть в подсистеме, внутри которой находится этот блок). Для того, чтобы передать данные внутрь подсистемы, необходимо присоединить связь к переменной-входу в этой подсистеме, затем найти внутри нее внешний вход, соответствующий этой переменной, и продолжить связь от него.

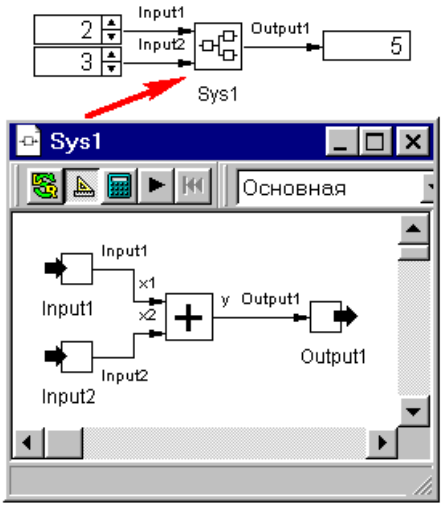


Рис. 2. Подсистема с внешними входами “Input1” и “Input2” и выходом “Output1”

- *Внешний выход* – блок, служащий для передачи данных изнутри подсистемы наружу. Он имеет единственную статическую переменную-вход, при этом ей всегда соответствует статическая переменная-выход того же типа в родительской подсистеме. Для передачи данных наружу необходимо присоединить связь к единственной переменной внешнего выхода, а затем продолжить ее от соответствующего выхода самой подсистемы.
- *Ввод шины* – блок, позволяющий соединить шину внутри подсистемы с шиной снаружи (рис. 3). Соединенные шины имеют один и тот же набор каналов и обеспечивают передачу данных между подсистемами. Для того, чтобы соединить внутреннюю и внешнюю шины, необходимо добавить в подсистему ввод, соединить с ним внутреннюю шину, а затем

соединить внешнюю шину с подсистемой, указав имя этого ввода (можно действовать и в обратной последовательности).

Блоки можно сохранять в отдельные файлы для использования в других схемах. Именно так устроены библиотека и панель стандартных блоков РДС: это просто набор отдельных файлов в соответствующих папках. Простые блоки, внешние входы и выходы, вводы шин обычно сохраняются в файлы с расширением ".blk", а подсистемы – в файлы с расширением ".rds". Это же расширение имеют и схемы, поскольку файл схемы – это, по сути, корневая подсистема схемы, сохраненная в файл. Таким образом, сохраненные схемы можно, при желании, использовать как блоки в других схемах.

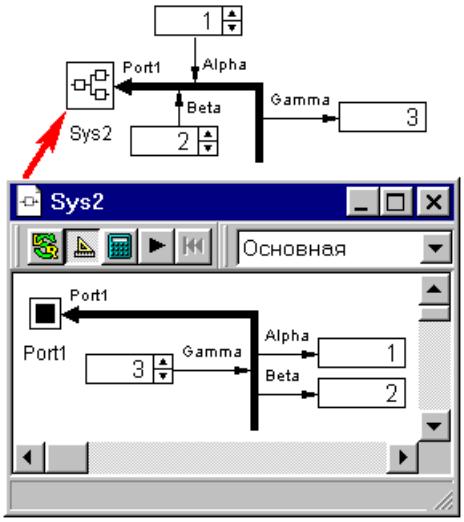


Рис. 3. Ввод шины в подсистему

§1.3. Режимы работы РДС

Описываются три режима работы РДС и циклический вызов моделей блоков в режиме расчета.

РДС может находиться в одном из трех режимов: *редактирования*, *моделирования* и *расчета*.

В режиме редактирования пользователь может изменять схему, добавляя и удаляя блоки и связи, вводя их параметры и т.п. При этом информация о движении курсора и нажатии кнопок мыши и клавиш не передается в модели блоков. Тем не менее, блоки продолжают реагировать на остальные события пользовательского интерфейса: вывод всплывающей подсказки, вызов настройки, перемещение и т.п.

В режиме моделирования пользователь не может ни редактировать схему, ни вызвать функцию настройки блока – все события мыши и клавиатуры передаются в функцию модели.

В режиме расчета все события мыши и клавиатуры также передаются в функции блоков. Кроме того, постоянно (как правило, в отдельном потоке) циклически выполняется следующая последовательность действий:

- По очереди в произвольном порядке вызываются все модели простых блоков (при этом в функцию модели передается идентификатор события `RDS_BFM_MODEL`), у которых установлен параметр "запуск каждый такт" или значение первой однобайтовой статической переменной (как правило, она называется "Start") равно единице. При этом перед вызовом каждой модели значение переменной `Start` сбрасывается в 0, и второй однобайтовой статической переменной блока (как правило, она называется "Ready") присваивается единица. Перед вызовом реакции `RDS_BFM_MODEL` у блоков, модели которых взвели флаг `RDS_CTRLCALC` в своей структуре данных (см. стр. 36), вызывается дополнительная реакция `RDS_BFM_PREMODEL` (см. §2.14.5).
- Для всех простых блоков, у которых переменная `Ready` имеет значение 1, выполняется передача данных по связям, подключенным к выходам (некоторые выходы блока могут быть запрещены, в этом случае передача данных этого выхода не производится). Таким образом, все сработавшие блоки передают свои данные. После передачи значение `Ready` сбрасывается в 0.
- Начинается следующий *такт* расчета: снова вызываются все модели простых блоков, передаются данные по связям, и т.д. до тех пор, пока пользователь или один из блоков не даст команду остановить расчет.

Переменные `Start` и `Ready` (их названия можно изменить, главное, чтобы это были первая и вторая переменные блока) играют роль сигналов запуска и готовности. Модель может сама изменять значения этих переменных для управления работой блока. Например, если присвоить переменной `Start` значение 1, модель блока будет снова запущена в следующем такте расчета, даже если связь, присоединенная к этой переменной, не сработает. Если же присвоить переменной `Ready` значение 0, данные с выхода блока не будут переданы по связям, несмотря на то, что блок сработал в этом такте.

В расчете участвуют только простые блоки. Ни подсистемы, ни внешние входы/выходы, ни вводы шин в расчете не участвуют, то есть их модели вообще никогда не вызываются в режиме `RDS_BFM_MODEL` в такте расчета. Если, например, данные передаются от простого блока “Block1” внутрь подсистемы “Sys1” через внешний вход “Input1” ко второму простому блоку “Block2” (рис. 4), то ни модель подсистемы, ни модель внешнего входа вызвана не будет. Будут вызваны только модели первого и второго блоков, при этом после срабатывания первого блока данные его выхода `y` передадутся непосредственно на вход второго блока `x`, на вход `Input1` подсистемы “Sys1” и на выход ее внешнего входа “Input1”. Такой подход позволяет ускорить расчет за счет исключения промежуточных ступеней. Следует иметь в виду, что статические переменные подсистемы и внешнего входа все равно получат данные первого блока, хотя их модели и не вызываются. Это может оказаться полезным в тех случаях, когда подсистема или внешний вход имеют векторную картинку: если какие-либо элементы этой картинки связаны с переменными, изменившиеся значения этих переменных отразятся на внешнем виде блоков. Например, если добавить в векторную картинку подсистемы “Sys1” строку текста и связать ее с входом `Input1` этой подсистемы, в режимах расчета и моделирования на изображении подсистемы будет выводиться числовое значение входа `Input1`, полученное по связи от блока “Block1”.

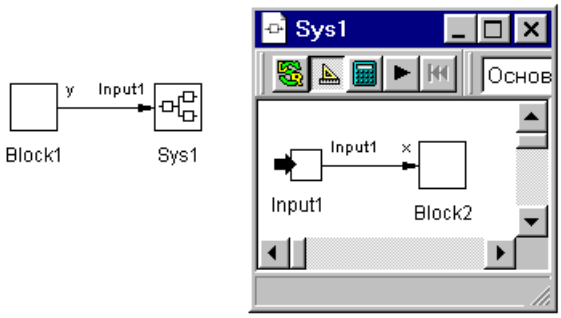


Рис. 4. В расчете участвуют только блоки 1 и 2

§1.4. Параметры и внешний вид блоков

Описывается внутренняя структура блоков РДС, способы задания их внешнего вида в окне подсистем, возможности взаимодействия блоков с пользователем.

Помимо модели блока, на его поведение и внешний вид влияют различные параметры. Эти параметры могут различаться у разных блоков с одной и той же моделью, что позволяет использовать одну модель для целого семейства похожих блоков. Например, стандартные блоки числовой индикации и ввода числа используют одну и ту же функцию модели из библиотеки “Common.dll”.

Каждый блок схемы имеет *имя*, уникальное в пределах родительской подсистемы. Имя блока – это строка произвольной длины, которая не может содержать символов двоеточия, доллара и коммерческого АТ (“ : ”, “$” и “@”). При создании нового блока или загрузке его из библиотеки имя ему присваивается автоматически, позже оно может быть изменено пользователем (рис. 5) или сервисной функцией. Для удобства пользователя имя

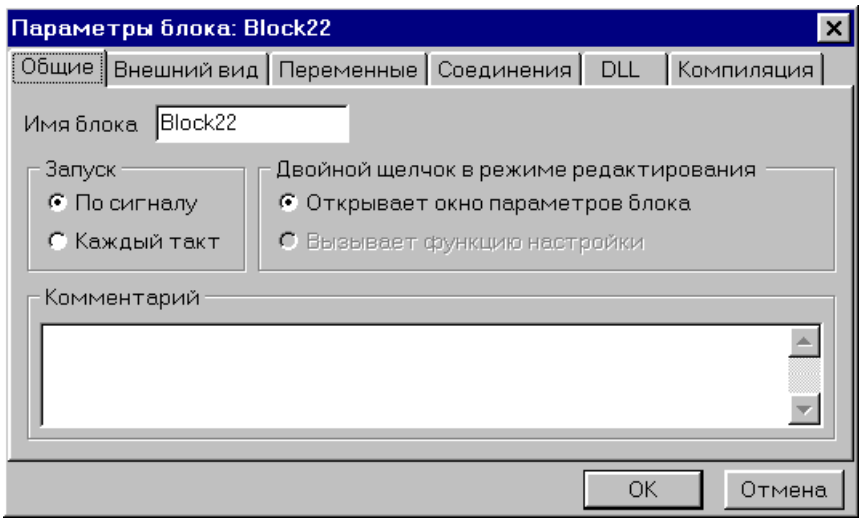


Рис. 5. Параметры блока – общие

блока обычно отображается в окне подсистемы непосредственно под изображением этого блока, но, при необходимости, оно может быть перемещено или вообще отключено. *Полное имя* блока начинается с двоеточия, за которым следует последовательное перечисление через двоеточие всех имен подсистем на пути от корневой подсистемы до этого блока, которое завершается именем самого блока. Например, полное имя ":Sys1:Sys100:Block1" говорит о том, что блок с именем Block1 находится в подсистеме Sys100, которая, в свою очередь, находится в подсистеме Sys1 корневой подсистемы. Полное имя блока в схеме всегда уникально.

Комментарий блока – это произвольный текст, связанный с блоком, который может быть задан пользователем (в окне параметров блока) или сервисными функциями. РДС позволяет искать и выделять блоки, содержащие какой-либо текст в комментарии. При необходимости, модель блока может хранить в комментарии какие-либо данные в текстовом виде, однако, для этого есть более удобные способы.

Внешний вид блока в окне подсистемы может задаваться одним из трех способов: картинкой, прямоугольником с текстом, или рисоваться моделью блока (рис. 6).

Картинка блока – это набор векторных элементов: прямоугольников, линий, многоугольников, блоков текста и т.п. Некоторые элементы и группы элементов могут быть связаны с переменными блока, в этом случае в режимах моделирования и расчета внешний вид блока будет отражать изменения этих переменных – элементы будут появляться, исчезать, перемещаться и поворачиваться, будет меняться их текст, цвет и т.п. Картинка задается в векторном редакторе, встроенном в РДС, который вызывается из вкладки "Внешний вид" окна параметров блока кнопкой "Изменить". Если для блока задано отображение картинкой, а сама картинка не задана, блок будет иметь стандартный внешний вид, определяемый его типом.

Прямоугольник с текстом обычно используется в тех случаях, когда блоку не требуется анимированное векторное изображение – для большинства простых блоков достаточно названия в прямоугольной рамке. В окне параметров блока можно задать цвет прямоугольника, шрифт, текст (может быть несколько строк) и выравнивание текста внутри прямоугольника. Связать текст с какой-либо переменной блока в этом случае нельзя.

Рисование функцией модели – самый сложный способ задания внешнего вида блока. Модель может использовать стандартные функции Windows API [8, 9] (в модель передается контекст, на котором нужно рисовать) или сервисные функции-оболочки РДС. Рисование целесообразно применять для сложных блоков, внешний вид которых зависит от многих

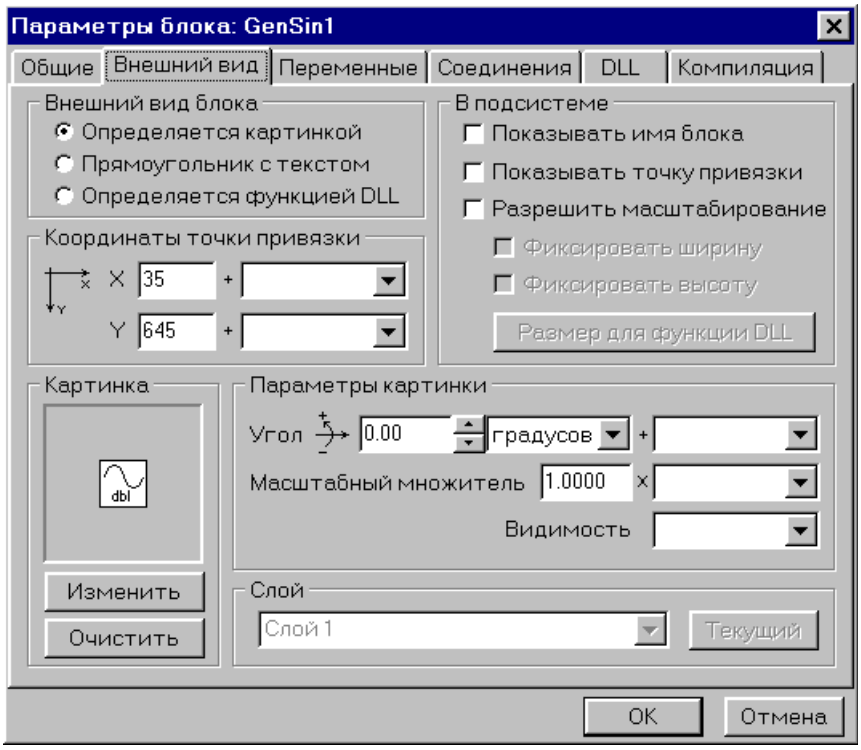


Рис. 6. Параметры блока – внешний вид

параметров (например, для графиков). Программное рисование внешнего вида блока рассматривается в §2.10.

Независимо от способа задания внешнего вида блока, его модель может нарисовать какие-либо дополнительные элементы поверх изображения (для этого существует специальное событие, на которое может отреагировать модель, см. §2.10.3). Можно, например, отображать таким образом состояние блока, или привлечь внимание пользователя к блоку, перечеркнув его изображение при возникновении каких-либо ошибок.

Способ запуска блока определяет момент срабатывания модели блока в режиме расчета. При запуске *по сигналу* модель будет запущена только в том случае, если значение первой статической переменной блока (“Start”) будет равно единице. При запуске *каждый такт* (см. рис. 5) модель будет запускаться постоянно независимо от значения переменной. Этот параметр может быть изменен как пользователем, так и вызовом сервисной функции из модели блока.

Отдельная группа параметров разрешает или запрещает реакцию блока на действия пользователя (рис. 7). Например, для того, чтобы блоки реагировали на нажатие кнопок мыши, нужно не только включить в модель обработку соответствующего события, но и разрешить реакцию на мышь в параметрах конкретных блоков с этой моделью. Можно разрешить или запретить следующие реакции:

- реакция на мышь: нажатие и отпускание кнопок мыши и перемещение курсора с нажатыми кнопками;
- перемещение курсора мыши без нажатия кнопок (включается, только если включена реакция на мышь);
- *функция настройки* – специальный вызов модели для ввода параметров блока, который связан с пунктом “Настройка” контекстного меню этого блока в режиме редактирования (при желании, можно изменить имя этого пункта);
- всплывающая подсказка блока – модель блока может сформировать текст подсказки (см. §2.11) и, при необходимости, указать время, которое подсказка будет отображаться;
- реакция на клавиатуру;

- для подсистем можно также разрешить реакцию на мышь и клавиатуру в окне этой подсистемы, если ни один блок внутри этой подсистемы не среагировал на событие.

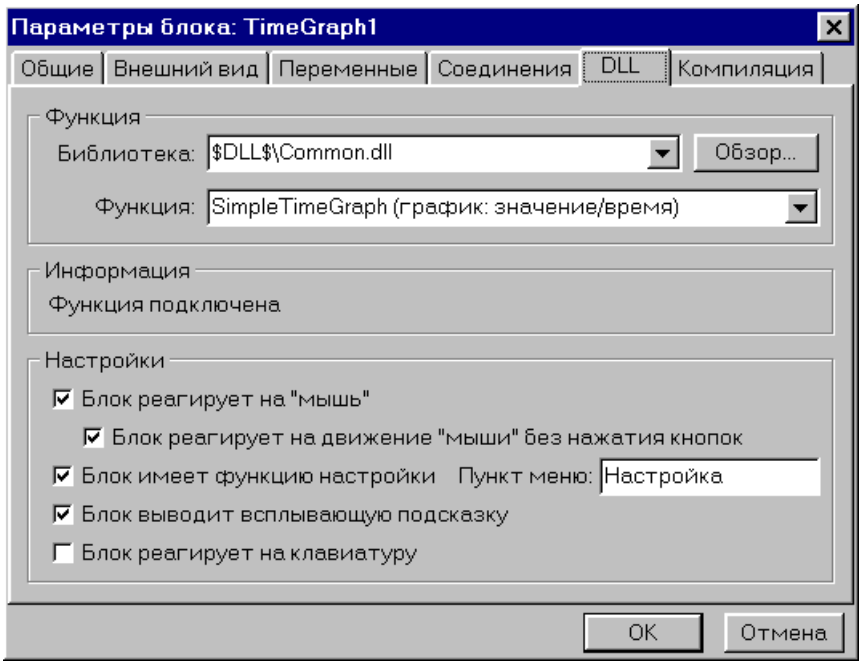


Рис. 7. Параметры блока – модель и реакции

Если для блока разрешена функция настройки, можно также включить ее вызов по двойному щелчку в режиме редактирования (см. рис. 5, по умолчанию по двойному щелчку открывается окно параметров блока).

Кроме стандартных, блок может иметь произвольное количество специализированных параметров, с которыми может работать только его модель. Для хранения таких параметров предназначена *личная область данных* блока. Для каждого блока в РДС хранится указатель на область памяти, которая может создаваться (в языке C++ – при помощи оператора `new` или функции `malloc`) и освобождаться (в C++ – оператором `delete` или функцией `free`) моделью блока. РДС не имеет доступа к личной области данных блока, все действия по ее обслуживанию должны выполняться в модели. При создании нового блока указателю на личную область данных присваивается нулевое значение (`NULL`), после чего РДС больше никогда к нему не обращается. Если модель отвела себе память под личную область и хранит там какие-либо параметры, она должна освободить эту область самостоятельно при уничтожении блока, иначе возможны утечки памяти. Поскольку РДС не работает с личной областью, при записи и загрузке схемы все действия по сохранению и загрузке параметров, которые хранятся в личной области, а также по созданию пользовательского интерфейса для редактирования этих параметров тоже должны выполняться моделью блока. Обычно для редактирования параметров используется функция настройки блока (пример модели с функцией настройки параметров приведен на стр. 119).

§1.5. Статические переменные блоков, входы и выходы

Описывается структура и возможные типы статических переменных блока, то есть тех переменных, которые обычно существуют все время жизни блока и не создаются и не уничтожаются в процессе работы схемы. Входы и выходы блока, к которым могут присоединяться связи, всегда являются статическими переменными. Описывается способ формирования строки типа, используемой для проверки правильности структуры переменных.

Статические переменные, в основном, служат для подключения связей к блоку, а также для хранения параметров, которые создатель модели блока решил не размещать в

личной области данных. Для простых блоков они, чаще всего, задаются при создании блока и указании его модели, после чего их структура уже не изменяется. Редактор переменных

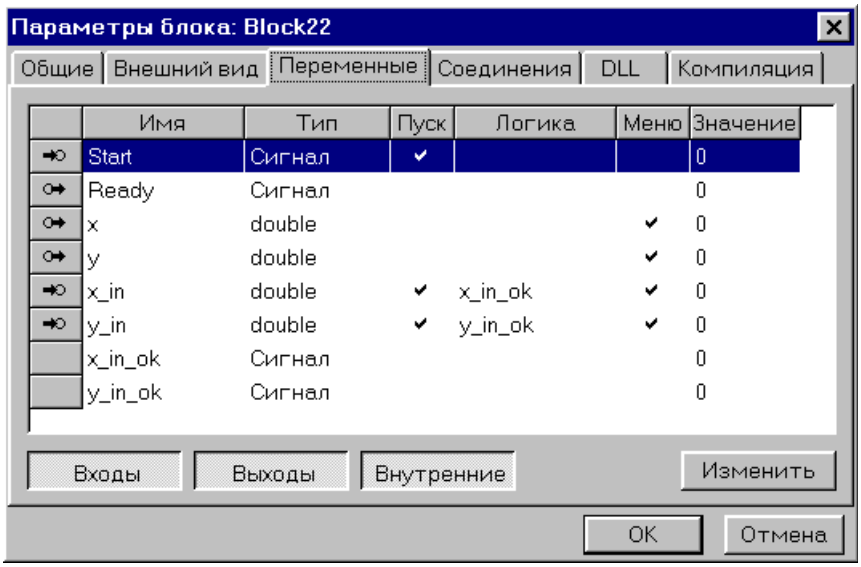


Рис. 8. Параметры блока – переменные

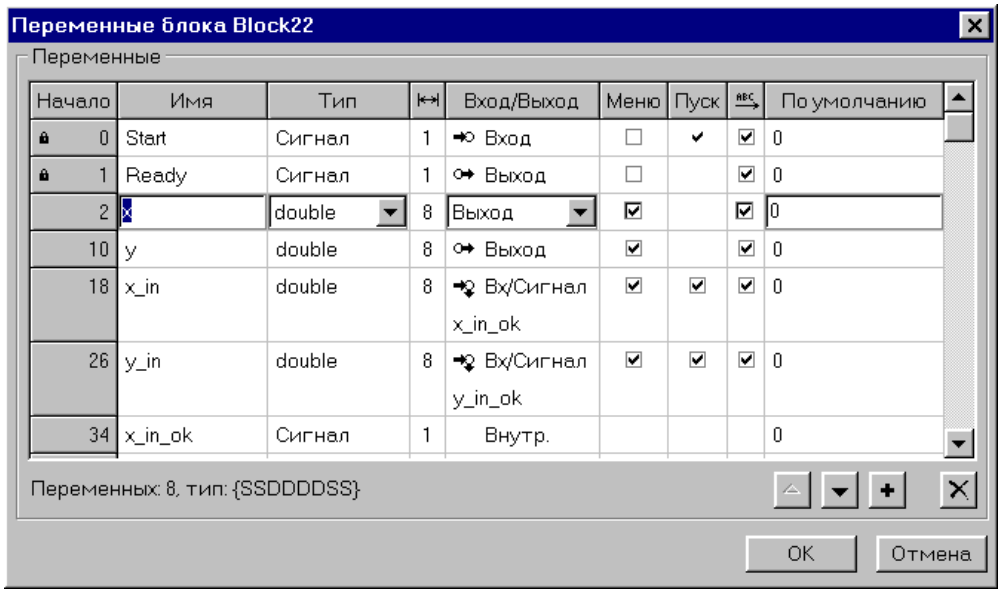


Рис. 9. Редактор переменных

(рис. 9) вызывается кнопкой “Изменить” вкладки “Переменные” окна параметров блока (рис. 8). В памяти эти переменные расположены друг за другом, поэтому, зная указатель на область переменных и размер каждой из них в памяти, модель может обращаться к различным переменным по фиксированным смещениям. Для простых переменных (целые и вещественные числа, логические значения) в памяти хранятся сами значения, а для сложных, размер которых может изменяться (массивы, матрицы, переменные произвольного типа), в памяти хранится указатель на область памяти, занимаемой сложной переменной. Таким образом, при наличии у блока сложных переменных, область переменных представляет собой дерево, поэтому набор статических переменных блока обычно называется *деревом переменных*.

Каждая статическая переменная имеет имя, уникальное в данном блоке – оно задается в колонке “Имя” редактора (см. рис. 9). Для того, чтобы имена переменных блоков можно

было использовать в программах их моделей, они должны подчиняться правилам, принятым в большинстве языков программирования: имя переменной должно содержать только буквы латинского алфавита, цифры и знак подчеркивания, и при этом оно не должно начинаться с цифры. Имена переменных чувствительны к регистру, поэтому, например, “Start” и “start” будут считаться разными переменными.

Переменные в РДС могут быть одного из следующих типов, выбираемых в колонке “Тип” редактора переменных:

- Сигнал (размер – 1 байт). Сигналы служат для передачи информации о наступлении какого-либо события, и могут принимать два значения: 1 (событие наступило) и 0 (событие не наступило). Сигнал передается по связи только в том случае, если значение выхода блока, соединенного с этой связью, равно 1. При этом после передачи значение выхода сбрасывается в 0, таким образом блокируется повторная передача информации о том же самом событии. Блок, обнаруживший на сигнальном входе единицу, должен сам сбросить ее в 0, чтобы подготовится к приему информации о следующем событии. Первая переменная простого блока всегда является сигнальным входом (обычно она называется “Start”), появление единицы в котором в режиме расчета приводит к запуску модели блока, а вторая переменная – сигнальным выходом (“Ready”), единица в котором активизирует передачу данных с выходов блока (см. режимы работы РДС в §1.3).
- Логический (размер – 1 байт). Логические переменные могут принимать одно из двух значений: 0 (ложь) и 1 (истина).
- char (размер – 1 байт). Целая однобайтовая переменная, предназначенная для хранения небольших целых чисел (диапазон значений: –128 ... 127) или кодов символов, полностью эквивалентная типу signed char в C++. Этот тип используется в блоках редко, в основном, для совместимости со старыми моделями. Для работы с целыми числами чаще всего используется тип int.
- short (размер – 2 байта). Целая двухбайтовая переменная с диапазоном значений –32768 ... 32767, эквивалентная типу `short int` в C++. В настоящее время используется редко.
- int (размер – 4 байта). Целая четырехбайтовая переменная с диапазоном значений –2147483648 ... 2147483647. Основной тип для работы с целыми числами в РДС. Эквивалентен тридцатидвухбитному типу `int` в C++.
- float (размер – 4 байта). Вещественная переменная одинарной точности с диапазоном значений модуля числа 1.18×10^{-38} ... 3.40×10^{38}. Этот тип эквивалентен одноименному типу в C++. Используется редко.
- double (размер – 8 байтов). Вещественная переменная двойной точности с диапазоном значений модуля числа 2.23×10^{-308} ... 1.79×10^{308}. Этот тип эквивалентен одноименному типу в C++, он используется во всех стандартных блоках для работы с вещественными числами.
- Строка (размер – 4 байта). Содержит указатель на строку символов произвольной длины, завершающуюся кодом 0. Если строка пустая, может содержать нулевой указатель (`NULL`). В строках РДС всегда используется кодировка Windows CP1251, многобайтовые символы Unicode не поддерживаются.
- Массив/матрица (размер – 8 байтов). Первые 4 байта переменной содержат указатель на область данных массива или матрицы или нулевой указатель (`NULL`) при отсутствии элементов. Вторые 4 байта – служебные, они используются РДС для определения типа элемента массива. Область данных матрицы содержит число строк и столбцов (первые и вторые 4 байта соответственно), за которыми следуют последовательно записанные данные элементов. Массив отличается от матрицы только тем, что при указании номера элемента используется только один индекс (второй считается нулевым). Область данных

массива ничем не отличается от области данных матрицы (в качестве числа строк всегда хранится значение 1).

- Произвольный тип (размер – 8 байтов). Универсальный тип переменной, который может изменяться в процессе работы блока. К входам произвольного типа можно подключать связи от выходов любого типа – в момент срабатывания связи такой вход получит тот же фактический тип, что и выход, передавший ему значение. Выход произвольного типа может подстраиваться под разные типы значений при работе модели блока. Первые 4 байта переменной содержат указатель на область данных переменной, вторые 4 байта – служебные, используются РДС для хранения текущего типа переменной. Переменные произвольного типа обрабатываются РДС медленнее, чем другие, поэтому их следует использовать только для данных, тип которых заранее неизвестен (например, при создании моделей универсальных мультиплексоров или демультиплексоров).
- Структура (размер – 4 байта). Содержит указатель на область памяти, в которой последовательно размещены значения полей структуры. Каждое поле – это отдельная переменная, имеющая собственные имя и тип. Связи могут подключаться как к структуре в целом, так и к ее отдельным полям. У структуры есть имя типа – произвольная строка, связанная с данным конкретным набором полей. Структуры чаще всего используются для работы со сложными данными, состав их полей может редактироваться пользователем.

Статические переменные блока могут быть внутренними, входами или выходами – их роль задается в колонке “Вход/Выход” редактора переменных. К внутренним переменным нельзя подключать связи, они обычно используются для хранения параметров блока или для управления элементами его векторной картинки.

Входы могут получать данные через связи от других блоков (автоматически в режиме расчета или при вызове специальной сервисной функции в других режимах). Можно указать необходимость автоматического запуска модели блока в режиме расчета при срабатывании связи, подключенной к данному входу – для этого служит колонка “Пуск” редактора переменных. Например, в структуре переменных, изображенной на рис. 9, включен автоматический запуск модели при срабатывании связей, подключенных к переменным `x_in` и `y_in`. Кроме того, к каждому входу блока можно привязать другую переменную сигнального типа: при срабатывании связи, подключенной к данному входу, в эту переменную автоматически запишется значение 1. Анализируя значения таких связанных сигналов, модель блока может выяснить, какие из связей, подключенных к блоку, сработали в предыдущем такте расчета. Поскольку в РДС нулевые значения сигнальных переменных не передаются при срабатывании связей, после анализа связанной сигнальной переменной модель блока должна самостоятельно сбросить эти сигналы в 0, чтобы получить возможность определить следующее срабатывание связи – иначе в переменной навсегда останется единичное значение, и модель не сможет отличить срабатывание следующей связи от предыдущей. Для привязывания сигнала к входу в редакторе переменных в колонке “Вход/Выход” нужно выбрать для данной переменной вариант “Вход/Сигнал”, и указать в этой же колонке имя связанного сигнала. На рис. 9 к входам `x_in` и `y_in` привязаны внутренние сигнальные переменные `x_in_ok` и `y_in_ok` соответственно.

К выходам блока могут подключаться связи, передающие данные на входы других блоков. С каждым выходом может быть связана дополнительная переменная логического типа – при этом связи, подключенные к этому выходу, сработают только в том случае, если эта переменная будет иметь значение 1 (истина). Для такой связи нужно в колонке “Вход/Выход” редактора переменных выбрать вариант “Выход/Логическая” и указать имя связанной логической переменной. С выходами-массивами можно связать не только логическую переменную, управляющую передачей всего выхода в целом, но и целую – в этом случае будут работать только связи, присоединенные к элементу массива, номер которого определяется этой целой переменной, и связи, присоединенные ко всему массиву как к единому целому. Следует помнить, что в режиме расчета связи, присоединенные к

выходам блока, срабатывают только в том случае, если вторая сигнальная переменная этого блока (выход “Ready”) имеет единичное значение. При запуске модели блока в режиме расчета в эту переменную автоматически записывается единица, но блок может обнулить ее, чтобы блокировать передачу по связям значений всех своих выходов.

В колонке “По умолчанию” редактора переменных задается исходное значение переменной, которое она получит сразу после загрузки схемы, добавлении нового блока с этой структурой или после сброса расчета. Для массивов и матриц можно задать только исходный размер и значение по умолчанию для одного элемента, задать разные значения по умолчанию для разных элементов массива нельзя.

Флаг в колонке редактора “Меню” указывает на необходимость добавить имя данного входа или выхода в меню подключении связи (рис. 10). К входам и выходам без этого флага все равно можно подключить связь, но в меню их имена отображаться не будут, пользователь должен будет выбрать в нем пункт “Список” и указать имя подключаемой переменной в отдельном окне. Обычно флаг “меню” устанавливают для основных входов и выходов блока, а у дополнительных и редко используемых его сбрасывают, чтобы их имена не загромождали меню подключения связи.

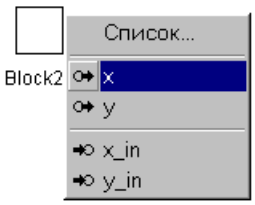


Рис. 10. Меню подключения связи

Флаг в колонке “ABC” управляет отображением имени входа или выхода на схеме по умолчанию: если он включен, при подключении новой связи к блоку рядом с точкой подключения будет выведено имя переменной, с которой соединена эта связь (см. имена переменных на рис. 1). Позже, независимо от состояния этого флага, изображение имени переменной можно включать и выключать по желанию – для этого предназначен отдельный пункт в контекстном меню точки связи.

Колонки “Начало” и “Размер” (с изображением двунаправленной стрелки), недоступные для изменения, показывают смещение к данной переменной от начала дерева и ее размер в байтах соответственно. Эти значения вычисляются автоматически, они могут пригодиться при написании модели блока для обращения к конкретной переменной при известном адресе начала дерева переменных в памяти. При написании функций моделей на языке C/C++ указатели на переменные не нужно вычислять вручную, для этого можно использовать макроопределения, генерируемые РДС (см. §2.5.1).

Для однозначного определения типов всех переменных блока используется *строка типа* (см. рис. 9, внизу слева). В этой строке каждой переменной соответствует один или несколько символов, указывающих на тип этой переменной. Большинству типов соответствует один символ: сигнал – “S”, логический – “L”, char – “C”, short – “H”, int – “I”, float – “F”, double – “D”, строка – “A”, произвольный – “V”. Массивам и матрицам соответствует символ “M”, за которым следует тип элемента (например, “MD” – матрица переменных double, “MMA” – матрица матриц строк). Структурам соответствуют символы “{” и “}”, между которыми указываются типы всех полей структуры (например, структура, состоящая из двух вещественных полей двойной точности, описывается строкой “{DD}”). На самом деле, весь набор переменных блока тоже является структурой, поэтому строка типа переменных блока всегда начинается с символа “{” и заканчивается символом “}”. Например, тип переменных стандартного блока вычитания (имеющего, помимо двух обязательных сигналов “Start” и “Ready”, два входа типа double и выход того же типа) описывается строкой “{SSDDD}” , где две буквы “S” соответствуют двум сигналам, а три буквы “D” – двум вещественным входам и выходу.

Модели блоков обычно используют строку типа для проверки, может ли данная модель работать с данной структурой статических переменных (чаще всего это делается в реакции на событие `RDS_BFM_VARCHECK`). Эта проверка очень важна, так как модели обычно работают со статическими переменными по фиксированным смещениям

относительно начала дерева переменных, и при несоответствии структуры переменных ожиданиям модели, в лучшем случае, будут считаны и записаны неверные значения, в худшем – возникнет ошибка общей защиты. Например, уже упомянутая стандартная модель блока вычитания откажется работать с любым блоком, строка типа переменных которого не “{SSDDD}”. Эта модель обращается к переменным блока по следующим смещениям: 0 – первый сигнал (`Start`), 1 – второй сигнал (`Ready`), 2 (1+1) – первая вещественная (`x1`), 10 (1+1+8) – вторая вещественная (`x2`), 18 (1+1+8+8) – третья вещественная (`y`). Если бы модель не проверяла типы переменных, при попытке подключить ее к блоку со строкой типа, например, “{SSD}”, при обращении к переменной `y` (смещение 18) возникла бы ошибка общей защиты, т. к. байты 18-25 от начала области переменных находится за пределами отведенной памяти. В строке типа нигде не указывается, является ли переменная входом или выходом, она определяет только типы переменных и их последовательность в памяти. Если в структуре переменных блока вычитания поменять местами вход `x1` и выход `y`, строка типов не изменится. Модель при этом будет работать неправильно, но, поскольку и переменная `x1`, и переменная `y` имеют одинаковый размер и тип, фатальных ошибок, вроде ошибки общей защиты, не произойдет.

§1.6. Взаимодействие блоков между собой

Описываются способы взаимодействия блоков друг с другом через динамические переменные, которые создаются в процессе работы одним блоком подсистемы и автоматически становятся доступными для других блоков. Использование динамических переменных для передачи данных не загромождает подсистему лишними связями и не требует запуска режима расчета, что иногда может оказаться очень удобным.

В РДС предусмотрено несколько способов передачи информации между блоками. Самый простой способ – передача данных по связям. Она осуществляется автоматически, однако требует запуска расчета, что не всегда удобно. Кроме того, слишком большое число связей загромождает схему. Чтобы можно было передавать данные в любом режиме, следует использовать другие способы.

Если нескольким блокам требуется получить доступ к одной, общей для всех, переменной (например, к значению текущего времени при моделировании в динамике), они могут использовать *динамическую* переменную. Динамическая переменная создается одним из блоков, после чего остальные блоки могут найти ее по имени и типу и получить к ней доступ – *подписаться* на переменную. Динамическая переменная всегда принадлежит какому-нибудь блоку, при этом не обязательно тому, который ее создал. Например, если эта переменная нужна всем блокам какой-либо подсистемы, ее целесообразно создать в этой подсистеме, чтобы все заинтересованные блоки могли ее найти. Блок может создать динамическую переменную в себе самом, в родительской подсистеме или в корневой подсистеме. При подписке на переменную можно указать режим поиска – в этом случае блок попытается найти переменную в родительской подсистеме, и, если ее там не окажется, будет двигаться вверх по иерархии до тех пор, пока не найдет переменную с заданными именем и типом или пока не дойдет до корневой подсистемы. Блок также может подписаться на переменную, которой еще не существует. Факт подписки будет запомнен, и, как только переменная будет создана, блок получит к ней доступ. Сервисные функции для работы с динамическими переменными будут рассмотрены в §2.6.

Если блоку необходимо передать информацию одному или нескольким другим блокам и получить ответ, можно использовать механизм вызова *функций блоков*. Можно вызвать функцию конкретного блока (для этого нужно знать его идентификатор) или всех блоков какой-либо подсистемы (при этом можно включать вложенные подсистемы или ограничиться только одним уровнем иерархии). Каждая функция в схеме должна иметь уникальное имя-строку, поэтому целесообразно давать им длинные осмысленные имена с указанием имени библиотеки, блоки которой ее используют (например, функция

уведомления органов управления об изменении значения, которую используют блоки из стандартной библиотеки “Common.dll”, называется “Common.ControlValueChanged”). Для того, чтобы вызвать функцию блока или ввести в блок реакцию на вызываемую функцию, ее сначала нужно *зарегистрировать* в РДС при помощи сервисной функции `rdsRegisterFunction`. Она присвоит функции блока уникальный целый идентификатор, который будет использоваться для вызова. Регистрировать функции блоков можно при загрузке DLL или при создании блока, использующего функцию. Повторная регистрация функции с тем же именем не приведет к ошибке, будет возвращен целый идентификатор, присвоенный этой функции при первой регистрации.

В модель блока, функция которого вызвана, передается структура, содержащая уникальный целый идентификатор функции, идентификатор вызвавшего блока и указатель на параметры функции. Параметры функции – это область данных (обычно структура или класс), в которой находятся входные параметры функции блока. При необходимости, туда же помещаются результаты возврата функции. Такая структура создается при написании модели блока для каждой конкретной функции. РДС никак не контролирует правильность передачи параметров, указанный при вызове функции адрес передается в модель блока без изменений. По этой причине очень важно следить за уникальностью имен функций – если параметры двух функций с одинаковыми именами будут различаться, блок может принять одну функцию за другую, и либо считать неверные значения параметров, либо, если размеры структур отличаются, обратиться к не отводившейся памяти и вызвать ошибку общей защиты. Для исключения таких ситуаций имеет смысл во все структуры параметров всех функций включать специальное поле, содержащее размер переданной структуры (причем лучше всего делать это поле самым первым в структуре), как это делается во многих функциях Windows API и сервисных функциях РДС. Несоответствие значения этого поля ожиданиям модели блока будет говорить о неправильном вызове функции.

Механизм вызова функций блоков можно использовать по-разному. Например, блок может активировать передачу данных по своим выходным связям при помощи сервисной функции `rdsActivateOutputConnections`, после чего получить из РДС список блоков, соединенных с его выходами, и информировать каждый из них об изменении входных данных вызовом функции (естественно, в моделях этих блоков должна быть реакция на эту функцию). Именно так работает функция “Common.ControlValueChanged”, которую поддерживают стандартные поля ввода и индикаторы из библиотеки “Common.dll”. Если, например, соединить связями вертикальную рукоятку и поле ввода (рис. 11), то в режиме моделирования (без запуска расчета) при перемещении рукоятки будет автоматически изменяться значение в поле ввода. Как только пользователь переместит рукоятку в другое положение, модель блока, отображающего рукоятку, вызовет сервисную функцию `rdsActivateOutputConnections`, что приведет к передаче по связи нового значения в блок, отображающий поле ввода. После этого модель рукоятки вызовет функцию “Common.ControlValueChanged” для соединенных с ней блоков, то есть для поля ввода. Модель поля ввода, реагируя на функцию, считает новое значение со своего входа и отобразит его. Эта же функция, вызываемая моделью поля ввода, приведет к тому, что при изменении значения в поле ввода рукоятка будет перемещаться в новое положение.

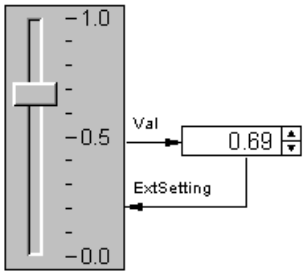


Рис. 11. Соединение рукоятки и поля ввода.

При помощи вызова функций можно также реализовать поиск блока по какому-нибудь значению, хранящемуся в личной области данных. Для этого следует зарегистрировать функцию поиска и вызвать ее для всех блоков схемы, передав в структуре параметров искомое значение. Модель блока, реагирующего на эту функцию, должна сравнить переданное значение с хранящимся в личной области данных, и, при совпадении,

записать в структуру параметров функции идентификатор блока. Примеры взаимодействия блоков при помощи вызова функций будут рассмотрены в §2.13.

§1.7. Реакция на действия пользователя

Описываются способы реакции модели блока на действия пользователя: нажатие кнопок и перемещение курсора мыши, нажатие и отпускание клавиш клавиатуры. Также описывается создание собственных пунктов меню блока и функции настройки параметров.

Самый простой способ заставить блок реагировать на действия пользователя – это разрешить данному блоку реакцию на мышь и клавиатуру (установив флаги в окне параметров блока, см. рис. 7) и добавить в модель обработку соответствующих событий. Следует, однако, помнить, что информация об этих событиях будет передаваться блоку только в режимах моделирования и расчета, в режиме редактирования часть реакций блока отключается. Кроме того, на мышь и клавиатуру не реагируют блоки, расположенные на неактивных и невидимых слоях.

Для того, чтобы блок мог реагировать на манипуляции мышью (например, если блоки должны изображать какие-либо кнопки и переключатели, они должны реагировать на щелчки мыши), необходимо включить в окне параметров блока флаг “блок реагирует на мышь”. При этом, если курсор мыши будет находиться в пределах *описывающего прямоугольника* блока (минимального прямоугольника, полностью покрывающего изображение блока в окне подсистемы), модель будет вызываться каждый раз при нажатии кнопки мыши, при ее отпускании, а также при перемещении курсора, когда хотя бы одна кнопка нажата. Чтобы модель блока вызывалась при перемещении курсора даже тогда, когда ни одна кнопка не нажата, необходимо включить в окне параметров блока флаг “блок реагирует на движение мыши без нажатия кнопок”. Это может потребоваться при создании блоков, изображающих диаграммы и графики – при перемещении курсора мыши над полем диаграммы можно выводить координаты точки под курсором.

Все описанные выше события передаются в модель блока только тогда, когда курсор мыши находится в пределах описывающего прямоугольника блока. При необходимости, модель блока может *захватить* мышь, после чего вся информация о перемещениях курсора и нажатии кнопок будет поступать только в эту модель, даже если курсор выйдет за пределы блока, до тех пор пока модель не *освободит* мышь. Этот механизм может быть полезен, например, при создании модели блока, имитирующего рукоятку для задания какого-либо значения. При нажатии левой кнопки такая модель может захватить мышь и использовать информацию о перемещении курсора для изменения положения нарисованной рукоятки, а при отпускании кнопки – освобождать мышь и фиксировать получившееся значение. Таким образом, рукоятка, рисуемая блоком, будет перемещаться, даже если пользователь выведет курсор за пределы изображения блока, до тех пор, пока он не отпустит кнопку.

Если внешний вид блока определяется векторной картинкой и ее элементам присвоены целые идентификаторы, модель при обработке событий может определить, какой элемент картинки находится под курсором мыши. Для этого используется сервисная функция `rdsGetPictureObjectId`, возвращающая идентификатор элемента картинки, на который приходится точка с заданными координатами. Можно, например, создать в картинке блока специальные активные зоны и присвоить им разные идентификаторы, а затем, обрабатывая событие нажатия кнопки мыши, увеличивать или уменьшать значение какой-либо переменной блока в зависимости от зоны, в которой находится курсор.

Для того, чтобы блок мог реагировать на клавиатуру, необходимо включить в окне параметров блока флаг “блок реагирует на клавиатуру” и добавить в модель обработку соответствующих событий (нажатия и отпускания клавиш). Модель блока будет вызываться только в том случае, если РДС находится в режиме моделирования или расчета и окно подсистемы, в которую входит блок, открыто и активно (имеет фокус). При нажатии и

отпускании клавиш по очереди вызываются модели всех блоков активного окна подсистемы, для которых разрешена реакция на клавиатуру. Вызванная модель может прекратить перебор блоков, проинформировав РДС о том, что дальнейшая обработка не требуется.

Если ни один из блоков в окне не среагировал на действия пользователя, может быть вызвана модель подсистемы, которой принадлежит это окно (для этого необходимо разрешить обработку соответствующих событий в окне параметров подсистемы). Для каждого события, связанного с мышью и клавиатурой, на которое реагируют блоки, существует похожее событие для подсистемы. Например, при нажатии кнопки мыши модели блоков вызываются с параметром `RDS_BFM_MOUSEDOWN`, а модель подсистемы, которой принадлежит окно, – с параметром `RDS_BFM_WINDOWMOUSEDOWN`. Подсистема может реагировать на оба этих события: если пользователь нажмет кнопку мыши на изображении подсистемы в окне ее родителя (где она выглядит как обычный блок, рис. 12а), ее модели будет передано событие `RDS_BFM_MOUSEDOWN`, а при нажатии кнопки на свободном месте ее собственного окна (рис. 12б) – событие `RDS_BFM_WINDOWMOUSEDOWN`.

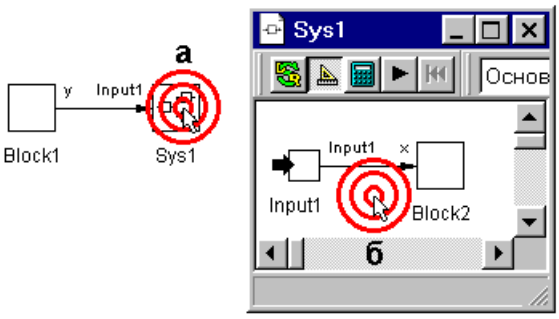


Рис. 12. Реакция подсистемы на кнопку мыши на ее изображении (а) и внутри ее окна (б)

Все описанные выше реакции работают только в режимах моделирования и расчета. Для взаимодействия с пользователем в режиме редактирования модель блока может добавить свои пункты в главное (сервисной функцией `rdsRegisterMenuItem`, описана в §2.12.7) или контекстное (сервисными функциями `rdsRegisterContextMenuItem` и `rdsAdditionalContextMenuItem`, описаны в §2.12.6) меню РДС. При добавлении пункта меню модель указывает целый идентификатор, который будет передаваться ей при выборе пользователем этого пункта. Расширения контекстного меню блока, добавленные моделью, отображаются во всех режимах при щелчке правой кнопкой мыши на изображении блока. В режиме редактирования они дублируются в подменю “Редактирование” главного меню, если добавивший их блок выделен. Пункты, добавленные моделями блоков в главное меню, отображаются в подменю “Система | Дополнительно” и активны независимо от режима или выделенности какого-либо блока (рис. 13). Кроме того, с пунктами главного меню могут быть связаны “горячие клавиши”, нажатие которых вызывает данный пункт меню независимо от того, какое окно в данный момент активно. Таким образом, добавив в главное меню РДС собственный пункт и задав для него сочетание клавиш, модель блока сможет реагировать на нажатие этих клавиш в любом режиме, даже если окно с данным блоком не

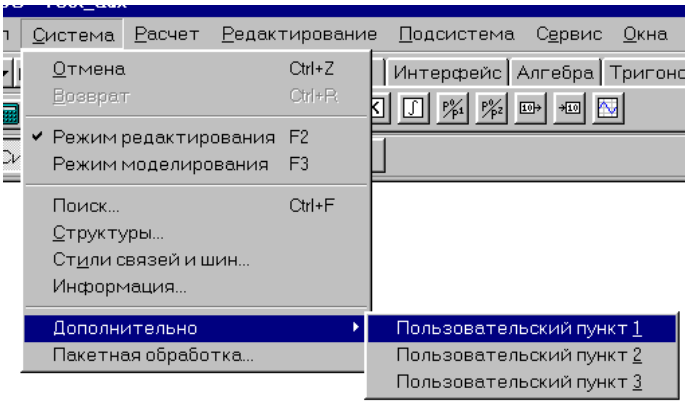


Рис. 13. Расширения системного меню

открыто. Обычно в главное меню добавляют свои пункты уникальные, то есть присутствующие в схеме в единственном экземпляре, блоки.

В режиме редактирования модель блока может также реагировать на вызов пользователем функции настройки, если он разрешен в окне параметров блока. Функции настройки соответствует пункт контекстного меню или подменю “Редактирование” (название пункта можно изменять), при выборе которого модель вызывается с параметром `RDS_BFM_SETUP`. В ответ на этот вызов модель должна самостоятельно организовать диалог с пользователем средствами Windows или при помощи сервисных функций, которые будут рассмотрены в §2.7.2.

§1.8. Открытие окон в модели блока

Описываются особенности открытия модальных (блокирующих доступ к остальным окнам приложения до своего закрытия) и немодальных (позволяющих переключаться в другие окна) окон из программы модели блока и связанные с этим проблемы, на которые следует обратить внимание. Различные примеры работы с окнами в моделях блоков будут приведены в §2.7.

Как и любая другая программа в Windows, модель блока может открывать модальные и немодальные окна для диалога с пользователем. Модальные окна отличаются от немодальных тем, что при открытом модальном окне пользователь не может перейти в какое-либо другое окно приложения, пока модальное не будет закрыто. Модель блока, открывшая модальное окно, получит управление обратно только после закрытия этого окна. Если же функция модели открывает немодальное окно, она получает управление обратно немедленно, в то время как окно остается открытым. Пользователь при этом может свободно переключаться между окнами.

Для корректного взаимодействия создаваемых моделью окон с РДС необходимо соблюдать некоторые правила. Если модель открывает немодальное окно, процедура обработки сообщений этого окна будет вызываться в произвольные моменты времени, тогда, когда это необходимо Windows. В режиме расчета к главному потоку РДС, занимающемуся обслуживанием окон и интерфейса, добавляется поток расчета, по очереди вызывающий модели простых блоков и передающий данные по связям между блоками. При этом необходимо синхронизировать работу процедуры окна, которая, вероятнее всего, будет работать в главном потоке, с потоком расчета. Если оба потока одновременно обратятся к каким-либо данным блока, могут возникнуть серьезные ошибки. При обращении к данным изнутри функции модели таких проблем не возникает, поскольку РДС самостоятельно синхронизирует вызовы моделей блоков в обоих потоках. Однако, процедура окна может быть вызвана Windows в любой момент, без синхронизации, за которой следит РДС. Поэтому, чтобы избежать коллизий, в процедуре окна (и связанных с ней функциях, если они есть) необходимо перед любым обращением к данным блока и вызовом сервисных функций РДС вызывать функцию `rdsLockBlockData`, которая заблокирует данные блока и запретит другому потоку доступ к ним. После выполнения всех необходимых действий с данными блока необходимо вызвать функцию `rdsUnlockBlockData`, которая разблокирует данные и снова разрешит к ним доступ. Если второй поток попытается обратиться к данным блока после вызова `rdsLockBlockData`, он будет остановлен до тех пор, пока не будет вызвана `rdsUnlockBlockData`, поэтому желательно разблокировать данные блока как можно быстрее. Крайне необходимо следить за тем, чтобы за каждым вызовом функции `rdsLockBlockData` обязательно следовал вызов `rdsUnlockBlockData`, иначе один из потоков может зависнуть. Если компилятор поддерживает конструкции типа `try...finally`, целесообразно использовать их для слежения за парностью этих вызовов, например (для Borland C++):

```
rdsLockBlockData(); // Блокировка данных
try
```

```
  { // .....
    // Действия с данными блока или вызов сервисных функций
    // .....
  }
__finally
  { rdsUnlockBlockData(); // Снятие блокировки
  }
```

Немодальные окна чаще всего используются моделями блоков для постоянной индикации каких-либо параметров, которую невозможно или неудобно осуществлять в окне подсистемы (например, для вывода трехмерного изображения) и для предоставления пользователю возможности менять параметры блока в реальном времени. В таких случаях целесообразно блокировать данные только на время чтения параметров из внутренних структур блока и записи изменений, сделанных пользователем. Все остальное время данные не должны быть заблокированы, иначе поток расчета будет постоянно простаивать.

Для удобства пользователя модель блока может зарегистрировать созданные ей немодальные окна в РДС при помощи сервисной функции `rdsRegisterWindow`, которая добавит название созданного окна в меню “Окна” и кнопку для вызова этого окна на панель окон в главном окне РДС. При этом следует информировать РДС об активации зарегистрированного окна функцией `rdsRegWinActivateNotify`.

Обычно открытие окон связывают с выбором каких-либо пунктов меню, созданных блоком, или с нажатием кнопок мыши или клавиш. Все эти действия выполняются в главном потоке РДС. Открывать окна из потока расчета (при вызове модели с параметром `RDS_BFM_MODEL`) не рекомендуется. Следует также помнить, что при удалении блока модель обязана уничтожить все открытые этим блоком окна.

Модальные окна обычно применяются для организации диалога с пользователем, ввода параметров блока и т.п. Поскольку при открытии модального окна управление возвращается вызвавшей программе только после закрытия этого окна, в вызове функций синхронизации `rdsLockBlockData` и `rdsUnlockBlockData` нет необходимости. Открытие модального окна происходит внутри функции модели блока, а перед вызовом модели данные блокируются автоматически. Таким образом, в процедуре модального окна можно в любой момент обращаться к данным блока и вызывать сервисные функции РДС. Это сильно упрощает написание процедуры модального окна, но порождает следующее ограничение: ***крайне нежелательно открывать модальные окна в режиме расчета***, когда одновременно работают два потока. Поскольку на момент открытия окна данные уже заблокированы, второй поток, тоже попытавшийся обратиться к данным, будет простаивать до тех пор, пока окно не будет закрыто пользователем и функция модели, открывшая окно, не завершится, вернув управление РДС. В режимах моделирования и редактирования, в которых работает только один поток, открытие модальных окон не вызывает никаких проблем.

Если логика работы блока требует открытия модального окна в режиме расчета, и это окно не может быть заменено немодальным, можно воспользоваться механизмом вызова функции без блокировки данных. Для этого нужно создать функцию специального вида, которая будет открывать модальное окно, после чего передать указатель на эту функцию сервисной функции РДС `rdsUnlockAndCall`. Эта функция снимет блокировку данных, после чего вызовет указанную функцию пользователя, по завершении которой блокировка будет восстановлена. Поскольку в этом случае на время открытия модального окна блокировка данных снимается, в процедуре окна, несмотря на его модальность, необходимо вызывать `rdsLockBlockData` перед обращением к данным блока и `rdsUnlockBlockData` после него. Пример использования функции приведен в §2.7.6.

В потоке расчета, то есть при вызове модели с параметром `RDS_BFM_MODEL` для выполнения одного такта расчета (см. §1.3) открывать модальные окна ***недопустимо***.

Независимо от способа синхронизации, функция модели, вызванная в потоке расчета, не получит обратно управления до закрытия модального окна, что приведет к остановке всего потока расчета.

После закрытия модального окна управление возвращается функции, открывшей его, поэтому ни в коем случае нельзя допускать удаления блока или выгрузки DLL с моделью, пока окно не будет закрыто. Если это случится, после закрытия окна управление будет передано в освобожденную область памяти, что, вероятнее всего, приведет к ошибке общей защиты. DLL с функцией модели блока всегда выгружается при очистке всей схемы (например, перед загрузкой другой схемы или перед завершением РДС). Она также может быть выгружена при удалении блока или отключении его модели, если в схеме больше не осталось блоков, использующих эту модель.

Пока модальное окно открыто, пользователь не сможет ни загрузить другую схему, ни закрыть РДС, ни удалить какой-либо блок. Модальное окно блокирует доступ к другим окнам, включая главное окно РДС и окна подсистем, поэтому пользователь просто не сможет выполнить соответствующие действия. Может показаться, что выгрузка DLL с моделью блока, открывшего модальное окно, невозможна. Однако, загрузить схему или удалить блок может не только пользователь. Если РДС работает под управлением другого приложения (например, использующего библиотеку RdsCtrl.dll, подробно описанную в главе 2), это приложение может приказать РДС загрузить другую схему независимо от наличия открытых модальных окон. Открытое модальное окно также не может помешать какой-либо модели блока, вызванной по таймеру или в потоке расчета, загрузить другую схему или удалить блок при помощи соответствующих сервисных функций. Чтобы избежать катастрофических последствий из-за не вовремя выгруженной DLL, модель блока должна информировать РДС об открытии и закрытии модальных окон при помощи сервисных функций `rdsBlockModalWinOpen` и `rdsBlockModalWinClose` соответственно (для слежения за парностью вызовов этих функций можно использовать конструкцию `try...finally`, но в приведенный ниже пример она не включена):

```
// Информация об открытии окна блоком Block
rdsBlockModalWinOpen(Block);
// ...
// Открытие модального окна средствами Windows
// ...
rdsBlockModalWinClose(Block); // Информация о закрытии окна
```

Обе функции принимают единственный параметр – идентификатор блока, открывающего модальное окно (в примере – `Block`). Если функция вызывается непосредственно из модели, можно вместо идентификатора блока указать `NULL` – РДС самостоятельно определит, модель какого блока вызывается в данный момент. Если же модальное окно открывается из немодального (например, при нажатии какой-либо кнопки в окне), идентификатор блока указывать обязательно. Поскольку процедура немодального окна вызывается Windows в произвольные моменты времени без всякой синхронизации с вызовами моделей, РДС в этом случае не сможет определить, к какому именно блоку относится данное модальное окно.

Вызов функции `rdsBlockModalWinOpen` сообщает РДС об открытии нового модального окна, которое будет считаться открытым до вызова функции `rdsBlockModalWinClose`. В промежутке между этими двумя вызовами удаление блока, открывшего окно, будет запрещено. Если после вызова `rdsBlockModalWinOpen` от управляющего приложения или от другой модели блока поступит команда загрузить другую схему, РДС попытается закрыть модальное окно, по очереди передавая в окна верхнего уровня стандартное сообщение Windows `WM_CLOSE` до тех пор, пока данное окно не закроется (о закрытии окна РДС узнает, получив вызов `rdsBlockModalWinClose`). Чтобы этот механизм сработал, все модальные окна, открываемые моделями блоков, должны принадлежать главному окну РДС (его дескриптор можно получить при помощи сервисной

функции rdsGetAppWindowHandle). Пример открытия модального окна средствами Windows с использованием сервисных функций rdsBlockModalWinOpen из rdsBlockModalWinClose приведен в §2.7.5.

В некоторых случаях логика работы блока требует вывода запроса о необходимости сохранения введенных пользователем данных при закрытии модального окна. Обычно окно запроса содержит три кнопки: “Да”, “Нет” и “Отмена”, причем при нажатии кнопки “Отмена” модальное окно не закрывается. Написание процедуры модального окна в этом случае требует осторожности, поскольку механизм закрытия окон при помощи сообщения WM_CLOSE, описанный выше, может привести к появлению бесконечного цикла: модальное окно получает сообщение WM_CLOSE и выводит запрос о сохранении данных, окно запроса тоже получает сообщение WM_CLOSE (что равносильно нажатию кнопки “Отмена”) и возвращает управление модальному окну, не закрывая его, модальное окно снова получает сообщение WM_CLOSE и т.д. Чтобы избежать этого цикла, перед выводом запроса следует вызвать сервисную функцию rdsModalWindowMustClose, которая вернет значение TRUE, если в данный момент идет принудительное закрытие модальных окон. В этом случае модальное окно необходимо закрыть, не выводя запросов пользователю.

Если модальные окна открываются не стандартными средствами Windows, а специальными сервисными функциями РДС (rdsMessageBox, rdsCallColorDialog, rdsFORMShowModalEx и т.п.), в вызове функций rdsBlockModalWinOpen и rdsBlockModalWinClose нет необходимости. Сервисные функции, открывающие модальные окна, самостоятельно информируют РДС об их открытии и закрытии.

Глава 2. Создание моделей блоков

В главе подробно разбираются различные аспекты написания моделей блоков, реакции на системные события, вызов сервисных функций РДС. Приводятся примеры исходных текстов моделей на языке C++ с подробным описанием работы этих моделей.

§2.1. Программы моделей и DLL

Описывается общий принцип размещения моделей в динамически подключаемых библиотеках (DLL) и способ компиляции примеров моделей, рассматриваемых в этой главе.

Модель блока в РДС – это экспортированная из динамически подключаемой библиотеки (DLL) функция с определенным набором параметров. В одной библиотеке может находится несколько моделей, поэтому некоторые действия, общие для всех моделей (инициализация, регистрация функций блоков и т.п.), часто выносятся в главную функцию DLL (обычно она называется `DllEntryPoint` или `DllMain`, но может иметь и другое имя). РДС всегда загружает DLL с моделями блоков динамически, при помощи функции Windows API `LoadLibrary` [8, 9], и всегда выгружает их перед загрузкой новой схемы, даже если в новой схеме используются те же самые модели. Все примеры, приведенные ниже, написаны на C или C++ и рассчитаны на Borland C++ Builder 5/6 или бесплатный консольный компилятор Borland C++ 5.5. Для того, чтобы консольный компилятор создал именно DLL, а не обычный исполняемый файл (EXE), который он формирует по умолчанию, можно использовать следующие параметры командной строки, в которые нужно подставить пути к папкам и компилируемым файлам, как показано ниже:

для bcc32: `-I"`*`папки_заголовков`*`" -O2 -Vx -Ve -X- -a8 -k- -vi -tWD -tWM -c -q "`*`файл_C`*`"`

для ilink32: `-L"`*`папки_библиотек`*`" -D"" -aa -Tpd -x -Gn -Gi -q c0d32.obj "`*`файл_OBJ`*`" , "`*`файл_DLL`*`" , , import32.lib cw32mt.lib`

В командной строке компилятора bcc32 вместо слова “*папки_заголовков*” следует подставить полные пути ко всем папкам, в которых находятся заголовочные файлы с описаниями констант и функций стандартных библиотек C (обычно они входят в состав компилятора), а также путь к папке “\Include” из состава РДС, а вместо “*файл_C*” – путь к файлу исходного текста программы. В командной строке редактора связей ilink32 вместо слова “*папки_библиотек*” следует подставить путь к стандартным библиотекам компилятора, вместо “*файл_OBJ*” – путь к объектному файлу, сформированному компилятором, и вместо “*файл_DLL*” – путь к библиотеке DLL, которую должен создать редактор связей. Например, если Borland C++ 5.5 установлен в папку “C:\Prog\bcpp55”, РДС – в папку “C:\RDS”, а исходный текст программы находится в файле “C:\Work\model.cpp”, команды для вызова компилятора и редактора связей будут выглядеть так (жирным выделены подставленные пути):

```
C:\Prog\bcpp55\bin\bcc32
-I"C:\Prog\bcpp55\include;C:\Prog\bcpp55\include\sys"
-I"C:\RDS\include" -O2 -Vx -Ve -X- -a8 -k- -vi -tWD -tWM -c -q
C:\Work\model.cpp
C:\Prog\bcpp55\bin\ilink32 -c -L"C:\Prog\bcpp55\lib" -D"" -aa -Tpd
-x -Gn -Gi -q c0d32.obj C:\Work\model.obj , C:\Work\model.dll , ,
import32.lib cw32mt.lib
```

При этом созданная в результате компиляции библиотека будет размещаться в файле “C:\Work\model.dll”.

Для рассматриваемых примеров можно использовать и другие компиляторы. В частности, модели блоков РДС успешно компилировались при помощи Open Watcom C++ 1.4, Digital Mars C++ v846, gcc 3.4.2 и MS Visual C toolkit (все эти компиляторы бесплатны и

могут быть загружены с web-сайтов их производителей). Разумеется, для каждого компилятора нужно подобрать параметры командной строки, которые укажут ему на необходимость создать именно DLL, а не исполняемый exe-файл, указать пути к папкам с файлами заголовков РДС, перечислить необходимые для компиляции библиотеки и т.д. Эта информация содержится в описании каждого компилятора.

Многие модели блоков в этом тексте написаны с использованием классов, поэтому для их сборки нужен именно компилятор C++. Тем не менее, при желании, модели блоков можно писать и на "чистом" C: обмен данными между РДС и моделью блока осуществляется только через структуры и функции обратного вызова, при этом классы и прочие расширения C++ не используются.

Можно писать модели блоков и на других языках, если, конечно, используемый компилятор поддерживает создание DLL и работу с функциями Windows API. Как правило, если компилятор позволяет обращаться к функциям API, он также позволит обратиться к сервисным функциям РДС – они имеют похожие типы вызова, описанные в следующем параграфе.

§2.2. Главная функция DLL и файлы заголовков

Описывается главная функция DLL, которую должна иметь каждая библиотека с моделями блоков, а также реализация взаимодействия программы модели с главной программой РДС при помощи сервисных функций. Приводится пример использования макроса, позволяющего получить доступ ко всем функциям РДС сразу.

Для создания библиотеки с моделью блока в файл исходного текста необходимо включить по крайней мере два файла заголовков: "windows.h", содержащий стандартные описания Windows, и "RdsDef.h" из комплекта РДС (папка "Include\", см. §1.1), содержащий описания типов и констант, необходимые для моделей блоков. Для создания простейших моделей ничего больше не требуется. Если в модели будут использоваться какие-либо сервисные функции РДС, надо предпринять несколько дополнительных шагов, чтобы получить к ним доступ.

Сервисные функции РДС – это обычные функции, экспортированные из модуля "rds.exe". Все они имеют тип вызова `RDSCALL`, описанный в файле "RdsDef.h". Этому типу полностью соответствуют `CALLBACK` в Windows API и `__stdcall` в Borland C++ (аргументы функции передаются в стеке справа налево, стек освобождается вызванной функцией). Для получения их адресов (указателей на функции) следует использовать функцию Windows API `GetProcAddress`.

Проще всего получить указатели на все необходимые сервисные функции в момент загрузки библиотеки с моделями блоков. Допустим, что для написания модели какого-либо блока требуется сервисная функция `rdsMessageBox`, выводящая окно с сообщением (при написании моделей она может быть удобнее стандартной функции Windows API `MessageBox`, поскольку для функции РДС не нужно указывать окно-владелец, и, кроме того, при вызове этой функции из режима расчета расчет не останавливается). Ниже приведен фрагмент исходного текста библиотеки с главной функцией DLL, включающей команды получения доступа к этой сервисной функции.

```
#include <windows.h>        // Стандартные описания Windows
#include <RdsDef.h>         // Описания РДС

// Описание переменной-указателя на функцию
RDS_ISSI rdsMessageBox;

// Главная функция DLL
int WINAPI DllEntryPoint(
        HINSTANCE /*hinst*/,      // Дескриптор модуля этой DLL
```

```
        unsigned long reason,     // Причина вызова (загрузка или
                                  // выгрузка DLL)
        void* /*lpReserved*/      // Способ загрузки – динамический
                                  // или статический
    )
{ if(reason==DLL_PROCESS_ATTACH) // Загрузка DLL
    // Получение указателя на функцию rdsMessageBox
    rdsMessageBox=(RDS_ISSI)GetProcAddress(GetModuleHandle(NULL),
                                           "rdsMessageBox");
  // Возврат – успешное завершение
  return 1;
}
//-----------------------------------------
// ... здесь должны быть функции моделей ...
//-----------------------------------------
```

Сначала описывается глобальная переменная `rdsMessageBox` – указатель на функцию соответствующего типа. Тип `RDS_ISSI` описан в файле "RdsDef.h" следующим образом:

```
typedef int (RDSCALL *RDS_ISSI)(LPSTR,LPSTR,int);
```

Таким образом, переменная `rdsMessageBox` становится указателем на функцию, получающую в качестве параметров две строки (`LPSTR` – тип, эквивалентный типу `char*`) и целое число. Теперь нужно присвоить этой переменной указатель на одноименную функцию в "rds.exe" – для этого мы используем главную функцию DLL.

Каждая динамически подключаемая библиотека в Windows обязательно имеет главную функцию, которая автоматически вызывается при загрузке и выгрузке этой библиотеки, а также при создании и уничтожении потоков в загрузившем библиотеку процессе. В компиляторе Borland C++ 5.5 она обычно имеет название `DllEntryPoint`, в других компиляторах может иметь другое имя, но она всегда принимает три параметра и возвращает целое число:

```
int WINAPI DllEntryPoint(
           HINSTANCE hinst,      // Дескриптор модуля этой DLL
           unsigned long reason, // Причина вызова (загрузка или
                                 // выгрузка DLL)
           void* lpReserved)     // Способ загрузки – динамический
                                 // или статический
```

В первом параметре функции `hinst` передается дескриптор (уникальный идентификатор) модуля данной DLL – он может понадобиться, если функции библиотеки будут загружать из нее же какие-либо ресурсы (строки, растровые изображения и т.п.). Во втором параметре `reason` передается одна из четырех целых констант, описанных в файлах заголовков Windows API, указывающая на причину вызова главной функции DLL: загрузка библиотеки (`DLL_PROCESS_ATTACH`), ее выгрузка (`DLL_PROCESS_DETACH`), создание или уничтожение потока внутри процесса (`DLL_THREAD_ATTACH` и `DLL_THREAD_DETACH` соответственно). Нас будет интересовать именно загрузка библиотеки – это самый подходящий момент для инициализации ее глобальных переменных. Наконец, третий параметр `lpReserved` указывает на способ загрузки и выгрузки DLL – нас он интересовать не будет. Возвращаемое главной функцией целое число информирует загружающий библиотеку процесс об успешности инициализации библиотеки: если все в порядке, нужно вернуть ненулевое число, если при инициализации произошли какие-либо ошибки и библиотеку нельзя использовать, нужно вернуть 0. Моменты вызова главной функции DLL и ее параметры подробно описаны в документации по Windows API.

В приведенном выше примере в главной функции DLL `DllEntryPoint` при загрузке библиотеки (когда параметр `reason` равен константе `DLL_PROCESS_ATTACH` из стандартных описаний Windows) глобальной переменной `rdsMessageBox` присваивается указатель на функцию с именем "rdsMessageBox", экспортированную из главного модуля, то

есть из “rds.exe”. Для получения дескриптора, то есть уникального идентификатора, главного модуля приложения используется вызов API `GetModuleHandle`, а для получения указателя на функцию с заданным именем из этого модуля – `GetProcAddress`. Затем главная функция возвращает значение 1, что сигнализирует об успешной загрузке библиотеки.

В этом примере используется только один параметр главной функции – причина вызова `reason`. Два других параметра – дескриптор модуля DLL `hinst` и дополнительный параметр `lpReserved`, указывающий способ загрузки или выгрузки DLL, не используются, поэтому их имена записаны внутри комментария. При создании библиотек с моделями блоков РДС эти параметры используются крайне редко. Если же они, тем не менее, понадобятся, комментарий можно будет убрать.

После получения указателя на функцию, ее можно вызывать непосредственно по имени, например:

```
rdsMessageBox(”Проверка вызова функции”, // Текст сообщения
              ”Сообщение”,               // Заголовок сообщения
              MB_OK|MB_ICONINFORMATION); // Кнопка OK и иконка ”i”
```

Окно сообщения, выводимое этой функцией (рис. 14), знакомо практически каждому программисту, когда-либо создававшему программу для Windows.

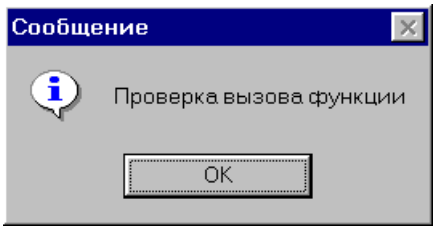


Рис. 14. Результат вызова функции `rdsMessageBox`

Если для написания моделей блоков требуется несколько сервисных функций, необходимо описать столько глобальных переменных и сделать столько вызовов `GetProcAddress`, сколько функций предполагается использовать. При большом количестве функций это не очень удобно, поэтому в таких случаях лучше включить в исходный текст заголовочный файл “RdsFunc.h”, который позволяет получить доступ сразу ко всем сервисным функциям РДС. Ниже приведен пример использования этого файла (отличия от предыдущего примера выделены жирным).

```
#include <windows.h>
#include <RdsDef.h>

// Подготовка описаний сервисных функций
#define RDS_SERV_FUNC_BODY GetInterfaceFunctions
#include <RdsFunc.h>

// Главная функция DLL
int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                         unsigned long reason,
                         void* /*lpReserved*/)
{ if(reason==DLL_PROCESS_ATTACH) // Загрузка DLL
    { // Получение доступа к функциям
      if(!GetInterfaceFunctions())
        MessageBox(NULL,”Нет доступа к функциям”,”Ошибка”,MB_OK);
    }
  return 1;
}
//----------------------------------------
// ... здесь должны быть функции моделей ...
//----------------------------------------
```

В этом примере перед включением файла “RdsFunc.h” находится описание вида

```
#define RDS_SERV_FUNC_BODY имя_функции_пользователя
```

Наличие этого описание приведет к тому, что в месте включения файла (в приведенном примере – в следующей строке) будет вставлен полный набор переменных-указателей на все

сервисные функции РДС, после которого будет автоматически сформирована дополнительная функция с именем, указанным пользователем (в данном случае – GetInterfaceFunctions), которая получает указатели на функции и присваивает их этим переменным. Теперь, вместо того, чтобы вручную присваивать указатели на функции глобальным переменными, нужно просто вызвать эту функцию внутри DllEntryPoint и, при желании, проверить возвращаемое логическое (BOOL) значение. Возврат FALSE указывает на то, что не все указатели на сервисные функции удалось получить (чаще всего это говорит о том, что версия "rds.exe" более старая, чем версия описаний в "RdsFunc.h", и некоторые сервисные функции в ней еще не реализованы).

Если проект DLL состоит из нескольких модулей, заголовочный файл "RdsFunc.h" следует включить в те из них, которые используют сервисные функции РДС. При этом следует помнить, что описание константы #define RDS_SERV_FUNC_BODY может находиться ***только в одном*** модуле. При его наличии в нескольких модулях будет создано несколько комплектов глобальных переменных с одинаковыми именами и несколько одинаковых функций, что приведет к ошибкам при компоновке библиотеки.

По понятным причинам, описание #define RDS_SERV_FUNC_BODY и файл "RdsFunc.h" можно использовать только тогда, когда модель блока пишется на языке C или C++. В других языках необходимо получать указатель на каждую сервисную функцию вручную, при помощи функции Windows API GetProcAddress, как описано в первом примере.

Замечание для Borland C++ Builder. Если в моделях блоков будут использоваться функции и классы библиотеки VCL, и если модели будут открывать собственные окна, в главной функции DLL необходимо добавить два присваивания (выделены жирным):

```
int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                         unsigned long reason,
                         void* /*lpReserved*/)
{ switch(reason)
    { case DLL_PROCESS_ATTACH: // Загрузка DLL
        if(!GetInterfaceFunctions())
          MessageBox(NULL,"Нет доступа к функциям","Ошибка",MB_OK);
        else
          Application->Handle=rdsGetAppWindowHandle();
        break;
      case DLL_PROCESS_DETACH: // Выгрузка DLL
        Application->Handle=NULL;
        break;
    }
  return 1;
}
```

Здесь при загрузке библиотеки инициализируется глобальная переменная библиотеки VCL Application: ее свойству Handle присваивается дескриптор главного окна РДС, возвращаемый сервисной функцией rdsGetAppWindowHandle. Это необходимо для того, чтобы привязать все окна, открываемые в этой DLL, к приложению-владельцу, то есть к "rds.exe" (подробнее это описано в документации к Borland C++ Builder и библиотеке VCL). При выгрузке библиотеки (когда параметр reason принимает значение DLL_PROCESS_DETACH) необходимо очистить это свойство, присвоив ему NULL.

§2.3. Структура функции модели блока

Рассматривается структура любой функции модели блока на языке C/C++, объясняется смысл ее параметров. Описывается структура данных, которую РДС хранит для каждого блока.

Любая функция модели блока РДС имеет следующий вид:

```
extern "C" __declspec(dllexport) int RDSCALL имя_функции(
  int CallMode,                  // Режим вызова
  RDS_PBLOCKDATA BlockData,      // Данные блока
  LPVOID ExtParam)               // Дополнительные параметры
```

Вызывая эту функцию, РДС передает ей три параметра:

- `int CallMode` – режим вызова. В этом параметре передается одна из констант `RDS_BFM_*`, описанных в файле "RdsDef.h". Каждой константе соответствует определенное событие, на которое может среагировать блок: переключение режима, такт расчета, нажатие кнопки мыши и т.п.
- `RDS_PBLOCKDATA BlockData` – указатель на структуру данных блока. В этой структуре содержится имя блока, адрес начала дерева (области памяти) статических переменных, адрес личной области данных блока и т.п. (см. §1.4 и §1.5).
- `LPVOID ExtParam` – дополнительные параметры, зависящие от конкретного значения `CallMode`, то есть режима вызова блока. Чаще всего это указатель на какую-либо структуру из описанных в "RdsDef.h" – например, при щелчке на изображении блока в этом параметре передается указатель на структуру `RDS_MOUSEDATA` (см. §2.12.1), в которой находятся координаты курсора мыши, текущие размеры блока и т.п.

Каждое значение режима вызова (параметра `CallMode` функции модели) соответствует определенному событию, на которое, при необходимости, может отреагировать модель блока. При этом для каждого события через параметр `ExtParam` передается указатель на данные (как правило, на структуру), описывающие произошедшее событие. Общее число событий, на которые может реагировать модель, довольно велико, их полный список приведен в приложении А. Среди них – инициализация и очистка данных блока, выполнение одного такта в режиме расчета, загрузка и сохранение параметров блока, реакция на мышь и клавиатуру, и т.п. Возвращаемое функцией значение интерпретируется РДС по-разному, в зависимости от цели вызова модели блока, то есть значения параметра `CallMode`. В большинстве случаев, возврат целой константы `RDS_BFR_DONE`, описанной в "RdsDef.h" и равной нулю, говорит об успешном завершении функции.

Структура данных блока `RDS_BLOCKDATA` устроена следующим образом:

```
typedef struct
{ LPVOID VarData;     // Начало дерева переменных
  LPVOID BlockData;   // Указатель на личные данные
  RDS_BHANDLE Block;  // Идентификатор блока
  LPSTR BlockName;    // Имя блока
  RDS_BHANDLE Parent; // Идентификатор подсистемы
  DWORD Flags;        // Флаги
  int Width,Height;   // Размеры блока
  int Tag;            // Пользовательское поле
} RDS_BLOCKDATA;
typedef RDS_BLOCKDATA *RDS_PBLOCKDATA; // Указатель на структуру
```

- `LPVOID VarData` – указатель на начало дерева статических переменных (см. §1.5). Перед обращением к переменным модель должна убедиться, что их структура соответствует ее ожиданиям (для этого предусмотрен вызов модели с `CallMode` равным `RDS_BFM_VARCHECK`). Модель не должна изменять значение этого поля.
- `LPVOID BlockData` – указатель на личную область данных блока. Перед первым вызовом модели РДС присваивает этому полю значение `NULL`, после чего никогда к нему не обращается. Обычно блок, у которого есть личная область данных, отводит под нее память (например, оператором C++ `new`) при вызове модели с `CallMode` равным `RDS_BFM_INIT`, и присваивает указатель на отведенную область памяти полю `BlockData`. При этом при вызове модели с `CallMode` равным `RDS_BFM_CLEANUP` необходимо освободить отведенную память (например, оператором `delete`).

- RDS_BHANDLE Block – уникальный идентификатор данного блока. Такие идентификаторы используются во многих сервисных функциях для указания конкретного блока, с которым производится то или иное действие. Модель не должна изменять значение этого поля.
- LPSTR BlockName – строка с именем блока. Это поле указывает на строку (стандартную строку символов, завершенную кодом 0) имени блока в подсистеме. Эта строка находится во внутренней памяти РДС, поэтому модель не должна изменять значение этого поля или менять какие-либо символы в этой строке – для переименования блока существует специальная сервисная функция rdsRenameBlock.
- RDS_BHANDLE Parent – идентификатор родительской подсистемы блока. Для корневой подсистемы, у которой нет родительской, это поле равно NULL. Модель не должна изменять значение этого поля.
- DWORD Flags – битовые флаги, управляющие поведением блока. Некоторые примеры их использования будут приведены позже. Это поле может содержать любую комбинацию следующих флагов:
  - RDS_VARCHECKFAILED – структура переменных блока не соответствует требованиям модели, то есть модель, вызванная с параметром RDS_BFM_VARCHECK, сообщила об ошибке. Модель может только читать этот флаг, его установка игнорируется РДС.
  - RDS_NEEDSDLLREDRAW – изображение блока следует перерисовать при следующем обновлении окна (только для блоков, модели которых рисуют их самостоятельно). Этот флаг взводится РДС автоматически перед вызовом модели блока. Модель может сбросить его, если перерисовка не требуется, это позволяет избежать замедления работы РДС из-за излишне частого обновления окон.
  - RDS_MOUSECAPTURE – блок захватил мышь. Модель может взвести или сбросить этот флаг только при реакции на перемещение мыши или нажатие и отпускание ее кнопок. Если модель взведет этот флаг, информация обо всех манипуляциях мышью в окне, в котором находится блок, будет поступать только в этот блок, независимо от того, над каким блоком находится курсор. После сброса этого флага восстановится нормальный порядок работы.
  - RDS_NOWINREFRESH – перерисовка немодальных окон блока (если они есть) или окна подсистемы временно запрещена. Этот флаг используется для того, чтобы приостановить обновление окон на время какой-либо сложной операции, занимающей несколько тактов расчета. Например, при моделировании переходных процессов нежелательно обновлять окна между изменениями значения времени, когда часть блоков схемы еще не успела сработать. Обычно этот флаг устанавливается и сбрасывается сервисной функцией rdsEnableWindowRefresh, но модель может управлять им и самостоятельно.
  - RDS_WINREFRESHWAITING – необходимо обновить немодальные окна блока или окно подсистемы, как только обновление будет разрешено. Этот флаг работает в паре с флагом RDS_NOWINREFRESH: если обновление окон запрещено и поступает команда обновления (по таймеру или от сервисной функции rdsRefreshBlockWindows), этот флаг взводится автоматически. Как только обновление окон снова будет разрешено сервисной функцией rdsEnableWindowRefresh, РДС проверит флаг RDS_WINREFRESHWAITING и, если он установлен, даст повторную команду на обновление окон.
  - RDS_DISABLED – блок не реагирует на действия пользователя. Пользователь не может выделить, отредактировать или удалить этот блок. В режиме моделирования блок также не будет реагировать на мышь и клавиатуру. РДС никогда не устанавливает этот флаг, модель должна управлять им самостоятельно, если есть такая необходимость (по умолчанию он сброшен).

- RDS_CTRLCALC – перед началом каждого такта расчета модель этого блока должна быть вызвана в режиме RDS_BFM_PREMODEL, если для блока установлен запуск каждый такт или его первая однобайтовая переменная (“Start”, см. стр. 20) имеет значение 1. По умолчанию этот флаг сброшен, обычно его взводят модели управляющих блоков, которым необходимо выполнить какие-либо действия до того, как начнется очередной такт расчета.

- int Width,Height – ширина и высота изображения блока в точках экрана при масштабе 100%. Значения этих полей используются только тогда, когда модель самостоятельно рисует внешний вид блока в подсистеме. Если внешний вид блока определяется векторной картинкой или прямоугольником с текстом, эти поля не используются.
- int Tag – целое поле, которое разработчик модели может использовать по своему усмотрению, например, для хранения каких-либо меток или флагов. РДС его не инициализирует и не обрабатывает.

Далее будут рассмотрены примеры моделей, в которых реализованы реакции на различные события.

§2.4. Инициализация и очистка данных блока

Описывается событие инициализации RDS_BFM_INIT – самое первое событие в “жизни” модели блока, и событие очистки RDS_BFM_CLEANUP – самое последнее. Приводится пример модели, отводящей себе память под личные нужды при инициализации и освобождающей ее при очистке.

Самое первое событие, на которое реагирует блок – это событие инициализации RDS_BFM_INIT. Оно возникает в момент подключения модели к блоку, то есть при загрузке блока из файла в составе схемы, при вставке блока из буфера обмена, при указании новой модели в окне параметров блока и т.п. Как только конкретному блоку схемы ставится в соответствие конкретная функция модели, эта функция вызывается для данного блока с параметром RDS_BFM_INIT. Обычно этот вызов используется для создания личной области данных блока – области памяти, используемой моделью по своему усмотрению. В C личная область данных чаще всего представляет собой структуру, а в C++ – объект какого-либо класса. В личной области хранятся параметры блока, которые неудобно или невозможно хранить в статических переменных – строки, вспомогательные объекты, информация о динамических переменных, с которыми работает блок, и т.п. При инициализации модель должна отвести память под личную область данных (например, функцией malloc или оператором new) и присвоить указатель на созданную область полю BlockData структуры данных блока RDS_BLOCKDATA, указатель на которую передается в функцию модели через параметр BlockData при каждом вызове (см. §2.3).

Событие очистки RDS_BFM_CLEANUP – это самое последнее событие, на которое может среагировать блок. Оно возникает тогда, когда по какой-либо причине модель отключается от блока: при удалении блока пользователем, при очистке памяти в момент завершения РДС или при загрузке новой схемы, перед подключением к блоку новой модели и т.п. Если при реакции на RDS_BFM_INIT была создана личная область данных, при реакции на RDS_BFM_CLEANUP она должна быть удалена.

В качестве примера рассмотрим модель блока, работающую с личной областью, в которой будут храниться два параметра: целый IParam и вещественный DParam (пример не имеет практического применения, он просто иллюстрирует реакцию блока на события инициализации и очистки). В этом примере будет приведена и главная функция DLL, и модель блока. В дальнейшем главная функция DLL чаще всего будет опущена.

```
#include <windows.h>
#include <RdsDef.h>
```

```
// Подготовка описаний сервисных функций
#define RDS_SERV_FUNC_BODY GetInterfaceFunctions
#include <RdsFunc.h>

//========== Главная функция DLL ==========
int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                         unsigned long reason,
                         void* /*lpReserved*/)
{ if(reason==DLL_PROCESS_ATTACH) // Загрузка DLL
    { // Получение доступа к функциям
      if(!GetInterfaceFunctions())
        MessageBox(NULL,"Нет доступа к функциям","Ошибка",MB_OK);
    }
  return 1;
}
//========= Конец главной функции =========

//====== Класс личной области данных ======
class TTest1Data
{ public:
    int IParam;       // Целый параметр
    double DParam;    // Вещественный параметр
    TTest1Data(void)  // Конструктор класса
      { IParam=0; DParam=0.0;
        rdsMessageBox("Область создана","TTest1Data",MB_OK);
      };
    ~TTest1Data()     // Деструктор класса
      { rdsMessageBox("Область удалена","TTest1Data",MB_OK);};
};
//=========================================

//============= Модель блока ==============
extern "C" __declspec(dllexport)
  int RDSCALL Test1(int CallMode,
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{ TTest1Data *data;
  switch(CallMode)
    { case RDS_BFM_INIT:    // Инициализация
        BlockData->BlockData=new TTest1Data();
        break;
      case RDS_BFM_CLEANUP: // Очистка
        data=(TTest1Data*)(BlockData->BlockData);
        delete data;
        break;
    }
  return RDS_BFR_DONE;
}
//=========================================
```

Перед функцией модели блока находится описание класса `TTest1Data`, объект которого используется блоком в качестве личной области данных. В конструкторе и деструкторе класса (функциях, автоматически вызываемых при создании и уничтожении объекта соответственно) вызывается сервисная функция `rdsMessageBox`, выводящая сообщение о создании и удалении объекта. Функция модели `Test1` реагирует всего на два события: инициализации `RDS_BFM_INIT` и очистки `RDS_BFM_CLEANUP`. При вызове функции с параметром `RDS_BFM_INIT` создается объект `TTest1Data`, указатель на который

присваивается полю структуры данных блока `BlockData`. При вызове функции с параметром `RDS_BFM_CLEANUP` значение `BlockData->BlockData` приводится к типу “указатель на `TTest1Data`” (для ясности примера в функцию модели введена вспомогательная переменная `data`), после чего созданный при инициализации объект уничтожается.

Чтобы проверить работоспособность этого примера необходимо скомпилировать DLL с этой моделью, запустить РДС, создать новую схему и создать в ней новый простой блок (пункт “Создать | Новый блок” в контекстном меню или в меню редактирования). Затем необходимо открыть окно параметров этого блока (пункт “Параметры” в контекстном меню блока), выбрать в окне вкладку “DLL”, указать путь к скомпилированной DLL и имя экспортированной функции в ней (см. рис. 15). Именем экспортированной функции, в данном случае, будет “`Test1`”, если используется компилятор Borland C++. Для других

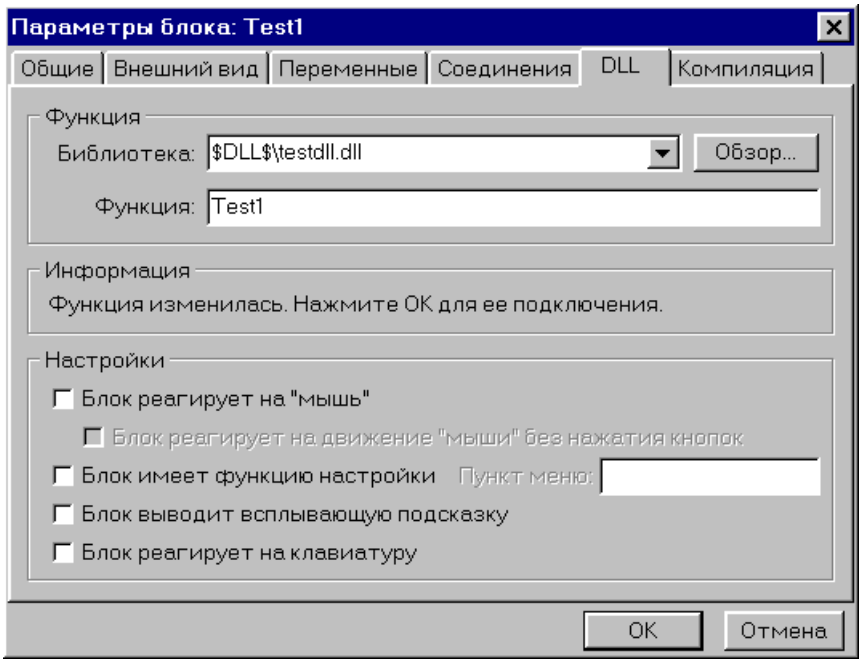


Рис. 15. Подключение модели к блоку – после нажатия “OK” к блоку будет подключена модель “Test1” из библиотеки “testdll.dll”, находящейся в стандартной папке DLL

компиляторов может потребоваться что-нибудь вроде “`Test1@12`” или “`_Test1@12`” – способ формирования экспортированного имени из имени функции обычно указывается в описании компилятора). При нажатии кнопки “OK” модель будет подключена к созданному блоку, при этом она будет вызвана с параметром `RDS_BFM_INIT` и должна вывести сообщение “Область создана”. Если после этого удалить созданный блок, перед удалением модель будет отключена от него, что приведет к вызову функции с параметром `RDS_BFM_CLEANUP` и выводу сообщения “Область удалена”. Это же сообщение будет выведено, если закрыть РДС не удаляя блок (в момент отключения модели от блока при очистке памяти перед завершением программы).

§2.5. Статические переменные блоков

Описывается работа со статическими переменными блоков (то есть переменными фиксированной структуры), которые могут служить им входами и выходами. Рассматриваются особенности работы с сигнальными переменными, матрицами и массивами, структурами, строками произвольной длины, а также с переменными, меняющими свой тип в процессе работы. Приводятся примеры моделей блоков, иллюстрирующие доступ к переменным каждого типа.

§2.5.1. Доступ к статическим переменным и работа в режиме расчета

Описывается способ чтения и записи значений статических переменных блока. Рассматривается событие проверки допустимости типа статических переменных RDS_BFM_VARCHECK и событие выполнения такта расчета RDS_BFM_MODEL. Приводится пример блока, выдающего в режиме расчета на выход разность двух вещественных входов. В примере предусмотрена проверка значений входов блока на специальное значение, символизирующее ошибку вещественных вычислений.

Для написания простейшей модели, работающей в режиме расчета, необходимо задать реакцию всего на два события: проверки типа статических переменных `RDS_BFM_VARCHECK` и выполнения одного такта расчета `RDS_BFM_MODEL`.

Обычно модели блоков, работающих в режиме расчета, производят какие-либо вычисления со значениями входов и внутренних переменных и присваивают результаты своим выходам, то есть работают, в основном, со статическими переменными блока. Как уже было написано в §1.5, перед обращением к статическим переменным модель должна проверить, удовлетворяет ли их структура необходимым требованиям. Если этого не сделать, при подключении модели к блоку с неподходящими для этой модели переменными могут возникнуть серьезные проблемы, например, ошибки общей защиты из-за обращения к не отведенной памяти. При этом такие ошибки могут проявится не сразу: если окажется, что блок памяти, к которому ошибочно обращается модель, вместо данных переменной содержит какие-либо важные для РДС данные, при записи в этот блок работа РДС может нарушиться самым непредсказуемым образом: например, начнут возникать ошибки в других моделях, ошибки при попытке сохранения схемы (с возможной потерей данных) и т.п. Поэтому к проверке типов переменных блока программист должен относиться очень ответственно.

Для проверки типов статических переменных служит вызов модели с параметром `RDS_BFM_VARCHECK`, при этом через параметр `ExtParam` передается строка типа переменных блока, которую модель должна проанализировать и возвратить один из следующих результатов:

- `RDS_BFR_DONE` – структура переменных блока удовлетворяет требованиям модели;
- `RDS_BFR_ERROR` – структура переменных блока не удовлетворяет требованиям модели;
- `RDS_BFR_BADVARSMSG` – структура переменных блока не удовлетворяет требованиям модели, РДС выведет предупреждающее сообщение для пользователя.

Если в ответ на событие `RDS_BFM_VARCHECK` модель вернет `RDS_BFR_ERROR` или `RDS_BFR_BADVARSMSG`, все вызовы модели, кроме последующих `RDS_BFM_VARCHECK` при новых изменениях структуры переменных, а также реакции на отключение модели `RDS_BFM_CLEANUP`, будут заблокированы. Разумеется, вызов `RDS_BFM_INIT` также не блокируется, поскольку он производится самым первым, еще до `RDS_BFM_VARCHECK`. Таким образом, введя в модель реакцию на это событие, в реакциях на все остальные события (кроме `RDS_BFM_INIT` и `RDS_BFM_CLEANUP`, в которых обращение к статическим переменным запрещено) можно обращаться к статическим переменным без каких-либо дополнительных проверок – все проверки уже сделаны в `RDS_BFM_VARCHECK`.

В качестве примера рассмотрим блок с двумя вещественными входами `x1` и `x2` и выходом `y`, на который выдается разность входов (`x1-x2`). С учетом двух обязательных

сигналов `Start` и `Ready` структура переменных блока будет выглядеть следующим образом:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	x1	double	8	Вход
10	x2	double	8	Вход
18	y	double	8	Выход

Структура переменных задается в окне параметров блока на вкладке “Переменные” (см. рис. 8, кнопка “Изменить”). Для редактирования переменных блока открывается отдельное окно, в котором также отображается смещение для каждой из них (вычисляется автоматически) и строка типа для всей структуры переменных (см. рис. 9).

Функция модели блока будет выглядеть следующим образом:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestSub(int CallMode,
                      RDS_PBLOCKDATA BlockData,
                      LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define x1      (*((double *)(pStart+2)))
#define x2      (*((double *)(pStart+10)))
#define y       (*((double *)(pStart+18)))
  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDDD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта расчета
      case RDS_BFM_MODEL:
        y=x1-x2;      // Вычисление значения выхода
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef y
#undef x2
#undef x1
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

В тексте этой модели используются макроопределения для доступа к статическим переменным, что позволяет сделать текст программы более читаемым. Сначала вводится определение `pStart` – это указатель на начало дерева переменных, приведенный к типу “указатель на `char`”:

```
#define pStart ((char *)(BlockData->VarData))
```

Тип `char` выбран из-за своего размера в один байт: прибавление целого числа к такому указателю смещает его на заданное число байтов. Теперь, если требуется обратиться к переменной со смещением N, указатель на нее можно получить, добавив N к `pStart`. Например, чтобы получить начальный адрес области памяти, занимаемой переменной `y`, смещение к которой равно восемнадцати байтам, можно написать `pStart+18`. Естественно, для доступа к этой переменной необходимо привести этот адрес к типу "указатель на `double`", то есть использовать конструкцию `(double*) (pStart+18)`. А для того, чтобы можно было непосредственно использовать имя переменной в выражениях (как в правой части, так и в левой), вводится макроопределение вида

```
#define y ( *((double*)(pStart+18)) )
```

В конце функции модели все макроопределения уничтожаются командами `#undef`, чтобы в следующей функции модели можно было, при желании, ввести новые определения для других переменных с теми же именами.

Вместо того, чтобы писать все эти определения вручную, можно получить их у РДС, нажав правую кнопку мыши на строке типа в левом нижнем углу окна редактирования переменных и выбрав в меню пункт "Копировать список (#define, по возможности сами данные)" (рис. 16). При этом в буфер обмена Windows будет помещен текст описаний для

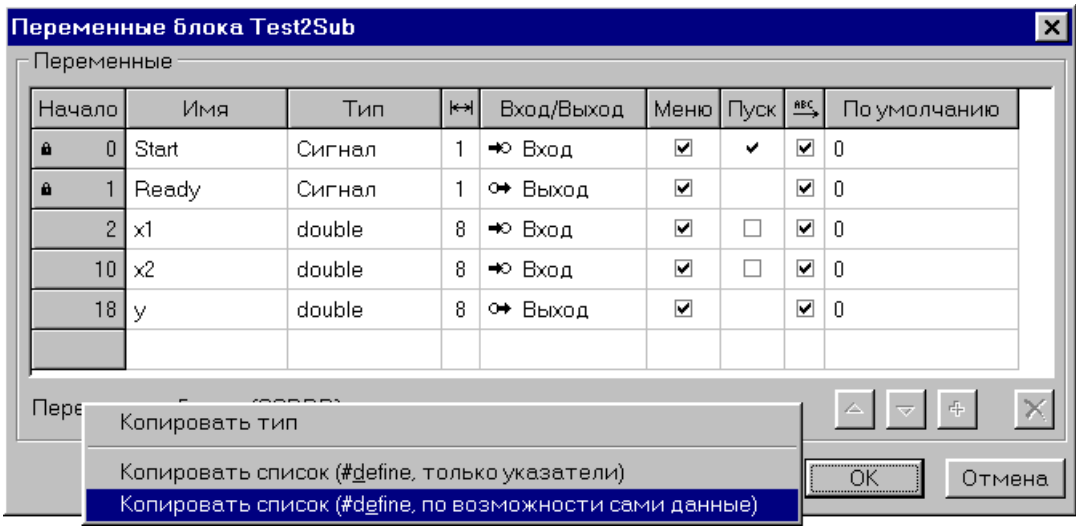


Рис. 16. Вызов меню для копирования в буфер обмена всех макросов переменных блока

всех статических переменных блока вместе с командами `#undef`.

Следует помнить, что макроопределения для доступа к статическим переменным можно использовать только в функциях, в которые передается указатель на структуру данных блока `BlockData`, поскольку все макросы ссылаются на этот указатель. Сначала вводится описание указателя на начало дерева переменных `pStart`, а затем все остальные переменные определяются относительно него. Когда препроцессор языка С, обрабатывая текст приведенной выше модели, встретит в нем слово "`y`", он сначала подставит вместо него текст

```
(*((double *)(pStart+18)))
```

а затем, раскрывая макроопределение для `pStart`, преобразует его в

```
(*((double *)((char *)(BlockData->VarData))+18)))
```

Пока переменная `y` используется внутри функции модели блока, развернутое макроопределение для нее будет компилироваться успешно – в функцию модели передается параметр `BlockData`, и конструкция `BlockData->VarData` будет понята компилятором. Если же переменную необходимо использовать и в других функциях, вызываемых из функции модели, в эти функции также необходимо передавать параметр

RDS_PBLOCKDATA BlockData, иначе развернутый макрос не будет скомпилирован. Кроме того, тексты всех этих функций должны размещаться после макроопределений для переменных, чтобы на момент их компиляции макросы уже существовали. Это несколько ухудшает читаемость текста программы: в функцию передается параметр, который, на первый взгляд, нигде не используется, при этом переменная y, фигурирующая в функции, не описана ни как параметр, ни как внутренняя. К тому же, вероятнее всего, значение y нельзя будет посмотреть в отладчике, поскольку это не настоящая переменная, а макрос. Тем не менее, при написании функций, интенсивно работающих с переменными блока, использование макросов позволяет передавать в функцию один параметр BlockData вместо множества указателей на все переменные блока, а ухудшение читаемости текста можно компенсировать подробными комментариями.

Вернемся к модели блока, вычисляющего разность своих входов. При вызове этой модели с параметром RDS_BFM_VARCHECK производится сравнение строки типа переменных блока, переданной через параметр ExtParam, со строкой, заложенной в программу. В данном случае строка, соответствующая структуре переменных, для которой разработана эта модель, выглядит как "{SSDDD}" – две сигнальных ("SS") и три вещественных ("DDD") переменных двойной точности. Сравнение производится стандартной библиотечной функцией strcmp (из-за нее может потребоваться включение в исходный текст команды #include <string.h>), перед вызовом которой ExtParam приводится к типу "char*". Если строки совпадают, эта функция возвращает значение 0. Функция модели в этом случае вернет константу RDS_BFR_DONE, информируя РДС о том, что структура статических переменных подходит для данной модели. Если же строки не совпадут, функция модели вернет константу RDS_BFR_BADVARSMSG, в результате чего РДС заблокирует все последующие вызовы этой модели (за исключением RDS_BFM_VARCHECK и RDS_BFM_CLEANUP) и выведет пользователю сообщение о несовместимости данной модели с данной структурой переменных. Если бы модель вернула константу RDS_BFR_ERROR, все вызовы также были бы заблокированы, но без какого-либо сообщения пользователю.

Строку типа, соответствующую структуре переменных блока, не обязательно записывать вручную. Так же, как и макросы переменных, ее можно получить у РДС, нажав правую кнопку мыши на строке типа, отображаемой в левом нижнем углу окна редактирования переменных, и выбрав в появившемся меню пункт "Копировать тип" (см. рис. 16). При этом в буфер обмена Windows будет помещена указанная строка, которую можно будет вставить в текст программы.

При вызове модели нашего блока с параметром RDS_BFM_MODEL значение выхода y вычисляется как разность входов x1 и x2, при этом для доступа к переменным используются описанные выше макросы. Чтобы значение y всегда соответствовало значениям входов, наша модель должна вызываться либо постоянно в каждом такте расчета, либо при изменении любого из входов. В первом случае можно задать для блока с этой моделью запуск каждый такт расчета (в окне параметров блока на вкладке "Общие"), во втором – задать для переменной Start начальное значение 1 и установить флаги "Пуск" для входов x1 и x2 (в окне редактирования переменных). Второй вариант предпочтительнее, поскольку он позволит избежать ненужных вызовов модели, если входы не изменялись: модель будет вызываться в режиме RDS_BFM_MODEL только тогда, когда сработают связи, подключенные к x1 или x2, то есть тогда, когда значение входов изменится.

Приведенная выше модель имеет один серьезный недостаток: в ней не проверяется допустимость выполнения операции вычитания. Если на одном из входов блока окажется значение, используемое математическими функциями для индикации ошибки или переполнения, попытка выполнить с ним какую-либо операцию приведет к возникновению исключения. Чтобы этого избежать, необходимо перед вычитанием сравнить значения входов с этим специальным значением-индикатором. Можно использовать константу

HUGE_VAL, описанную в стандартном заголовочном файле “math.h”, но лучше всего получить это значение у РДС при помощи сервисной функции rdsGetHugeDouble. Этот вариант более надежен, потому что все модели, использующие эту функцию, гарантированно получают одно и то же значение, не зависящее от версии РДС и версий библиотек, которые используются в DLL моделей.

Лучше всего вызывать функцию rdsGetHugeDouble один раз в главной функции DLL и записывать полученное значение в какую-нибудь глобальную переменную, которую смогут использовать все функции моделей этой DLL. В этом случае главная функция примет следующий вид (изменения выделены жирным):

```
// Глобальная переменная для значения ошибки
double DoubleErrorValue;
//========== Главная функция DLL ==========
int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                         unsigned long reason,
                         void* /*lpReserved*/)
{ if(reason==DLL_PROCESS_ATTACH) // Загрузка DLL
    { // Получение доступа к функциям
      if(!GetInterfaceFunctions())
        MessageBox(NULL,"Нет доступа к функциям","Ошибка",MB_OK);
      else
        rdsGetHugeDouble(&DoubleErrorValue);
    }
  return 1;
}
//=========================================
```

Функции rdsGetHugeDouble передается указатель на глобальную переменную DoubleErrorValue, в которую она заносит значение-индикатор ошибки. Теперь реакцию модели на выполнение одного такта моделирования можно изменить следующим образом:

```
// Выполнение такта моделирования
case RDS_BFM_MODEL:
  if(x1==DoubleErrorValue || x2==DoubleErrorValue)
    y=DoubleErrorValue;
  else
    y=x1-x2;
  break;
```

Если хотя бы на одном входе блока появляется значение, сигнализирующее об ошибке, выход тоже получает это значение.

§2.5.2. Особенности использования сигналов

Описываются особенности работы с переменными сигнального типа, то есть с переменными, используемыми для передачи информации о факте наступления какого-либо события. Использование сигналов позволяет организовать сложное логическое взаимодействие блоков. Приводится пример модели блока-счетчика, увеличивающего значение своего выхода на единицу при каждом поступлении сигнала на вход.

Во многих случаях модели блока бывает нужно не передать другим блокам по связям какое-либо конкретное значение, а уведомить их о том, что произошло какое-то событие, на которое эти блоки должны отреагировать согласно логике работы своих моделей. Допустим, например, что нам для каких-то целей нужно подсчитывать число нажатий кнопки пользователем. Мы можем для этого взять из библиотеки РДС стандартные блоки: кнопку, счетчик и индикатор, и соединить их, как показано на рис. 17. Если бы таких стандартных блоков в библиотеке не было, нам пришлось бы

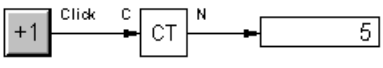


Рис. 17. Подсчет числа нажатий кнопки

создавать модели кнопки и счетчика самостоятельно. Рассмотрим возможные варианты решения этой задачи.

Передавать факт нажатия кнопки по связи при помощи обычных переменных, конечно, можно, но не очень удобно. Можно было бы, например, сделать выход кнопки логическим и присваивать ему единицу при нажатии кнопки и ноль при отпускании. В этом случае счетчик, который подключен к этой кнопке, должен был бы увеличивать на единицу свой выход только при изменении значения своего входа с ноля на единицу. Просто увеличивать выход при единичном значении входа было бы ошибкой: если пользователь продержит кнопку нажатой несколько тактов расчета, на протяжении всех этих тактов вход счетчика будет иметь единичное значение, и его выход будет каждый раз увеличиваться, хотя пользователь нажал на кнопку всего один раз. Таким образом, в этом случае мы подсчитали бы не число фактических нажатий кнопки, а число тактов, в течение которых кнопка была нажата. Значит, нам нужно отслеживать именно изменение входа с нуля на единицу (переход кнопки в нажатое состояние), а для этого в блок придется добавить дополнительную переменную, в которую нужно записывать значение входа в прошлом такте и сравнивать с ней его текущее значение – только так мы сможем обнаружить изменение входа.

Кроме усложнения модели блока из-за необходимости работы с дополнительной переменной, хранящей прошлое значение входа, этот метод передачи информации о событиях имеет еще один недостаток: блок сможет реагировать на события не чаще одного раза в два такта. Действительно, обнаружив событие мы должны присвоить выходу блока единицу, а сбросить его в ноль мы сможем только в следующем такте, иначе единица не будет передана по связи, и блок на другом ее конце на получит информацию об этом событии. Таким образом, передача информации о событии занимает два такта, в течение которых новое событие будет проигнорировано блоком. Для реакции на нажатие кнопки этот недостаток не имеет большого значения, поскольку такты расчета занимают очень небольшое время, и за один такт пользователь просто физически не успеет отпустить кнопку и снова ее нажать. Однако, во многих случаях игнорирование двух последовательных событий может помешать нормальной работе схемы.

Во избежание описанных выше проблем, в РДС введен сигнальный тип переменных, специально предназначенный для передачи по связям информации о событиях. Как и логические переменные, сигнальные могут принимать значения 0 и 1, но передаются от блока к блоку они совсем по-другому. Во-первых, по связи передается только единичное значение сигнального выхода, при его нулевом значении связь не срабатывает. Фактически, передается только передний фронт сигнала: при изменении значения выхода с 0 на 1 во все входы, соединенные с этим выходом, запишется значение 1. Во-вторых, после передачи единичного значения сигнальному выходу автоматически присваивается ноль. Таким образом, отпадает необходимость в дополнительном такте расчета для обнуления значения выхода: выход обнуляется автоматически в конце такта, и в следующем же такте блок будет готов передать информацию о новом событии. Кроме того, в блоках, к входам которых подключена сигнальная связь, нет необходимости создавать дополнительные переменные для отслеживания факта изменения входа. Вместо этого при обнаружении на сигнальном входе единицы им достаточно сбросить его в ноль: поскольку выход соединенного блока будет сброшен автоматически, а нулевое значение сигнального выхода по связи не передается, единица на входе появится только при следующем событии. Следует учитывать, что если модель блока с сигнальным входом не сбросит этот вход после реакции на событие, то в следующем такте расчета, обнаружив единицу на входе, она может ошибочно посчитать, что произошло еще одно событие. По этой причине при написании моделей крайне важно следить за сбросом сигнальных входов. Однако, в некоторых случаях, единицу, не исчезающую автоматически с сигнального входа после срабатывания связи, модель может использовать в своих целях. Например, если модель выполняет какие-то действия по таймеру

раз в несколько секунд, она может не беспокоиться о том, что пропустит какое-то событие: единица, появившаяся на ее входе в результате срабатывания связи, так на нем и останется до следующего вызова модели по таймеру. Конечно, в этом случае модель не сможет узнать, сколько именно событий произошло между ее вызовами, но, во многих случаях, это и не нужно. Если же число событий важно, всегда можно переписать модель так, чтобы она, как положено, сбрасывала сигнальный вход в каждом такте расчета и увеличивала какой-нибудь внутренний счетчик.

Описанные принципы передачи сигнальных переменных по связям можно проиллюстрировать условной временной диаграммой, приведенной на рис. 18. Пусть в схеме есть три блока: “Блок 1” с сигнальным выходом `y`, “Блок 2” с сигнальным входом `x1` и “Блок 3” с сигнальным входом `x2`, причем выход первого блока соединен с входами второго и третьего.

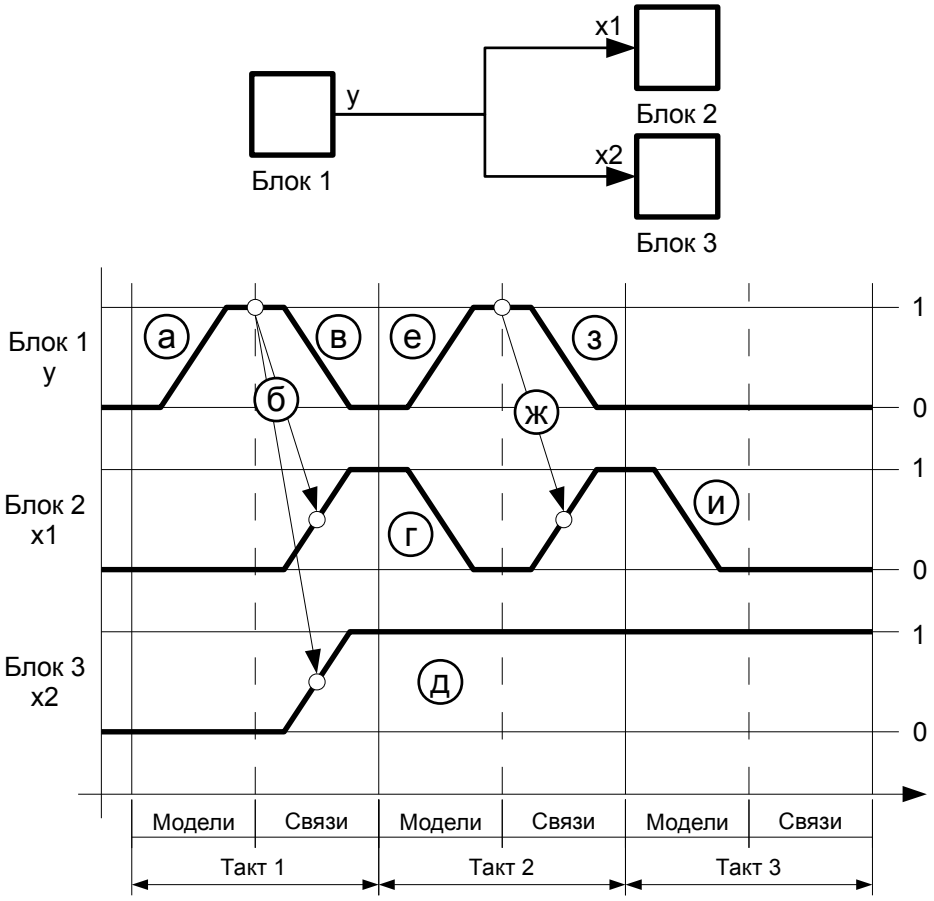


Рис. 18. Временная диаграмма передачи переменных сигнального типа на примере трех соединенных блоков

На рисунке изображены три такта расчета, каждый из которых разделен на две части: полутакт срабатывания моделей блоков и следующий за ним полутакт срабатывания связей. Пусть в первом такте модель первого блока присвоила сигнальному выходу `y` значение 1 (на рисунке этот момент обозначен буквой “а”), информируя соединенные блоки о каком-то событии. В конце этого такта, при срабатывании связей, эта единица будет передана на входы `x1` и `x2` второго и третьего блоков соответственно (“б”), а значение выхода `y` будет автоматически обнулено (“в”).

Во втором такте расчета модель второго блока обнаружит единицу на входе x1, выполнит связанные с данным событием действия, определяемые логикой модели, и обнулит вход, подготавливая его к приему информации о следующем событии (“г”). Модель третьего блока в это же время по какой-то причине (по своей внутренней логике или из-за ошибки программиста) не обнулит вход x2, и в нем так и останется единичное значение (“д”). В это же время модель первого блока снова присвоит выходу y единицу, чтобы сообщить соединенным блокам о втором событии (“е”). Когда в конце второго такта расчета начнут срабатывать связи, эта единица снова будет передана на входы первого и второго блоков (“ж”), однако, поскольку значение x2 не было сброшено моделью второго блока, там оставалась единица, полученная им в прошлом такте расчета – таким образом, значение x2 не изменится. После передачи выходу первого блока снова будет автоматически присвоено нулевое значение (“з”).

В начале третьего такта расчета выход y имеет нулевое значение, а входы x1 и x2 – единичные. В этом такте первый блок не будет информировать остальные о новом событии, его выход y так и останется нулевым и не будет передаваться по связям. Модель второго блока, отреагировав на событие, информация о котором передана ей по связи в конце второго такта, снова сбросит свой вход x1 (“и”) и будет ждать нового события. Значение же x2, не сброшенное моделью третьего блока, так и останется единичным, поскольку нулевое значение сигнального выхода y, соединенного с ним, в конце третьего такта не будет передано по связи.

Из приведенной диаграммы видно, что модель первого блока передала по связям информацию о двух событиях в двух последовательных тактах (такт 1 и такт 2). Модель второго блока узнала об обоих этих событиях, несмотря на то, что они возникли в соседних тактах, и отреагировала на них с запаздыванием в один такт (любая передача данных по связи создает такое запаздывание, потому что модель блока, на вход которого переданы данные, запустится только в начале следующего такта расчета). Модель же третьего блока не сбросила вовремя свой сигнальный вход и поэтому не сможет узнать, сколько событий произошло с момента выполнения первого такта расчета.

При работе с сигналами нужно также иметь в виду, что в РДС к одному и тому же входу блока может быть одновременно присоединено несколько связей. Если несколько связей подключено к сигнальному входу, значение этого входа станет единицей при срабатывании хотя бы одной из них. Модель не может узнать, сколько связей одновременно сработало. Значение входа либо останется нулем, если не сработала ни одна, либо станет равным единице, если сработала одна или несколько связей. Если модели необходимо знать число сработавших связей, их нужно подключать к разным входам блока и анализировать независимо. Можно, например, сделать вход массивом сигналов и подключать связи к его элементам.

Обязательные для каждого простого блока переменные Start и Ready тоже являются сигналами. При задании набора переменных блока (см. рис. 9) можно изменить только их имена и начальные значения, при этом первая переменная блока всегда останется сигнальным входом, а вторая – сигнальным выходом. Эти сигналы управляют работой модели и ее выходных связей в режиме расчета: единица на входе Start запускает модель блока, единица на выходе Ready разрешает передачу данных выходов. Модель может работать с этими переменными как с обычными сигналами, однако следует помнить, что РДС автоматически сбрасывает сигнал Start и взводит сигнал Ready перед запуском модели для выполнения такта моделирования, то есть в режиме RDS_BFM_MODEL. Если модели необходимо запуститься в следующем такте независимо от срабатывания связей, присоединенных к ее входам, и состояния флага блока “работать каждый такт” (рис. 5), она может самостоятельно взвести сигнал Start. Модель также может запретить передачу данных своих выходов по связям, сбросив сигнал Ready.

Для примера рассмотрим модель блока-счетчика, который по сигналу изменяет значение выхода от 0 до 9, после чего выдает на выход сигнал переноса и начинает счет с нуля. Блок будет иметь сигнальные входы `Clk` и `Reset`, целый выход `Count` и сигнальный выход `Carry`. Блок будет работать следующим образом: по сигналу, приходящему на вход `Clk`, блок будет увеличивать значение выхода `Count` на 1 до тех пор, пока оно не достигнет девяти. При следующем срабатывании `Clk` модель обнулит выход `Count` и выдаст сигнал переноса на выход `Carry`. По сигналу `Reset` модель должна обнулить оба выхода. На рис. 19 изображены два таких счетчика, соединенные последовательно.

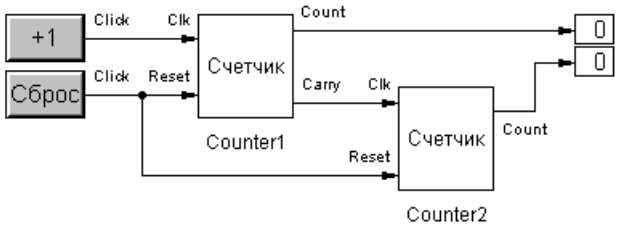


Рис. 19. Два последовательно соединенных счетчика

К входу `Clk` первого счетчика подключена кнопка “+1”, позволяющая увеличивать значение его выхода на единицу. Его выход `Carry` подключен к входу `Clk` второго счетчика – в результате этого, когда первый счетчик досчитает до десяти и его выход обнулится, второй увеличит свой выход на единицу. Кнопка “Сброс” подключена к входам `Reset` обоих счетчиков, ее нажатие сбрасывает их одновременно. При таком соединении выходы первого и второго счетчиков можно считать двумя разрядами двузначного десятичного числа, таким образом, вместе они могут досчитать до ста.

Структура переменных такого блока-счетчика будет выглядеть следующим образом:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	Reset	Сигнал	1	Вход
3	Clk	Сигнал	1	Вход
4	Count	int	4	Выход
8	Carry	Сигнал	1	Выход

Модель блока будет такой:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestCounter(int CallMode,
  RDS_PBLOCKDATA BlockData,
  LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define Reset   (*((char *)(pStart+2)))
#define Clk     (*((char *)(pStart+3)))
#define Count   (*((int *)(pStart+4)))
#define Carry   (*((char *)(pStart+8)))
  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
```

```
        if(strcmp((char*)ExtParam,"{SSSSIS}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        if(Reset)     // Поступил сигнал Reset
          { Reset=0;  // Сброс этого сигнала
            Clk=0;    // Сброс сигнала Clk
            Count=0;  // Обнуление счетчика
            Carry=0;  // Сброс сигнала переноса
          }
        else if(Clk)  // Поступил сигнал Clk
          { Clk=0;    // Сброс этого сигнала
            Count++;  // Увеличение счетчика
            if(Count>=10)  // Досчитали до 10
              { Count=0;   // Обнуление счетчика
                Carry=1;   // Выдача сигнала переноса
              }
          }
        break;
    }

  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Carry
#undef Count
#undef Clk
#undef Reset
#undef Ready
#undef Start
#undef pStart
}
//=========================================
```

При вызове этой модели с параметром `RDS_BFM_VARCHECK` производится сравнение переданной строки типа переменных блока со строкой “{SSSSIS}”. Все переменные блока – сигналы (“S”), кроме целой переменной `Count`, которой соответствует символ “I”. Сравнив строки, модель возвращает РДС соответствующую константу – `RDS_BFR_DONE`, если структура переменных блока правильная, и `RDS_BFR_BADVARSMSG` в противном случае.

При вызове модели с параметром `RDS_BFM_MODEL` для выполнения такта расчета сначала проверяется, не поступил ли на вход блока сигнал сброса `Reset`. Этот сигнал считается более важным, поэтому его проверка производится до проверки сигнала `Clk`. Если в одном такте на вход блока придут оба сигнала, `Reset` будет иметь приоритет. Если `Reset` имеет значение 1, он обнуляется, чтобы модель могла среагировать на него еще раз (иначе единица останется на этом входе навсегда). Сигнал `Clk` также сбрасывается – если он поступил одновременно с `Reset`, его нужно игнорировать. Затем обнуляется счетчик `Count` и сбрасывается сигнал переноса `Carry`. Сброс сигнала переноса может показаться излишним, поскольку выходные сигналы автоматически сбрасываются при передаче по связям. Однако, если к выходу `Carry` не было подключено ни одной связи, его значение не будет никуда передаваться, и, следовательно, сбрасываться он также не будет. Если пользователь остановит расчет и соединит сигнал `Carry` с другими блоками, сохранившаяся в нем единица будет немедленно передана на их входы при повторном запуске расчета, даже если на вход данного блока поступит сигнал `Reset`. Поэтому лучше подстраховаться и принудительно обнулить `Carry` при поступлении сигнала сброса.

Если сигнал сброса не поступал (Reset имеет значение 0), модель проверяет значение сигнала Clk. Если он имеет ненулевое значение, он обнуляется, чтобы модель могла среагировать на него в дальнейшем, и значение счетчика Count увеличивается на 1. Если значение Count достигло 10, оно сбрасывается в 0 и взводится сигнал переноса Carry. Сбрасывать сигнал переноса не нужно – он либо сбросится автоматически, если его значение будет передано по связям, либо останется взведенным, запомнив факт переполнения счетчика, до тех пор, пока к нему не будет подключена какая-либо связь.

Для блока с этой моделью необходимо либо включить запуск каждый такт расчета, либо установить флаги “Пуск” для входов Reset и Clk. Если этого не сделать, модель будет запускаться только при срабатывании связи, соединенной со входом Start, что для данного блока не имеет никакого смысла. У блока есть два собственных управляющих сигнала, поэтому его модель должна либо запускаться при их срабатывании (если для них установлены флаги “Пуск”), либо проверять их значения в каждом такте расчета.

Следует помнить, что в РДС все события, произошедшие в одном такте расчета, считаются произошедшими одновременно. Если в одном такте на оба сигнальных входа поступит по единице, модель должна считать их пришедшими одновременно и действовать исходя из приоритета этих сигналов. В описанной модели приоритетным считается сигнал Reset, и поступивший одновременно с ним Clk будет проигнорирован. Если при одновременном поступлении сигналов ни один из них игнорировать нельзя, можно сначала сбросить счетчик, а потом, реагируя на Clk, увеличить его на 1. В этом случае реакцию модели на такт расчета необходимо изменить следующим образом:

```
// Выполнение такта моделирования
case RDS_BFM_MODEL:
  if(Reset)     // Поступил сигнал Reset
    { Reset=0;  // Сброс этого сигнала
      Count=0;  // Обнуление счетчика
      Carry=0;  // Сброс сигнала переноса
      // Сигнал Clk теперь не сбрасывается
    }
  // "else if(Clk)" заменено на "if(Clk)"
  if(Clk)       // Поступил сигнал Clk
    { Clk=0;    // Сброс этого сигнала
      Count++;  // Увеличение счетчика
      if(Count>=10)  // Досчитали до 10
        { Count=0;   // Обнуление счетчика
          Carry=1;   // Выдача сигнала переноса
        }
    }
  break;
```

Как и в предыдущем варианте, модель сначала проверяет значение сигнала Reset и, если оно равно единице, обнуляет счетчик. Однако, если раньше модель в этом случае сбрасывала Clk и завершалась, теперь она проверяет значение Clk и увеличивает счетчик независимо от состояния Reset. Теперь при одновременном поступлении сигналов значение Count будет равно 1, а не 0, поскольку сначала оно будет сброшено из-за сигнала Reset, а потом увеличено на единицу из-за сигнала Clk.

Для проверки работы созданной модели можно собрать схему с рис. 19. Если запустить расчет и нажимать на кнопку “+1”, каждое нажатие должно приводить к увеличению числа на выходе первого счетчика – оно отображается на верхнем индикаторе. Когда счетчик досчитает до девяти, следующее нажатие на кнопку “+1” должно привести к обнулению выхода первого счетчика и увеличению выхода второго (нижний индикатор). Нажатие кнопки “Сброс” должно приводить к обнулению выходов обоих счетчиков.

§2.5.3. Доступ к матрицам и массивам

Описываются особенности работы с матрицами и массивами: их размещение в памяти, макросы для доступа к ним, сервисные функции для изменения размера матриц. Приводится пример блока, умножающего вход-матрицу на вход-число и выдающего результат на выход.

Работать с массивами и матрицами несколько сложнее, чем с простыми переменными. Простые переменные имеют фиксированный размер, в то время как размер массивов и матриц может изменяться в процессе расчета. По этой причине в дереве переменных блока хранятся не сами ячейки матрицы, а указатель на динамически отводимый блок данных переменного размера, в котором они находятся (рис. 20).

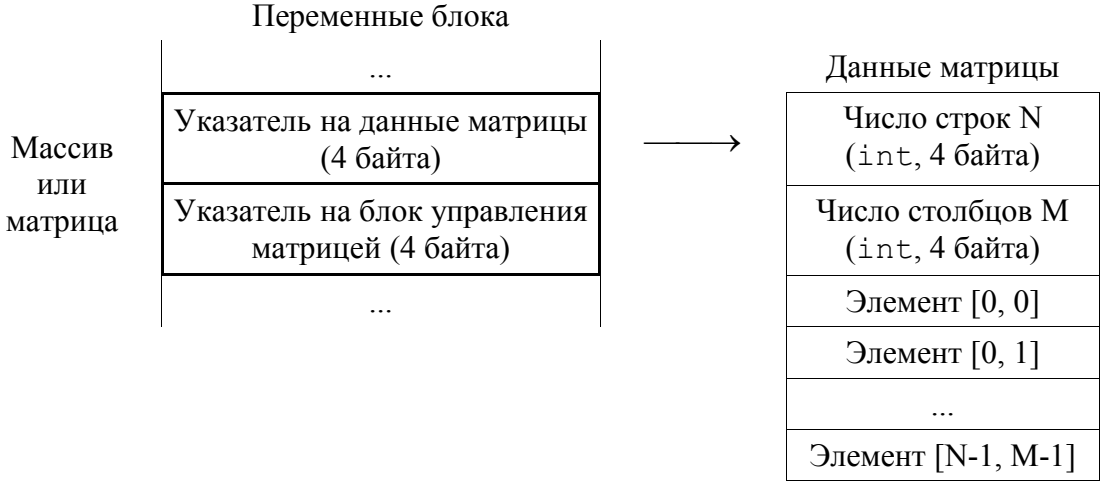


Рис. 20. Размещение в памяти данных матрицы

Данные матрицы занимают в дереве переменных блока 8 байтов. Первые 4 байта содержат указатель на область данных матрицы, в которой хранится ее текущий размер и элементы. Вторые 4 байта представляют собой указатель на блок управления матрицей. Это служебные данные, модель блока не должна изменять эти четыре байта. Область данных матрицы отводится динамически только в том случае, если матрица не пустая, то есть ее размер не 0x0 (для пустых матриц поле указателя на данные содержит `NULL`). Эта область начинается с двух четырехбайтовых целых чисел, указывающих текущий размер матрицы, за которыми последовательно, строка за строкой, записываются ее элементы. Размер элемента зависит от типа переменной. Например, если переменная определена как “матрица double”, каждый ее элемент будет представлять собой вещественное число двойной точности (`double`) и занимать 8 байтов.

Для того, чтобы получить значение заданного элемента матрицы, функция модели сначала должна проверить матрицу на пустоту. Если указатель на данные матрицы равен `NULL`, то матрица пуста, и дальнейшие вычисления не имеют смысла. Если же указатель ссылается на какую-то область памяти, необходимо считать из нее размеры матрицы, и, при необходимости, проверить, существует ли в матрице элемент с указанными индексами. После этого смещение в байтах к элементу матрицы, находящемуся в строке `r` и столбце `c`, можно вычислить по следующей формуле:

$$(\texttt{r} \times \textit{число\_столбцов} + \texttt{c}) \times \textit{размер\_элемента} + 8$$

Поскольку элементы матрицы хранятся в памяти построчно, для того, чтобы получить номер первого элемента строки `r`, необходимо умножить `r` на число элементов в строке, то есть на число столбцов. Затем к получившемуся значению добавляется номер элемента в строке `c`, в результате чего получается порядковый номер элемента `[r,c]` в массиве элементов матрицы. Остается умножить этот номер на размер элемента, чтобы получить смещение в байтах, и

добавить 8, чтобы пропустить первые 8 байтов, занимаемые данными о размере матрицы (4 + 4).

Для изменения размера или очистки матрицы следует использовать сервисную функцию `rdsResizeVarArray`. Она позволяет изменять размеры матриц без потери текущих данных: при указании размера, большего текущего, новые строки и столбцы дописываются в конец матрицы и заполняются значением по умолчанию, указанным для матрицы в редакторе переменных (см. рис. 9), при уменьшении размера матрица обрезается справа и снизу. Если же вызвать эту функцию, указав размер 0x0, матрица будет очищена.

Массив в РДС – это матрица с числом строк, равным 1, поэтому данные массивов хранятся в памяти точно так же, как и данные матриц.

Для работы с массивами и матрицами модель блока может использовать два способа: один из них более простой, другой – более быстрый. В этом параграфе будут рассмотрены оба.

Чтобы упростить написание функции модели можно использовать сервисную функцию `rdsGetVarArrayAccessData`, которая записывает всю информацию о матрице в специальную структуру `RDS_ARRAYACCESSDATA`. Эта структура имеет следующие поля:

- `BOOL Exists` – принимает значение `TRUE`, если матрица содержит элементы, и `FALSE`, если она пуста;
- `int Rows` – число строк в матрице;
- `int Cols` – число столбцов в матрице;
- `int ItemSize` – размер элемента матрицы в байтах;
- `LPVOID Data` – указатель на первый элемент матрицы.

Вместе с этой структурой можно использовать макрос `RDS_ARRAYITEM`, с помощью которого можно обращаться к элементу по двум индексам. Например, чтобы обратиться к элементу `[r,c]` переменной типа “матрица double”, параметры которой считаны в структуру с именем `str`, можно написать `RDS_ARRAYITEM(double,&str,r,c)`.

В качестве примера рассмотрим блок, который умножает каждый элемент входной матрицы вещественных чисел `X` на константу `k` и выдает результат на выход `Y`. С учетом обязательных сигналов `Start` и `Ready` (см. стр. 20) структура переменных блока будет выглядеть следующим образом:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	X	Матрица double	8	Вход
10	k	double	8	Вход
18	Y	Матрица double	8	Выход

Функция модели блока будет выглядеть так:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestMatr(int CallMode,
                       RDS_PBLOCKDATA BlockData,
                       LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define pX      ((void **)(pStart+2))
#define k       (*((double *)(pStart+10)))
```

```
#define pY      ((void **)(pStart+18))
  // Структуры с информацией о матрицах
  RDS_ARRAYACCESSDATA XD,YD;
  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSMDDMD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        // Считать информацию о матрице X в структуру XD
        rdsGetVarArrayAccessData(pX,&XD);
        if(XD.Exists) // Матрица X существует
          { // Задать размер Y и считать информацию о ней в YD
            rdsResizeVarArray(pY,XD.Rows,XD.Cols,FALSE,&YD);
            // Присвоить значения элементам Y
            for(int r=0;r<XD.Rows;r++)
              for(int c=0;c<XD.Cols;c++)
                RDS_ARRAYITEM(double,&YD,r,c)=
                  k*RDS_ARRAYITEM(double,&XD,r,c);
          }
        else // Матрица X не существует
          // Очистить матрицу Y
          rdsResizeVarArray(pY,0,0,FALSE,NULL);
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef pY
#undef k
#undef pX
#undef Ready
#undef Start
#undef pStart
}
//=========================================
```

Макроопределения для этой модели отличаются от предыдущих примеров. Для матриц `X` и `Y` вместо определений для доступа к самим переменным вводятся определения для указателей `pX` и `pY`. В языке С нет стандартного типа, описывающего конструкцию, аналогичную матрице РДС, поэтому создать макроопределение для самой переменной-матрицы невозможно (при желании, можно создать класс в С++, который будет описывать матрицу РДС и облегчать обращение к ее элементам при помощи своих функций-членов). Сервисные функции РДС, обслуживающие матрицы и массивы, работают именно с указателями, поэтому такое описание вполне оправдано. В модели также используются локальные переменные-структуры типа `RDS_ARRAYACCESSDATA`, в которые будет записываться информация о матрицах.

При вызове этой модели с параметром `RDS_BFM_VARCHECK` производится сравнение переданной строки типа переменных блока со строкой “{SSMDDMD}”. Первые две буквы “S” соответствуют двум обязательным сигналам, “MD” – первой матрице double (`X`), “D” – вещественной переменной `k`, и завершающие буквы “MD” – матрице `Y`. Сравнив эти строки, модель возвращает РДС соответствующую константу: `RDS_BFR_DONE`, если строки совпали (переменные имеют правильный тип), и `RDS_BFR_BADVARSMSG`, если они отличаются (пользователю будет выведено сообщение о недопустимой структуре переменных блока).

При вызове модели с параметром RDS_BFM_MODEL сначала вызывается сервисная функция rdsGetVarArrayAccessData, записывающая информацию о матрице X в локальную структуру XD. Если матрица X существует (XD.Exists имеет значение TRUE), формируется выходная матрица Y. Для этого сначала размер Y делается равным размеру X при помощи вызова сервисной функции rdsResizeVarArray. Кроме указателя на изменяемую матрицу (pY) и нового числа строк и столбцов, этой функции передается значение FALSE, указывающее на то, что старые данные матрицы Y можно не сохранять, и указатель на локальную структуру YD, в которую будет записана информация об измененной матрице Y. В данном примере функция rdsResizeVarArray будет вызываться даже тогда, когда размеры матриц Y и X совпадают. Это не приведет к дополнительным задержкам, поскольку эта функция изменяет размер матрицы только тогда, когда это необходимо. Можно было бы вызывать ее только при несовпадении размеров, но это потребовало бы дополнительного вызова rdsGetVarArrayAccessData для выяснения размера Y и не дало бы выигрыша в быстродействии.

После того, как размер матрицы Y установлен, каждому ее элементу присваивается значение произведения переменной k и соответствующего элемента матрицы X. Для перебора элементов матриц служат два цикла, вложенные один в другой: внешний цикл по переменной r (номер строки), изменяющейся от 0 до XD.Rows-1, и внутренний цикл по переменной c (номер столбца), изменяющейся от 0 до XD.Cols-1. Для обращения к элементам матриц служит макрос RDS_ARRAYITEM, которое используется и в левой, и в правой части выражения. В этом макросе вычисляется смещение к заданному элементу матрицы с использованием данных вспомогательной структуры RDS_ARRAYACCESSDATA. Например, текст

```
RDS_ARRAYITEM(double,&XD,r,c)
```

эквивалентен выражению

```
* (double*) (  ((char*)XD.Data) + (r*XD.Cols + c) * XD.ItemSize )
```

К указателю на первый элемент матрицы XD.Data, приведенному к указателю на однобайтовый тип char, добавляется вычисленное смещение к элементу [r,c] в байтах, после чего указатель приводится к типу double*, чтобы можно было работать с вещественными переменными.

Все указанные действия выполняются только тогда, когда матрица X существует, то есть имеет ненулевые размеры. Если же она не существует (XD.Exists имеет значение FALSE), матрица Y очищается вызовом rdsResizeVarArray с указанием нулевого числа строк и столбцов.

При создании блоков, работающих с массивами и матрицами в режиме расчета, следует помнить, что операции с матрицами могут занимать достаточно продолжительное время, особенно при больших размерах матриц. Поэтому для таких блоков крайне нежелательно устанавливать флаг срабатывания в каждом такте расчета. Вместо этого следует вызывать модель таких блоков только при срабатывании входных связей. Для описанного блока, например, следует на вкладке “Общие” окна параметров (рис. 5) включить запуск по сигналу, после чего в окне редактирования переменных (рис. 9) задать для переменной Start начальное значение 1 и установить флаг “Пуск” для входов X и k. Таким образом, модель блока будет вызвана при первом запуске расчета для вычисления начального значения выхода (сигнал запуска Start в этот момент будет иметь значение 1, после чего автоматически сбросится) и при каждом поступлении на входы X и k новых значений. Во всех остальных случаях она вызываться не будет, не тратя тем самым процессорное время на повторные вычисления значения выхода при неизменных входах.

Использование структур RDS_ARRAYACCESSDATA делает текст программы более читаемым, но это не самый быстрый способ работы с матрицами. Хотя вызов rdsGetVarArrayAccessData выполняется достаточно быстро, всю информацию о

матрице можно получить и без него. Изменим приведенную выше модель блока, убрав из нее локальные переменные XD и YD. Реакция модели на выполнение одного такта моделирования теперь будет выглядеть следующим образом:

```
// Выполнение такта моделирования
case RDS_BFM_MODEL:
  if(RDS_ARRAYEXISTS(pX)) // Матрица X существует
    { // Вспомогательные переменные
      int xr,xc;
      double *ydata,*xdata;
      // Получить размеры матрицы X
      xr=RDS_ARRAYROWS(pX);
      xc=RDS_ARRAYCOLS(pX);
      // Задать размер Y равным размеру X
      rdsResizeVarArray(pY,xr,xc,FALSE,NULL);
      // Получить указатель на первый элемент X
      xdata=(double*)RDS_ARRAYDATA(pX);
      // Получить указатель на первый элемент Y
      ydata=(double*)RDS_ARRAYDATA(pY);
      // Присвоить значения элементам Y
      for(int r=0;r<xr;r++)
        for(int c=0;c<xc;c++)
          ydata[r*xc+c]=k*xdata[r*xc+c];
    }
  else // Матрица X пуста
    rdsResizeVarArray(pY,0,0,FALSE,NULL);
  break;
```

Как и в предыдущем варианте примера, сначала необходимо проверить, существует ли матрица X. Для этого используется макрос RDS_ARRAYEXISTS, который возвращает FALSE, если указатель на область данных матрицы равен NULL, то есть если в матрице нет элементов (поскольку в данном примере в определении pX указатель уже приведен к типу void**, вместо RDS_ARRAYEXISTS(pX) можно было просто написать *pX!=NULL). Далее, если матрица существует, число ее строк присваивается вспомогательной переменной xr при помощи макроса RDS_ARRAYROWS, а число столбцов – переменной xc при помощи RDS_ARRAYCOLS. Оба этих макроса нельзя использовать для пустых матриц, поскольку они считывают размеры из области данных, которая у пустых матриц отсутствует. Например, текст

```
RDS_ARRAYROWS(pX)
```

преобразуется в

```
*( *( (int**)pX ) )
```

В этом выражении указатель pX приводится к типу “указатель на указатель на int”. Данные, на которые ссылается указатель pX, в свою очередь также являются указателем, указывающим на область данных матрицы, первые 4 байта которой представляют собой целое число (int) – число строк матрицы (см. рис. 20). Если бы матрица была пуста, выполнение операции *((int**)pX) дало бы значение NULL, и попытка получить данные по этому указателю привела бы к возникновению ошибки.

После того, как размеры матрицы X считаны, размеры Y делаются такими же при помощи уже описанного вызова rdsResizeVarArray, только в данном случае вместо указателя на структуру RDS_ARRAYACCESSDATA функции передается значение NULL – вспомогательные структуры теперь не используются. Далее указатели на первый элемент матриц X и Y присваиваются локальным переменным xdata и ydata при помощи макроса RDS_ARRAYDATA. Поскольку элементы обеих матриц – вещественные числа двойной точности (double), оба указателя приводятся к типу double*, что позволяет использовать адресную арифметику и обращаться к элементам матриц как к элементам одномерных

массивов xdata и ydata. В результате всех этих присвоений вспомогательные переменные будут ссылаться на матрицу X так, как показано на рис. 21.

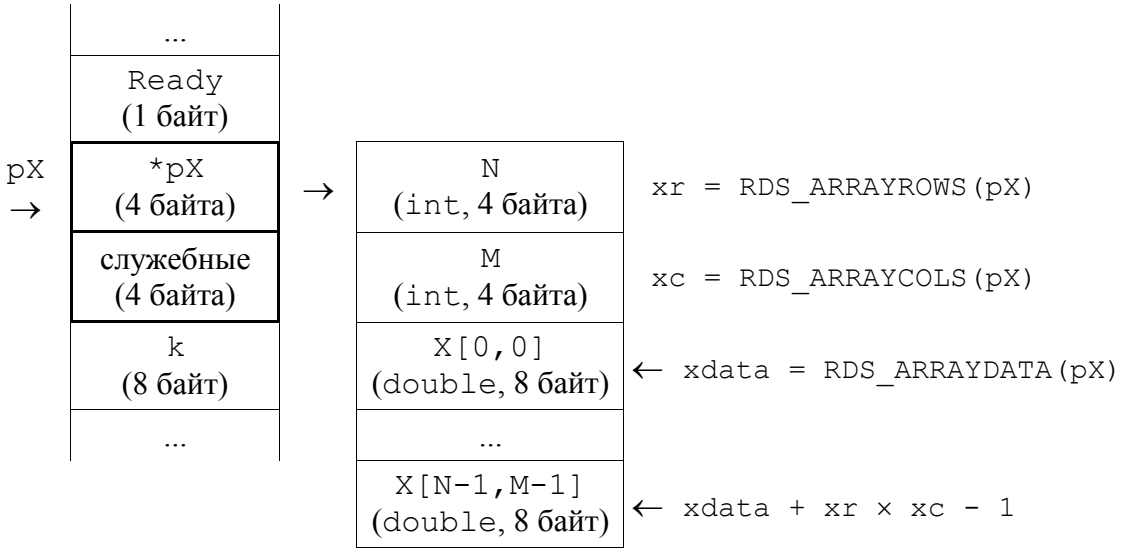


Рис. 21. Матрица X и вспомогательные переменные модели

Переменная ydata ссылается на первый элемент матрицы Y аналогичным образом. Вспомогательные переменные для размеров Y не вводятся, т.к. после вызова rdsResizeVarArray они должны быть равны xr и xc.

Далее, как и в первом варианте модели, в двух циклах производится вычисление элементов матрицы Y. Для получения индекса элемента в одномерных массивах xdata и ydata, соответствующего элементу матриц [r,c], номер строки r умножается на число элементов в строке xc (что дает индекс первого элемента строки r), после чего к нему добавляется номер столбца c.

Можно заметить, что в обоих вариантах этого примера все элементы матрицы X обрабатываются одинаково – каждый из них умножается на одно и то же число k независимо от номера элемента. Поскольку все элементы матриц хранятся в памяти последовательно, в данном случае можно упростить модель, заменив два цикла по номеру строки и столбца одним, перебирающим все элементы общего массива элементов размером xr × xc:

```
// Присвоить значения элементам Y
for(int i=0;i<xr*xc;i++)
  ydata[i]=k*xdata[i];
```

Если бы номера строки и столбца были важны для вычисления элементов матрицы Y (например, если после умножения на k необходимо было бы транспонировать результат), замена двух циклов на один была бы невозможна.

Для проверки этой модели можно подключить к созданному блоку три стандартных блока: поле ввода к входу k, редактор матриц к входу X и блок отображения матриц к выходу Y (рис. 22). При запущенном расчете любые изменения, внесенные в окно редактора матриц или в поле ввода, должны немедленно отражаться на выходной матрице.

Чтобы не загромождать этот пример, в тексте модели мы не проверяем логическое значение, возвращаемое функцией rdsResizeVarArray. В настоящих моделях блоков такая проверка необходима. Если размер матрицы изменить не удалось (например, из-за нехватки памяти), функция возвращает FALSE. Если при этом модель, не проверив возвращенное значение, будет обращаться к элементам матрицы, под которые не удалось отвести память, вероятнее всего произойдет ошибка общей защиты. Также в этом примере не производится сравнение k и элементов матрицы X со специальным значением, возвращаемым функцией rdsGetHugeDouble, которое используется для сигнализации об

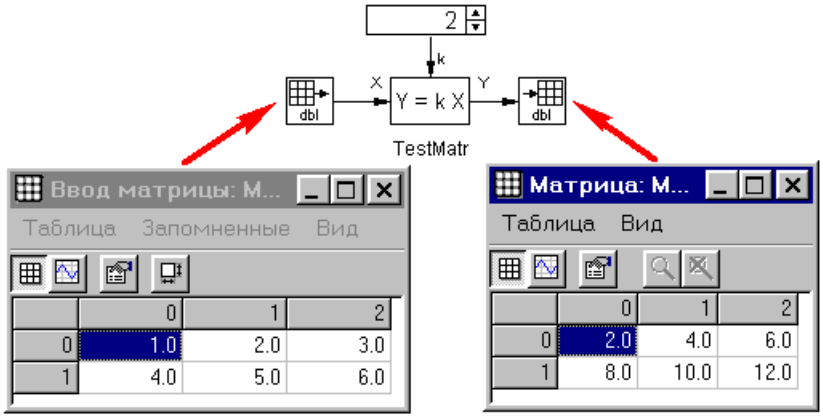


Рис. 22. Тестирование блока умножения матрицы на константу

ошибке (см. стр. 44). В модели, оперирующей вещественными числами двойной точности, такая проверка позволяет избежать возникновения исключений при выполнении арифметических операций.

§2.5.4. Работа со строками

Описываются особенности работы с переменными-строками произвольной длины и сервисные функции для их создания, уничтожения и сложения. Приводится пример блока, преобразующего поступившее на вход целое число в строку.

Для строк, как и для матриц, в дереве переменных хранится только указатель на динамически отводимую область памяти (рис. 23).

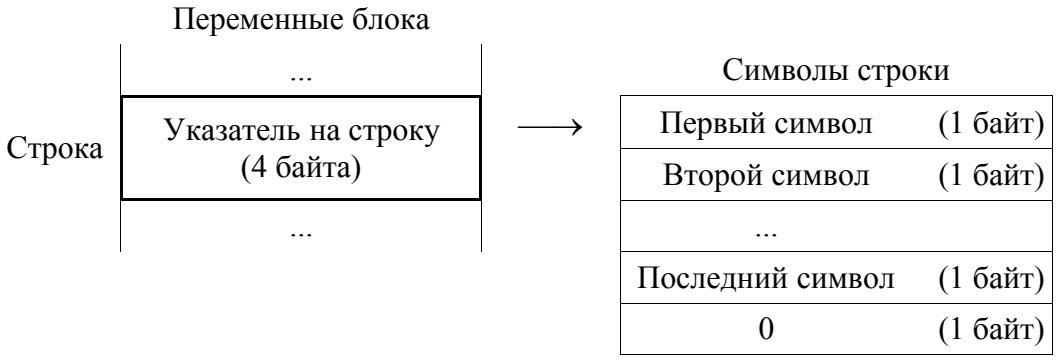


Рис. 23. Размещение в памяти данных строки

Для того, чтобы можно было использовать стандартные функции обработки строк, символы строки завершаются нулевым байтом. Таким образом, размер области памяти строки должен быть на единицу больше длины этой строки. Например, для размещения строки “ABCD” необходимо отвести область памяти размером в 5 байтов, в которой последовательно будут записаны коды символов 'A', 'B', 'C', 'D' и 0. Для пустой строки память не отводится, при этом указатель на строку в дереве переменных блока принимает значение `NULL`.

Модель блока может самостоятельно отводить память для строк при помощи сервисной функций `rdsAllocate` и освобождать ее при помощи `rdsFree`. ***Не следует*** пользоваться для этого стандартными библиотечными функциями `malloc` и `free` или операторами C++ `new` и `delete`. Функции отведения и освобождения памяти рассчитаны на

совместную работу: память, отведенная функцией malloc, должна быть освобождена функцией free; память, отведенная функцией rdsAllocate, должна быть освобождена функцией rdsFree. Если модель блока отведет память под строку при помощи malloc, при удалении блока или перед загрузкой другой схемы РДС попытается удалить эту память функцией rdsFree вместо free, что приведет к возникновению ошибки.

Для отведения памяти под строку можно также использовать любые сервисные функции РДС, формирующие динамические строки, поскольку все эти функции базируются на rdsAllocate. Например, для объединения двух строк можно вызвать функцию rdsDynStrCat и записать возвращенный ей указатель в дерево переменных блока.

В качестве примера рассмотрим блок, формирующий выходную строку str, добавляя к строке prefix текстовое представление целого числа val (например, если подать на вход val число 10 и на вход prefix строку “v=”, блок должен сформировать на выходе str строку “v=10”). Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	prefix	Строка	4	Вход
6	val	int	4	Вход
10	str	Строка	4	Выход

Функция модели этого блока будет выглядеть следующим образом:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestIntStr(int CallMode,
                         RDS_PBLOCKDATA BlockData,
                         LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define prefix  (*((char **)(pStart+2)))
#define val     (*((int *)(pStart+6)))
#define str     (*((char **)(pStart+10)))

  char buf[40]; // Буфер, в котором формируется строка
  int l1,l2;

  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSAIA}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        // Преобразование числа val в строку buf
        itoa(val,buf,10);
        // Определение длин строк prefix и buf
        l1=prefix==NULL?0:strlen(prefix);
        l2=strlen(buf);
        // Освобождение прежнего значения str
        rdsFree(str);
```

```
            // Отведение памяти под новую строку
            str=(char*)rdsAllocate(l1+l2+1);
            // Занесение строки в отведенную память
            if(prefix!=NULL) // Строка prefix не пуста
              { strcpy(str,prefix); // Копировать prefix
                strcat(str,buf);    // Дописать buf
              }
            else // Строка prefix пуста
              strcpy(str,buf);      // Копировать buf
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef str
    #undef val
    #undef prefix
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Поскольку в этом примере используется стандартная библиотечная функция `itoa`, для успешной компиляции необходимо включить в исходный текст стандартный файл заголовка “stdlib.h” (файл “string.h”, необходимый для стандартных функций `strlen`, `strcpy` и `strcat`, уже должен быть включен из-за присутствия в тексте модели функции `strcmp`, которая используется для сравнения строк при проверке типов переменных).

Вызов функции с параметром `RDS_BFM_VARCHECK` уже неоднократно описывался. В этом примере реализована точно такая же проверка типа переменных, как и в остальных. В данном случае строка, с которой сравнивается переданная в функцию строка типа, состоит из двух букв “S” для обязательных сигналов `Start` и `Ready`, буквы “A” для строки `prefix`, буквы “I” для целой переменной `val` и еще одной “A” для строки `str`.

При вызове модели с параметром `RDS_BFM_MODEL` в локальном массиве `buf` стандартной функцией `itoa` формируется текстовое представление значения переменной `val` (параметр 10 указывает на использование десятичной системы счисления). Размер массива `buf` выбран равным 40 поскольку тридцати девяти символов заведомо хватит для текстового представления тридцатидвухбитного целого числа типа `int`. Далее определяются длины строк `prefix` и `buf` и заносятся во вспомогательные переменные `l1` и `l2` соответственно. Макрос `prefix` представляет собой ссылку на одноименный вход блока, то есть указатель на данные входной строки. Для пустой строки этот указатель будет равен `NULL`, поэтому длина `l2` вычисляется при помощи условного оператора: если `prefix` равен `NULL`, `l2` присваивается значение 0 и вызов функции `strlen` не производится.

Далее необходимо объединить строки `prefix` и `buf` и присвоить получившуюся строку выходу блока `str`. Сначала вызовом `rdsFree(str)` освобождается память, занятая прежним значением строки (функцию `rdsFree` можно безопасно вызывать для нулевых указателей, поэтому если прежнее значение выхода было пустой строкой, и `str` равнялась `NULL`, ошибок не будет). Затем при помощи функции `rdsAllocate` отводится память для новой строки. В функцию передается размер отводимой области, который должен быть на единицу большим суммарной длины обеих объединяемых строк – один лишний байт требуется для нуля, завершающего строку. Функция возвращает указатель на отведенную область, который, после приведения к типу `char*`, записывается в переменную `str`. После того, как память отведена, в ней формируется объединение строк. Если строка `prefix` не пустая, она копируется в `str` при помощи функции `strcpy`, после чего функцией `strcat` к

ней в конец дописывается строка buf. Если же строка prefix пуста, в str копируется только строка buf.

Этот пример можно упростить, использовав сервисную функцию РДС rdsDynStrCat, которая самостоятельно отводит память, необходимую для размещения объединяемых строк. С использованием этой функции реакция модели на выполнение такта расчета будет выглядеть так:

```
case RDS_BFM_MODEL:
  // Преобразование числа в строку
  itoa(val,buf,10);
  // Освобождение прежнего значения str
  rdsFree(str);
  // Формирование выходной строки
  str=rdsDynStrCat(prefix,buf,TRUE);
  break;
```

Сначала, как и в предыдущем варианте модели, целое число val преобразуется в строку во вспомогательном массиве buf и освобождается старое значение str. Все остальные действия по отведению памяти, проверке строки prefix на пустоту и объединению строк выполняет функция rdsDynStrCat. Ей передаются указатели на объединяемые строки (любой из них может равняться NULL) и значение TRUE, указывающее на то, что если в результате объединения строк получается пустая строка, функция должна вернуть значение NULL, не отводя память. Возвращаемый функцией указатель записывается в переменную str. Вспомогательные переменные l1 и l2 в этом варианте модели не используются.

Как и при работе с матрицами, при работе со строками желательно вызывать модель блока только при срабатывании входных связей – это позволит увеличить быстродействие, избежав лишних вызовов модели, когда входные данные не изменяются. Для описанного блока следует включить запуск по сигналу, после чего в окне редактирования переменных задать для переменной Start начальное значение 1 и установить флаг “Пуск” для входов prefix и val.

§2.5.5. Работа со структурами

Описываются особенности работы с переменными-структурами, их размещение в памяти и способ доступа к их полям при помощи макросов. Приводится пример блока, суммирующего два комплексных входа, в котором комплексные числа представлены структурами с двумя вещественными полями Re и Im.

Достаточно часто необходимо передавать от блока к блоку несколько значений одновременно. Например, если блоки моделируют поведение каких-либо объектов в трехмерном пространстве, им нужно передавать друг другу координаты этих объектов, то есть шесть вещественных чисел (три числа описывают положение центра объекта в пространстве, и еще три – углы его поворота). Делать каждую их шести координат отдельным входом или выходом не очень удобно: для передачи информации об объекте между двумя блоками их придется соединить шестью связями – это загромождает схему, особенно если таких блоков много, и нагружает пользователя лишней работой. Можно записывать координаты в массивы: массив вещественных чисел из шести элементов может хранить все шесть координат объекта и передаваться между блоками как единое целое (пользователю потребуется провести только одну связь). Однако, при этом программист должен постоянно следить за размером переданного массива и помнить соответствие между координатами объекта и номерами элементов массива. Помнить это соответствие придется и пользователю: если он, например, захочет вывести координату X на числовой индикатор, он должен знать номер элемента массива, в котором эта координата находится. А если кроме шести вещественных координат необходимо передавать еще какую-либо информацию об объекте (например, строку с его названием), использование массивов становится невозможным: все элементы массива должны иметь один и тот же тип.

Решение этой проблемы давно известно во всех языках программирования высокого уровня: для хранения и передачи набора разнородных данных обычно используются структуры (в некоторых языках их называют “записями”), которые состоят из полей разного типа, каждое из которых имеет свое собственное имя. Имена полей существенно облегчают работу и пользователю, и программисту: запись “`Coords.X`” гораздо понятнее записи “`Coords[0]`”. РДС позволяет создавать в блоках переменные структурного типа (см. §1.5), в качестве полей они могут содержать как простые переменные, так и массивы и другие структуры. По связям структура всегда передается как единое целое, этим структуры отличаются от шин: каналы передачи данных в шинах тоже имеют свои имена, как и поля структур, но работают полностью независимо, в структуре же нельзя передать по связи только одно поле.

Для структуры в дереве переменных хранится указатель на область памяти, в которой последовательно размещаются ее поля (рис. 24).

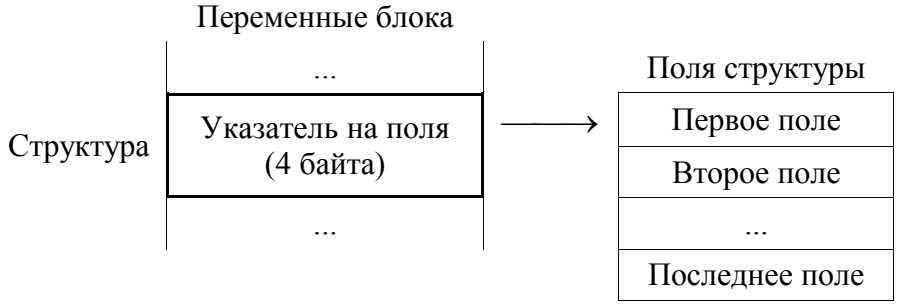


Рис. 24. Размещение в памяти данных структуры

Зная указатель на эту область, с полями структуры можно работать так же, как и с переменными блока – фактически, с точки зрения РДС, набор переменных блока тоже является структурой, указатель на которую хранится в поле `VarData` параметров блока `RDS_BLOCKDATA`. В отличие от матриц и строк, дополнительная память которых может освобождаться и отводиться заново при их изменении, область памяти с полями структуры отводится один раз при создании переменных блока и больше не изменяется до момента уничтожения блока или задания нового набора переменных. По этой причине к полям структуры, как и к переменным блока, можно обращаться по фиксированным смещениям относительно базового адреса (указателя на начало области данных).

Для примера введем в РДС структуру `TestComplex`, описывающую комплексное число и состоящую из двух полей `Re` и `Im` типа `double`:

Смещение	*Имя*	*Тип*	*Размер*
0	Re	double	8
8	Im	double	8

Для редактирования и создания новых структур служит пункт меню “Система | Структуры” (рис. 25), он открывает окно со списком структур, содержащим имена всех созданных в схеме структур и число блоков схемы, использующих каждую структуру в данный момент (при попытке изменить структуру, которая в данный момент используется, РДС выведет предупреждение, так как добавление, удаление и перестановка полей в структуре может привести к неработоспособности блоков, модели которых ссылаются на ее поля по фиксированным смещениям).

В окне списка структур можно добавить новую структуру (кнопка “+”) и изменить уже существующую (кнопка “{...}”). Нам нужно создать новую структуру – после нажатия кнопки “+” откроется окно редактора структур (рис. 26), похожее на окно редактора переменных блока. Фактически, эти два окна отличаются только тем, что в редакторе

структур нельзя сделать поле структуры входом или выходом (входом или выходом блока может быть только вся структура целиком), задать запуск модели блока, связанные логические и сигнальные переменные, а также отображение имени в точке присоединения связи (см. §1.5) – все эти параметры структура вместе со всеми своими полями получит, когда она будет использована в каком-либо блоке в качестве входа или выхода. В окне редактора структур нужно задать имя нашей структуры (“TestComplex”) ввести два вещественных поля `Re` и `Im`, и запомнить структуру нажатием кнопки “OK”. Теперь структура TestComplex зарегистрирована в схеме и может быть указана в качестве типа любой статической переменной любого блока.

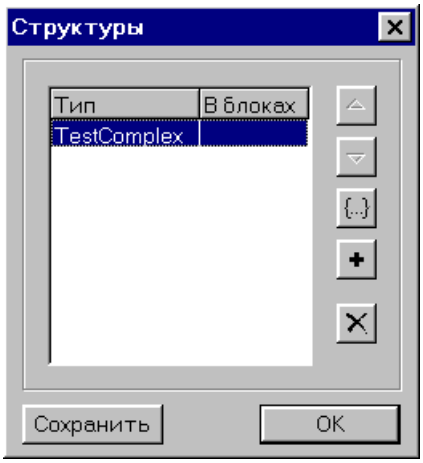


Рис. 25. Окно списка структур схемы

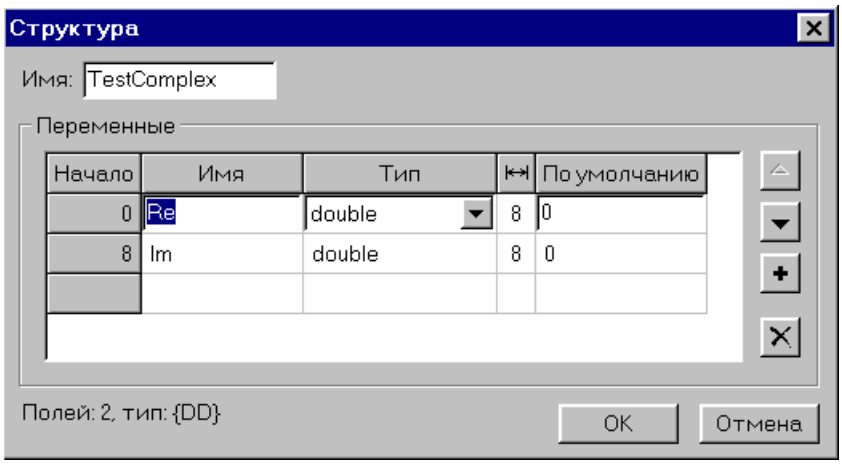


Рис. 26. Окно редактирования полей структуры

Создадим модель блока, суммирующего значения комплексных (типа “TestComplex”) входов `x1` и `x2` и выдающего сумму на выход `y`. Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	x1	Структура TestComplex	4	Вход
6	x2	Структура TestComplex	4	Вход
10	y	Структура TestComplex	4	Выход

Функция модели блока будет довольно простой:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestStructSum(int CallMode,
                            RDS_PBLOCKDATA BlockData,
                            LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define x1      (*((void **)(pStart+2)))
```

```
#define x2      (*((void **)(pStart+6)))
#define y       (*((void **)(pStart+10)))
// Макроопределения для полей структуры TestComplex
#define Re(base)      (*((double *)(base)))
#define Im(base)      (*((double *)(((char*)(base))+8)))

  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SS{DD}{DD}{DD}}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        Re(y)=Re(x1)+Re(x2);
        Im(y)=Im(x1)+Im(x2);
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Im
#undef Re
#undef y
#undef x2
#undef x1
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Макроопределения для структур `x1`, `x2` и `y` представляют собой указатели универсального типа (`void*`) на начало области данных каждой структуры. В отличие от определений для переменных блока, которые жестко привязаны к началу дерева переменных `BlockData->VarData`, определения для полей структуры `Re` и `Im` – это макросы с параметром, через который передается указатель на данные конкретной структуры. Например, для доступа к полю `Re` структуры `x1` необходимо написать `Re(x1)` (рис. 27).

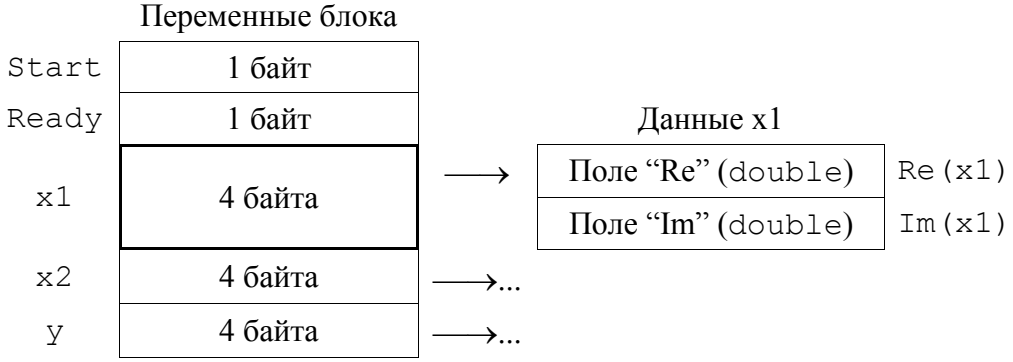


Рис. 27. Обращение к полям структуры из модели блока

Чтобы не писать такие макроопределения вручную, можно заставить РДС сформировать их в буфере обмена, нажав правую кнопку мыши на строке типа в левом нижнем углу окна редактирования структуры и выбрав в открывшемся меню (см. аналогичное меню в редакторе переменных блока на рис. 16) пункт “Копировать список (#define, по возможности сами данные)”.

Строка типа, передаваемая функции модели при вызове с параметром RDS_BFM_VARCHECK, состоит из двух букв “S” для обязательных сигналов Start и Ready, и трех фрагментов “{DD}”, описывающих тип TestComplex для переменных x1, x2 и y. При проверке переменных название типа структуры не имеет значения. Важно только то, что структура состоит из двух полей double (два символа “D”). Если, например, ввести в РДС структуру TestStruct, состоящую из двух полей Field1 и Field2 типа double, и заменить в описываемом блоке тип TestComplex на TestStruct, в модель будет передаваться точно такая же строка типа. Обе структуры будут описываться строкой “{DD}”, и модель не увидит между ними разницы. Тем не менее, это не приведет к возникновению ошибок, поскольку структуры с одинаковыми типами полей размещаются в памяти одинаково. Макрос Re(x1) может с одинаковым успехом обращаться к полю Re, если переменная x1 имеет тип TestComplex, и к полю Field1, если x1 имеет тип TestStruct, поскольку эти поля имеют одинаковые смещение и размер. Эта модель, предназначенная для суммирования комплексных чисел, то есть одноименных полей структуры TestComplex, сможет суммировать и одноименные поля структуры TestStruct (целесообразность такого суммирования остается на совести пользователя, заменившего одну структуру на другую).

При вызове модели с параметром RDS_BFM_MODEL в поля Re и Im выхода y записывается сумма одноименных полей входов x1 и x2. Макросы для полей структур, как и макросы статических переменных, могут находиться и в левой, и в правой части выражения. Чтобы присвоить y.Re сумму x1.Re и x2.Re, нужно записать

```
Re(y)=Re(x1)+Re(x2)
```

Точно так же записывается суммирование полей Im. Для упрощения примера мы здесь не проверяем допустимость сложения полей структуры. Одно из них может оказаться равным специальному значению, сигнализирующему об ошибке вычисления, возникшей в предыдущем блоке (оно возвращается функцией rdsGetHugeDouble, см. стр. 44) – в этом случае выполнять сложение вещественных чисел нельзя, вместо этого соответствующему полу выхода y следует присвоить это же значение-индикатор ошибки.

Поскольку структуру TestComplex мы только что создали для этого примера, единственная модель, которая с ней работает – это написанная нами TestStructSum. Хотя у нас нет других блоков и моделей, поддерживающих нашу новую структуру, проверить работу нового блока мы все равно можем: РДС позволяет подключать связи не только ко всей структуре как к единой переменной, но и к отдельным ее полям. Можно собрать схему, изображенную на рис. 28: к полям Re и Im входов блока x1 и x2 подключены поля ввода, а к полям выхода y – индикаторы. При запущенном расчете на индикаторе, соединенном с y.Re должна отображаться сумма x1.Re и x2.Re, а на индикаторе, соединенном с y.Im – сумма x1.Im и x2.Im.

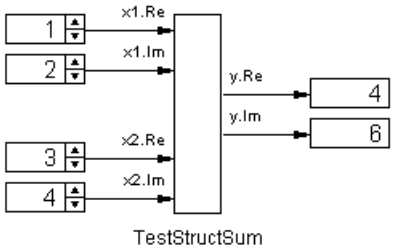


Рис. 28. Схема для проверки работы со структурами

Изображенная на рис. 28 схема не показывает преимущества структур перед отдельными переменными – как было указано выше, у нас пока есть только один блок, работающий со структурой “TestComplex”, поэтому все связи в схеме присоединяются к отдельным полям структур. Однако, выход этого блока мы уже можем соединить с входом другого такого же (например, чтобы добавить к вычисленной сумме третье комплексное число), и для этого потребуется всего одна связь. Мы также можем ввести выход блока в какую-либо подсистему через один внешний вход (см. стр. 13) типа “TestComplex”, подключить его к одному каналу шины этого же типа и т.п. – нам теперь не нужно создавать пару связей для каждой передачи комплексного числа.

§2.5.6. Работа с переменными произвольного типа

Описываются особенности работы с переменными, тип которых может изменяться в процессе работы системы. Приведен пример блока – универсального выключателя, пропускающего или не пропускающего значение входа произвольного типа на выход в зависимости от дополнительного логического входа. Также приводится пример блока, выдающего на выходы разные значения в зависимости от типа значения, поступившего на вход. В третьем примере модель меняет тип выхода в зависимости от значения целого числа на входе.

Переменные произвольного типа могут содержать данные любого из других типов, используемых в РДС. Фактический тип такой переменной, то есть тип хранимых в ней данных, может изменяться в процессе работы, поэтому переменные произвольного типа устроены сложнее и передаются по связям медленнее, чем все остальные. Чаще всего они применяются при создании универсальных блоков-переключателей, передающих данные с заданного входа на заданный выход в зависимости от каких-либо условий. Такие блоки обычно не интересуются фактическим значением своего входа, они просто передают его на один из выходов без изменения. Если входы и выходы такого блока имеют произвольный тип, его можно использовать для коммутации любых значений: матриц, структур, целых и вещественных чисел и т.д. В противном случае пришлось бы писать отдельную модель для коммутации значений каждого из возможных типов.

Данные переменной произвольного типа в дереве переменных занимают 8 байтов (рис. 29). В первых четырех хранится указатель на динамически отводимую память, в которой размещаются данные того фактического типа, который в данный момент имеет эта переменная. Если переменная в данный момент не имеет фактического типа, в первых четырех байтах ее данных содержится нулевой указатель `NULL`. Во вторых четырех байтах хранится служебная информация, по которой РДС определяет фактический тип переменной (модель не должна изменять их значения).

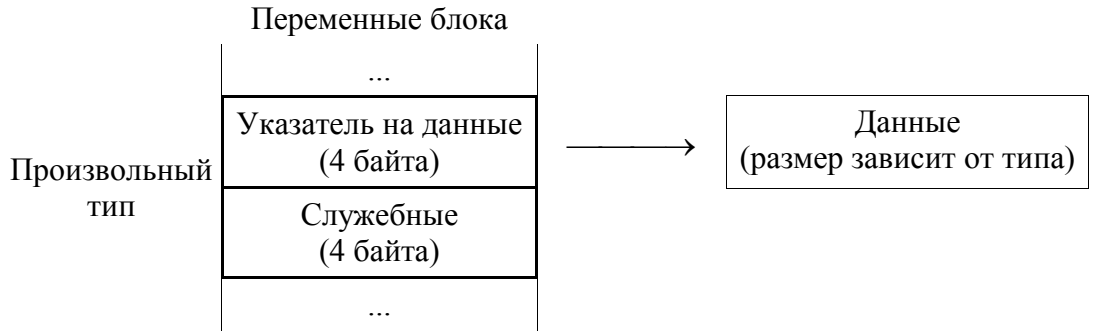


Рис. 29. Размещение в памяти данных переменной произвольного типа

Для любого фактического типа отводимая область данных устроена точно так же, как у статической переменной такого же типа в дереве переменных блока. На рис. 30 приведено несколько примеров размещения в памяти переменных с разными фактическими типами.

Самое простое и наиболее часто используемое действие над переменными произвольного типа – копирование данных одной переменной в другую при помощи сервисной функции `rdsCopyRuntimeType`. Именно эта функция используется в моделях различных блоков-переключателей. В качестве примера рассмотрим простой блок со входом произвольного типа `x`, логическим входом `Enable` и выходом произвольного типа `y`. Если значение входа `Enable` равно единице, блок должен передавать данные со входа `x` на выход `y` независимо от их типа. Если же значение `Enable` равно нулю, блок не должен пропускать данные на выход. Такой блок можно использовать в качестве универсального выключателя, разрешающего или запрещающего передачу данных в зависимости от значения `Enable`. Его можно вставить в разрыв связи любого типа – на рис. 31 слева он управляет передачей

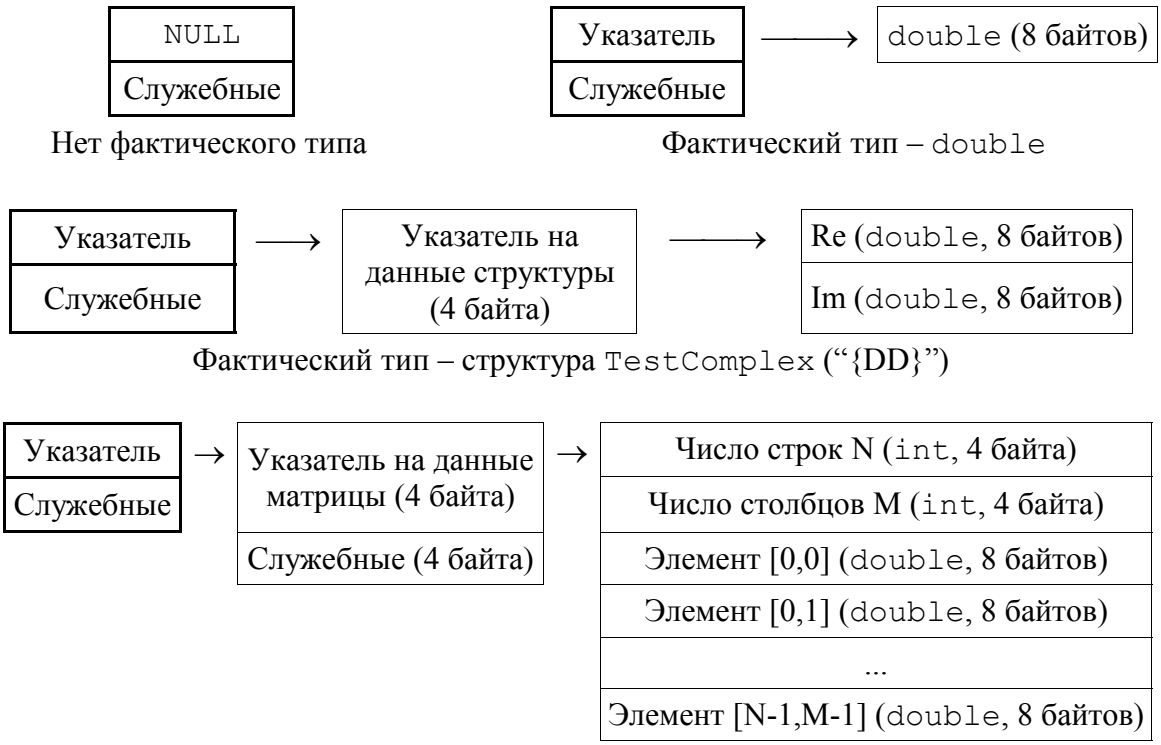


Рис. 30. Примеры отведения памяти в переменной произвольного типа для разных фактических типов

вещественного числа, справа – передачей матрицы. Блок сам изменит тип своего выхода `y` согласно типу входа `x`.

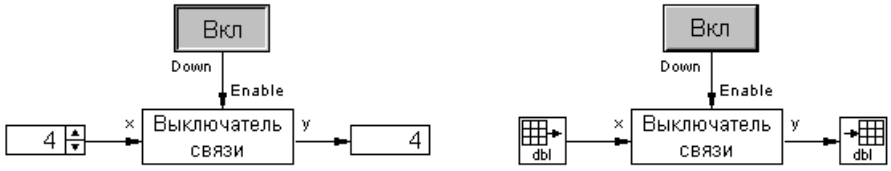


Рис. 31. Универсальный выключатель, работающий с переменными любых типов

Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	x	Произвольный	8	Вход
10	Enable	Логический	1	Вход
11	y	Произвольный	8	Выход

Поскольку тип передаваемых данных для этого блока не важен, в его модели целесообразно использовать функцию `rdsCopyRuntimeType`:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestVarSwitch(int CallMode,
                            RDS_PBLOCKDATA BlockData,
```

```
                                LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define px      ((void **)(pStart+2))
#define Enable  (*((char *)(pStart+10)))
#define py      ((void **)(pStart+11))

  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSVLV}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        if(Enable) // Передача разрешена
          rdsCopyRuntimeType(py,px); // Копирование x в y
        else // Передача запрещена
          Ready=0; // Блокировка передачи данных по связям
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef py
#undef Enable
#undef px
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Для переменных произвольного типа `x` и `y` вместо определений для доступа к самим переменным вводятся определения для указателей `px` и `py`. Эти указатели передаются в сервисные функции, обслуживающие переменные произвольного типа (в том числе и в `rdsCopyRuntimeType`).

Проверка типа переменных производится стандартным образом – в данном случае переменным произвольного типа соответствуют буквы “V” в строке, передаваемой в модель блока при вызове `RDS_BFM_VARCHECK`. При вызове модели с параметром `RDS_BFM_MODEL` проверяется значение логического входа `Enable`. Если это значение ненулевое, то есть передача данных разрешена, вызывается функция `rdsCopyRuntimeType (py,px)`, копирующая значение входа `x` в выход `y`. Эта функция самостоятельно отводит память для нового значения `y` и, при необходимости, освобождает память, занимаемую прежним значением – никаких дополнительных действий от программиста не требуется. Если же значение входа `Enable` равно нулю, т.е. передача данных запрещена, сигнальному выходу `Ready` присваивается 0. Как уже упоминалось выше, сигнал `Ready` – один из двух обязательных сигналов, присутствующих в каждом простом блоке. Если значение `Ready` будет равно нулю, связи, соединенные с выходом этого блока, не сработают, что и требуется при запрещении передачи данных. Значение 1 присваивается сигналу `Ready` автоматически при каждом запуске модели, поэтому явно присваивать ему единицу при разрешении передачи не требуется.

Приведенную модель, как и любую модель со сложными переменными, желательно вызывать только при изменении входных данных. Для этого следует включить для блока с этой моделью запуск по сигналу (см. рис. 5), после чего в окне редактирования переменных (рис. 9) задать для переменной `Start` начальное значение 1 и установить флаг “Пуск” для входов `x` и `Enable`.

Модель блока-выключателя получилась очень простой, поскольку, несмотря на сложную структуру переменных произвольного типа, копирование данных из одной переменной в другую производится внутри сервисной функции `rdsCopyRuntimeType`. Модель блока, самостоятельно анализирующая фактический тип переменной и считывающая ее данные, или самостоятельно присваивающая значение выходу произвольного типа, будет гораздо сложнее. На самом деле, необходимость в создании таких моделей возникает очень редко – как правило, не существует действия, которое можно было бы выполнить и над числами, и над матрицами, и над строками, и над всеми возможными структурами. Для выполнения действий над группой похожих типов (например, над числами всех видов) можно обойтись и без произвольного типа. РДС позволяет соединять связями входы и выходы разных типов, если эти типы могут быть приведены один к другому (например, допускается присоединение выхода типа `double` ко входу типа `int`, и наоборот). Для выполнения какого-либо действия обычно достаточно создать модель, выполняющую это действие над переменными наиболее общего типа. Например, блок, вычисляющий сумму двух вещественных чисел типа `double`, может также использоваться для сложения чисел типа `int`, `char`, `short` и `float`.

Тем не менее, модель блока может, при необходимости, работать с данными переменной произвольного типа непосредственно. Рассмотрим в качестве примера модель блока с входом произвольного типа `x`, выдающую строку фактического типа этого входа на выход `type`. Кроме того, если вход `x` имеет фактический тип `double` или `int`, эта модель должна выдать его значение на вещественный выход `val`. Если `x` – массив или матрица любого типа, на `val` необходимо выдать число элементов в матрице. Если же `x` имеет другой фактический тип, выход `val` должен быть равен нулю. Практическая ценность этого примера сомнительна, но он позволит проиллюстрировать работу с произвольным типом. Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	x	Произвольный	8	Вход
10	type	Строка	4	Выход
14	val	double	8	Выход

Для получения строки фактического типа входа `x` и указателя на его область данных будем использовать сервисную функцию `rdsGetRuntimeTypeData`. Модель блока будет выглядеть следующим образом:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestVar1(int CallMode,
                       RDS_PBLOCKDATA BlockData,
                       LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define px      ((void **)(pStart+2))
```

```
#define type    (*((char **)(pStart+10)))
#define val     (*((double *)(pStart+14)))
  // Вспомогательные переменные
  char *s; // Строка фактического типа входа x
  void *v; // Указатель на данные входа x

  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSVAD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        // Освобождение прежнего значения строки type
        rdsFree(type);
        // Получение указателя на данные (v)
        // и строки фактического типа (s) входа x
        v=rdsGetRuntimeTypeData(px,&s);
        // Занесение строки типа в выход type
        type=s;
        // Анализ типа, если переменная не пуста
        if(v!=NULL) // У входа x есть фактический тип
          switch(*s) // Анализ первого символа строки типа
            { case 'D': // Фактический тип - double
                val=*((double*)v);
                break;
              case 'I': // Фактический тип - int
                val=*((int*)v);
                break;
              case 'M': // Фактический тип - матрица или массив
                if(RDS_ARRAYEXISTS(v)) // Матрица не пуста
                  val=RDS_ARRAYROWS(v)*RDS_ARRAYCOLS(v);
                else // Матрица пуста (0x0)
                  val=0;
                break;
              default: // Другой фактический тип
                val=0.0;
            } // Конец switch(*s)
        else // У входа x нет фактического типа
          val=0.0;
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef val
#undef type
#undef px
#undef Ready
#undef Start
#undef pStart
}
//=========================================
```

При вызове модели с параметром `RDS_BFM_MODEL` сначала вызывается функция `rdsFree` для освобождения данных прежнего значения выходной строки `type`. Далее при помощи функции `rdsGetRuntimeTypeData` определяется фактический тип входа `x`. В первом

параметре функции передается указатель на исследуемую переменную произвольного типа (px), во втором – указатель на переменную, в которую нужно записать указатель на динамически сформированную строку фактического типа (в данном случае передается указатель на вспомогательную переменную s). Функция возвращает указатель на область данных входа, который присваивается вспомогательной переменной v. Если у входа x нет фактического типа, переменным v и s будет присвоено значение NULL. На самом деле, если бы в этой модели не нужна была строка типа, можно было бы не использовать сервисную функцию rdsGetRuntimeTypeData, а получить указатель на область данных входа v при помощи оператора v=*px. Однако, в данном случае удобнее получить оба указателя в одном вызове.

Все динамически сформированные строки, возвращаемые сервисными функциями РДС, совместимы с функцией rdsFree, поэтому указатели на эти строки могут непосредственно присваиваться переменным блока. В данном случае указатель на строку типа, который функция rdsGetRuntimeTypeData записала в переменную s, присваивается выходу type.

Далее необходимо проверить фактический тип входа x, и, в зависимости от него, вычислить значение выхода val. Если значение v, которое вернула функция rdsGetRuntimeTypeData, не равно NULL, значит, тип у входа есть. В этом случае выполнятся оператор switch(*s), анализирующий первый символ строки фактического типа входа.

Если x имеет тип double, строка s будет состоять из единственного символа “D”. В этом случае v указывает на восьмибайтовую область, в которой хранится вещественное значение входа. Указатель v приводится к типу double*, и значение входа присваивается выходу блока val. Аналогично, если x имеет тип int, s будет содержать единственный символ “I”, и, после приведения типа указателя v к int*, четырехбайтовое целое значение входа будет присвоено выходу блока. Если же x будет матрицей, строка s будет состоять из символа “M” и типа элемента матрицы (например, для матрицы double – “MD”, для матрицы строк – “MA”, для матрицы упоминавшихся ранее структур TestComplex – “M{DD}”). В данной модели выходу val в этом случае присваивается число элементов в матрице, равное произведению числа строк и числа столбцов, поэтому тип элементов матрицы анализировать не нужно – достаточно считать ее размеры. Для проверки матрицы на пустоту и получения ее размеров используются уже знакомые по прошлым примерам макросы RDS_ARRAYEXISTS, RDS_ARRAYROWS и RDS_ARRAYCOLS, только в этом случае в них подставляется не макроопределение для какой-нибудь статической переменной, а указатель на область данных входа (вспомогательная переменная v), возвращенный функцией rdsGetRuntimeTypeData. Область данных переменной произвольного типа устроена точно так же, как статическая переменная соответствующего фактического типа, поэтому эти макросы будут работать для нее так же, как и для обычной статической матрицы.

На рис. 32 изображена схема, собранная для проверки работы созданной модели блока. В ней к входу блока x подключены три связи: первая соединяет его с выходом библиотечного блока выбора варианта (его выход – целое число), вторая – с обычным полем ввода вещественного числа, третья – с блоком ввода матрицы. Две из трех связей заблокированы, чтобы работала только одна. К выходу val подключен числовой индикатор, а к выходу type – блок, отображающий строку.

В зависимости от того, какая из трех входных связей блока не заблокирована, фактический тип его входа x будет либо целым, либо вещественным, либо матрицей вещественных чисел. На рисунке работает только связь, соединяющая наш блок с блоком ввода матрицы, поэтому на выход type выдается строка “MD” (матрица double), а на выход val – число элементов матрицы – в данном случае, шесть. Если заблокировать эту связь, и

разблокировать соединение с блоком выбора варианта, на выходе `type` появится строка “I” (`int`), если же разблокировать соединение с полем ввода – строка “D” (`double`), а выход `val` в обоих случаях будет равен поступившему на вход `x` числу, независимо от его типа.

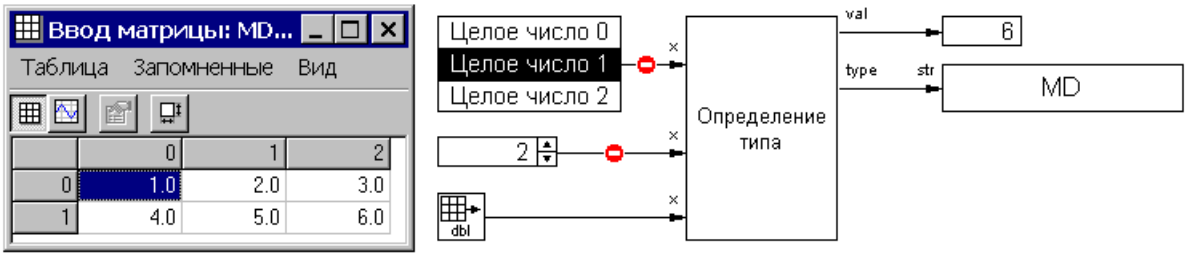


Рис. 32. Пример блока, обрабатывающего данные входа произвольного типа

Приведенный пример демонстрирует возможность получения данных из переменной произвольного типа. Рассмотрим другой пример, в котором модель блока будет присваивать переменной произвольного типа данные разных типов, то есть программно задавать фактический тип выхода блока. Пусть у блока будет целый вход `Type` и выход `y` произвольного типа. При нулевом значении `Type` модель должна передать на выход целое число 1, при `Type`, равном единице – вещественное число 2, а при любых других значениях `Type` – сформировать на выходе матрицу вещественных чисел 2x2 и заполнить ее значениями 1, 2, 3 и 4. Блок должен иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	Type	int	4	Вход
6	y	Произвольный	8	Выход

Модель блока будет такой:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestVar2(int CallMode,
                       RDS_PBLOCKDATA BlockData,
                       LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define Type    (*((int *)(pStart+2)))
#define py      ((void **)(pStart+6))
// Вспомогательные переменные
int *i_ptr;     // Указатель на данные для целого типа
double *d_ptr;  // Указатель на данные для вещественного типа
void *v_ptr;    // Указатель на данные для матрицы

  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSIV}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;
```

```
        // Выполнение такта моделирования
        case RDS_BFM_MODEL:
          switch(Type)
            { case 0: // Выдать целое число
                // Установить целый тип выхода
                i_ptr=(int*)rdsSetRuntimeType(py,"I");
                // Присвоить выходу значение
                if(i_ptr) *i_ptr=1;
                break;
              case 1: // Выдать вещественное число
                // Установить вещественный тип выхода
                d_ptr=(double*)rdsSetRuntimeType(py,"D");
                // Присвоить выходу значение
                if(d_ptr) *d_ptr=2.0;
                break;
              default: // Выдать матрицу
                // Тип выхода - матрица double
                v_ptr=rdsSetRuntimeType(py,"MD");
                if(v_ptr)
                  { double *array;
                    // Установить размер матрицы
                    rdsResizeVarArray(v_ptr,2,2,FALSE,NULL);
                    // Получить указатель на первый элемент
                    array=(double*)RDS_ARRAYDATA(v_ptr);
                    // Заполнить матрицу числами 1,2,3,4
                    for(int i=0;i<4;i++)
                      array[i]=i+1;
                  }
            } // Конец switch(Type)
          break;

      }
    return RDS_BFR_DONE;
  // Отмена макроопределений
  #undef py
  #undef Type
  #undef Ready
  #undef Start
  #undef pStart
  }
  //=========================================
```

При вызове модели для выполнения такта расчета (параметр `CallMode` равен `RDS_BFM_MODEL`) анализируется значение входа `Type`. Если значение `Type` равно 0, на выход блока необходимо выдать целое число. Для этого вызывается функция `rdsSetRuntimeType`, присваивающая выходу произвольного типа `y` фактический тип `int`. В первом параметре функции передается указатель на переменную произвольного типа (`py`), во втором – строка типа, присваиваемая переменной (для типа `int` передается строка “I”). Функция возвращает указатель на созданную область данных заданного типа, в которую, после приведения указателя к типу `int*`, записывается число 1.

Если значение `Type` равно 1, на выход блока выдается вещественное число. Для этого в функцию `rdsSetRuntimeType` передается строка типа “D”, соответствующая типу `double`. Указатель, возвращенный функцией, приводится к типу `double*` и используется для занесения в созданную область данных числа 2.0.

При любом другом значении `Type` для выхода блока устанавливается фактический тип “матрица `double`”, для чего в функцию `rdsSetRuntimeType` передается строка

“MD”. Указатель, возвращенный этой функцией, передается в уже знакомую нам функцию `rdsResizeVarArray` (см. §2.5.3), которая устанавливает для созданной матрицы размер 2x2. Далее при помощи описывавшегося ранее макроса `RDS_ARRAYDATA` указатель на первый элемент матрицы присваивается вспомогательной переменной `array`. Первому элементу матрицы будет соответствовать значение `array[0]`, второму – `array[1]` и т.д. В матрице 2x2 содержится четыре элемента, поэтому далее в цикле четырем элементам `array` присваиваются числа, на единицу большие их индексов, то есть 1, 2, 3 и 4.

Для проверки работы этой модели можно собрать схему, изображенную на рис. 33. Вход блока `Type` связан с выходом стандартного библиотечного блока выбора варианта, а выход `y` одновременно подан на входы числового индикатора и блока отображения матрицы. Выходы произвольного типа могут быть связаны с входами любого типа, поэтому такое соединение допустимо. Если в блоке выбора варианта будут выбраны пункты “int” или “double”, на его выходе, а, значит, и на входе `Type` нашего блока, появится целое число 0 или 1 соответственно. При этом фактическим типом выхода `y` станет `int` или `double`, и число с этого выхода будет передано по верхней ветви связи на индикатор. Нижняя ветвь связи не сработает, поскольку она соединена с входом типа “матрица double”, а целое или вещественное число не может быть передано на вход такого типа. Если же будет выбран пункт “матрица”, вход `Type` получит значение 2, в результате чего наш блок сформирует на своем выходе `y` матрицу из четырех вещественных чисел, и она будет передана на вход блока отображения матриц по нижней ветви выходной связи. Верхняя ветвь связи не сработает, поскольку матрица вещественных чисел не может быть передана на вход числового индикатора.

Рис. 33. Пример блока, программно задающего фактический тип своего выхода

Приведенные примеры показывают, как создавать модели блоков, тип входов и выходов которых заранее не известен и может изменяться в процессе расчета. Хотя использование переменных произвольного типа и приводит к усложнению модели и некоторому замедлению расчета, блоки с такими переменными получаются универсальными, что, в некоторых случаях, может оказаться полезным.

Следует, однако, помнить, что в тех случаях, когда для работы блока необходимы входы или выходы не известного заранее типа, но тип их будет определяться не в процессе расчета, а до него (например, при настройке блока на работу с определенными данными), вместо использования переменных произвольного типа лучше в нужный момент программно изменить тип входа или выхода в структуре переменных блока с помощью сервисных функций РДС. При этом переменная сразу окажется нужного типа и будет передаваться по связям значительно быстрее. Модель блока тоже упростится: не нужно будет выяснять фактический тип переменной в каждом такте расчета, достаточно просто запомнить его при вызове модели в момент изменения структуры переменных (режим `RDS_BFM_VARCHECK`). Разумеется изменять структуру переменных нужно до того, как к блоку подключены связи: если, например, вход блока имел тип `double`, а модель сделала его матрицей, то связь, подключенная к этому блоку, не сможет работать и будет помечена как ошибочная.

Примеры программного изменения структуры переменных блока приведены в §2.16.1.

§2.5.7. Использование входов со связанными сигналами

Описывается работа с входами блока, для которых заданы связанные сигналы – по этим сигналам можно понять, какие из входов блока сработали в данном такте расчета. Приводится пример модели блока, вычисляющей произведение минимального элемента первой входной матрицы и максимального элемента второй входной матрицы. Анализируя связанные сигналы входов, эта модель вычисляет максимальный или минимальный элемент только для изменившейся матрицы, экономя тем самым процессорное время.

В режиме расчета, когда модели блоков схемы постоянно вызываются в цикле и с заданной периодичностью перерисовываются все открытые окна, нагрузка на систему увеличивается. Если модель производит какие-либо сложные вычисления (например, итеративный расчет или обработку матриц), задержка, вызванная этими вычислениями, повторяемыми в цикле, может снизить быстродействие системы. Поэтому обычно стараются сделать так, чтобы модели блоков вызывались только тогда, когда это действительно необходимо. Обычно для этого в параметрах блока отключают флаг “запуск каждый такт” и устанавливают флаги “пуск” для всех входов блока, в результате чего модель будет вызываться только при срабатывании связи, подключенной к любому из входов блока. Если ни одна из связей не сработала, значит, значения входов блока не изменились, и, в большинстве случаев, новые значения выходов вычислять не обязательно (если блок подписан на динамические переменные, ему необходимо также отслеживать и их изменения, но для этого существуют другие методы, которые будут описаны в §2.6).

Иногда информации о том, что значение какого-то входа блока изменилось, бывает недостаточно. Часто модели необходимо знать, какой именно вход получил новое значение – это позволит ей выполнить только те вычисления, которые связаны с изменившимся входом. Для того, чтобы модель могла узнать об изменении конкретного входа, необходимо добавить в блок дополнительную переменную-сигнал (неважно, будет она входом, выходом или внутренней) и связать с ней нужный вход блока, задав для этого входа тип “вход / сигнал” вместо “вход” и указав имя сигнальной переменной (см. стр. 19). При срабатывании связи, подключенной к данному входу, связанной сигнальной переменной автоматически будет присвоено значение 1. Появление единицы в одной или нескольких связанных переменных укажет модели блока на входы, значения которых изменились в предыдущем такте расчета. Выполнив необходимые вычисления, модель должна присвоить всем связанным переменным значение 0, подготовив их тем самым к следующему срабатыванию. Связанные переменные, как и обычные сигналы, не могут получить нулевое значение по подключенной связи – при срабатывании связи сигнальная переменная принимает значение 1 и сохраняет его до тех пор, пока модель блока самостоятельно не обнулит переменную. Допускается связывание нескольких входов с одним сигналом, если модель блока должна выполнять одинаковые действия при изменении любого из этих входов.

Рассмотрим в качестве примера блок, вычисляющий произведение минимального элемента входной матрицы вещественных чисел `M1` и максимального элемента такой же входной матрицы `M2`. Для того, чтобы определить максимальное и минимальное значения, необходимо перебрать все элементы матриц. Для матриц больших размеров этот перебор может занять значительное время, поэтому модель блока будет запоминать значения максимального и минимального элементов матриц `M1` и `M2` во вспомогательных переменных `M1min` и `M2max` соответственно. Значение `M1min` будет вычисляться только при изменении входа `M1`, значение `M2max` – только при изменении `M2`. Для того, чтобы модель могла узнать о поступлении новых значений на входы, вход `M1` будет связан с внутренней сигнальной переменной `s1`, вход `M2` – с переменной `s2`. Таким образом, блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Логика*	*Начальное значение*
0	Start	Сигнал	1	Вход		1
1	Ready	Сигнал	1	Выход		–
2	s1	Сигнал	1	Внутренняя		1
3	s2	Сигнал	1	Внутренняя		1
4	M1	Матрица double	8	Вход/сигнал	s1	–
12	M2	Матрица double	8	Вход/сигнал	s2	–
20	M1min	double	8	Внутренняя		?
28	M2max	double	8	Внутренняя		?
36	y	double	8	Выход		–

Следует обратить внимание на то, что в качестве начальных значений переменных `M1min` и `M2max` указан вопросительный знак. Вопросительным знаком в РДС обозначается специальное значение, используемое в качестве индикатора ошибки вычисления. Действительно, пока максимальный элемент `M2` и минимальный элемент `M1` не вычислены, мы ничего не можем присвоить переменным `M1min` и `M2max`.

Поскольку необходимость запуска модели будет определяться срабатыванием входных связей, необходимо включить флаг “запуск по сигналу” (вместо “запуск каждый такт”) на вкладке “Общие” окна параметров этого блока и установить флаг “Пуск” для входов `M1` и `M2` в окне редактирования переменных. Также необходимо задать переменным `Start`, `s1` и `s2` начальное значение 1. При первом запуске расчета, до того, как сработает какая-либо из входных связей, модель должна вычислить значения `M1min` и `M2max` по начальным значениям `M1` и `M2`. Единичное начальное значение переменной `Start` заставит модель запуститься в самом первом такте расчета, единичные начальные значения `s1` и `s2` заставят ее обработать обе входных матрицы. Начальные значения всех остальных переменных могут быть любыми.

Модель блока будет выглядеть следующим образом:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestMatMinMax(int CallMode,
                            RDS_PBLOCKDATA BlockData,
                            LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define s1      (*((char *)(pStart+2)))
#define s2      (*((char *)(pStart+3)))
#define pM1     ((void **)(pStart+4))
#define pM2     ((void **)(pStart+12))
#define M1min   (*((double *)(pStart+20)))
#define M2max   (*((double *)(pStart+28)))
#define y       (*((double *)(pStart+36)))

  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSSSMDMDDDD}")==0)
          return RDS_BFR_DONE;
```

```
      return RDS_BFR_BADVARSMSG;

    // Выполнение такта моделирования
    case RDS_BFM_MODEL:
      if(s1) // Изменилось значение входа M1
        { // Необходимо найти новый минимальный элемент
          s1=0; // Сброс сигнала
          if(RDS_ARRAYEXISTS(pM1)) // Матрица M1 не пуста
            { // Вспомогательные переменные
              int m1count;
              double *m1data;
              // Число элементов в матрице
              m1count=RDS_ARRAYROWS(pM1)*RDS_ARRAYCOLS(pM1);
              // Получить указатель на первый элемент
              m1data=(double*)RDS_ARRAYDATA(pM1);
              // Поиск минимального элемента
              M1min=DoubleErrorValue;
              for(int i=0;i<m1count;i++)
                if(m1data[i]!=DoubleErrorValue &&
                   (M1min==DoubleErrorValue || M1min>m1data[i]))
                  M1min=m1data[i];
            }
          else // Матрица M1 пуста
            M1min=DoubleErrorValue;
        } // if(s1)
      if(s2) // Изменилось значение входа M2
        { // Необходимо найти новый максимальный элемент
          s2=0; // Сброс сигнала
          if(RDS_ARRAYEXISTS(pM2)) // Матрица M2 не пуста
            { // Вспомогательные переменные
              int m2count;
              double *m2data;
              // Число элементов в матрице
              m2count=RDS_ARRAYROWS(pM2)*RDS_ARRAYCOLS(pM2);
              // Получить указатель на первый элемент
              m2data=(double*)RDS_ARRAYDATA(pM2);
              // Поиск максимального элемента
              M2max=DoubleErrorValue;
              for(int i=0;i<m2count;i++)
                if(m2data[i]!=DoubleErrorValue &&
                   (M2max==DoubleErrorValue || M2max<m2data[i]))
                  M2max=m2data[i];
            }
          else // Матрица M2 пуста
            M2max=DoubleErrorValue;
        } // if(s2)
      // Значения M1min и M2max определены -
      // вычисление произведения
      if(M1min!=DoubleErrorValue && M2max!=DoubleErrorValue)
        y=M1min*M2max;
      else
        y=DoubleErrorValue;
      break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef y
#undef M2max
```

```
#undef M1min
#undef pM2
#undef pM1
#undef s2
#undef s1
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

При вызове модели с параметром `RDS_BFM_MODEL` прежде всего проверяется сигнал `s1`. Если значение `s1` ненулевое, значит, в конце предыдущего такта расчета сработала связь, соединенная со входом `M1`, в результате чего вход получил новое значение. В этом случае переменной `s1` присваивается 0 (сигнал подготавливается к следующему срабатыванию) и, если матрица `M1` не пуста, производится поиск минимального элемента, который записывается во внутреннюю переменную блока `M1min`. Если матрица пуста, переменной `M1min` присваивается значение-индикатор ошибки из глобальной переменной `DoubleErrorValue` (подразумевается, что эта переменная инициализирована в главной функции DLL, как указано на стр. 44).

Далее проверяется сигнал `s2`. Его ненулевое значение сигнализирует об изменении значения матрицы `M2` и необходимости вычисления нового значения ее максимального элемента. В этом случае сигнал `s2` сбрасывается, и найденный максимальный элемент матрицы записывается во внутреннюю переменную `M2max`. Затем произведение `M1min` и `M2max` присваивается выходу блока `y`, если ни одна из этих двух внутренних переменных не равна значению `DoubleErrorValue`, указывающему на ошибку вычисления (максимальный или минимальный элемент матрицы найти не удалось – одна из матриц пуста или во всех элементах одной из матриц находится значение-индикатор ошибки).

В результате единственная операция, выполняющаяся в модели при любом запуске – это вычисление произведения `M1min` и `M2max`. Перебор элементов матрицы `M1` выполняется только при срабатывании подключенной к ней связи, так же как и перебор элементов `M2`. Если значения на одном из входов блока меняются редко, такая модель позволит получить выигрыш в быстродействии, поскольку вычисления для этого входа будут выполняться только при его изменениях. Если бы у блока не было связанных сигнальных переменных и

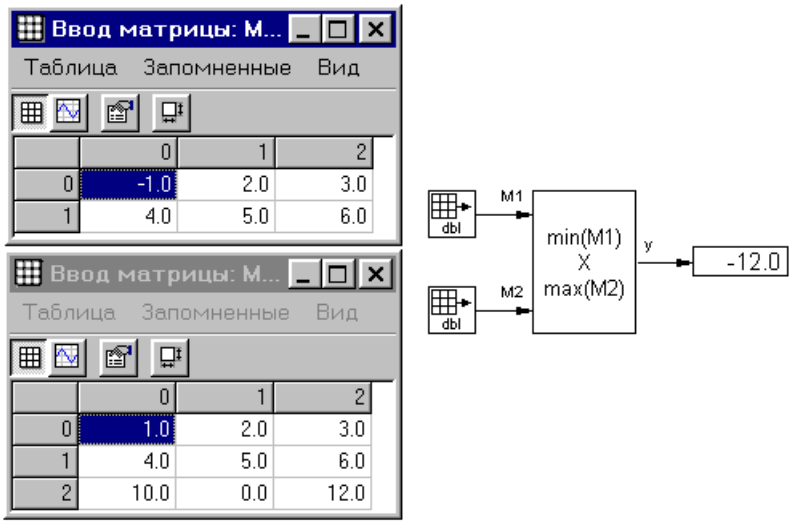


Рис. 34. Блок, вычисляющий произведение минимального элемента одной матрицы и максимального элемента другой

модель запускалась бы при срабатывании любой связи, ей пришлось бы каждый раз перебирать элементы обеих матриц, независимо от того, какая из них изменилась.

Проверить работу созданного блока можно при помощи схемы, изображенной на рис. 34. В процессе расчета при любом изменении значений в обеих входных матрицах на числовом индикаторе должно отображаться произведение минимального элемента первой матрицы и максимального элемента второй. Если это произведение не может быть вычислено (например, если одна из матриц пуста), на индикаторе отобразится вопросительный знак.

§2.5.8. Использование выходов с управляющими переменными

Описывается использование управляющих переменных, которые позволяют запретить передачу по связям значения конкретного выхода блока, а также активировать связь, подключенную к конкретному элементу выхода-массива. Приведен пример модели переключателя с двумя выходами, передающего входное значение на один или на оба выхода в зависимости от значения дополнительного входа (работа неактивного выхода блокируется управляющей переменной). Также приведен пример демультиплексора на произвольное число выходов (активный выход выбирается целой управляющей переменной массива).

Связи, подключенные к выходам простого блока, срабатывают в режиме расчета только тогда, когда значение стандартного сигнала `Ready` (сигнального выхода блока со смещением 1) равно единице. Перед запуском модели блока в режиме `RDS_BFM_MODEL` РДС автоматически взводит этот сигнал, поэтому, если модель не предпримет никаких действий, в конце такта расчета значения всех выходов блока будут переданы на входы других блоков, связанных с ним. Если модель присвоит переменной `Ready` нулевое значение, ни одна из связей, соединенных с выходами блока, не сработает. Чаще всего модели сбрасывают `Ready`, если значения выходов блока не изменились, поскольку в этом случае нет никакого смысла передавать по связям те же самые данные еще раз. Повторная передача данных не приведет к возникновению каких-либо ошибок, но из-за срабатывания связей могут запуститься модели блоков, ко входам которых эти связи подключены. Поскольку входные данные этих моделей не изменились, их запуск был бы напрасной тратой времени.

Разрешение и запрещение передачи данных отдельных выходов по связям часто требуется при создании блоков-выключателей и демультиплексоров, управляющих передачей данных со входа на один или несколько выходов. Использование сигнала `Ready` позволяет управлять только всеми выходами одновременно: или сработают все связи, или не сработает ни одна из них. Для переключателей с единственным выходом (см. пример на стр. 66) это вполне подходит, однако, если выходов несколько, бывает необходимо управлять ими независимо – разрешать передачу данных для одних выходов, запрещая при этом передачу других.

Для управления передачей данных конкретного выхода блока необходимо ввести дополнительную логическую переменную и связать с ней выход, задав для него тип “выход / логическая” вместо “выход” и указав имя логической переменной (см. стр. 19). При этом связи, подключенные к этому выходу, будут передавать данные только в том случае, если связанная логическая переменная будет иметь значение 1. Разумеется, сигнал `Ready` также должен быть равен единице, иначе ни одна выходная связь блока не сработает, какие бы значения не имели управляющие логические переменные выходов. Если выход блока – массив, можно вместо логической управляющей переменной указать для него целую. Целая переменная будет управлять только связями, подключенными к отдельным элементам массива, никак не влияя на связи, подключенные ко всему массиву как к одной сложной переменной. Значение переменной будет определять номер элемента массива, для которого разрешена передача данных. Если переменная будет иметь значение 0, будут работать только

связи, присоединенные к нулевому элементу массива, если она будет равна единице – только связи, присоединенные к первому элементу и т.д.

Для примера сначала рассмотрим блок, который должен передавать данные вещественного входа `x` на один из выходов `y0` и `y1` в зависимости от значения переменной `N`: при нулевом `N` значение должно передаваться на выход `y0`, при `N`=1 – на выход `y1`, при `N`=2 – на оба выхода (рис. 35). Для этого блока потребуется раздельное управление выходами, поэтому придется ввести в него две дополнительных логических переменных `L0` и `L1`, которые будут управлять выходами `y0` и `y1` соответственно:

Рис. 35. Управление отдельными выходными связями блока

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Логика*
0	Start	Сигнал	1	Вход	
1	Ready	Сигнал	1	Выход	
2	x	double	8	Вход	
10	N	int	4	Вход	
14	L0	Логическая	1	Внутренняя	
15	L1	Логическая	1	Внутренняя	
16	y0	double	8	Выход/логическая	L0
24	y1	double	8	Выход/логическая	L1

Модель блока будет выглядеть так:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestSW2(int CallMode,
                      RDS_PBLOCKDATA BlockData,
                      LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define x       (*((double *)(pStart+2)))
#define N       (*((int *)(pStart+10)))
#define L0      (*((char *)(pStart+14)))
#define L1      (*((char *)(pStart+15)))
#define y0      (*((double *)(pStart+16)))
#define y1      (*((double *)(pStart+24)))
  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDILLDD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        switch(N)
          { case 0: // Передать данные на выход y0
              y0=x;
              L0=1; // Разрешить y0
```

```
                L1=0; // Запретить y1
                break;
              case 1: // Передать данные на выход y1
                y1=x;
                L0=0; // Запретить y0
                L1=1; // Разрешить y1
                break;
              default: // Передать данные на оба выхода
                y0=y1=x;
                L0=L1=1; // Разрешить оба выхода
            }
          break;
      }
    return RDS_BFR_DONE;
  // Отмена макроопределений
  #undef y1
  #undef y0
  #undef L1
  #undef L0
  #undef N
  #undef x
  #undef Ready
  #undef Start
  #undef pStart
  }
  //=======================================
```

В режиме `RDS_BFM_MODEL` модель выполняет разные действия в зависимости от значения входа `N`. Если значение `N` равно нулю, выходу `y0` присваивается значение входа `x`, после чего логической переменной `L0` присваивается единица, а `L1` – ноль. При таких значениях управляющих переменных сработает только связь, присоединенная к выходу `y0`, которым управляет `L0`. Если значение `N` равно единице, все происходит наоборот – переменной `L0` присваивается 0, а `L1` – единица, при этом будет работать только связь, присоединенная к `y1`. Если же `N` имеет какое-либо другое значение, обеим управляющим переменным присваивается значение 1, что разрешает работу связей, присоединенных к обоим выходам блока.

Теперь рассмотрим пример, в котором целая переменная будет управлять передачей данных элементов массива. Предположим, что необходимо создать модель блока-демультиплексора, который будет передавать значение вещественного входа `x` на выход, номер которого определяется целым входом `N`. Поскольку заранее неизвестно, сколько выходов должно быть у блока, его выходом будет вещественный массив `Y`, к элементам которого будут подключаться связи (рис. 36). Чтобы для любого значения `N` срабатывала только связь, соединенная с элементом `Y[N]`, целая переменная `N` будет указана в качестве управляющей для массива `Y`:

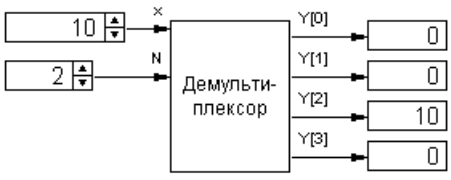


Рис. 36. Управление передачей отдельных элементов массива

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Логика*
0	Start	Сигнал	1	Вход	
1	Ready	Сигнал	1	Выход	
2	x	double	8	Вход	
10	N	int	4	Вход	
14	Y	Массив double	8	Выход/логическая	N

В данном случае управляющая массивом `Y` переменная `N` одновременно является входом блока, поэтому в модели не нужно отдельно управлять номером выхода, связь которого будет работать – это произойдет автоматически при поступлении на вход `N` нового значения. Модель блока будет выглядеть следующим образом:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestSW(int CallMode,
                     RDS_PBLOCKDATA BlockData,
                     LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define x       (*((double *)(pStart+2)))
#define N       (*((int *)(pStart+10)))
#define pY      ((void **)(pStart+14))
  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDIMD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        if(N<0) // Значение N не должно быть отрицательным
          Ready=0; // Не передавать ничего по связям
        else // Значение N не отрицательно
          { int count;
            double *array;
            // Число элементов в массиве Y
            count=RDS_ARRAYEXISTS(pY)?RDS_ARRAYCOLS(pY):0;
            if(N>=count) // Число элементов недостаточно
              { // Увеличение размера Y
                if(!rdsResizeVarArray(pY,1,N+1,TRUE,NULL))
                  { // Ошибка: не удалось увеличить размер массива
                    rdsStopCalc(); // Остановка расчета
                    // Не передавать ничего по связям
                    Ready=0;
                    // Вывод сообщения
                    rdsMessageBox("Мало памяти",
                                  "Ошибка",
                                  MB_OK | MB_ICONERROR);
                    return RDS_BFR_DONE;
                  }
              }
```

```
                // Получить указатель на первый элемент Y
                array=(double*)RDS_ARRAYDATA(pY);
                // Записать в Y[N] значение входа
                array[N]=x;
              } // else (N>=0)
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef pY
    #undef N
    #undef x
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Поскольку у массива не может быть элементов с отрицательными индексами, модель в режиме `RDS_BFM_MODEL` сначала проверяет неотрицательность значения `N`. Если `N` отрицательно, сигнал `Ready` обнуляется и работа модели на этом завершается. В противном случае текущее число элементов массива `Y` записывается во вспомогательную переменную `count` и сравнивается с `N` (поскольку число строк массива всегда равно единице, число элементов в нем всегда равно числу столбцов и вычисляется при помощи макроса `RDS_ARRAYCOLS`). Если значение `N` больше или равно числу элементов в `Y`, значит, элемент `Y[N]` еще не существует, и размер массива нужно увеличить до `N+1`. Для этого вызывается сервисная функция `rdsResizeVarArray` и проверяется возвращаемое ей значение. Если размер массива увеличить не удалось (из-за нехватки памяти или слишком большого значения `N`), функция вернет `FALSE`. В этом случае модель остановит расчет, вызвав функцию `rdsStopCalc`, присвоит сигналу `Ready` значение 0 и выведет сообщение “Мало памяти” при помощи функции `rdsMessageBox`. Обнуление сигнала `Ready` может показаться лишним, поскольку расчет все равно будет остановлен, однако остановка расчета произойдет только после завершения текущего такта моделирования. Если не обнулить `Ready`, значение выхода `Y` будет передано по связям в конце такта, что нежелательно.

После того, как модель удостоверилась в существовании элемента массива `Y[N]` (или создала его, увеличив размер `Y`), этому элементу присваивается значение входа `x`. Поскольку переменная `N` указана как управляющая для выхода `Y`, только это значение будет передано по связям в конце такта расчета.

§2.6. Динамические переменные

Описывается работа с динамическими переменными, то есть с переменными, которые модели блоков создают и уничтожают в процессе работы. Модели могут создавать такие переменные в корневой или родительской подсистеме блока, поэтому несколько блоков могут получать доступ к одной и той же динамической переменной и использовать ее для связи.

§2.6.1. Использование динамических переменных

Описываются общие принципы работы с динамическими переменными – их создание, уничтожение, получение доступа к созданным (“подписка”). Описаны основные сервисные функции РДС, относящиеся к динамическим переменным.

Динамические переменные позволяют нескольким блокам обмениваться данными через общую для этих блоков область памяти, содержащую переменную одного из используемых в РДС типов. В отличие от передачи данных по связям через статические входы и выходы, которая определяется пользователем, соединяющим выходы одних блоков

со входами других, передача данных через динамические переменные задается на этапе проектирования модели блока. РДС не позволяет пользователю, не разрабатывающему модели блоков, самостоятельно создавать динамические переменные в произвольном блоке и указывать, с какими переменными должен быть связан блок – все эти действия только выполняются моделями блоков при помощи различных сервисных функций. Иногда модель дает пользователю возможность указать имя динамической переменной, используемой для связи с другими блоками (например, в окне настроек блока-графика можно указать имя переменной, из которой блок берет значение времени), но, чаще всего, обмен данными через динамические переменные скрыт от пользователя и не требует от него никаких действий. Это позволяет организовать взаимодействие большого числа блоков, не загромождая схему многочисленными связями. Кроме того, передача данных по связям работает только в режиме расчета, в то время как данные динамических переменных доступны блокам во всех режимах.

Для того, чтобы блоки могли работать с динамической переменной, один из них должен ее создать, а остальные – подписаться на нее, то есть найти динамическую переменную по имени и типу и, если такая переменная существует, получить к ней доступ. Динамическая переменная всегда находится в каком-либо блоке, при этом чаще всего не в том, который ее создал. Механизм подписки устроен таким образом, что блок может получить доступ к своей динамической переменной или к переменной любой из родительских подсистем в иерархии, начиная от непосредственного родителя и заканчивая корневой подсистемой. Таким образом, если простой блок создаст динамическую переменную в себе самом, никто кроме него не сможет на нее подписаться, поскольку он не может являться ничьим родителем, то есть внутри него нет других блоков. Обычно простые блоки создают динамические переменные либо в родительской, либо в корневой подсистеме. Если блок создал переменную в родительской подсистеме, на нее смогут подписаться другие блоки этой же подсистемы и блоки вложенных в нее подсистем. Если блок создал переменную в корневой подсистеме, к ней смогут получить доступ все блоки схемы.

Подсистемы, в отличие от простых блоков, иногда создают динамические переменные, принадлежащие им самим, для передачи данных во вложенные блоки. Кроме того, собственные динамические переменные могут использоваться для управления элементами векторной картинки в подсистемах и внешних входах и выходах. В простом блоке для этой цели можно создать несколько внутренних статических переменных, но все остальные типы блоков лишены такой возможности – структура их переменных не может быть задана произвольно. Для того, чтобы связать элемент картинки с собственной динамической переменной, необходимо указать ее имя с префиксом “$DYN.”. Например, если для управления элементом должна использоваться динамическая переменная с именем “`Value`”, при редактировании этого элемента необходимо ввести в соответствующее поле текст “`$DYN.Value`”. При необходимости, элементы картинки блока можно также связать с динамической переменной его родительской подсистемы (используется префикс “$PARENT.”) или с первой встреченной в иерархии подсистем переменной с заданным именем (используется префикс “$SEARCH.”). На рис. 37 изображено окно настройки системы координат в редакторе векторной картинки блока, в котором координаты этой системы связаны с динамическими переменными `ObjX` и `ObjY` в родительской подсистеме блока, угол поворота системы координат – с динамической переменной `Angle` в самом блоке, а масштаб – с первой встреченной динамической переменной с именем `GlobK` в иерархии родительских подсистем.

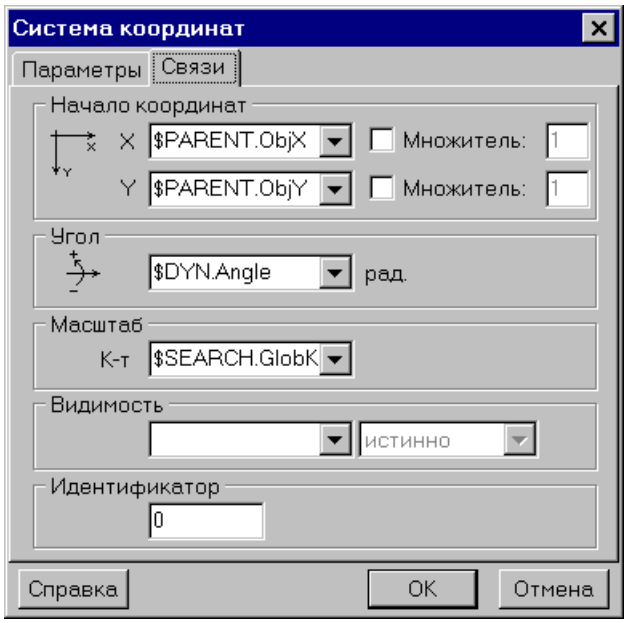


Рис. 37. Связь элементов векторной картинки
блока с динамическими переменными

Чтобы получить доступ к какой-либо динамической переменной, блок должен на нее подписаться. Это касается и переменных, созданных самим блоком – сервисная функция, создающая динамическую переменную, автоматически подписывает на нее блок-создатель. При подписке указывается имя переменной, ее тип, а также блок, в котором необходимо ее найти (данный блок, родительская подсистема или корневая подсистема). В результате, независимо от того, найдена переменная или нет, РДС создает структуру `RDS_DYNVARLINK`, содержащую информацию о переменной, и возвращает указатель на нее модели блока. Структура `RDS_DYNVARLINK` описана в файле “RdsDef.h” следующим образом:

```
typedef struct
{ LPVOID Data;              // Указатель на область данных
  LPSTR  VarName;           // Имя переменной
  LPSTR  VarType;           // Строка типа переменной
  RDS_BHANDLE Provider;     // Блок-владелец
  LPVOID UID;               // Служебный идентификатор переменной
  RDS_VHANDLE Var;          // Идентификатор переменной для
                            // сервисных функций
} RDS_DYNVARLINK;
typedef RDS_DYNVARLINK *RDS_PDYNVARLINK; // Указатель на структуру
```

- `LPVOID Data` – указатель на область данных переменной. Структура этой области данных в точности соответствует данным такой же статической переменной в дереве переменных блока (см. §2.5). Например, для динамической переменной типа `double` поле `Data` будет указывать на восьмибайтовую область памяти, содержащую вещественное число, для строки – на четырехбайтовую область, содержащую указатель на первый символ строки и т.п. (см. рис. 38). Если переменная не найдена, значение этого поля равно `NULL`.
- `LPSTR VarName` – строка имени переменной.
- `LPSTR VarType` – строка типа переменной (такая же, как и у статических переменных, см. стр. 22).
- `RDS_BHANDLE Provider` – идентификатор блока, в котором располагается найденная переменная.

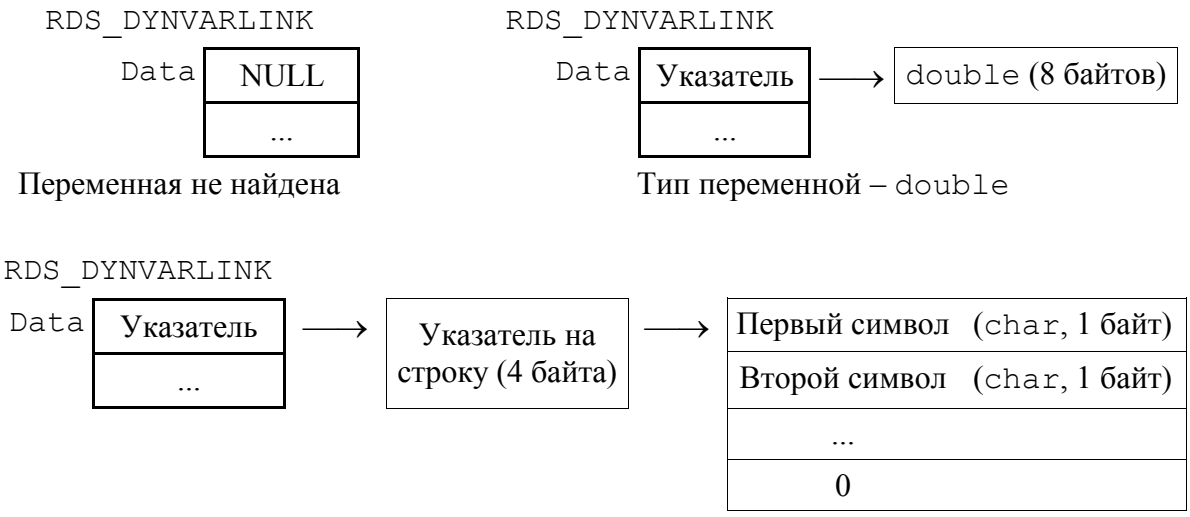


Рис. 38. Размещение в памяти динамических переменных разных типов

- `LPVOID UID` – уникальный идентификатор динамической переменной (служебная, используется внутри РДС).
- `RDS_VHANDLE Var` – идентификатор переменной, используемый в некоторых сервисных функциях.

Если переменная с заданным именем и типом не существует, РДС запомнит факт подписки, и, как только такая переменная будет создана каким-либо блоком, подписавшийся блок немедленно получит к ней доступ. Структура `RDS_DYNVARLINK` создается один раз при подписке на переменную, после чего РДС самостоятельно обновляет значение ее поля `Data` при создании и удалении соответствующей динамической переменной. Перед каждым использованием переменной модель блока должна проверять ее существование: если в поле `Data` находится значение `NULL`, значит, запрошенная переменная еще не создана или уже удалена. Таким образом, при создании моделей блоков нужно иметь в виду, что, в отличие от статической переменной, динамическая может появляться и исчезать в процессе расчета.

Для подписки на динамическую переменную используется сервисная функция `rdsSubscribeToDynamicVar`:

```
RDS_PDYNVARLINK RDSCALL rdsSubscribeToDynamicVar(
      int Block,       // В каком блоке искать переменную
      LPSTR VarName,   // Имя переменной
      LPSTR VarType,   // Строка типа переменной
      BOOL Search);    // Искать по иерархии
```

Функция принимает следующие параметры:

- `Block` – одна из трех констант `RDS_DV*`, определяющая, в каком блоке нужно искать переменную (`RDS_DVSELF` – в вызвавшем функцию блоке, `RDS_DVPARENT` – в родительской подсистеме, `RDS_DVROOT` – в корневой подсистеме);
- `VarName` – имя переменной;
- `VarType` – строка типа переменной (строится так же, как и для статических переменных, см. стр. 22);
- `Search` – нужно ли искать переменную в цепочке родительских подсистем, если она не найдена в блоке, указанном в параметре `Block`.

Функция `rdsSubscribeToDynamicVar` возвращает указатель на созданную РДС структуру подписки `RDS_DYNVARLINK`. Эта структура будет находиться в памяти до тех

пор, пока блок-подписчик не откажется от подписки на данную переменную или не будет удален. В параметрах VarName и VarType указывается соответственно имя динамической переменной и строка ее типа. Как и имена статических переменных, имена динамических чувствительны к регистру и не должны содержать некоторых специальных символов (знака доллара, точек, запятых и скобок). Строка типа для динамических переменных строится по тому же принципу, что и строка типа статических. Например, если необходимо подписаться на переменную Var1 типа double, следует передать в параметре VarName строку "Var1" и в параметре VarType строку "D". Подписка будет успешной, если будет найдена переменная с указанным именем и типом, при этом переменная с тем же именем, но другим типом будет проигнорирована. Блоку-подписчику не нужно следить за типом динамической переменной – если поле Data структуры RDS_DYNVARLINK не равно NULL, значит, найдена переменная, тип которой в точности соответствует запросу.

Если подписка на указанную в параметрах функции переменную принципиально невозможна (например, указано недопустимое имя переменной), функция вернет значение NULL.

Блок-владелец переменной, на которую необходимо подписаться, задается параметром Block. В этом параметре может быть передана одна из трех констант, описанных в файле "RdsDef.h": RDS_DVSELF (искать переменную в блоке, модель которого вызвала функцию rdsSubscribeToDynamicVar), RDS_DVPARENT (искать в родительской подсистеме) или RDS_DVROOT (искать в корневой подсистеме). Если переменная будет найдена в указанном блоке, в поле Data структуры RDS_DYNVARLINK будет записан указатель на ее область данных. Если же переменная в указанном блоке не найдена, дальнейшие действия РДС определяются параметром Search, разрешающим или запрещающим поиск переменной вверх по иерархии. Если в Search передано значение FALSE, полю Data структуры подписки будет присвоено значение NULL и работа функции rdsSubscribeToDynamicVar на этом завершится. Если же в Search передано TRUE, РДС попытается найти переменную в подсистеме, родительской по отношению к блоку, указанному в параметре Block. Если и в этой подсистеме указанная переменная не будет обнаружена, будет сделана попытка найти ее в родительской подсистеме этой подсистемы и так далее, пока, перебирая подсистемы вверх по иерархии, РДС не доберется до корневой подсистемы. Только если и в корневой подсистеме переменная не будет найдена, полю Data структуры RDS_DYNVARLINK будет присвоено значение NULL. Таким образом, если параметр Search равен TRUE, РДС попытается найти динамическую переменную в ближайшей к указанному блоку родительской подсистеме. Если параметр Block имеет значение RDS_DVROOT, значение параметра Search не имеет значения – у корневой подсистемы нет родительской и, при отсутствии динамической переменной, ее больше негде будет искать.

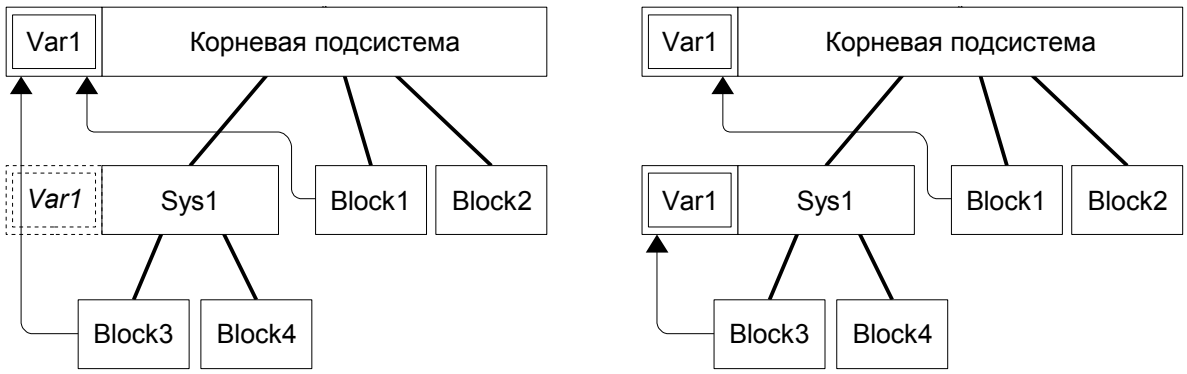
Подписка на динамическую переменную с поиском вверх по иерархии подсистем применяется достаточно часто, поскольку дает вложенным блокам какой-либо подсистемы возможность получить доступ к переменным этой подсистемы независимо от глубины вложенности этих блоков. С помощью этого механизма, например, стандартные блоки для динамического расчета получают значение времени. Блок, управляющий динамическим расчетом (так называемый планировщик) создает в подсистеме, в которой он находится, динамическую переменную "DynTime" типа double и изменяет ее значение согласно шагу расчета, заданной синхронизацией с реальным временем и другими своими параметрами. Блоки, выполняющие динамический расчет, подписываются на эту переменную с поиском по иерархии. В результате, любой блок вложенной подсистемы всегда имеет доступ к текущему значению времени, если где-нибудь в цепочке его родительских подсистем находится блок-планировщик. В схеме может независимо работать несколько планировщиков, каждый из которых будет предоставлять доступ к текущему значению времени блокам своей

подсистемы и вложенных в нее подсистем. Это дает возможность, например, моделировать какие-либо процессы с разным шагом по времени в разных подсистемах, что часто бывает полезно при расчете динамических систем.

Если при подписке с поиском по иерархии была найдена какая-либо переменная, РДС продолжает следить за цепочкой подсистем, и, если в более близкой подсистеме появится переменная с тем же именем и типом, блок-подписчик будет переключен на нее.

Пусть, например, в корневой подсистеме находится динамическая переменная Var1 типа `double` (рис. 39а). Модели блоков Block1 в корневой подсистеме и Block3 в подсистеме Sys1 вызвали сервисную функцию `rdsSubscribeToDynamicVar` со следующими параметрами:

```
rdsSubscribeToDynamicVar(RDS_DVPARENT,"Var1","D",TRUE);
```



а) Блоки Block1 и Block3 подписаны на переменную Var1 с поиском по иерархии

б) В подсистеме Sys1 появилась переменная Var1, и блок Block3 получил к ней доступ

Рис. 39. Подписка на динамическую переменную с поиском по иерархии

то есть оба блока запросили подписку на динамическую переменную Var1 типа `double` (строка “D”) в родительской подсистеме (константа `RDS_DVPARENT`) с поиском по иерархии. Родительской подсистемой для блока Block2 является корневая, поэтому он сразу же получит доступ к динамической переменной Var1 корневой подсистемы (функция `rdsSubscribeToDynamicVar` запишет в поле `Data` структуры подписки `RDS_DYNVARLINK` указатель на область данных Var1). Блок Block3 находится в подсистеме Sys1, в которой нет запрашиваемой переменной. Поскольку при вызове сервисной функции был указан поиск по иерархии, функция продолжит искать переменную с указанным именем и типом, найдет ее в корневой подсистеме и запишет в поле `Data` структуры подписки указатель на ее область данных. Таким образом, блоки Block1 и Block3 получили доступ к одной и той же динамической переменной, находящейся в корневой подсистеме. При этом РДС запомнит тот факт, что блок Block3 получил доступ к переменной не в той подсистеме, в которой он его запрашивал.

Теперь представим себе, что какой-либо блок создал динамическую переменную типа `double` с именем “Var1” в подсистеме Sys1. Эта переменная удовлетворяет условиям запроса на подписку, сделанного блоком Block3, и при этом находится ближе к нему по иерархии, чем переменная Var1 корневой подсистемы, на которую Block3 подписан в данный момент. В результате РДС переключит этот блок на использование переменной Var1 подсистемы Sys1 вместо одноименной переменной корневой подсистемы (рис. 39б). При этом от модели блока Block3 не потребуется никаких действий – все произойдет автоматически. В структуре подписки `RDS_DYNVARLINK`, которая была создана при вызове сервисной функции `rdsSubscribeToDynamicVar`, в поле `Data` будет записан новый

указатель, и когда модель блока Block3 в очередной раз обратится к этому полю, она считает данные переменной из подсистемы Sys1, а не из корневой подсистемы.

Если через некоторое время переменная Var1 в подсистеме Sys1 будет удалена, РДС снова изменит поле `Data` структуры подписки таким образом, чтобы оно ссылалось на ближайшую переменную с заданным именем и типом, в данном случае – снова на переменную Var1 корневой подсистемы.

Подписка на динамическую переменную предоставляет блокам доступ к общей области памяти, в которой расположены данные этой переменной, поэтому как только один из блоков присвоит переменной новое значение, все остальные смогут немедленно считать его. Однако, эти блоки не узнают об изменении переменной, пока не обратятся к ней за данными. Запись в область памяти динамической переменной производится внутри модели блока без использования каких-либо сервисных функций, и РДС никак не может ее отследить, чтобы сообщить об этом другим блокам-подписчикам.

Как правило, это не вызывает проблем, если блок-подписчик срабатывает каждый такт расчета или по таймеру. Независимо от того, изменилась ли динамическая переменная, блок будет постоянно считывать ее значение. Это надежный, но неэкономный способ слежения за динамической переменной – модель блока будет постоянно запускаться и тратить время процессора впустую. Более целесообразно заложить в модель блока, присваивающего значение динамической переменной, какой-либо механизм, позволяющий уведомить об этом всех подписчиков. Проще всего это сделать при помощи сервисной функции `rdsNotifyDynVarSubscribers`, специально предназначенной для уведомления блоков-подписчиков о возможных изменениях динамической переменной. Функция принимает единственный параметр типа `RDS_PDYNVARLINK` – указатель на структуру `RDS_DYNVARLINK`, полученный блоком при подписке на переменную, об изменении которой нужно сообщить другим блокам. При вызове этой функции модели всех блоков, подписанных на ту же самую переменную, будут вызваны в режиме `RDS_BFM_DYNVARCHANGE`, при этом через параметр модели `ExtParam` (см. §2.3) будет передан указатель на структуру подписки на переменную, созданную для вызываемого блока. Если блок подписан на несколько динамических переменных, его модель сможет выяснить, какая из них изменилась, сравнив значение `ExtParam` с указателями на их структуры. Следует обратить внимание на то, что указатель, передаваемый через `ExtParam` в модель блока-подписчика, это не тот же самый указатель, который был использован блоком, изменившим переменную, в качестве параметра функции `rdsNotifyDynVarSubscribers`. Хотя переменная одна и та же, для каждого подписавшегося на нее блока создается своя структура `RDS_DYNVARLINK`, поэтому каждый блок-подписчик получит через `ExtParam` указатель на собственную структуру подписки. Примеры использования механизма уведомления об изменении динамических переменных будут приведены далее.

Следует помнить, что модели блоков-подписчиков будут вызваны в режиме `RDS_BFM_DYNVARCHANGE` при изменении значения переменной, только если в модели блока, изменившего переменную, предусмотрен вызов сервисной функции `rdsNotifyDynVarSubscribers`. Все стандартные блоки, работающие с динамическими переменными (например, уже упоминавшийся выше планировщик динамического расчета из библиотеки “Common.dll”), поддерживают этот вызов. Однако, если создается модель блока, которая будет работать с какими-либо динамическими переменными вместе с блоками других разработчиков, необходимо узнать из описания этих блоков, поддерживается ли ими вызов функции `rdsNotifyDynVarSubscribers`. Если этот вызов не поддерживается или в описании ничего об этом не сказано, надежнее проверять значение переменной каждый такт расчета, не полагаясь на вызов модели в режиме `RDS_BFM_DYNVARCHANGE`.

Модели блоков, подписанных на динамические переменные, вызываются в режиме RDS_BFM_DYNVARCHANGE не только при вызове каким-либо блоком функции rdsNotifyDynVarSubscribers, но и при создании или удалении переменной, а также при переключении на другую переменную при подписке с поиском по иерархии (рис. 39 а-б). Эти события, в отличие от изменения значения переменной, происходят при вызове различных сервисных функций, поэтому РДС в состоянии самостоятельно уведомить о них блоки-подписчики.

Модель блока может в любой момент прекратить подписку на динамическую переменную при помощи сервисной функции rdsUnsubscribeFromDynamicVar, в которую передается указатель на структуру подписки. При вызове этой функции созданная при подписке структура RDS_DYNVARLINK уничтожается, и указатель на нее, при помощи которого осуществлялся доступ к переменной, не может быть использован в дальнейшем – теперь он указывает на освобожденную область памяти. При уничтожении блока не обязательно прекращать подписку на переменные, на которые блок подписался в процессе работы – это будет сделано автоматически. Однако, для соблюдения хорошего стиля программирования и улучшения читаемости программы лучше явно прекратить подписку на переменные при вызове модели в режиме RDS_BFM_CLEANUP.

§2.6.2. Подписка на динамическую переменную

Приводится пример модели блока, получающего доступ к стандартной динамической переменной времени DynTime, создаваемой блоком-планировщиком, и используемой большинством библиотечных блоков. Блок в примере вычисляет синус от времени, умножает его на значение входа и выдает на выход. В примере также рассматривается реакция модели на событие RDS_BFM_DYNVARCHANGE, сигнализирующее о том, что динамическая переменная изменилась.

Рассмотрим пример, в котором модель блока получит доступ к динамической переменной, созданной другим блоком. Создадим блок, вычисляющий значение $y=A\times\sin(T)$, где A – вход блока, y – выход, а T – значение текущего времени из динамической переменной “DynTime” типа double, которая создается и изменяется планировщиком динамического расчета (см. стр. 87). Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	1
1	Ready	Сигнал	1	Выход		–
2	A	double	8	Вход	✓	–
10	y	double	8	Выход		–

В параметрах блока следует отключить флаг “запуск каждый такт” (см. рис. 5) и установить флаг “пуск” для входа A – теперь блок будет автоматически запускаться при срабатывании связи, подключенной к этому входу. Также желательно дать сигналу Start начальное значение 1, чтобы блок запустился в самом первом такте расчета, вычислив значение выхода y по начальному значению A. Чтобы блок запускался и при изменении переменной “DynTime”, необходимо будет ввести в его модель реакцию на событие RDS_BFM_DYNVARCHANGE. Модель блока будет иметь следующий вид:

```
extern "C" __declspec(dllexport)
     int RDSCALL TestDynSinT(int CallMode,
                             RDS_PBLOCKDATA BlockData,
                             LPVOID ExtParam)
{
```

```
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define A       (*((double *)(pStart+2)))
#define y       (*((double *)(pStart+10)))
  // Вспомогательная переменная – указатель на структуру подписки
  RDS_PDYNVARLINK Link;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Подписка на динамическую переменную DynTime
        Link=rdsSubscribeToDynamicVar(RDS_DVPARENT, // В родителе
                                      "DynTime",    // Переменная
                                      "D",          // Тип
                                      TRUE);        // Искать
        // Запомнить указатель на структуру подписки
        BlockData->BlockData=Link;
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        // Прекратить подписку на DynTime
        rdsUnsubscribeFromDynamicVar(Link);
        break;

      // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        // Проверка существования DynTime
        if(Link!=NULL && Link->Data!=NULL)
          { // DynTime существует – привести указатель на область
            // данных переменной к типу "double*"
            double *pT=(double*)Link->Data;
            // pT – указатель на данные DynTime
            y=A*sin(*pT); // Вычисление выхода
          }
        else // DynTime не существует
          Ready=0; // Не передавать данные по связям
        break;

      // Реакция на изменение динамической переменной
      case RDS_BFM_DYNVARCHANGE:
        Start=1; // Запустить модель в следующем такте расчета
        break;
    }
  return RDS_BFR_DONE;
```

```
// Отмена макроопределений
#undef y
#undef A
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

При инициализации блока (вызове модели с параметром `RDS_BFM_INIT`) модель вызывает функцию `rdsSubscribeToDynamicVar`, запрашивая у РДС подписку на переменную “DynTime” типа `double` (строка “D”) с поиском по иерархии (параметр функции `Search` равен `TRUE`) в родительской подсистеме (константа `RDS_DVPARENT`). Таким образом, если где-нибудь между родительской подсистемой этого блока и корневой подсистемой будет находиться планировщик динамического расчета, эта модель получит доступ к созданной им переменной “DynTime”. Возвращаемый функцией указатель на созданную структуру подписки присваивается вспомогательной переменной `Link`. Для того, чтобы динамическую переменную можно было использовать в модели блока при реакции на другие события (например, при выполнении такта моделирования), необходимо как-нибудь запомнить этот указатель, поскольку вспомогательная переменная `Link`, как и любая автоматическая переменная в языке С, будет уничтожена при завершении функции модели и ее значение будет потеряно. У некоторых программистов может возникнуть соблазн объявить переменную `Link` статической, чтобы ее значение сохранялось между вызовами функции модели, но этого ни в коем случае нельзя делать. В РДС одна и та же функция вызывается для всех блоков с данной моделью, сколько бы их ни было, при этом в параметре `BlockData` передаются данные конкретного блока, для которого вызвана функция. Если объявить статической какую-либо переменную внутри функции модели, она станет общей для всех блоков с этой моделью. В данном случае это приведет к тому, что блоки будут обращаться не к тем динамическим переменным, на которые они подписались, а к переменной, на которую подписался последний из них, поскольку он присвоил общей переменной `Link` новое значение. При удалении любого из этих блоков структура подписки будет уничтожена, и все остальные блоки, пытаясь получить доступ к динамической переменной, обратятся к уже освобожденной области памяти, что с большой вероятностью вызовет ошибку общей защиты.

Для хранения данных, относящихся к конкретному блоку, в том числе и указателей на структуры подписки, чаще всего используется личная область данных блока. Обычно при инициализации блока модель отводит необходимую для личной области данных память и присваивает указатель на эту область полю `BlockData` структуры данных блока (пример модели, работающей с личной областью данных, приведен на стр. 38). При написании модели на языке C/C++ личная область данных блока обычно оформляется как объект какого-либо класса или структуры, и указатели на структуры подписки делают членами этого класса. В рассматриваемом примере модели необходимо хранить для каждого блока только один параметр – указатель на структуру подписки на переменную “DynTime”. Можно было бы описать структуру личной области данных с единственным полем типа `RDS_PDYNVARLINK` (указатель на структуру подписки), отвести память под эту структуру при инициализации блока и присвоить этому полю значение вспомогательной переменной `Link`, но в данном случае можно поступить проще – будем считать личной областью данных блока ту самую структуру подписки, указатель на которую находится в переменной `Link`. Отведением и освобождением памяти под эту структуру занимается РДС, поэтому никаких дополнительных действий в функции модели не потребуется – нужно просто присвоить значение переменной `Link` переменной `BlockData->BlockData`, предназначенной для

хранения указателя на личную область данных блока, что и делается в последнем операторе реакции модели на событие RDS_BFM_INIT.

При вызове модели в режиме RDS_BFM_CLEANUP (при отключением модели от блока, перед уничтожением блока и т.п.), блок прекращает подписку на переменную “DynTime”. Для этого вызывается функция rdsUnsubscribeFromDynamicVar, в которую передается указатель на структуру подписки. Поскольку этот указатель был сохранен в поле структуры данных блока BlockData->BlockData, которая имеет тип void* (произвольный указатель), перед вызовом функции он приводится к типу RDS_PDYNVARLINK.

При вызове модели с параметром RDS_BFM_MODEL вспомогательной переменной Link присваивается (с приведением типа) запомненный указатель на структуру подписки, хранящийся в поле BlockData структуры данных блока. Затем модель проверяет, удалось ли ей получить доступ к запрошенной переменной. Если РДС по каким-либо причинам не удалось выполнить подписку и создать соответствующую структуру (например, при нехватке памяти), значение переменной Link будет равно NULL. Если переменной “DynTime” нет ни в одной из подсистем, просмотренных в процессе поиска, значение поля Data структуры подписки (Link->Data) будет равно NULL. Если же и Link, и Link->Data имеют ненулевые значения, значит, РДС удалось найти переменную “DynTime” и записать указатель на ее область данных в поле Link->Data. В этом случае указатель на область данных переменной приводится к типу double* (указатель на double) и присваивается вспомогательной переменной pT, после чего с использованием этой переменной вычисляется значение выхода y.

Для того, чтобы значение выхода y вычислялось заново при каждом изменении значения “DynTime”, нужно каким-либо образом отслеживать эти изменения. Поскольку эта переменная формируется стандартным блоком-планировщиком, про который точно известно, что он уведомляет всех подписчиков о ее изменении, в модель можно ввести реакцию на событие RDS_BFM_DYNVARCHANGE. При вызове функции модели в этом режиме стандартному сигналу Start будет присвоено значение 1, что приведет к запуску модели в следующем такте расчета. Тогда и будет вычислено новое значение y.

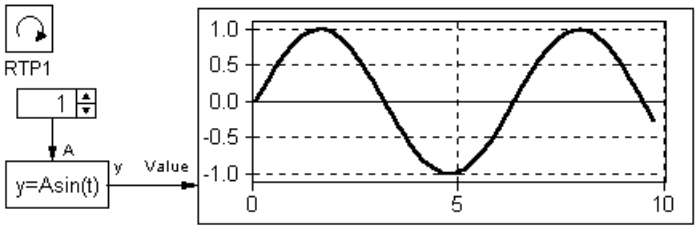


Рис. 40. Доступ к динамической переменной времени

Чтобы проверить работу блока с этой моделью, следует поместить его в схему (см. рис. 40) вместе со стандартным блоком-планировщиком (на рисунке он располагается в левом верхнем углу) и подключить ко входу блока A поле ввода, а к выходу y – график, получающий значение времени из той же самой переменной “DynTime” (это задается в параметрах графика, впрочем, по умолчанию стандартные графики работают именно с этой переменной). При запуске расчета график должен отобразить синусоиду с амплитудой, равной значению в поле ввода, подключенном ко входу A.

§2.6.3. Создание и удаление динамической переменной

Описываются сервисные функции создания и удаления динамических переменных. Приводится пример двух блоков, организующих связь между разными частями схемы при помощи динамической переменной, создаваемой в корневой подсистеме. В примере также рассматривается событие перехода в режим моделирования RDS_BFM_CALCMODE и событие запуска расчета RDS_BFM_STARTCALC.

В описанном выше примере блок получал доступ к уже существующей переменной, которая была создана другим блоком. Теперь рассмотрим процедуры создания и удаления динамических переменных.

Динамическая переменная обычно создается при помощи сервисной функции `rdsCreateAndSubscribeDV`:

```
RDS_PDYNVARLINK RDSCALL rdsCreateAndSubscribeDV(
     int Block,            // В каком блоке создать
     LPSTR VarName,        // Имя переменной
     LPSTR VarType,        // Тип переменной
     BOOL Fixed,           // Запретить удаление
     LPSTR Init);          // Начальное значение
```

Функция принимает следующие параметры:

- `Block` – как и в уже описанной функции `rdsSubscribeToDynamicVar` – одна из трех констант `RDS_DV*`, определяющая, в каком блоке создается переменная (`RDS_DVSELF` – в вызвавшем функцию блоке, `RDS_DVPARENT` – в родительской подсистеме, `RDS_DVROOT` – в корневой подсистеме);
- `VarName` – имя создаваемой переменной;
- `VarType` – строка типа создаваемой переменной (строится так же, как и для статических переменных, см. стр. 22);
- `Fixed` – разрешить удаление переменной любому блоку (`FALSE`) или только блоку, создавшему переменную (`TRUE`);
- `Init` – строка со значением переменной по умолчанию или `NULL`, если значение по умолчанию не важно.

Первые три параметра этой функции похожи на параметры функции `rdsSubscribeToDynamicVar` – они задают блок-владелец, имя и тип переменной, которую нужно создать. Параметр `Fixed`, установленный в `TRUE`, позволяет запретить удаление созданной переменной всем блокам, кроме создавшего, а через параметр `Init` можно передать строку со значением переменной по умолчанию в том же формате, который используется в окне редактирования статических переменных. Если значение `Init` будет равно `NULL`, переменная получит стандартное значение (0, пустой массив, пустая строка и т.п. в зависимости от типа переменной). При создании переменной нельзя указать поиск по иерархии, как при подписке – переменная будет создана именно в том блоке, который указан константой в параметре `Block`: в вызвавшем блоке (`RDS_DVSELF`), в родительской подсистеме (`RDS_DVPARENT`) или в корневой подсистеме (`RDS_DVROOT`). Если в указанном блоке на данный момент нет динамической переменной с указанным именем, функция создаст в нем переменную указанного типа и автоматически подпишет на нее вызывавший блок. Если переменная успешно создана, функция `rdsCreateAndSubscribeDV` возвращает указатель на структуру подписки, точно такой же, как и функция `rdsSubscribeToDynamicVar`. Если переменную создать не удалось (например, в указанном блоке уже есть переменная с указанным именем), функция вернет значение `NULL`.

Поскольку функция `rdsCreateAndSubscribeDV` автоматически подписывает вызвавший блок на созданную динамическую переменную, обращение к созданной переменной производится таким же образом, как и при обычной подписке – через структуру `RDS_DYNVARLINK`, указатель на которую возвращает функция, а точнее, через ее поле `Data`, указывающее на область данных динамической переменной. Как и модели блоков-

подписчиков, модель блока, создавшего переменную, будет реагировать на события RDS_BFM_DYNVARCHANGE, и может, при необходимости, вызывать функцию rdsNotifyDynVarSubscribers для уведомления других блоков об изменении переменной.

Иногда в старых текстах моделей блоков можно встретить создание динамических переменных при помощи устаревшей функции rdsCreateDynamicVar. Эта функция не подписывает создавший блок на созданную переменную, поэтому если модели блока необходимо сообщать подписчикам об изменении переменной и, в свою очередь, получать уведомление об изменениях, за вызовом этой функции должен следовать вызов rdsSubscribeToDynamicVar. Функция rdsCreateAndSubscribeDV объединяет эти два вызова, поэтому ее использование предпочтительнее.

Для удаления динамической переменной обычно используется сервисная функция rdsDeleteDVByLink, в которую передается указатель на структуру подписки на переменную, которую необходимо удалить. Эта функция не только удаляет переменную, но и прекращает подписку на нее для вызвавшего блока – после ее вызова структура подписки уничтожается, и указатель на нее, переданный в функцию, больше использовать нельзя. Если блоку необходимо удалить переменную, но, при этом, сохранить подписку на нее (чтобы позже получить доступ к переменной с тем же именем и типом, если ее создаст другой блок), следует воспользоваться функцией rdsDeleteDynamicVar, в параметрах которой указывается блок, в котором нужно удалить переменную, и ее имя. Следует помнить, что если при создании переменной в параметре Fixed было передано значение TRUE, ее сможет удалить только тот блок, модель которого создала эту переменную.

Рассмотрим модели двух блоков, позволяющих организовать передачу данных между разными участками схемы через динамическую переменную. Блок-передатчик с моделью TestTunnelIn создаст в корневой подсистеме динамическую переменную типа double и будет записывать в нее значение своего входа. Блок-приемник с моделью TestTunnelOut (таких блоков может быть несколько в разных местах схемы) будет выдавать значение этой динамической переменной на свой выход. Таким образом, подав какое-нибудь значение на вход блока-передатчика, его можно будет снимать с выходов блоков-приемников, не соединяя эти блоки связями. Даже если приемники и передатчик находятся в разных подсистемах, удаленных друг от друга в иерархии, приемники смогут получать данные от передатчика, поскольку динамическая переменная, через которую они связаны, находится в корневой подсистеме и доступна всем блокам схемы. В блоках-приемниках и блоках-передатчиках необходимо предусмотреть возможность задания имени связывающей их переменной, чтобы можно было организовать несколько независимых групп из передатчика и приемников, каждая из которых будет работать со своей переменной, не мешая остальным. Чтобы не перегружать этот пример лишними функциями, связанными с организацией диалога с пользователем, имя динамической переменной для связи будет читаться из комментария блока, который пользователь может изменить в режиме редактирования на вкладке “Общие” окна параметров блока (рис. 5). Такой способ настройки параметров блоков не очень удобен для пользователя, поэтому при разработке моделей для практического применения следует включать в них функцию настройки, позволяющую пользователю ввести параметры блока в отдельном окне в удобной для него форме (пример функции настройки приведен ниже, на стр. 119).

Сначала создадим модель блока-передатчика. Помимо двух стандартных сигналов, блок будет иметь единственный вещественный вход x:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	1
1	Ready	Сигнал	1	Выход		–
2	x	double	8	Вход	✓	–

Как и для большинства уже описанных блоков, для этого блока желательно отключить запуск каждый такт и установить флаг “пуск” для входа `x` и начальное значение 1 для сигнала `Start`, чтобы модель автоматически запускалась при каждом изменении `x` и при первом запуске расчета.

Блок-передатчик должен создать динамическую переменную в корневой подсистеме и записывать в нее значения своего входа `x`, причем имя создаваемой переменной необходимо считать из комментария блока. Пользователь может в любой момент остановить расчет, перейти в режим редактирования и изменить комментарий, поэтому блок должен проверять, совпадает ли имя переменной, с которой он в данный момент работает, с текстом комментария блока. Если комментарий изменился, блок должен удалить старую динамическую переменную и создать новую, с именем, взятым из нового комментария. Поскольку пользователь может изменить комментарий блока только в режиме редактирования, можно производить проверку соответствия имени переменной комментарию при входе в режим моделирования (событие `RDS_BFM_CALCMODE`) или при запуске расчета (событие `RDS_BFM_STARTCALC`). Кажется логичным вставить в модель реакцию на одно из этих событий, в которой имя динамической переменной будет сравниваться с комментарием блока, и, если они не совпадают, созданная переменная будет уничтожаться и вместо нее будет создаваться новая. Однако, в такой конструкции модели скрывается не совсем очевидная проблема, на которой следует остановиться подробнее.

Допустим, в схеме находятся два блока-передатчика: Block1, работающий с переменной Var1, и Block2, работающий с переменной Var2. Эта схема некоторое время работала в режиме расчета, после чего пользователь перешел в режим редактирования и изменил комментарий блока Block1 на “Var2”, а блока Block2 – на “Var1”, то есть поменял переменные блоков местами. При запуске расчета сначала вызовется модель блока Block1. Она сравнит имя переменной блока “Var1” с новым текстом комментария “Var2” и обнаружит, что они не совпадают. Модель удалит переменную Var1 и попытается создать в корневой подсистеме переменную с именем “Var2”. Однако, это ей не удастся, поскольку переменная Var2 уже есть в корневой подсистеме – ее создала модель блока Block2, которая еще не вызывалась и не знает о том, что ей тоже нужно удалить старую переменную и создать новую. Модель блока Block1 завершится, так и не создав новую переменную, после чего вызовется модель блока Block2, которая сработает так, как и предполагалось – переменная Var1 уже удалена и ничто не помешает модели создать переменную с таким именем. В результате из двух блоков-передатчиков останется работоспособным только один – тот, модель которого была вызвана позже.

У этой проблемы есть простое решение – необходимо разнести по времени моменты удаления старых переменных и создания новых. Когда блоки-передатчики начнут создавать новые переменные, все старые переменные всех блоков должны быть уже удалены, тогда они не помешают созданию новых с теми же именами. Этого можно добиться, удаляя переменные в реакции на переход в режим моделирования и создавая новые в реакции на запуск расчета. Из режима редактирования невозможно попасть в режим расчета, минуя режим моделирования. Если пользователь нажмет кнопку “Запуск расчета”, находясь в режиме редактирования, РДС все равно сначала перейдет в режим моделирования (при этом все модели блоков будут вызваны с параметром `RDS_BFM_CALCMODE`), и только после этого запустит расчет (модели всех блоков будут вызваны с параметром `RDS_BFM_STARTCALC`). В результате, на момент реакции модели на событие `RDS_BFM_STARTCALC`, все старые

переменные должны быть уже удалены в процессе реакции моделей всех блоков на событие RDS_BFM_CALCMODE, которое произошло раньше.

С учетом изложенного выше, модель блока-передатчика будет выглядеть следующим образом:

```
extern "C" __declspec(dllexport)
     int RDSCALL TestTunnelIn(int CallMode,
                              RDS_PBLOCKDATA BlockData,
                              LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define x       (*((double *)(pStart+2)))
  // Вспомогательная переменная - указатель на структуру подписки
  RDS_PDYNVARLINK Link;
  // Вспомогательная переменная - структура описания блока
  RDS_BLOCKDESCRIPTION Descr;

  switch(CallMode)
    { // Очистка
      case RDS_BFM_CLEANUP:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        // Удалить созданную динамическую переменную
        rdsDeleteDVByLink(Link);
        break;

      // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Переход в режим моделирования
      case RDS_BFM_CALCMODE:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        if(Link!=NULL) // Динамическая переменная была создана
          { // Заполнить структуру описания блока
            Descr.servSize=sizeof(Descr);
            rdsGetBlockDescription(BlockData->Block,&Descr);
            // Сравнить имя переменной с текстом комментария
            if(strcmp(Link->VarName,Descr.BlockComment)!=0)
              { // Имя переменной не совпадает с комментарием -
                // переменная будет удалена
                rdsDeleteDVByLink(Link);
                // Очистить запомненный указатель на структуру
                // подписки
                BlockData->BlockData=NULL;
              }
          }
        break;

      // Запуск расчета
      case RDS_BFM_STARTCALC:
```

```
        if(BlockData->BlockData!=NULL) // Переменная была создана
          break;
        // Динамической переменной нет (не было или удалена
        // в реакции на RDS_BFM_CALCMODE) - надо создать новую.
        // Сначала надо получить комментарий блока
        Descr.servSize=sizeof(Descr);
        rdsGetBlockDescription(BlockData->Block,&Descr);
        if(*Descr.BlockComment==0) // Комментарий пуст
          break;
        // Комментарий блока - не пустая строка. Создаем
        // переменную.

        Link=rdsCreateAndSubscribeDV(RDS_DVROOT,
                                     Descr.BlockComment,
                                     "D",
                                     TRUE,
                                     NULL);
        // Запомнить новый указатель на структуру подписки
        BlockData->BlockData=Link;
        // В конце реакции на запуск расчета нет оператора break,
        // поэтому сразу после нее выполнится реакция на
        // такт расчета (чтобы начальное значение входа
        // записалось в динамическую переменную)

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        // Проверка существования переменной
        if(Link!=NULL && Link->Data!=NULL)
          { // Переменная существует - привести указатель
            // на область данных переменной к типу "double*"
            double *pV=(double*)Link->Data;
            if(*pV!=x) // Значение входа изменилось
              { // Записать в динамическую переменную новое
                // значение входа
                *pV=x;
                // Уведомить всех подписчиков об изменении
                // переменной
                rdsNotifyDynVarSubscribers(Link);
              }
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef x
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Как и в предыдущем примере, в этой модели указатель на структуру подписки на созданную динамическую переменную будет запоминаться в поле структуры данных блока `BlockData->BlockData`, предназначенном для хранения указателя на личную область данных блока. Однако, в этом блоке создание переменной и подписка на нее будет происходить не при инициализации блока, а при запуске расчета, поэтому эта модель не

имеет реакции на событие инициализации RDS_BFM_INIT. РДС автоматически присваивает переменной BlockData->BlockData значение NULL при создании блока, что в данном случае будет означать, что динамическая переменная блока еще не создана.

Несмотря на отсутствие в модели реакции на инициализацию блока, в ней есть реакция на событие очистки RDS_BFM_CLEANUP, в которой созданная блоком динамическая переменная уничтожается при помощи функции rdsDeleteDVByLink. В эту функцию передается запомненный в BlockData->BlockData указатель на структуру подписки на созданную переменную. Если переменная так и не была создана, этот указатель будет равен NULL, и в этом случае функция rdsDeleteDVByLink не выполнит никаких действий.

При переходе РДС в режим моделирования модели всех блоков вызываются с параметром RDS_BFM_CALCMODE. По причинам, изложенным выше, именно в этот момент модель блока-передатчика должна сравнить текст комментария блока с именем созданной переменной и удалить переменную, если они не совпадают. Очевидно, что если блок до сих пор не создал динамическую переменную, эту проверку выполнять бессмысленно, поэтому сначала запомненный в BlockData->BlockData указатель на структуру подписки присваивается вспомогательной переменной Link и его значение сравнивается с NULL. Все дальнейшие действия производятся только в том случае, если Link не равно NULL, то есть у блока на данный момент есть какая-то динамическая переменная.

Текст комментария блока не содержится в структуре RDS_BLOCKDATA, указатель на которую передается в модель блока в параметре BlockData. Указатель на этот текст, как и многие другие параметры блока, можно получить только заполнив специальную структуру описания блока RDS_BLOCKDESCRIPTION при помощи сервисной функции rdsGetBlockDescription. В этой модели структура описания блока объявлена как вспомогательная переменная Descr в самом начале функции модели. Для того, чтобы записать в эту структуру параметры данного блока, необходимо вызвать функцию rdsGetBlockDescription, передав ей идентификатор блока, описание которого нужно получить (BlockData->Block), и указатель на структуру описания &Descr. Однако, сначала следует присвоить полю servSize структуры Descr размер этой структуры:

```
Descr.servSize=sizeof(Descr);
```

Такое присваивание необходимо выполнять перед вызовами большинства сервисных функций РДС, заполняющих какие-либо структуры – это позволяет избежать ошибок при работе со старыми библиотеками. По мере развития РДС в некоторые структуры добавлялись дополнительные поля, поэтому может случиться так, что какая-нибудь устаревшая библиотека вызовет сервисную функцию, передав ей структуру меньшего размера, в которой дополнительные поля еще не были предусмотрены. Если функция попытается обратиться к отсутствующим полям за пределами переданной ей структуры, это вызовет серьезные ошибки, поэтому любая сервисная функция сначала считывает значение поля servSize переданной структуры и сравнивает его с ожидаемым размером этой структуры. Если переданное значение окажется меньше ожидаемого, сервисная функция не будет пытаться обращаться к полям структуры, находящимися за пределами servSize.

После того, как функция rdsGetBlockDescription заполнит структуру описания блока Descr, указатель на текст комментария блока будет находиться в поле BlockComment этой структуры. Комментарий блока представляет собой строку символов, завершенную нулевым байтом. Технически комментарий может состоять из множества строк, разделенных кодами перевода строки, но в данном примере мы считаем, что комментарий блока должен содержать единственную строку – имя переменной. Модель должна сравнить текст комментария с именем динамической переменной, с которой в данный момент работает блок, и удалить эту переменную, если они отличаются. Указатель на имя переменной находится в поле VarName структуры подписки RDS_DYNVARLINK,

указатель на которую в данный момент хранится во вспомогательной переменной Link. Для сравнения двух строк будет использоваться стандартная функция strcmp, описанная в файле заголовка “string.h”:

```
strcmp(Link->VarName,Descr.BlockComment)
```

Если эта функция вернет значение, отличное от нуля, значит, имя переменной не совпадает с комментарием блока. В этом случае вызывается сервисная функция rdsDeleteDVByLink, которая удалит переменную и прекратит подписку на нее, после чего переменной BlockData->BlockData, в которой хранился указатель на структуру подписки, будет присвоено значение NULL.

При запуске расчета модель будет вызвана с параметром RDS_BFM_STARTCALC. К этому моменту все динамические переменные блоков-передатчиков, имена которых не совпадают с комментариями, должны быть уже удалены. Если на момент запуска расчета динамическая переменная существует (запомненный в BlockData->BlockData указатель на структуру подписки не равен NULL), значит, комментарий блока не изменялся, и никаких действий предпринимать не нужно – можно продолжать работать со старой переменной. В этом случае выполнение функции модели на этом прекращается. Если же указатель на структуру подписки нулевой, переменная либо еще не была создана, либо была удалена в реакции на включение режима моделирования из-за изменения комментария блока. В любом случае необходимо создать в корневой подсистеме переменную с именем, указанным в комментарии блока. Для доступа к комментарию используется описанная выше сервисная функция rdsGetBlockDescription, заполняющая вспомогательную структуру описания блока Descr. Получив строку комментария, имеет смысл проверить, содержится ли в ней какой-нибудь текст. Для этого первый символ комментария (*Descr.BlockComment) сравнивается с нулевым байтом, который завершает строку. Если первый же символ строки – нулевой, в строке ничего нет. Динамическая переменная должна иметь какое-нибудь имя, поэтому при пустой строке комментария модель завершается, не пытаясь создать переменную (для упрощения этого примера здесь не проверяется, является ли строка комментария блока допустимым именем переменной).

Если динамической переменной нет, и комментарий блока не пустой, модель вызывает функцию rdsCreateAndSubscribeDV, передавая ей в качестве имени переменной строку комментария блока (Descr.BlockComment). Эта функция создаст переменную типа double (строка “D”) с указанным именем в корневой подсистеме (RDS_DVROOT) и запретит ее удаление всем блокам кроме данного (значение параметра Fixed равно TRUE). Вместо строки значения по умолчанию передается NULL – значение по умолчанию в данном случае не важно, переменной сразу же будет присвоено новое значение. Возвращаемый функцией указатель на структуру подписки на созданную переменную запоминается в BlockData->BlockData.

Следует обратить внимание на то, что, в отличие от всех остальных реакций, в конце реакции на запуск расчета отсутствует оператор break. Все реакции в этой модели, как и во всех предыдущих примерах, оформлены в виде операторов case с различными константами внутри оператора switch (CallMode). При вызове функции модели с параметром CallMode, равным RDS_BFM_STARTCALC, выполняются все операторы после “case RDS_BFM_STARTCALC” до первого оператора break или до конца оператора switch. Отсутствие break в конце реакции на запуск расчета приведет к тому, что сразу после нее выполнится часть программы, отвечающая за реакцию на такт расчета. Это сделано намеренно, чтобы при запуске расчета созданной динамической переменной было немедленно присвоено значение входа блока (именно этим и занимается функция модели в реакции на такт расчета).

При запуске модели в режиме реакции на такт расчета (RDS_BFM_MODEL) во вспомогательную переменную Link записывается запомненный указатель на структуру

подписки из BlockData->BlockData, после чего проверяется, существует ли динамическая переменная. Если структура подписки была создана (значение Link не равно NULL) и ее поле Data указывает на какую-то область памяти (Link->Data!=NULL), переменная существует, и в нее можно записывать значение входа блока. В этом случае указатель на область данных переменной Link->Data приводится к типу double* и присваивается вспомогательной переменной pV – теперь для доступа к данным переменной можно использовать выражение "*pV". Если значение динамической переменной не равно значению входа блока x, ей присваивается новое значение, после чего все блоки-приемники информируются об изменении значения переменной при помощи функции rdsNotifyDynVarSubscribers.

Перейдем к созданию модели блока-приемника. Эта модель будет несколько проще – блоку-приемнику не нужно создавать и удалять динамическую переменную. Конечно, ему тоже придется следить за текстом комментария и подписываться на переменную заново при ее изменении, но, в отличие от передатчиков, которые могут мешать друг другу, пытаясь создать переменные с одинаковыми именами, приемники просто запрашивают подписку на необходимые им переменные и ждут, пока РДС не предоставит им доступ к этим переменным.

Помимо двух стандартных сигналов, блок-приемник будет иметь единственный вещественный выход y:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	y	double	8	Выход

Этот блок не будет отрабатывать такты моделирования, поэтому у него следует отключить запуск каждый такт, чтобы он не тратил время процессора впустую. Передача данных из динамической переменной на выход блока будет осуществляться в реакции модели на изменение динамической переменной.

Модель блока будет выглядеть следующим образом:

```
extern "C" __declspec(dllexport)
     int RDSCALL TestTunnelOut(int CallMode,
                               RDS_PBLOCKDATA BlockData,
                               LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define y (*((double *)(pStart+2)))
  // Вспомогательная переменная – указатель на структуру подписки
  RDS_PDYNVARLINK Link;
  // Вспомогательная переменная – структура описания блока
  RDS_BLOCKDESCRIPTION Descr;

  switch(CallMode)
   { // Очистка
      case RDS_BFM_CLEANUP:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        // Прекратить подписку на переменную
        rdsUnsubscribeFromDynamicVar(Link);
        break;
```

```
    // Проверка типа статических переменных
    case RDS_BFM_VARCHECK:
      if(strcmp((char*)ExtParam,"{SSD}")==0)
        return RDS_BFR_DONE;
      return RDS_BFR_BADVARSMSG;

    // Запуск расчета
    case RDS_BFM_STARTCALC:
      // Запомненный указатель на структуру подписки
      Link=(RDS_PDYNVARLINK)BlockData->BlockData;
      // Получение описания блока (с комментарием)
      Descr.servSize=sizeof(Descr);
      rdsGetBlockDescription(BlockData->Block,&Descr);
      // Проверка наличия комментария
      if(*Descr.BlockComment==0) // Пустая строка
        { // Прекратить подписку на переменную
          rdsUnsubscribeFromDynamicVar(Link);
          // Очистить запомненный указатель на структуру
          // подписки
          BlockData->BlockData=NULL;
          break;
        }
      // Если переменной нет (Link==NULL) или ее имя не
      // соответствует комментарию блока (strcmp... !=0),
      // нужно подписаться на новую переменную
      if(Link==NULL ||
         strcmp(Link->VarName,Descr.BlockComment)!=0)
        { // Прекратить подписку на старую переменную
          rdsUnsubscribeFromDynamicVar(Link);
          // Подписаться на новую
          Link=rdsSubscribeToDynamicVar(RDS_DVROOT,
                                        Descr.BlockComment,
                                        "D",
                                        FALSE);
          // Запомнить новый указатель на структуру подписки
          BlockData->BlockData=Link;
        }
      // В конце реакции на запуск расчета нет оператора break,
      // поэтому сразу после нее выполнится реакция на
      // изменение динамической переменной (чтобы ее значение
      // было немедленно передано на выход блока)

    // Реакция на изменение динамической переменной
    case RDS_BFM_DYNVARCHANGE:
      // Запомненный указатель на структуру подписки
      Link=(RDS_PDYNVARLINK)BlockData->BlockData;
      // Проверка существования переменной
      if(Link!=NULL && Link->Data!=NULL)
        { // Присвоить выходу блока значение динамической
          // переменной
          y=*(double*)Link->Data;
          // Взвести сигнал Ready, чтобы значение выхода
          // было передано по связям
          Ready=1;
        }
      break;
  }
```

```
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef y
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

В этой модели, как и в модели блока-передатчика, нет реакции на инициализацию блока. Чтобы можно было отслеживать изменение комментария блока пользователем, подписка на динамическую переменную для получения данных от блока-передатчика производится не при инициализации, а при реакции на запуск расчета, то есть заведомо после того, как пользователь мог изменить комментарий. Указатель на структуру подписки, так же, как и в предыдущих двух моделях, будет храниться в поле `BlockData->BlockData`, которому РДС автоматически присваивает значение `NULL` при создании блока (в данном случае это будет означать, что блок еще не подписывался на переменную). При вызове с параметром `RDS_BFM_CLEANUP` модель прекращает подписку блока на динамическую переменную, вызывая функцию `rdsUnsubscribeFromDynamicVar`, в которую передается указатель на структуру подписки из `BlockData->BlockData` (предварительно он приводится к типу `RDS_PDYNVARLINK` и присваивается вспомогательной переменной `Link`).

При запуске расчета (вызов с параметром `RDS_BFM_STARTCALC`) модель блока-приемника должна подписаться на динамическую переменную для связи с передатчиком, если она еще не подписана на нее, или если комментарий блока изменился. Запомненный в `BlockData->BlockData` указатель на структуру подписки присваивается вспомогательной переменной `Link`, после чего при помощи сервисной функции `rdsGetBlockDescription` заполняется структура описания блока `Descr`, в поле `BlockComment` которой эта функция записывает указатель на строку комментария блока (получение указателя на текст комментария было подробно описано выше, в пояснениях к модели блока-передатчика). Затем, модель проверяет, содержит ли комментарий блока какой-либо текст. Если комментарий пуст, модель не сможет продолжать работу – для подписки на динамическую переменную необходимо знать ее имя. Если первый символ строки комментария имеет код 0, значит, строка комментария пуста – в этом случае модель прекращает подписку на старую переменную при помощи функции `rdsUnsubscribeFromDynamicVar` и возвращает управление РДС (если в данный момент блок не был подписан ни на какую переменную, в функцию `rdsUnsubscribeFromDynamicVar` вместо указателя на структуру подписки будет передано значение `NULL`, и она завершится, на выполнив никаких действий).

После того, как модель установила, что комментарий блока содержит какой-то текст, она может сравнить этот текст с именем переменной, на которую подписан блок. Если блок вообще не подписан на переменную (значение `Link` равно `NULL`) или имя переменной не совпадает с текстом комментария (строки `Link->VarName` и `Descr.BlockComment` не совпадают, проверка производится при помощи функции `strcmp`), модель прекращает подписку на старую переменную, если она существовала, и подписывается на новую при помощи функции `rdsSubscribeToDynamicVar`. Переменная типа `double` (строка “D”) ищется в корневой подсистеме (константа `RDS_DVROOT`) без поиска по иерархии (параметр `Search` равен `FALSE`), в качестве имени переменной передается строка комментария блока `Descr.BlockComment`. Функция возвращает указатель на структуру подписки, который запоминается в `BlockData->BlockData` для дальнейшей работы.

Как и в модели блока-передатчика, в модели блока-приемника после реакции на запуск расчета отсутствует оператор `break`. Из-за этого сразу после этой реакции будет

выполнена часть программы модели, отвечающая за реакцию на изменение динамической переменной, что позволит немедленно передать данные переменной на выход блока, не дожидаясь уведомления от блока-передатчика.

Каждый раз, когда модель блока-передатчика вызывает функцию `rdsNotifyDynVarSubscribers` после изменения своей динамической переменной, модели блоков-приемников, подписанных на эту переменную, будут вызваны с параметром `RDS_BFM_DYNVARCHANGE`. Реагируя на это событие, модель присваивает выходу блока `y` значение динамической переменной (для этого указатель на область данных переменной сначала приводится к типу "указатель на `double`") и взводит стандартный сигнал `Ready`, что приведет к передаче данных выхода блока по связям в ближайшем такте расчета.

Для проверки работы созданных моделей следует поместить в схему блок-приемник и блок-передатчик (они могут находиться в разных подсистемах) и соединить вход передатчика с полем ввода, а выход приемника – с индикатором (рис. 41). В комментарий обоих блоков следует ввести одну и ту же строку имени переменной (например, "Var1"). При запуске расчета значения, вводимые в поле ввода, подключенное к передатчику, должны отображаться на индикаторе, подключенном к приемнику.

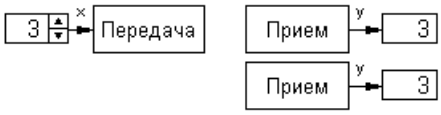


Рис. 41. Связь блоков через динамическую переменную

§2.6.4. Работа с несколькими динамическими переменными

Рассматриваются особенности работы модели блока с несколькими динамическими переменными одновременно. Приводится пример, в котором блок перемещается в окне подсистемы с заданной скоростью и в заданном направлении, при этом скорость и направление он берет из динамических переменных, созданных в подсистеме другим рассматриваемым в примере блоком, а время – из динамической переменной DynTime стандартного библиотечного блока-планировщика.

Во всех приведенных примерах каждый блок работал только с одной динамической переменной. При работе с несколькими переменными одновременно необходимо запоминать указатель на структуру подписки для каждой из них, и здесь уже не обойтись без отведения дополнительной памяти под личную область данных блока.

Рассмотрим модель блока, изображение которого должно перемещаться в окне подсистемы с заданной скоростью в заданном направлении. Значения скорости и направления блок будет получать через динамические переменные, созданные в его родительской подсистеме другим блоком – блоком управления, который записывает в эти переменные значения своих входов. Для перемещения блока также необходимо значение времени, которое блок будет получать из динамической переменной "DynTime", обслуживаемой планировщиком динамического расчета из библиотеки "Common.dll" – без значения времени невозможно будет вычислить величину перемещения блока по его скорости. Можно было бы получать значение времени у операционной системы (например, с помощью функции Windows API `GetTickCount`), но, в данном случае, удобнее работать с блоком-планировщиком, поскольку в его параметрах можно настроить шаг изменения значения времени, а также ускорить или замедлить его, если это будет необходимо.

Для того, чтобы изображение блока перемещалось в окне подсистемы, необходимо связать его горизонтальную и вертикальную координаты с какими-нибудь статическими переменными, значения которых в режимах моделирования и расчета будут определять смещение блока от заданной в режиме редактирования позиции. Например, если блок установлен в точку (10, 20), а переменные горизонтального и вертикального смещения равны 5 и 7 соответственно, в режиме редактирования блок будет по-прежнему изображаться в точке (10, 20), а в режимах моделирования и расчета – в точке (15, 27).

Будем считать, что нулевое значение угла направления соответствует движению блока вдоль горизонтальной оси вправо, а значение 90° – движению блока вдоль вертикальной оси вверх (увеличение угла направления будет соответствовать повороту блока против часовой стрелки). В окне подсистемы, как и в любом другом окне Windows, используется система координат с перевернутой вертикальной осью. Это означает, что увеличение вертикальной координаты соответствует перемещению блока вниз, а не вверх – нужно будет учитывать это при вычислении вертикального смещения блока.

Если в какой-либо момент времени блок изображается в точке (x_0, y_0) и двигается со скоростью v в направлении α (рис. 42), его координаты через промежуток времени Δt можно вычислить следующим образом:

$$\begin{cases} x_1 = x_0 + \Delta t \cdot v \cdot \cos\alpha \\ y_1 = y_0 - \Delta t \cdot v \cdot \sin\alpha \end{cases}$$

Смещение блока по вертикали вычитается из текущего значения вертикальной координаты, поскольку ось Y в окне подсистемы направлена вниз, и при положительном синусе угла направления α, когда блок двигается вверх, его вертикальная координата должна уменьшаться.

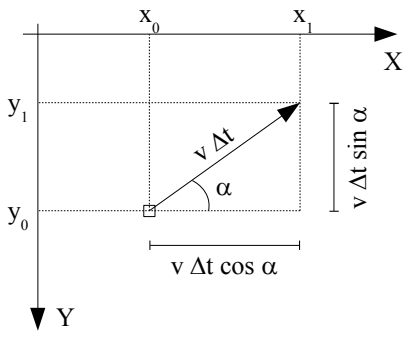


Рис. 42. Перемещение блока за промежуток времени Δt

Начнем с создания модели блока управления, на вещественные входы `Speed` и `Angle` которого поступают значения скорости и направления движения соответственно, а он, в свою очередь, передает эти значения перемещающемуся блоку через динамические переменные подсистемы. Для простоты имена динамических переменных будут жестко заданы в моделях обоих блоков: переменная для передачи скорости будет называться “BlkMove_Speed”, переменная для передачи направления – “BlkMove_Angle”. Хотя жесткое задание имен переменных ограничивает функциональность модели, не позволяя поместить в одну подсистему несколько независимо управляемых перемещающихся блоков, для данного примера этот вариант вполне подходит. Можно было бы, как в предыдущем примере, хранить имена переменных в комментарии блока, однако теперь каждому блоку необходимо две переменных, и их имена либо пришлось бы извлекать из разных строк комментария, либо формировать динамически, добавляя к тексту комментария блока суффиксы “_Speed” и “_Angle”. И то, и другое неоправданно усложнило бы пример, целью которого является демонстрация работы с несколькими динамическими переменными, а не операций со строками.

Блок управления будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	1
1	Ready	Сигнал	1	Выход		–
2	Speed	double	8	Вход	✓	–
10	Angle	double	8	Вход	✓	–

В параметрах блока следует отключить запуск каждый такт, чтобы его модель вызывалась только при срабатывании одного из входов с установленным флагом “пуск”.

Модель блока управления будет в целом похожа на модель блока-передатчика из прошлого примера. Однако, блок-передатчик создавал единственную динамическую переменную, поэтому указатель на ее структуру подписки можно было запомнить в поле `BlockData` структуры `RDS_PBLOCKDATA`, предназначенном для хранения указателя на

личную область данных блока. Новый же блок создает две переменных, поэтому ему необходимо хранить два указателя на структуры подписки. Проще всего будет описать структуру, содержащую два поля RDS_PDYNVARLINK для хранения этих указателей, и использовать такую структуру в качестве личной области данных, динамически отводя под нее память при инициализации и освобождая эту память при очистке данных блока.

Запишем модель блока управления вместе с описанием структуры личной области данных:

```
// Структура личной области данных блока
typedef struct
{ // Указатель на структуру подписки на скорость
  RDS_PDYNVARLINK VLink;
  // Указатель на структуру подписки на направление
  RDS_PDYNVARLINK ALink;
} TestBlkMoveSetterData;

// Модель блока управления
extern "C" __declspec(dllexport)
  int RDSCALL TestBlkMoveSetter(int CallMode,
                                RDS_PBLOCKDATA BlockData,
                                LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define Speed   (*((double *)(pStart+2)))
#define Angle   (*((double *)(pStart+10)))
  // Вспомогательная переменная - указатель на личную область
  // данных блока, приведенный к правильному типу
  TestBlkMoveSetterData *privdata;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Отведение памяти под личную область данных
        privdata=(TestBlkMoveSetterData*)
                 malloc(sizeof(TestBlkMoveSetterData));
        BlockData->BlockData=privdata;
        // Создание переменной для передачи скорости
        privdata->VLink=rdsCreateAndSubscribeDV(
                        RDS_DVPARENT,    // В родительской
                        "BlkMove_Speed", // Имя переменной
                        "D",             // Тип double
                        TRUE,            // Запрет удаления
                        NULL);           // Без нач.значения
        // Создание переменной для передачи направления
        privdata->ALink=rdsCreateAndSubscribeDV(
                        RDS_DVPARENT,
                        "BlkMove_Angle",
                        "D",
                        TRUE,
                        NULL);
        break;

      // Очистка данных блока
      case RDS_BFM_CLEANUP:
        // Приведение указателя на личную область данных к
```

```
        // правильному типу
        privdata=(TestBlkMoveSetterData*)(BlockData->BlockData);
        // Удаление динамических переменных
        rdsDeleteDVByLink(privdata->VLink);
        rdsDeleteDVByLink(privdata->ALink);
        // Освобождение отведенной памяти
        free(privdata);
        break;

      // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Выполнение такта моделирования
      case RDS_BFM_MODEL:
        // Приведение указателя на личную область данных к
        // правильному типу
        privdata=(TestBlkMoveSetterData*)(BlockData->BlockData);
        // Проверка существования переменной направления
        if(privdata->ALink!=NULL && privdata->ALink->Data!=NULL)
          { // Переменная существует - привести к типу double*
            double *pa=(double*)privdata->ALink->Data;
            if(*pa!=Angle) // Значение направления изменилось
              { // Записать значение в динамическую переменную
                *pa=Angle;
                // Уведомить всех подписчиков об изменении
                rdsNotifyDynVarSubscribers(privdata->ALink);
              }
          }
        // Проверка существования переменной скорости
        if(privdata->VLink!=NULL && privdata->VLink->Data!=NULL)
          { // Переменная существует - привести к типу double*
            double *pv=(double*)privdata->VLink->Data;
            if(*pv!=Speed)
              { // Записать значение в динамическую переменную
                *pv=Speed;
                // Уведомить всех подписчиков об изменении
                rdsNotifyDynVarSubscribers(privdata->VLink);
              }
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Angle
#undef Speed
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Перед текстом модели блока описана структура `TestBlkMoveSetterData`, которая будет использоваться в качестве личной области данных блока. Структура содержит два поля типа `RDS_PDYNVARLINK` – указатели на структуры подписки на динамические переменные скорости `VLink` и направления `ALink`. При вызове модели с параметром `RDS_BFM_INIT`

под эту структуру динамически отводится память при помощи стандартной функции `malloc` (описанной в “stdlib.h”), в которую передается размер структуры, полученный при помощи оператора `sizeof`. Возвращенный функцией `malloc` указатель на отведенную область памяти записывается во вспомогательную переменную `privdata` и в поле `BlockData` структуры данных блока, в которой он будет храниться до тех пор, пока модель блока не освободит отведенную память.

В отличие от блока-передатчика из прошлого примера, в котором пользователь мог ввести новое имя переменной в комментарии блока, в этом примере пользователь не имеет возможности задавать имена динамических переменных. Это сильно упрощает написание модели, поскольку здесь не нужно проверять изменения текста комментария при запуске расчета и создавать новые переменные – их можно создавать один раз при инициализации блока. После того, как личная область данных создана, модель создает в родительской подсистеме динамические переменные “BlkMove_Speed” и “BlkMove_Angle” при помощи сервисной функции `rdsCreateAndSubscribeDV` и записывает полученные указатели на структуры подписки в поля `VLink` и `ALink` личной области данных.

При вызове модели с параметром `RDS_BFM_CLEANUP` модель приводит запомненный в `BlockData->BlockData` указатель на личную область данных к типу “`TestBlkMoveSetterData*`” и присваивает его вспомогательной переменной `privdata`, чтобы к личной области данных было удобнее обращаться. Затем при помощи сервисной функции `rdsDeleteDVByLink` модель удаляет обе динамические переменные и освобождает отведенную под личную область память стандартной функцией `free`.

При выполнении такта моделирования (параметр `RDS_BFM_MODEL`) модель блока должна переписать значения изменившихся входов в динамические переменные и уведомить другие блоки, подписанные на эти переменные, об их возможном изменении. Эта часть модели практически не отличается от такта моделирования в блоке-передатчике: для каждого из двух входов проверяется наличие соответствующей динамической переменной, и, если она существует и не равна входу, в нее записывается новое значение и об этом информируются все блоки-подписчики.

Теперь рассмотрим модель блока, который будет перемещаться в окне подсистемы. В этом блоке должно быть по крайней мере две вещественных статических переменных (`x` и `y`), чтобы было с чем связать координаты его изображения на вкладке “Внешний вид” окна параметров блока (рис. 43). Для наглядности добавим еще одну вещественную переменную `a`, с которой свяжем угол поворота изображения блока. В эту переменную модель блока будет записывать значение угла направления в радианах. Таким образом, если блок будет иметь векторную картинку со стрелкой, указывающей вправо (что соответствует нулевому углу направления), в режиме расчета эта стрелка будет указывать направление движения блока.

Координаты изображения блока `x` и `y` будут вычисляться по приведенным выше рекуррентным формулам для перемещения блока за интервал времени Δt (рис. 42). Из динамической переменной “DynTime” можно получить только текущее значение времени, поэтому интервал, прошедший с момента последнего вычисления координат, модель должна будет вычислять самостоятельно. Для этого придется добавить к статическим переменным блока дополнительную переменную `t0`, в которой будет запоминаться предыдущее значение времени, а интервал Δt будет вычисляться как `DynTime-t0`. Чтобы первый интервал времени был вычислен правильно, начальное значение переменной `t0` должно быть нулевым.

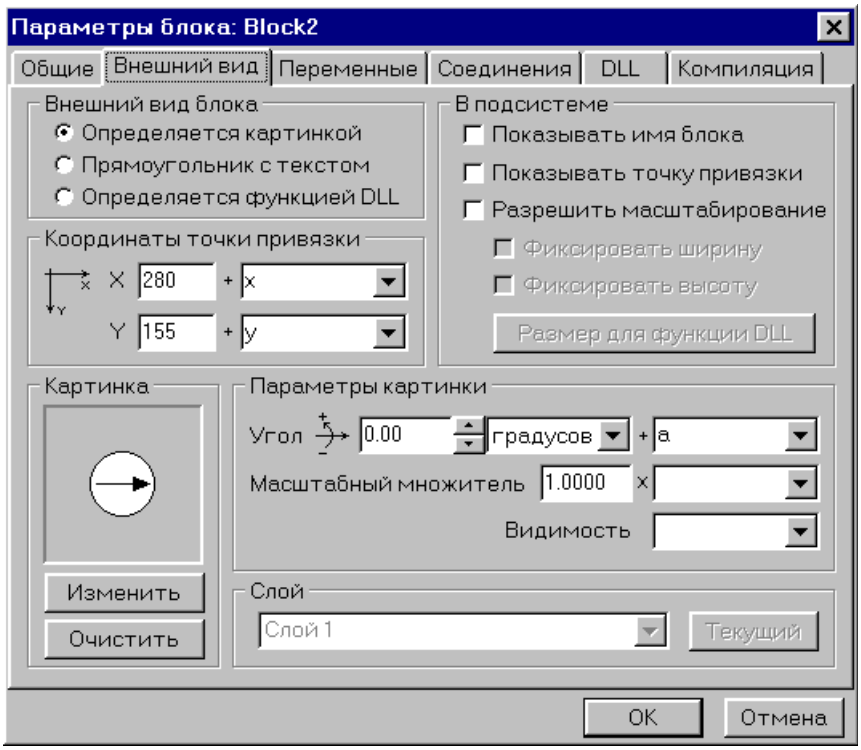


Рис. 43. Привязка координат подвижного блока
к переменным `x`, `y` и `a`

Таким образом, перемещающийся блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Начальное значение*
0	Start	Сигнал	1	Вход	–
1	Ready	Сигнал	1	Выход	–
2	x	double	8	Внутренняя	0
10	y	double	8	Внутренняя	0
18	a	double	8	Внутренняя	0
26	t0	double	8	Внутренняя	0

Этот блок не будет реагировать на событие `RDS_BFM_MODEL` (все действия будут производиться в реакции на изменение динамических переменных `RDS_BFM_DYNVARCHANGE`), поэтому в его параметрах имеет смысл выключить запуск каждый такт. В личной области данных блока необходимо запоминать три указателя на структуры подписки: для переменных “BlkMove_Speed”, “BlkMove_Angle” и “DynTime”. Кроме того, в структуре личной области данных будут два дополнительных вещественных поля `SinA` и `CosA`, в которых будут храниться значения синуса и косинуса угла `a`. Наличие этих полей позволит вычислять синус и косинус не при каждом перемещении блока, а только при изменении направления движения, что уменьшит нагрузку на систему. В данном случае это уменьшение будет практически незаметным, но при сложных вычислениях часто имеет смысл хранить промежуточные результаты.

Модель блока вместе с описанием структуры личной области данных будет выглядеть следующим образом:

```
// Структура личной области данных блока
typedef struct
```

```
{ // Указатель на структуру подписки на скорость
  RDS_PDYNVARLINK VLink;
  // Указатель на структуру подписки на направление
  RDS_PDYNVARLINK ALink;
  // Указатель на структуру подписки на время
  RDS_PDYNVARLINK Time;

  // Дополнительные поля для хранения значений синуса
  // и косинуса направления движения блока
  double SinA;
  double CosA;
} TestBlkMoveObjectData;

// Модель перемещающегося блока
extern "C" __declspec(dllexport)
  int RDSCALL TestBlkMoveObject(int CallMode,
                                RDS_PBLOCKDATA BlockData,
                                LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define x       (*((double *)(pStart+2)))
#define y       (*((double *)(pStart+10)))
#define a       (*((double *)(pStart+18)))
#define t0      (*((double *)(pStart+26)))
  // Вспомогательная переменная - указатель на личную область
  // данных блока, приведенный к правильному типу
  TestBlkMoveObjectData *privdata;
  // Вспомогательные переменные для скорости и времени
  double v,time;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Отведение памяти под личную область данных
        privdata=(TestBlkMoveObjectData*)
                 malloc(sizeof(TestBlkMoveObjectData));
        BlockData->BlockData=privdata;
        // Подписка на переменную скорости
        privdata->VLink=rdsSubscribeToDynamicVar(
                        RDS_DVPARENT,      // В родительской
                        "BlkMove_Speed",   // Имя переменной
                        "D",               // Тип double
                        FALSE);            // Без поиска
        // Подписка на переменную направления
        privdata->ALink=rdsSubscribeToDynamicVar(
                        RDS_DVPARENT,      // В родительской
                        "BlkMove_Angle",   // Имя переменной
                        "D",               // Тип double
                        FALSE);            // Без поиска
        // Подписка на переменную времени
        privdata->Time=rdsSubscribeToDynamicVar(
                        RDS_DVPARENT,      // В родительской
                        "DynTime",         // Имя переменной
                        "D",               // Тип double
                        TRUE);             // Поиск в иерархии
```

```
  break;

// Очистка данных блока
case RDS_BFM_CLEANUP:
  // Приведение указателя на личную область данных к
  // правильному типу
  privdata=(TestBlkMoveObjectData*)(BlockData->BlockData);
  // Прекращение подписки на все динамические переменные
  rdsUnsubscribeFromDynamicVar(privdata->VLink);
  rdsUnsubscribeFromDynamicVar(privdata->ALink);
  rdsUnsubscribeFromDynamicVar(privdata->Time);
  // Освобождение отведенной памяти
  free(privdata);
  break;

// Проверка типа статических переменных
case RDS_BFM_VARCHECK:
  if(strcmp((char*)ExtParam,"{SSDDDD}")==0)
    return RDS_BFR_DONE;
  return RDS_BFR_BADVARSMSG;

// Реакция на изменение динамической переменной
case RDS_BFM_DYNVARCHANGE:
  // Приведение указателя на личную область данных к
  // правильному типу
  privdata=(TestBlkMoveObjectData*)(BlockData->BlockData);
  // Проверка - удалось ли подписаться на все переменные
  // (если хотя бы один указатель - NULL, значит, не удалось)
  if(privdata->VLink==NULL ||   // Скорость
     privdata->ALink==NULL ||   // Направление
     privdata->Time==NULL)      // Время
    break;
  // Проверка - существуют ли все переменные
  // (если хотя бы один указатель - NULL, значит, не удалось)
  if(privdata->VLink->Data==NULL ||  // Скорость
     privdata->ALink->Data==NULL ||  // Направление
     privdata->Time->Data==NULL)     // Время
    break;
  // Если изменилась переменная направления, нужно вычислить
  // и запомнить новые значения синуса и косинуса угла
  if(ExtParam==(void*)(privdata->ALink))
    { // Изменилось направление - привести к типу double*
      double *pa=(double*)privdata->ALink->Data;
      // Значение угла в радианах
      a=(*pa)*M_PI/180.0;
      // Значения синуса и косинуса
      privdata->SinA=sin(a);
      privdata->CosA=cos(a);
    }
  // Записать во вспомогательные переменные v и time значения
  // скорости и времени
  v=*(double*)privdata->VLink->Data;
  time=*(double*)privdata->Time->Data;
  // Вычислить новое смещение изображения блока
  x+=v*(time-t0)*privdata->CosA;
  y-=v*(time-t0)*privdata->SinA;
```

```
            // Запомнить значение времени, для которого вычислены
            // новые координаты
            t0=time;
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef t0
    #undef a
    #undef y
    #undef x
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Структура личной области данных блока `TestBlkMoveObjectData` содержит три указателя на структуры подписки (`VLink` для динамической переменной скорости, `ALink` для направления и `Time` для времени) и два вещественных поля для хранения значений синуса и косинуса направления. Как и в модели блока управления, при вызове этой модели с параметром `RDS_BFM_INIT` под эту структуру отводится память функцией `malloc`, и полученный указатель записывается во вспомогательную переменную `privdata` и в поле `BlockData->BlockData`. Затем модель подписывается на три динамических переменные и записывает указатели на созданные структуры подписки в соответствующие поля личной области данных. Следует обратить внимание, что переменные “BlkMove_Speed” и “BlkMove_Angle” ищутся только в родительской подсистеме (параметр `Search` функции `rdsSubscribeToDynamicVar` равен `FALSE`), а переменная “DynTime” – во всех подсистемах вверх по иерархии, начиная с родительской. Это позволяет данному блоку получить доступ к переменной “DynTime”, созданной ближайшим к нему планировщиком в цепочке родительских подсистем. При вызове модели с параметром `RDS_BFM_CLEANUP` подписка на все динамические переменные прекращается, и память, занятая личной областью данных блока, освобождается функцией `free`.

Как только блок управления изменит направление или скорость, или блок-планировщик изменит значение времени, функция модели данного блока будет вызвана с параметром `RDS_BFM_DYNVARCHANGE`, при этом в параметре `ExtParam` в нее будет передан указатель на структуру подписки изменившейся переменной. Сравнив `ExtParam` с полями `VLink`, `ALink` и `Time` личной области данных, можно понять, какая из динамических переменных изменилась. Но сначала модель должна проверить, есть ли у нее доступ к динамическим переменным. Если хотя бы один указатель на структуру подписки, хранящийся в личной области данных, равен `NULL`, значит, модели не удалось подписаться на одну из переменных. В данном случае это маловероятно, поскольку выбранные имена переменных отвечают требованиям РДС, и функция `rdsSubscribeToDynamicVar` может вернуть значение `NULL`, сигнализирующее о невозможности подписки, только при каком-либо системном сбое (например, при нехватке памяти). Однако, проверить успешность подписки все-таки следует. Если один из указателей будет равен `NULL`, работа модели будет немедленно завершена – для работы этого блока необходимы все три переменные.

Если модели удалось подписаться на все переменные, это еще не значит, что у нее есть доступ к ним. Успешность подписки говорит только о том, что РДС теперь следит за тем, чтобы в поле `Data` структуры подписки находился указатель на область данных запрошенной переменной. Если такой переменной не существует, это поле будет иметь значение `NULL`. Поскольку блоку нужны все три динамические переменные, необходимо

проверить все три указателя: `VLink->Data`, `ALink->Data` и `Time->Data`. Если хотя бы один из них равен `NULL`, работа модели будет немедленно завершена.

После того, как модель убедилась, что в данный момент у нее есть доступ ко всем трем динамическим переменным, она сравнивает указатель на структуру подписки изменившейся переменной, переданный в `ExtParam`, с указателем переменной направления `ALink`, предварительно приведенным к типу универсального указателя (`void*`). Если два этих указателя равны, значит, изменилось направление движения блока. В этом случае переданное через динамическую переменную значение направления в градусах переводится в радианы и присваивается статической переменной `a`, чтобы векторная картинка блока, угол поворота которой связан с этой переменной, повернулась в направлении движения блока. Затем вычисляются косинус и синус `a` и запоминаются в полях `SinA` и `CosA` личной области данных блока для дальнейшего вычисления перемещения.

Затем, уже независимо от того, какая из трех переменных изменилась, вычисляются новые значения горизонтального `x` и вертикального `y` смещений изображения блока по приведенным на стр. 105 формулам. Интервал времени Δt, прошедший с момента последнего расчета, вычисляется как разность значения времени из динамической переменной "DynTime" (присвоенного вспомогательной переменной `time`) и запомненного значения времени `t0`. После этого `t0` присваивается текущее значение времени, чтобы при следующем изменении переменных использовать его как значение времени предыдущего расчета.

Чтобы убедиться в работоспособности созданной модели, необходимо поместить в одну подсистему блок управления, перемещающийся блок и планировщик (планировщик может также находиться выше по иерархии, например, в родительской подсистеме данной подсистемы), задать для перемещающегося блока какую-нибудь векторную картинку и подключить поля ввода ко входам `Speed` и `Angle` блока управления (рис. 44). После запуска расчета и ввода каких-либо значений в поля ввода блок должен начать двигаться в заданном направлении с указанной (в точках экрана в секунду) скоростью.

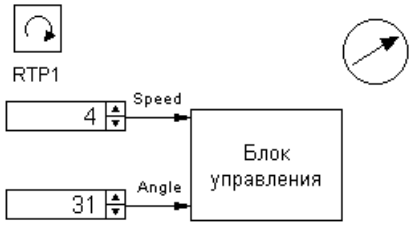


Рис. 44. Управление подвижным блоком

§2.6.5. Работа со сложными динамическими переменными

Рассматривается работа с динамическими переменными сложных типов: матрицами, структурами и т.п. Приводится пример, в котором два блока передают друг другу матрицу вещественных чисел через динамическую переменную в корневой подсистеме.

Во всех приведенных примерах модели блоков работали с простыми динамическими переменными типа `double`. В работе с динамическими переменными сложных типов (строками, матрицами, структурами) нет принципиальных отличий. Получив указатель на область данных переменной (поле `Data` структуры `RDS_DYNVARLINK`) и удостоверившись, что он не равен `NULL`, можно обращаться к данным динамической переменной так же, как если бы это была обычная статическая переменная.

Для примера создадим модели двух блоков, один из которых будет записывать поступившую ему на вход матрицу вещественных чисел в созданную им динамическую переменную "DynMatr" в корневой подсистеме, а другой – считывать эту матрицу из динамической переменной, передавать ее на свой выход и вычислять сумму ее элементов (рис. 45). Фактически эти блоки представляют собой упрощенные аналоги рассмотренных выше блока-передатчика и блока-приемника, передающие матрицу вместо вещественного числа и не имеющие возможности изменить имя динамической переменной, связывающей их.

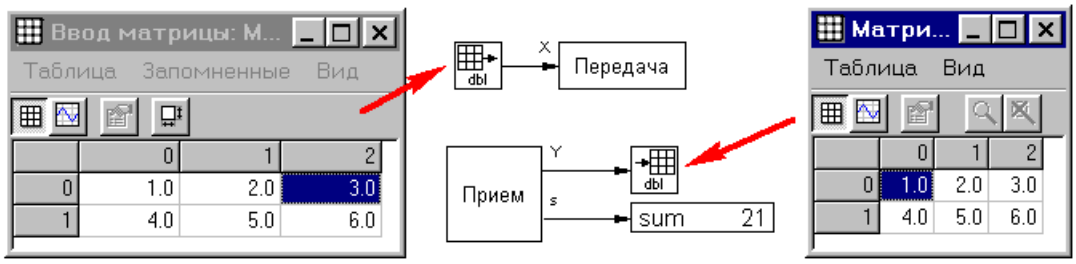


Рис. 45. Передача матрицы через динамическую переменную

Начнем с модели блока, записывающего матрицу в динамическую переменную. Помимо двух обязательных сигналов, этот блок будет иметь единственный вход `X` типа “матрица `double`”:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	1
1	Ready	Сигнал	1	Выход		–
2	X	Матрица double	8	Вход	✓	–

Как и в других моделях, работающих с единственной динамической переменной, указатель на ее структуру подписки будет запоминаться в поле `BlockData` структуры `RDS_BLOCKDATA`, поскольку как таковой личной области данных у блока не будет, и этот указатель можно использовать как угодно. Модель блока будет достаточно простой:

```
extern "C" __declspec(dllexport)
  int RDSCALL TestDynMatrCreate(int CallMode,
                                RDS_PBLOCKDATA BlockData,
                                LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define pX      ((void **)(pStart+2))
  // Вспомогательная переменная - указатель на структуру подписки
  RDS_PDYNVARLINK Link;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Создание динамической переменной
        Link=rdsCreateAndSubscribeDV(RDS_DVROOT, // В корневой
                                     "DynMatr",  // Имя переменной
                                     "MD",       // Тип
                                     TRUE,       // Запрет удаления
                                     NULL);      // Без нач.знач.
        // Запомнить указатель на структуру подписки
        BlockData->BlockData=Link;
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
```

```
        // Удалить переменную
        rdsDeleteDVByLink(Link);
        break;

      // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSMD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Такт расчета
      case RDS_BFM_MODEL:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        // Если переменная существует, копировать входную матрицу
        // в матрицу динамической переменной
        if(Link!=NULL && Link->Data!=NULL)
          { // Копирование X в динамическую переменную
            rdsCopyVarArray(Link->Data,pX);
            // Уведомление подписчиков
            rdsNotifyDynVarSubscribers(Link);
          }
        break;
    }

  return RDS_BFR_DONE;
// Отмена макроопределений
#undef pX
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

При вызове модели в режиме `RDS_BFM_INIT` в корневой подсистеме будет создана динамическая переменная “DynMatr” типа “MD”, то есть “матрица `double`”, указатель на структуру подписки которой будет запомнен в переменной `BlockData->BlockData`. При вызове модели в режиме `RDS_BFM_CLEANUP` эта переменная будет уничтожена. Эти действия уже несколько раз описывались в предыдущих примерах и останавливаться на них подробнее нет смысла. При вызове модели для выполнения такта расчета запомненный в `BlockData->BlockData` указатель на структуру подписки присваивается вспомогательной переменной `Link`, и, если этот указатель существует (`Link!=NULL`) и у динамической переменной есть данные (`Link->Data!=NULL`), вызывается сервисная функция `rdsCopyVarArray`, копирующая данные матрицы, на которую ссылается указатель `pX`, в матрицу, на которую ссылается указатель `Link->Data` – так вход блока копируется в динамическую переменную. Функция `rdsCopyVarArray` может работать как со статическими матрицами, так и с динамическими – все функции и макросы, которые работают с данными статических переменных, могут с тем же успехом работать и с данными динамических, поскольку их представление в памяти не отличается.

Теперь создадим модель блока, который будет считывать данные из динамической переменной “DynMatr”. Блок будет выдавать полученную матрицу на выход `Y`, а сумму ее элементов – на выход `s`:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Начальное значение*
0	Start	Сигнал	1	Вход	1
1	Ready	Сигнал	1	Выход	–
2	Y	Матрица double	8	Выход	–
10	s	double	8	Выход	–

Сигналу `Start` необходимо присвоить начальное значение 1 – это самый простой способ заставить модель блока выполниться в первом такте расчета. Также желательно выключить для блока запуск каждый такт. Модель будет передавать динамическую матрицу на выход и суммировать ее элементы и при каждом изменении переменной “DynMatr”, о котором ее информирует блок с моделью `TestDynMatrCreate` при помощи сервисной функции `rdsNotifyDynVarSubscribers`, и при выполнении такта расчета. На самом деле запуск модели в режиме `RDS_BFM_MODEL` произойдет всего один раз – при первом запуске расчета, из-за того, что сигнал `Start` в этот момент будет иметь значение 1. После первого же запуска значение этого сигнала будет сброшено, и, поскольку у блока нет входов, а запуск каждый такт отключен, в дальнейшем модель будет вызываться только при изменении динамической переменной.

```
extern "C" __declspec(dllexport)
  int RDSCALL TestDynMatrGet(int CallMode,
                             RDS_PBLOCKDATA BlockData,
                             LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define pY      ((void **)(pStart+2))
#define s       (*((double *)(pStart+10)))
  // Вспомогательная переменная – указатель на структуру подписки
  RDS_PDYNVARLINK Link;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Подписка на динамическую переменную
        Link=rdsSubscribeToDynamicVar(RDS_DVROOT, // В корневой
                                      "DynMatr",   // Имя
                                      "MD",        // Тип
                                      FALSE);      // Без поиска
        // Запомнить указатель на структуру подписки
        BlockData->BlockData=Link;
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        // Запомненный указатель на структуру подписки
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        // Прекратить подписку
        rdsUnsubscribeFromDynamicVar(Link);
        break;

     // Проверка типа статических переменных
     case RDS_BFM_VARCHECK:
```

```
      if(strcmp((char*)ExtParam,"{SSMDD}")==0)
        return RDS_BFR_DONE;
      return RDS_BFR_BADVARSMSG;

    // Изменение динамической переменой или такт расчета
    case RDS_BFM_DYNVARCHANGE:
    case RDS_BFM_MODEL:
      // Запомненный указатель на структуру подписки
      Link=(RDS_PDYNVARLINK)BlockData->BlockData;
      // В s будут суммироваться элементы матрицы
      s=0;
      // Проверка существования динамической переменной
      if(Link!=NULL && Link->Data!=NULL)
        { // Копировать динамическую матрицу в Y
          rdsCopyVarArray(pY,Link->Data);
          // Если матрица не нулевого размера - суммировать
          if(RDS_ARRAYEXISTS(Link->Data))
            { // Число элементов в матрице
              int count=RDS_ARRAYROWS(Link->Data)*
                        RDS_ARRAYCOLS(Link->Data);
              // Указатель на начало данных матрицы
              double *data=(double*)RDS_ARRAYDATA(Link->Data);
              // Суммирование элементов в цикле
              for(int i=0;i<count;i++)
                s+=data[i];
            }
          // Взвести Ready для передачи выходов по связям
          Ready=1;
        }
      break;
  }

  return RDS_BFR_DONE;
// Отмена макроопределений
#undef s
#undef pY
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

При инициализации блока эта модель подписывается на динамическую переменную “DynMatr”, при очистке – прекращает подписку. Как и в предыдущей модели, указатель на структуру подписки запоминается в `BlockData->BlockData`. При запуске модели в режимах `RDS_BFM_DYNVARCHANGE` (изменение динамической переменной) и `RDS_BFM_MODEL` (такт расчета) матрица, находящаяся в динамической переменной, копируется в выход блока `Y`, после чего все ее элементы суммируются и выдаются на выход `s`. Для проверки, существует ли матрица (не нулевой ли у нее размер) и для получения числа строк, числа столбцов и указателя на первый элемент матрицы применяются те же самые макросы, которые использовались для статических переменных. Если переписать отвечающую за суммирование матрицы часть модели, таким образом, чтобы она вместо динамической переменной обращалась к статическому выходу блока `Y`, в который эта динамическая переменная была только что скопирована, текст программы будет выглядеть следующим образом (изменения выделены жирным):

```
// ...
// Если матрица не нулевого размера - суммировать
if(RDS_ARRAYEXISTS(pY))
  { // Число элементов в матрице
    int count=RDS_ARRAYROWS(pY)*
              RDS_ARRAYCOLS(pY);
    // Указатель на начало данных матрицы
    double *data=(double*)RDS_ARRAYDATA(pY);
    // Суммирование элементов в цикле
    for(int i=0;i<count;i++)
      s+=data[i];
  }
// ...
```

В этом варианте указатель на область данных динамической переменной `Link->Data` заменен на указатель на статическую переменную `pY`, и модель полностью сохранила работоспособность, поскольку данные, расположенные по этим указателям, имеют одинаковую структуру.

Таким образом, если есть функция или макрос, выполняющая какое-либо действие над статической переменной, можно использовать ее и для выполнения этого действия над динамической, передав ей поле `Data` структуры подписки `RDS_DYNVARLINK` вместо указателя на статическую переменную. Это будет работать и для матриц, и для строк, и для структур, и для простых переменных.

§2.7. Настройка параметров блока

Описываются способы предоставления пользователю интерфейса для настройки параметров блока. Рассматриваются возможности открытия моделью блока модальных окон, возникающие при этом проблемы и их решения. Рассматривается использование специальных вспомогательных объектов РДС, позволяющих без особых усилий формировать модальные окна с полями ввода. Также рассматривается способ хранения параметров блока в файле схемы, не требующий введение в модель дополнительных реакций – запись значений параметров значения по умолчанию статических переменных блока.

§2.7.1. Функция настройки блока и открытие модальных окон

Описывается реакция модели на функцию настройки блока (RDS_BFM_SETUP), возможность вызова которой встроена в пользовательский интерфейс РДС. С помощью этой реакции в блоки, описанные в §2.6.3 (приемник и передатчик, организующие связь через динамическую переменную), добавлен интерфейс для настройки.

Функция настройки обычно используется для того, чтобы организовать какой-либо диалог с пользователем, например, дать ему возможность задать значения параметров блока. Эта функция может вызываться только в режиме редактирования, поэтому в ней можно открывать модальные окна, не опасаясь зависания потока расчета (проблемы, возникающие при открытии модальных окон в режиме расчета, были описаны в §1.8). Для того, чтобы пользователь мог вызвать функцию настройки, необходимо разрешить ее, установив на вкладке “DLL” окна параметров блока флаг “Блок имеет функцию настройки” (при этом можно указать название пункта контекстного меню, при выборе которого будет вызываться эта функция, см. рис. 7), и ввести в модель реакцию на событие `RDS_BFM_SETUP`. Модель, вызванная для реакции на это событие, должна самостоятельно открыть диалоговое окно (средствами Windows или с помощью сервисных функций РДС), и, после его закрытия, вернуть РДС одну из двух констант:

- `RDS_BFR_DONE` – параметры блока не изменились;
- `RDS_BFR_MODIFIED` – введены новые параметры блока, схема будет считаться измененной и при попытке выхода из РДС без сохранения схемы пользователю будет выдано предупреждение.

Для открытия диалогового окна функция модели может использовать любые доступные ей библиотечные функции (например, функцию `DialogBox` в Windows API, функцию-член `ShowModal` потомков класса `TForm` в Borland C++ Builder и т.п.). При этом, чтобы избежать проблем, описанных в §1.8 (стр. 29), модель должна сообщать РДС об открытии и закрытии диалогового окна при помощи сервисных функций `rdsBlockModalWinOpen` и `rdsBlockModalWinClose`. В РДС также предусмотрено несколько сервисных функций, позволяющих открывать диалоговые окна. Эти функции не предоставляют такой свободы в задании внешнего вида окна, как универсальные функции API, зато при их использовании можно не беспокоиться о синхронизации открытия и закрытия окон с РДС.

Если модели требуется запросить у пользователя только один параметр, проще всего использовать для этого сервисную функцию `rdsInputString`, которая открывает диалоговое окно для ввода строки. Функция принимает следующие параметры:

```
LPSTR RDSCALL rdsInputString(
      LPSTR      WinCaption,      // Заголовок окна
      LPSTR      StrCaption,      // Текст перед полем ввода
      LPSTR      Default,         // Исходное значение
      int        Width);          // Ширина поля ввода
```

- `WinCaption` – заголовок диалогового окна;
- `StrCaption` – текст перед полем ввода (обычно он сообщает пользователю, какой параметр он вводит);
- `Default` – исходное значение строки, которое помещается в поле ввода перед редактированием;
- `Width` – ширина поля ввода строки в точках экрана.

Функция открывает диалоговое окно с единственным полем ввода и кнопками “OK” и “Отмена”. Размер окна автоматически подбирается таким образом, чтобы в него уместилось поле ввода указанного в параметре `Width` размера и текст, указанный в параметре `StrCaption`, перед ним. Если пользователь нажмет кнопку “OK”, функция вернет указатель на динамически отведенную область памяти, в которую скопирована строка из поля ввода (эту область необходимо будет позже освободить при помощи сервисной функции `rdsFree`). Если пользователь нажмет кнопку “Отмена”, функция вернет константу `NULL`.

Эту функцию можно использовать в примере с блоками-передатчиками и приемниками (стр. 95) для задания имени динамической переменной, связывающей эти блоки между собой. В примере имя этой переменной хранится в комментарии блока, и пользователь может изменить его только в окне параметров этого блока на вкладке “Общие”. Это не очень удобно, поскольку пользователю не выдается никаких подсказок о том, где именно вводится имя переменной. При помощи функции `rdsInputString` можно организовать ввод имени переменной в отдельном окне, с заголовком, дающим пользователю понять, что он вводит имя переменной для связи блоков. Диалоговое окно будет открываться в момент реакции модели на вызов функции настройки. Для того, чтобы блок начал реагировать на это событие, необходимо в окне его параметров на вкладке “DLL” установить флаг “Блок имеет функцию настройки”. Кроме того, на вкладке “Общие” можно установить флаг “Двойной щелчок в режиме редактирования вызывает функцию настройки” (см. рис. 5), чтобы пользователю не нужно было выбирать пункт “Настройка” в контекстном меню каждый раз, как ему понадобится изменить имя переменной. Эти действия следует произвести со всеми блоками-приемниками и блоками-передатчиками (можно воспользоваться функцией групповой установки).

Теперь, когда блоки готовы откликаться на функцию настройки, необходимо ввести в их модели соответствующую реакцию. Реакция на функцию настройки будет одинаковой в модели блока-передатчика `TestTunnelIn` и в модели блока-приемника `TestTunnelOut`.

Обе эти модели хранят свой единственный параметр – имя переменной связи – в комментарии блока, поэтому обе они при вызове функции настройки должны позволить пользователю изменить значение комментария в отдельном окне, и при этом не дать ему ввести более одной строки. Для этого и будет использоваться сервисная функция rdsInputString.

Сначала в обе модели нужно добавить описание новой вспомогательной переменной str, которой будет присваиваться указатель, возвращаемый функцией rdsInputString:

```
// ....
  // Вспомогательная переменная – указатель на структуру подписки
  RDS_PDYNVARLINK Link;
  // Вспомогательная переменная – структура описания блока
  RDS_BLOCKDESCRIPTION Descr;
  // Вспомогательная переменная – указатель на строку
  char *str;

  switch(CallMode)
    { // Очистка
// ....
```

Внутрь оператора switch(CallMode) необходимо добавить новый оператор case с константой RDS_BFM_SETUP:

```
// ....
      // Реакция на функцию настройки
      case RDS_BFM_SETUP:
        // Получение текущего текста комментария блока
        Descr.servSize=sizeof(Descr);
        rdsGetBlockDescription(BlockData->Block,&Descr);
        // Открытие диалогового окна для ввода строки
        str=rdsInputString("Переменная связи", // Заголовок окна
                           "Имя переменной:",  // Заголовок поля
                           Descr.BlockComment, // Исходный текст
                           150);               // Ширина поля
        if(str!=NULL) // Пользователь нажал "OK"
          { // Установить новый комментарий
            rdsSetBlockComment(BlockData->Block,str);
            // Освободить динамическую строку
            rdsFree(str);
            // Информировать РДС об изменениях
            return RDS_BFR_MODIFIED;
          }
        break;
    }
  return RDS_BFR_DONE;
// ....
```

При вызове модели с параметром RDS_BFM_SETUP прежде всего необходимо получить текущее значение комментария блока, чтобы предъявить пользователю для редактирования имя переменной, используемой блоком в данный момент. Для этого вызывается функция rdsGetBlockDescription, заполняющая структуру описания блока (она уже использовалась в этих моделях). После ее вызова в поле структуры Descr.BlockComment будет записан указатель на строку комментария блока. Теперь это значение можно передать в функцию rdsInputString, которая откроет диалоговое окно для редактирования строки с заголовком “Переменная связи” (рис. 46). Перед полем ввода будет выведен заголовок “Имя переменной:”, а само поле будет содержать текст из комментария блока (или пустую строку, если комментарий был пуст). Если пользователь изменит текст в поле ввода и нажмет кнопку “OK”, функция rdsInputString вернет указатель на динамически

сформированную строку, скопированную из поля ввода, в противном случае она вернет NULL. Возвращенное функцией значение присваивается вспомогательной переменной str. Если оно не равно NULL (то есть пользователь нажал “OK”), эта строка устанавливается в качестве нового комментария блока при помощи сервисной функции rdsSetBlockComment, после чего занятая ей память освобождается при помощи функции rdsFree. Затем функция модели возвращает константу RDS_BFR_MODIFIED, информируя РДС о том, что параметры блока изменились и требуется предупредить об этом пользователя, если он захочет выйти из программы, не сохранив схему.

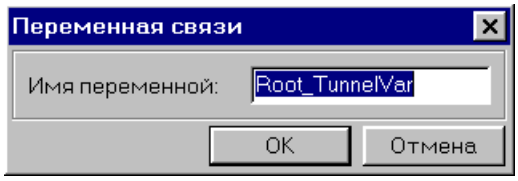


Рис. 46. Модальное окно ввода строки

§2.7.2. Использование объектов-окон РДС

Рассматривается использование вспомогательных объектов РДС, облегчающих создание и открытие модальных окно с полями ввода, описаны сервисные функции для работы с этими объектами. С их помощью в один из ранее описывавшихся блоков добавлен простой пользовательский интерфейс. Во другом примере рассматривается блок-генератор, выдающий на выход синусоиду, косинусоиду или прямоугольные импульсы по выбору пользователя, при этом некоторые поля в его окне настройки, формируемом при помощи вспомогательного объекта РДС, разрешаются или запрещаются в зависимости от значений других полей.

Простейшие диалоговые окна могут также открываться с помощью одного из *вспомогательных объектов* РДС. Вообще, вспомогательных объектов в РДС довольно много, они позволяют упростить выполнение различных сложных операций (разбор текста, программное задание переменных блока и т.п.). В данном случае мы воспользуемся объектом, создаваемым сервисной функцией rdsFORMCreate. При помощи этой и нескольких других сервисных функций можно создать модальное окно и добавить в него поля ввода для редактирования различных параметров. Подробно эти функции будут описаны ниже, здесь же будет приведен простой пример их использования. Добавим в модель Test1, приведенную в качестве примера реакции на события RDS_BFM_INIT и RDS_BFM_CLEANUP (стр. 38), возможность редактирования параметров, хранящихся в личной области данных. Класс личной области данных TTest1Data и модель блока должны быть изменены следующим образом (изменения выделены жирным):

```
//====== Класс личной области данных ======
class TTest1Data
{ public:
    int IParam;       // Целый параметр
    double DParam;    // Вещественный параметр
    int Setup(void);  // Функция настройки параметров
    TTest1Data(void)  // Конструктор класса
      { IParam=0; DParam=0.0;
        rdsMessageBox("Область создана","TTest1Data",MB_OK);
      };
    ~TTest1Data()     // Деструктор класса
      { rdsMessageBox("Область удалена","TTest1Data",MB_OK);};
};
//=======================================

//============= Модель блока ==============
extern "C" __declspec(dllexport)
  int RDSCALL Test1(int CallMode,
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{ TTest1Data *data;
```

```
  switch(CallMode)
    { case RDS_BFM_INIT:   // Инициализация
        BlockData->BlockData=new TTest1Data();
        break;
      case RDS_BFM_CLEANUP:// Очистка
        data=(TTest1Data*)(BlockData->BlockData);
        delete data;
        break;
      case RDS_BFM_SETUP:  // Функция настройки
        data=(TTest1Data*)(BlockData->BlockData);
        return data->Setup();
    }
  return RDS_BFR_DONE;
}
//=======================================
```

В этом примере часть программы, открывающая модальное окно, является функцией-членом класса `TTest1Data`. С тем же успехом она могла быть оформлена как обычная функция или вставлена непосредственно внутрь функции модели блока.

В класс личной области добавлена новая функция-член `int Setup(void)`, которая вызывается при реакции на событие `RDS_BFM_SETUP` и возвращает `RDS_BFR_DONE` или `RDS_BFR_MODIFIED` в зависимости от действий пользователя. Тело функции выглядит следующим образом:

```
// Функция настройки параметров
int TTest1Data::Setup(void)
{ RDS_HOBJECT window; // Идентификатор вспомогательного объекта
  BOOL ok;            // Пользователь нажал "OK"
  // Создание окна
  window=rdsFORMCreate(FALSE,-1,-1,"Ввод параметров");
  // Добавление полей ввода
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_EDIT,
                 "Целый параметр:",80);
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_EDIT,
                 "Вещественный параметр:",80);
  // Занесение исходных значений в поля ввода
  rdsSetObjectInt(window,1,RDS_FORMVAL_VALUE,IParam);
  rdsSetObjectDouble(window,2,RDS_FORMVAL_VALUE,DParam);
  // Открытие окна
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Нажата кнопка OK - запись параметров обратно в блок
      IParam=rdsGetObjectInt(window,1,RDS_FORMVAL_VALUE);
      DParam=rdsGetObjectDouble(window,2,RDS_FORMVAL_VALUE);
    }
  // Уничтожение окна
  rdsDeleteObject(window);
  // Возвращаемое значение
  return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
}
//=======================================
```

Сначала при помощи сервисной функции `rdsFORMCreate` в памяти создается вспомогательный объект-окно. Первый параметр `FALSE` указывает на то, что у окна нет вкладок, два значения `-1` заставляют РДС автоматически вычислить необходимую ширину и высоту окна. Последний параметр определяет текст в заголовке создаваемого окна, в данном случае – “Ввод параметров”. Затем, при помощи функции `rdsFORMAddEdit`, в созданное окно добавляются два поля ввода. Эта функция принимает следующие параметры:

```
rdsFORMAddEdit(
      RDS_HOBJECT      Win,       // Объект окна
      int              TabId,     // Идентификатор вкладки
      int              CtrlId,    // Идентификатор поля ввода
      DWORD            Type,      // Тип поля ввода
      LPSTR            Caption,   // Заголовок поля ввода
      int              Width);    // Ширина поля ввода
```

В первом параметре передается идентификатор вспомогательного объекта, созданного функцией `rdsFORMCreate`. Поскольку в созданном окне нет вкладок, вместо идентификатора вкладки передается 0. В качестве идентификатора поля передаются 1 для первого параметра и 2 для второго (это могут быть любые целые числа). Тип поля в обоих случаях равен `RDS_FORMCTRL_EDIT` (простое поле ввода). Текст, передаваемый в функцию, отображается слева от поля ввода, а ширина задает ширину поля в точках экрана (для обоих параметров задается ширина в 80 точек).

После создания полей ввода в них заносятся исходные значения параметров при помощи функций `rdsSetObjectInt` и `rdsSetObjectDouble`. В эти функции передаются: идентификатор объекта окна; идентификатор поля, присвоенный ему при вызове `rdsFORMAddEdit` (1 для первого поля, 2 для второго); константа `RDS_FORMVAL_VALUE`, указывающая, что устанавливается значение поля; и, наконец, собственно исходное значение параметра.

После того, как поля ввода созданы и их исходные значения установлены, окно открывается функцией `rdsFORMShowModalEx`. Эта функция вернет `TRUE`, если пользователь нажмет в окне кнопку “OK”, и `FALSE`, если он нажмет кнопку “Отмена” или просто закроет окно. При нажатии “OK” параметрам блока присваиваются новые значения, полученные из полей ввода функциями `rdsGetObjectInt` и `rdsGetObjectDouble` (их параметры аналогичны параметрам функций `rdsSetObjectInt` и `rdsSetObjectDouble`). Затем вспомогательный объект уничтожается функцией `rdsDeleteObject` и возвращается значение, соответствующее нажатой пользователем кнопке (`RDS_BFR_MODIFIED` для кнопки “OK” и `RDS_BFR_DONE` для кнопки “Отмена”). Следует отметить, что, хотя при изменении значений и нажатии “OK” функция возвращает константу `RDS_BFR_MODIFIED` и РДС будет считать схему измененной и предупреждать пользователя при выходе, значения параметров `IParam` и `DParam` будут потеряны, даже если пользователь сохранит схему. Эти параметры находятся в личной области данных блока, и за их сохранение и загрузку должна отвечать функция модели. Модель в этом примере не реагирует на события загрузки и сохранения, соответствующие реакции будут рассмотрены в §2.8.

Для того, чтобы пользователь мог вызвать функцию настройки блока, в окне параметров блока с этой моделью необходимо установить флаг “блок имеет функцию настройки” на вкладке “DLL”. Теперь можно проверить работу функции настройки: при выборе соответствующего пункта в контекстном меню блока должно открыться окно с двумя полями ввода (рис. 47).

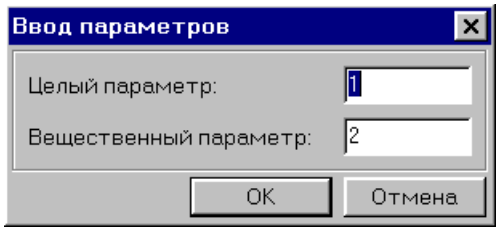


Рис. 47. Модальное окно для настройки параметров блока, созданное вспомогательным объектом РДС

Замечание для компилятора DigitalMars. Из-за некоторых особенностей этого компилятора (по крайней мере, версии 846) вместо функции `rdsGetObjectDouble` следует использовать функцию `rdsGetObjectDoubleP`. Чтобы скомпилировать приведенный выше пример при помощи DigitalMars, нужно будет заменить строку

```
DParam=rdsGetObjectDouble(window,2,RDS_FORMVAL_VALUE);
```

на

```
rdsGetObjectDoubleP(window,2,RDS_FORMVAL_VALUE,&DParam);
```

Функция rdsGetObjectDoubleP отличается от rdsGetObjectDouble только тем, что возвращает считанное из поля ввода значение через указатель, переданный в функцию.

Функция открытия модального окна rdsFORMShowModalEx возвращает управление модели блока только после того, как пользователь закроет окно. Достаточно часто возникает необходимость реагировать на действия пользователя до закрытия окна, например, разрешать или запрещать ввод данных в какие-либо поля в зависимости от значения других полей. Для этого в функцию rdsFORMShowModalEx можно передать дополнительный параметр – указатель на функцию, которая будет вызываться каждый раз при изменении значений в полях ввода. В этой функции можно разрешать или запрещать отдельные поля ввода окна, изменять их значения и т.п (в предыдущем примере этот параметр имел значение NULL, и никакая дополнительная функция не вызывалась).

Рассмотрим блок, выдающий на вещественный выход y синусоиду, косинусоиду или прямоугольные импульсы по выбору пользователя. Значение времени блок будет получать через динамическую переменную “DynTime” типа double, которая создается и изменяется планировщиком динамического расчета (см. также пример на стр. 90). В функции настройки блока пользователь сможет выбрать тип функции, ее период и, при формировании прямоугольных импульсов, длительность импульса. Для синусоиды и косинусоиды не требуется задание длительности импульса, поэтому, при выборе этих типов функции поле ввода длительности должно быть запрещено.

Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	y	double	8	Выход

У этого блока нет входов, участвующих в формировании значения y, поэтому в его модели не будет реакции на такт расчета – единственная переменная, изменение которой должно приводить к вычислению нового значения выхода, это “DynTime”. Все действия по вычислению y будут производиться в реакции на изменение этой динамической переменной, поэтому, чтобы блок зря не тратил процессорное время, в его параметрах следует отключить флаг “запуск каждый такт”.

Модель блока вместе с личной областью данных и функцией обратного вызова для запрещения поля ввода будет выглядеть следующим образом:

```
//====== Класс личной области данных ======
class TTestGenData
{ public:
    int Type;              // Тип (0-sin,1-cos,2-прямоугольные)
    double Period;         // Период
    double Impulse;        // Длительность импульса

    RDS_PDYNVARLINK Time;  // Связь с динамической
                           // переменной времени

    int Setup(void);       // Функция настройки
    TTestGenData(void)     // Конструктор класса
      { Type=0; Period=1.0; Impulse=0.5;
        // Подписка на динамическую переменную времени
        Time=rdsSubscribeToDynamicVar(RDS_DVPARENT,
                                      "DynTime",
                                      "D",
```

```
                                        TRUE);
      };
    ~TTestGenData(void)    // Деструктор класса
      { // Прекращение подписки
        rdsUnsubscribeFromDynamicVar(Time);
      };
};

//==== Прототип функции обратного вызова окна настроек ====
void RDSCALL TestGenDataCheckFunc(RDS_HOBJECT);

//====== Функция редактирования параметров ======
int TTestGenData::Setup(void)
{ RDS_HOBJECT window; // Идентификатор вспомогательного объекта
  BOOL ok;            // Пользователь нажал "ОК"
  // Создание окна
  window=rdsFORMCreate(FALSE,-1,-1,"Простой генератор");
  // Добавление полей ввода
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_COMBOLIST,
                 "Вид:",210);
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_EDIT,
                 "Период:",80);
  rdsFORMAddEdit(window,0,3,RDS_FORMCTRL_EDIT,
                 "Длительность:",80);
  // Установка списка вариантов
  rdsSetObjectStr(window,1,RDS_FORMVAL_LIST,
                  "Синус\nКосинус\nПрямоугольные импульсы");
  // Занесение исходных значений в поля ввода
  rdsSetObjectInt(window,1,RDS_FORMVAL_VALUE,Type);
  rdsSetObjectDouble(window,2,RDS_FORMVAL_VALUE,Period);
  rdsSetObjectDouble(window,3,RDS_FORMVAL_VALUE,Impulse);
  // Открытие окна с указанием функции обратного вызова
  ok=rdsFORMShowModalEx(window,TestGenDataCheckFunc);
  if(ok)
    { // Нажата кнопка ОК - запись параметров обратно в блок
      Type=rdsGetObjectInt(window,1,RDS_FORMVAL_VALUE);
      Period=rdsGetObjectDouble(window,2,RDS_FORMVAL_VALUE);
      Impulse=rdsGetObjectDouble(window,3,RDS_FORMVAL_VALUE);
    }
  // Уничтожение окна
  rdsDeleteObject(window);
  // Возвращаемое значение
  return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
}

//====== Функция обратного вызова для окна настроек ======
void RDSCALL TestGenDataCheckFunc(RDS_HOBJECT win)
{ // Считать номер пункта выпадающего списка
  int type=rdsGetObjectInt(win,1,RDS_FORMVAL_VALUE);
  // Разрешить ввод длительности, если выбран пункт 2
  rdsSetObjectInt(win,3,RDS_FORMVAL_ENABLED,type==2);
}

//============= Модель блока ==============
extern "C" __declspec(dllexport)
  int RDSCALL TestGen(int CallMode,
                      RDS_PBLOCKDATA BlockData,
```

```
                        LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define y (*((double *)(pStart+2)))
  // Вспомогательная переменная - указатель на личную область
  // данных блока, приведенный к правильному типу
  TTestGenData *data;

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        BlockData->BlockData=new TTestGenData();
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        data=(TTestGenData*)(BlockData->BlockData);
        delete data;
        break;

      // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Функция настройки
      case RDS_BFM_SETUP:
        data=(TTestGenData*)(BlockData->BlockData);
        return data->Setup();

      // Изменение динамической переменной или запуск расчета
      case RDS_BFM_STARTCALC:
      case RDS_BFM_DYNVARCHANGE:
        data=(TTestGenData*)(BlockData->BlockData);
        if(data->Period==0.0) // Нельзя вычислить частоту
          return 0;
        // Проверка наличия переменной "DynTime"
        if(data->Time!=NULL && data->Time->Data!=NULL)
          { // Динамическая переменная найдена - чтение значения
            double t=*((double*)data->Time->Data);
            switch(data->Type)
              { case 0: // Синус
                  y=sin(2*M_PI*t/data->Period);
                  break;
                case 1: // Косинус
                  y=cos(2*M_PI*t/data->Period);
                  break;
                case 2: // Прямоугольные импульсы
                  t=fmod(t,data->Period);
                  y=(t>data->Impulse)?-1.0:1.0;
                  break;
              }
            // Взвести Ready для передачи выхода по связям
            Ready=1;
```

```
            }
          break;
      }
    return RDS_BFR_DONE;
  // Отмена макроопределений
  #undef y
  #undef Ready
  #undef Start
  #undef pStart
  }
  //=======================================
```

Как и в предыдущем примере, личная область данных блока оформлена в виде класса. В нем размещаются три параметра блока: тип формируемой функции `Type`, период функции `Period` и длительность прямоугольного импульса `Impulse`. Также в личной области данных находится указатель на структуру подписки `Time` для доступа к динамической переменной “DynTime”. Подписка на переменную производится в конструкторе класса личной области данных, прекращение подписки – в деструкторе. Функция-член класса `Setup` отвечает за работу с окном настройки.

За описанием класса личной области данных следует прототип функции обратного вызова `TestGenDataCheckFunc`, отвечающей за запрет и разрешение поля ввода длительности импульса. Указатель на нее будет передаваться в функцию `rdsFORMShowModalEx` при открытии модального окна. Тело этой функции будет описано позже. Можно было бы разместить его прямо здесь и обойтись без описания прототипа, но для связности изложения лучше сначала рассмотреть функцию, открывающую окно настройки, а затем – используемую в ней функцию запрета и разрешения поля ввода.

Функция-член класса `TTestGenData::Setup` отвечает за редактирование параметров блока. Она похожа на аналогичную функцию из прошлого примера. Сначала при помощи функции `rdsFORMCreate` создается вспомогательный объект-окно. Затем, при помощи функции `rdsFORMAddEdit`, в него добавляются три поля ввода: выпадающий список (`RDS_FORMCTRL_COMBOLIST`) для выбора типа формируемой функции и два простых поля ввода (`RDS_FORMCTRL_EDIT`) для периода и длительности импульса (рис. 48). На этапе создания полей ввода никак не указывается, что возможность задания длительности импульса будет зависеть от выбранного в выпадающем списке типа функции – этим будет заниматься функция обратного вызова `TestGenDataCheckFunc`. Далее при помощи функции `rdsSetObjectStr` с параметром `RDS_FORMVAL_LIST` в поле ввода для выпадающего списка (поле номер 1) передается строка, содержащая список возможных вариантов выбора, разделенных символом перевода строки “\n”. Значение этого поля ввода равно номеру выбранного из списка варианта начиная с нуля. Таким образом, варианту “Синус” будет соответствовать значение 0, “Косинус” – 1, “Прямоугольные импульсы” – 2.

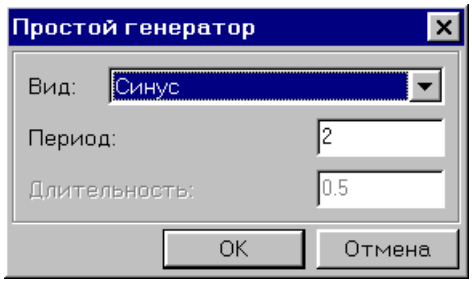


Рис. 48. Окно настройки параметров генератора (поле длительности импульса запрещено)

Затем в каждое из трех полей ввода записываются текущие значения параметров блока (в поле с выпадающим списком записывается целый номер варианта), после чего окно открывается при помощи функции `rdsFORMShowModalEx`. Вторым параметром в эту функцию передается указатель на функцию обратного вызова. Если пользователь закроет окно кнопкой “OK”, `rdsFORMShowModalEx` вернет значение `TRUE`, и значения полей ввода будут считаны при помощи функций `rdsGetObjectInt` и `rdsGetObjectDouble` и

записаны в соответствующие параметры блока. Затем объект окна будет уничтожен вызовом rdsDeleteObject, и функция вернет константу RDS_BFR_MODIFIED (если пользователь нажал “OK”) или RDS_BFR_DONE (если пользователь закрыл окно другим способом).

За функцией TTestGenData::Setup следует тело функции обратного вызова TestGenDataCheckFunc, прототип которой был описан ранее. Эта функция должна иметь тип void RDSCALL и принимать единственный параметр – идентификатор объекта окна типа RDS_HOBJECT. В самой функции производится всего два действия. Сначала при помощи функции rdsGetObjectInt считывается значение поля ввода с идентификатором 1, то есть номер варианта, выбранного в выпадающем списке. Этот номер присваивается вспомогательной переменной type. Затем в поле ввода номер 3 (длительность импульса) передается флаг разрешения редактирования (вызов rdsSetObjectInt с параметром RDS_FORMVAL_ENABLED), истинный при type == 2 (вариант “Прямоугольные импульсы”). В результате ввод данных в поле с идентификатором 3 будет разрешен только в том случае, если в выпадающем списке будет выбран последний вариант. Функция TestGenDataCheckFunc будет вызываться при любом изменении данных в полях ввода, в том числе и при выборе нового варианта из списка, в результате чего разрешенность поля ввода длительности импульса будет всегда соответствовать типу формируемой периодической функции. Например, на рис. 48 выбрано формирование синусоиды, и поле длительности запрещено.

В самой функции модели блока TestGen нет ничего необычного, все используемые в ней реакции на события уже рассматривались ранее. При вызове модели в режиме RDS_BFM_INIT создается объект класса C и указатель на него записывается в BlockData->BlockData. В режиме RDS_BFM_CLEANUP созданный объект удаляется. В режиме RDS_BFM_VARCHECK проверяется допустимость структуры статических переменных блока (должно быть два обязательных сигнала и вещественная переменная двойной точности, то есть строка типа должна равняться “{SSD}”). При вызове модели в режиме RDS_BFM_SETUP вызывается функция-член Setup класса личной области данных блока (указатель на объект класса предварительно присваивается вспомогательной переменной data). Наконец, при запуске расчета (RDS_BFM_STARTCALC) и при изменении единственной используемой в блоке динамической переменной (RDS_BFM_DYNVARCHANGE) вычисляется значение выхода блока в зависимости от текущего значения времени, полученного из “DynTime”, и типа формируемой функции, хранящегося в поле Type класса личной области данных блока. Для значений Type 0 или 1 вычисляется соответственно синус или косинус произведения времени на частоту, вычисленную по заданному пользователем периоду Period (чтобы синус или косинус имел период T, частота должна быть $2\pi/T$). Для значения 2, соответствующего формированию прямоугольных импульсов, вычисляется остаток от деления времени на период и выходу блока присваивается значение 1, если этот остаток меньше длительности импульса, и –1 в противном случае. Если бы значение y вычислялось в такте моделирования, больше ничего не требовалось бы – при запуске модели в режиме RDS_BFM_MODEL сигнал Ready автоматически взводится, что приводит к передаче выходов блока по связям в конце такта. В этой модели выход вычисляется при запуске расчета и при изменении динамического времени, поэтому функция модели должна самостоятельно присвоить сигналу Ready значение 1. При этом значение выхода блока будет передано по связям в конце ближайшего такта расчета, хотя его модель и не будет вызвана в этом такте.

В рассмотренных ранее моделях, использующих динамические переменные, для вычисления начального значения выхода сигналу Start давалось единичное значение по умолчанию, что приводило к обязательному запуску модели блока в первом такте расчета. Модель этого блока не вызывается в такте расчета, поэтому вычисление начального y

производится в реакции на запуск расчета (RDS_BFM_STARTCALC) – при этом просто выполняются те же самые действия, что и при изменении динамической переменной.

Чтобы проверить работу блока с этой моделью, следует установить для него флаг “блок имеет функцию настройки” на вкладке “DLL” окна параметров, поместить в схему блок-планировщик динамического расчета, который создаст переменную “DynTime” и будет управлять ей, и подключить к выходу блока стандартный график из библиотеки РДС, также получающий значение времени из переменной “DynTime” (рис. 49). При запуске расчета график должен отобразить функцию, тип которой выбран в настройках блока. В окне настроек блока поле ввода длительности импульса должно быть активно, только если в выпадающем списке выбран вариант “Прямоугольные импульсы”.

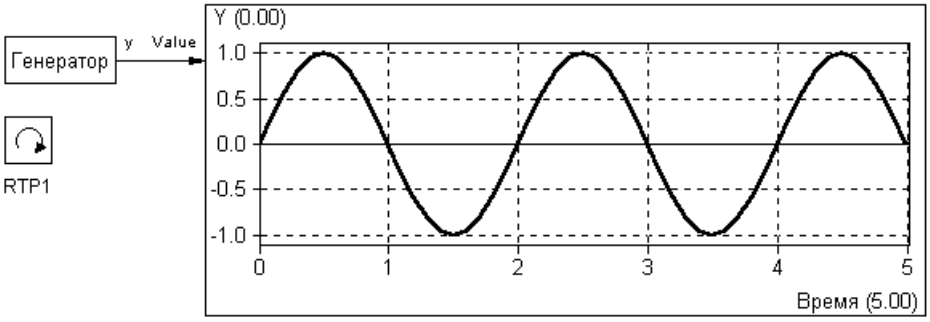


Рис. 49. Проверка работы генератора (выбрано формирование синусоиды)

Эта модель имеет тот же дефект, что и предыдущий пример – при сохранении схемы параметры Type, Period и Impulse не сохраняются, хотя РДС и предупреждает пользователя о наличии изменений в схеме. Позже (стр. 156) мы исправим этот недостаток, добавив в модель соответствующие реакции.

§2.7.3. Расширенные возможности функции обратного вызова

Рассматривается рисование произвольных изображений в модальном окне, формируемом вспомогательным объектом РДС. Для этого в окно настроек блока из предыдущего примера добавляется изображение внешнего вида формируемой генератором функции.

Вместо rdsFORMShowModalEx для открытия модального окна можно использовать функцию rdsFORMShowModalServ, которая позволяет использовать более сложную функцию обратного вызова. В частности, при использовании этой функции можно рисовать в специально выделенных полях окна произвольные изображения.

Усложним предыдущий пример: пусть теперь в окне настроек блока отображается примерный внешний вид формируемой периодической функции, причем прямоугольные импульсы должны рисоваться с правильной скважностью, то есть соотношение периода и длительности импульса у нарисованной функции должно соответствовать введенным пользователем значениям.

Прежде всего необходимо изменить функцию TTestGenData::Setup – нужно добавить в окно специальную панель для рисования и заменить вызов rdsFORMShowModalEx на rdsFORMShowModalServ. Функция приобретет следующий вид (изменения выделены жирным):

```
//====== Функция редактирования параметров ======
int TTestGenData::Setup(void)
{ RDS_HOBJECT window; // Идентификатор вспомогательного объекта
// .....
  // Занесение исходных значений в поля ввода
  rdsSetObjectInt(window,1,RDS_FORMVAL_VALUE,Type);
```

```
  rdsSetObjectDouble(window,2,RDS_FORMVAL_VALUE,Period);
  rdsSetObjectDouble(window,3,RDS_FORMVAL_VALUE,Impulse);
  // Включение дополнительной панели слева от полей ввода
  rdsFORMEnableSidePanel(window,1,-1);
  // Добавление области для рисования графика
  rdsFORMAddEdit(window,1,4,RDS_FORMCTRL_PAINTBOX,NULL,100);
  // Автоматическое вычисление высоты этой области
  rdsSetObjectInt(window,4,RDS_FORMVAL_PBHEIGHT,-1);
  // Открытие окна с другой функцией обратного вызова
  ok=rdsFORMShowModalServ(window,TestGenDataCheckFunc2);
  if(ok)
// .....
  return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
}
```

В этом примере график будет размещаться слева от полей ввода на дополнительной панели, поэтому сначала нужно разрешить отображение этой дополнительной панели при помощи функции `rdsFORMEnableSidePanel`. Второй параметр функции, равный 1, это идентификатор, который присваивается этой панели, при размещении на ней полей его нужно указывать при вызове `rdsFORMAddEdit` вместо идентификатора вкладки. Третий параметр функции (–1) указывает на необходимость автоматического вычисления ширины панели. Можно было бы разместить область рисования на основной панели над или под полями ввода, тогда в вызове функции `rdsFORMEnableSidePanel` не было бы необходимости. Здесь она используется только для иллюстрации возможности размещения полей на двух панелях.

После включения дополнительной панели на нее при помощи вызова `rdsFORMAddEdit` добавляется область рисования (`RDS_FORMCTRL_PAINTBOX`), которой присваивается идентификатор 4. В качестве идентификатора вкладки в функцию передается число 1, которое было использовано при вызове `rdsFORMEnableSidePanel`. Заголовка у области рисования нет (передается `NULL`), ширина устанавливается в 100 точек экрана. Для области рисования нужно задать не только ширину, но и высоту, поэтому за вызовом `rdsFORMAddEdit` следует вызов `rdsSetObjectInt` с параметрами `RDS_FORMVAL_PBHEIGHT` (установка высоты области рисования) и –1 (высота вычисляется автоматически). В результате на дополнительную панель будет добавлена область рисования с идентификатором 4, шириной в 100 точек и высотой во всю панель. Внутри этой области и будет рисоваться примерный вид графика функции.

Для открытия модального окна теперь используется функция `rdsFORMShowModalServ`, поэтому вместо старой функции обратного вызова `TestGenDataCheckFunc` теперь указана `TestGenDataCheckFunc2` – эту функцию еще предстоит написать. Она будет принимать не один, а два параметра: идентификатор окна `RDS_HOBJECT` и указатель на структуру `RDS_FORMSERVFUNCDATA`, в которой содержатся дополнительные параметры вызова. Эта структура описана в файле “RdsDef.h” следующим образом:

```
typedef struct
{ int Event;          // Событие (RDS_FORMSERVEVENT_*)
  int CtrlId;         // Идентификатор поля или -1
  // Для RDS_FORMSERVEVENT_DRAW
  HDC dc;             // Контекст устройства рисования Windows
  int Left,Top;       // Верхний левый угол зоны рисования
  int Width,Height;   // Размеры зоны рисования
} RDS_FORMSERVFUNCDATA;
// Указатель на структуру
typedef RDS_FORMSERVFUNCDATA *RDS_PFORMSERVFUNCDATA;
```

- `int Event` – событие, произошедшее в модальном окне. В данный момент может принимать два значения: `RDS_FORMSERVEVENT_CHANGE` (изменение одного или нескольких полей ввода) или `RDS_FORMSERVEVENT_DRAW` (рисование в специальных областях).
- `int CtrlId` – идентификатор поля, с которым связано событие, или –1. Например, для события `RDS_FORMSERVEVENT_DRAW` в этом поле передается идентификатор области, которую необходимо перерисовать, а для события `RDS_FORMSERVEVENT_CHANGE` – идентификатор изменившегося поля или –1, если изменилось сразу несколько полей.
- `HDC dc` – контекст устройства Windows, на котором нужно рисовать при реакции на `RDS_FORMSERVEVENT_DRAW`.
- `int Left, Top, Width, Height` – положение и размеры области рисования.

Для данного примера функция обратного вызова должна выглядеть следующим образом:

```
void RDSCALL TestGenDataCheckFunc2(RDS_HOBJECT win,
                                   RDS_PFORMSERVFUNCDATA data)
{ // Считать номер пункта выпадающего списка
  int type=rdsGetObjectInt(win,1,RDS_FORMVAL_VALUE);
  // Вспомогательные переменные для рисования графика
  int y0,y_ampl,x0,x1;
  double pix_period;

  switch(data->Event)
    { // Изменение поля ввода
      case RDS_FORMSERVEVENT_CHANGE:
        // Запретить ввод длительность импульса для sin и cos
        rdsSetObjectInt(win,3,RDS_FORMVAL_ENABLED,type==2);
        // Перерисовать график
        rdsCommandObject(win,RDS_FORM_INVALIDATE);
        break;

      // Рисование
      case RDS_FORMSERVEVENT_DRAW:
        // Заливка фона белым цветом
        rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xffffff);
        rdsXGFillRect(data->Left,
                      data->Top,
                      data->Left+data->Width,
                      data->Top+data->Height);
        // Координаты рисования
        x0=data->Left+10;             // Начало графика
        x1=data->Left+data->Width-10; // Конец графика
        y0=data->Top+data->Height/2;  // Центр по вертикали
        y_ampl=(data->Height-20)/2;   // Амплитуда
        pix_period=0.5*(x1-x0);       // Период на рисунке
        // Координатные оси
        rdsXGSetPenStyle(0,PS_SOLID,1,0,R2_COPYPEN);
        rdsXGMoveTo(data->Left+5,y0);
        rdsXGLineTo(data->Left+data->Width-5,y0);
        rdsXGMoveTo(x0,data->Top+5);
        rdsXGLineTo(x0,data->Top+data->Height-5);
        // График
        rdsXGSetPenStyle(RDS_GFWIDTH,0,3,0,0);
        if(type==2) // Прямоугольные импульсы
          { double period,impulse,pix_impulse;
            // Чтение введенных пользователем значений
            period=rdsGetObjectDouble(win,2,RDS_FORMVAL_VALUE);
            impulse=rdsGetObjectDouble(win,3,RDS_FORMVAL_VALUE);
```

```
                if(period==0.0) // Нельзя вычислить частоту
                  return;
                // Длительность импульса на рисунке
                pix_impulse=impulse*pix_period/period;
                // Первый период
                rdsXGMoveTo(x0,y0+y_ampl);
                rdsXGLineTo(x0,y0-y_ampl);
                rdsXGLineTo(x0+pix_impulse,y0-y_ampl);
                rdsXGLineTo(x0+pix_impulse,y0+y_ampl);
                rdsXGLineTo(x0+pix_period,y0+y_ampl);
                // Второй период
                rdsXGLineTo(x0+pix_period,y0-y_ampl);
                rdsXGLineTo(x0+pix_period+pix_impulse,y0-y_ampl);
                rdsXGLineTo(x0+pix_period+pix_impulse,y0+y_ampl);
                rdsXGLineTo(x1,y0+y_ampl);
              }
            else // Синус или косинус
              { double t,y;
                // Цикл по горизонтали с шагом в 3 точки
                for(int x=x0;x<=x1;x+=3)
                  { t=2*M_PI*(x-x0)/pix_period;
                    y=y_ampl*((type==0)?sin(t):cos(t));
                    if(x==x0) // Первая точка - установить позицию
                      rdsXGMoveTo(x,y0-y);
                    else // Рисовать линию от предыдущей точки
                      rdsXGLineTo(x,y0-y);
                  } // for(int x=x0...)
              }
            break;
        } // switch
    }
    //=======================================
```

Тело функции можно разместить перед функцией `TTestGenData::Setup` или, как в предыдущем примере, поместить там прототип, а тело функции разместить в любом другом месте.

В самом начале функции `TestGenDataCheckFunc2`, как и в `TestGenDataCheckFunc`, во вспомогательную переменную `type` считывается выбранный в выпадающем списке тип формируемой функции. Затем, в зависимости от события окна (`data->Event`), производятся различные действия.

При изменении какого-либо поля ввода (событие `RDS_FORMSERVEVENT_CHANGE`) разрешается или запрещается ввод данных в поле длительности импульса, так же, как и в предыдущем примере в функции `TestGenDataCheckFunc`. После этого вызов `rdsCommandObject` с параметром `RDS_FORM_INVALIDATE` информирует окно о необходимости перерисовки – если изменились тип функции, период или длительность импульса, внешний вид графика нужно изменить в соответствии с новыми параметрами.

Когда возникнет необходимость обновить изображение на панели рисования окна, функция `TestGenDataCheckFunc2` будет вызвана с параметром `data->Event`, равным `RDS_FORMSERVEVENT_DRAW`. В функцию передается контекст рисования `data->dc`, поэтому для рисования можно использовать любые функции Windows API, работающие с контекстами устройств. Чтобы не загромождать пример, вместо функций API здесь будут использоваться графические функции-оболочки РДС – хотя они и не так богаты, как функции рисования API, ими гораздо проще пользоваться, и для этого примера их вполне достаточно.

В этом примере мы будем рисовать на панели черный график функции на белом фоне. Независимо от того, какой график будет рисоваться на панели, сначала нужно закрасить всю панель белым цветом. Для этого прежде всего следует установить цвет заливки геометрических фигур при помощи сервисной функции `rdsXGSetBrushStyle`. Функция принимает следующие параметры:

```
void RDSCALL rdsXGSetBrushStyle(
     int        Mask,      // Набор устанавливаемых параметров
     int        Style,     // Стиль заливки
     COLORREF   Color);    // Цвет заливки
```

В данном случае параметр `Mask` равен нулю (устанавливаются оба параметра заливки), параметр `Style` – константе `RDS_GFS_SOLID` (сплошная заливка), а параметр `Color` – шестнадцатеричной константе `0xFFFFFF`, соответствующей белому цвету (интенсивности всех трех компонентов цвета равны 255 (`0xFF`), то есть максимальны). После этого вызова все геометрические фигуры, которые будут рисоваться на данной панели, будут иметь сплошной белый цвет фона. Теперь можно закрасить всю панель установленным цветом, вызвав функцию заполнения прямоугольника `rdsXGFillRect`:

```
void RDSCALL rdsXGFillRect(
     int   Left,      // Левая граница
     int   Top,       // Верхняя граница
     int   Right,     // Правая граница
     int   Bottom);   // Нижняя граница
```

Границы закрашиваемого прямоугольника определяются полями `Left`, `Top`, `Width` и `Height` структуры `RDS_FORMSERVFUNCDATA`, указатель на которую функция получает через параметр `data`. `Left` и `Top` – это левая и верхняя границы области рисования, `Width` и `Height` – ширина и высота. Таким образом, правая граница закрашиваемой области вычисляется как `data->Left+data->Width`, нижняя – `data->Top+data->Height`.

Для рисования графика необходимо вычислить несколько вспомогательных параметров. Будем считать, что горизонтальная ось графика проходит точно по центру панели, не доходя до левого и правого краев на 5 точек, а вертикальная ось отстоит от левого края панели на 10 точек, также не доходя до верхнего и нижнего краев на 5 точек (рис. 50). Для наглядности будем изображать на панели два периода графика, причем между правым краем графика и правым краем панели должен быть зазор в 10 точек. Максимум и минимум графика также должны отстоять от краев панели на 10 точек. Таким образом получаем:

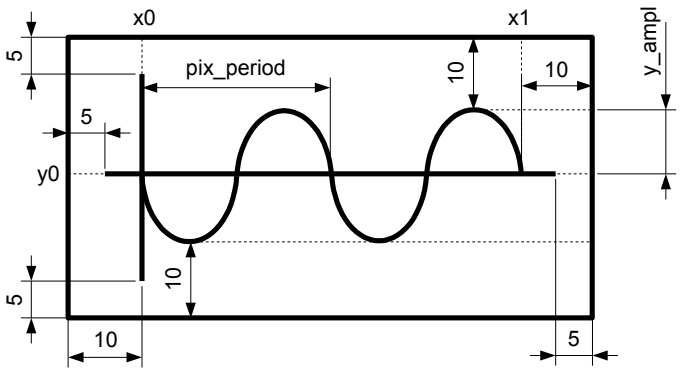


Рис. 50. Координаты и размеры графика в точках экрана

Начало графика отстоит на 10 точек от левого края панели:

```
x0 = data->Left + 10;
```

Конец графика отстоит на 10 точек от правого края панели:

```
x1 = data->Left + data->Width – 10;
```

Горизонтальная ось проходит по центру панели, то есть отстоит от верхнего края на половину высоты:

```
y0 = data->Top + data->Height/2;
```

Амплитуда графика равна половине высоты панели с запасом в 20 точек:

```
y_ampl = (data->Height – 20)/2;
```

Между x0 и x1 умещается два периода графика:

```
pix_period = 0.5*(x1 – x0);
```

После того, как вспомогательные параметры вычислены, нужно нарисовать оси координат. Для этого сначала следует установить тип и цвет рисуемых линий при помощи сервисной функции rdsXGSetPenStyle:

```
void RDSCALL rdsXGSetPenStyle(
     int        Mask,      // Набор устанавливаемых параметров
     int        Style,     // Стиль линии
     int        Width,     // Толщина линии
     COLORREF   Color,     // Цвет линии
     int        Mode);     // Режим рисования
```

Нужно установить все параметры линии, поэтому параметр Mask в вызове равен нулю. В качестве стиля линии передается стандартная константа Windows API PS_SOLID (сплошная линия), толщина линии устанавливается равной одной точке, линия будет рисоваться черным (параметр Color равен нулю, что означает нулевую интенсивность всех трех компонентов цвета, то есть черный цвет). В качестве режима рисования линии передается стандартная константа API R2_COPYPEN – точки линии будут окрашены в установленный цвет независимо от цвета фона под линией. Эти параметры будут использоваться при рисовании всех линий и контуров геометрических фигур до следующего вызова функции rdsXGSetPenStyle, или до тех пор, пока стиль линии не будет изменен средствами Windows API.

Теперь, когда установлен тип линий, можно рисовать оси координат. Для рисования линий будут использоваться функции rdsXGMoveTo и rdsXGLineTo. Первая устанавливает координаты точки начала рисования, вторая рисует линию от последней точки до точки с указанными в параметрах координатами (аналоги таких функций есть в большинстве графических библиотек). Для того, чтобы нарисовать прямую линию, нужно сделать один вызов rdsXGMoveTo, указав координаты начала линии, и один вызов rdsXGLineTo, указав координаты конца.

После того, как оси нарисованы, нужно изобразить внешний вид графика формируемой функции в зависимости от типа, выбранного в выпадающем списке (номер варианта уже считан во вспомогательную переменную type). Чтобы график четко выделялся на фоне координатных осей, будем рисовать его линией толщиной в три точки. Для установки новой толщины линии снова используется функция rdsXGSetPenStyle, только теперь в ее параметре Mask передается константа RDS_GFWIDTH (описана в “RdsDef.h”), указывающая на то, что будет изменена только толщина линии. В параметре Width передается новая толщина (3), а остальные параметры будут проигнорированы функцией, поэтому их значения не важны – в данном случае для их заполнения используются нули.

Рисование графика будет производиться по-разному для прямоугольных импульсов и для синуса или косинуса. Импульсы можно изобразить несколькими линиями, а тригонометрические функции лучше строить по точкам с заданным шагом по горизонтали. Кроме того, внешний вид прямоугольных импульсов будет зависеть от соотношения периода функции и длительности импульса.

При рисовании прямоугольных импульсов (значение type равно 2) необходимо вычислить длительность импульса на рисунке. Мы условились, что на рисунке должно изображаться два периода функции, и, исходя из этого, рассчитали период графика в точках экрана (вспомогательная переменная pix_period). Чтобы вычислить длительность импульса в точках экрана, нужно умножить pix_period на отношение введенной пользователем длительности к введенному им периоду. Введенные период и длительность считываются во вспомогательные переменные period и impulse соответственно, после чего, если значение периода не нулевое, вычисляется длительность импульса на рисунке

`pix_impulse`. Теперь можно нарисовать два периода графика одной ломаной линией. Сначала нужно установить начальную точку рисования при помощи функции `rdsXGMoveTo`, а затем рисовать ломаную линию последовательными вызовами `rdsXGLineTo` (каждый следующий вызов будет рисовать линию от конца предыдущей линии до указанных в параметрах координат).

Рисование тригонометрических функций (значение `type` не равно 2) осуществляется в цикле от `x0` до `x1` с шагом в 3 точки. Внутри цикла вычисляется значение синуса или косинуса (в зависимости от `type`) и вызывается `rdsXGMoveTo` для самой первой точки или `rdsXGLineTo` для всех остальных.

Теперь при вызове функции настройки этого блока на панели слева от полей ввода будет изображаться внешний вид графика функции, соответствующий выбранному в выпадающем списке варианту (рис. 51).

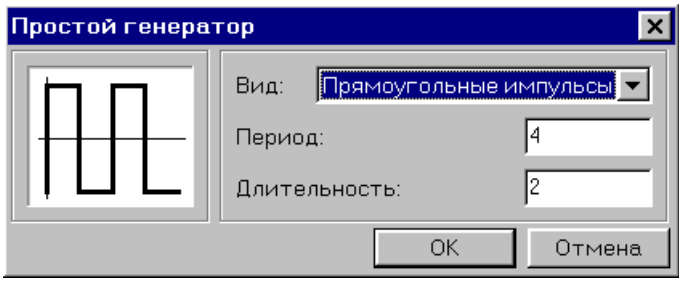


Рис. 51. Окно настроек генератора с изображением формируемой функции

§2.7.4. Хранение настроечных параметров в переменных блока

Описывается хранение настроечных параметров блока в значениях по умолчанию его статических переменных. Этот способ не требует введения в модель новых реакций, к тому же, он позволяет пользователю выбирать: задавать ли ему параметр вручную в окне настроек блока или подключить к нему связь и передавать ему значение откуда-нибудь из схемы. Приведен пример блока, вычисляющего $y=K \times x + C$, где x – вход, а K и C могут либо задаваться в окне настройки, либо получать свои значения по связям от других блоков.

В предыдущих примерах параметры, которые редактировались в функции настройки, были полями класса или структуры личной области данных блока. Это позволяет обращаться к ним по именам, и вообще оперировать ими, как обычными переменными в программе на С. Однако, при этом необходимо включать в функцию модели реакции на события загрузки и сохранения параметров блоков (они рассмотрены в §2.8), поскольку РДС не может самостоятельно сохранить личную область данных. Кроме того, при таком подходе приходится четко разделять параметры, задаваемые в функции настройки, и параметры, получаемые блоком по связям, так как к переменной внутри личной области данных не может быть подключена связь. Это не очень удобно, поскольку обычно нельзя заранее сказать, какие из параметров пользователь захочет вводить в функции настройки, а какие – подавать с полей ввода или других блоков.

Чтобы не писать реакции на загрузку и сохранение параметров, а заодно дать пользователю возможность самому выбирать, какие параметры он будет вводить в функции настройки, можно хранить настроечные параметры в значениях по умолчанию статических входов блока. Во-первых, значения переменных по умолчанию автоматически сохраняются при сохранении схемы. Во-вторых, если к такому входу не подключать связь, значение переменной останется равным значению по умолчанию, то есть значению, введенному в функции настройки, а если пользователь соединит этот вход с выходом другого блока, значение входа будет определяться сработавшей связью. Таким образом один и тот же параметр может и вводиться в функции настройки, и работать, как обычный вход блока.

Нужно только отслеживать наличие соединенной с этим входом связи и не разрешать пользователю изменять параметр, если эта связь существует. Многие блоки из стандартной библиотеки “Common.dll” устроены именно так.

Для примера создадим блок, вычисляющий значение выхода `y` по формуле `y=K x + C`, где `x` – обычный вход блока, а `K` и `C` – входы блока, которые также могут задаваться в функции настройки. Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*
0	Start	Сигнал	1	Вход	✓
1	Ready	Сигнал	1	Выход	
2	x	double	8	Вход	✓
10	K	double	8	Вход	✓
18	C	double	8	Вход	✓
26	y	double	8	Выход	

Перед тем, как начать писать функцию модели блока и функцию настройки, создадим пару вспомогательных функций, которые будут в них использоваться. Первая функция, `CheckBlockInputConnection`, будет получать описание входа блока с заданным номером и проверять, соединена ли с этим входом какая-нибудь связь:

```
// Проверка наличия связи у входа блока
BOOL CheckBlockInputConnection(
     RDS_BHANDLE Block,                // Идентификатор блока
     int num,                          // Номер входа
     RDS_PVARDESCRIPTION pVarDescr)    // Указатель на структуру
                                       // описания переменной
{ RDS_CHANDLE c;                  // Идентификатор связи
  RDS_POINTDESCRIPTION PtDescr; // Структура описания точки связи

  // Заполнение служебных полей структур их размерами
  PtDescr.servSize=sizeof(PtDescr);
  pVarDescr->servSize=sizeof(RDS_VARDESCRIPTION);

  // Получение описания переменной блока по номеру
  if(rdsGetBlockVar(Block,num,pVarDescr)==NULL)
    return FALSE; // Нет такой переменной

  // Перебор всех связей, подключенных к этому блоку
  c=NULL;
  for(;;)
    { // Найти связь, следующую за c, и заполнить структуру
      // описания точки соединения PtDescr
      c=rdsGetBlockLink(Block,c,TRUE,FALSE,&PtDescr);
      if(c==NULL) // Больше нет связей
        break;
      // Найдена очередная связь - сравнение имени заданной
      // переменной с именем переменной точки этой связи
      if(strcmp(PtDescr.VarName,pVarDescr->Name)==0)
        return TRUE; // Имена совпали - есть связь, соединенная
                     // с переменной блока
    }
  // Все подключенные связи перебраны, а связь, подключенная
  // к заданной переменной так и не была найдена
  return FALSE;
```

```
}
//=======================================
```

Функция принимает три параметра: идентификатор блока, для переменной которого проверяется наличие связи (Block), порядковый номер переменной в блоке (num) и указатель на внешнюю структуру описания переменной (pVarDescr). Возвращаемое функцией значение должно сигнализировать о наличии (TRUE) или отсутствии (FALSE) связи, соединенной с указанной переменной. Кроме того, функция будет записывать описание этой переменной в структуру, указатель на которую передан в параметре pVarDescr – из этой структуры можно будет считать имя переменной для формирования заголовка поля ввода.

В начале функции описаны две вспомогательные переменные: идентификатор связи c и структура описания точки связи PtDescr. Идентификатор связи будет использоваться при поиске связей, соединенных с блоком Block, а через структуру описания точки можно будет определить имя переменной блока, к которой присоединена найденная связь.

Прежде чем можно будет вызывать сервисные функции РДС, заполняющие структуры описания переменной и точки связи, необходимо присвоить полю servSize каждой из них размер соответствующей структуры – чтобы избежать конфликта версий, сервисные функции проверяют размер структур, с которыми они работают. Затем описание переменной с номером num записывается в структуру по указателю pVarDescr при помощи сервисной функции rdsGetBlockVar:

```
RDS_VHANDLE RDSCALL rdsGetBlockVar(
      RDS_BHANDLE Block,              // Блок
      int varnum,                     // Номер переменной блока
      RDS_PVARDESCRIPTION pDescr);    // Заполняемая структура
                                      // описания переменной
```

В функцию передается идентификатор блока (параметр Block) и порядковый номер переменной в этом блоке, начиная с нуля (параметр varnum). Если переменная с таким номером существует в блоке, функция возвращает уникальный идентификатор этой переменной и заполняет ее описанием структуру, указатель на которую передан в параметре pDescr. Если же такой переменной нет в блоке, функция возвращает NULL.

В этом примере функции CheckBlockInputConnection не требуется уникальный идентификатор переменной – возвращенное функцией значение просто проверяется на NULL. Если CheckBlockInputConnection вернула NULL, значит, переменной с номером num нет в блоке Block, и никакой связи у этой отсутствующей переменной быть не может – функция возвращает FALSE. В противном случае переменная существует, и структура pVarDescr (типа RDS_VARDESCRIPTION) заполнена ее описанием. Эта структура подробно описана в приложении А, сейчас в ней нас будет интересовать единственное поле Name – указатель на строку с именем переменной.

Теперь нужно перебрать все связи, подключенные ко входам блока, и попробовать найти ту, которая соединена с нужной переменной. Для перебора связей будет использоваться сервисная функция rdsGetBlockLink:

```
RDS_CHANDLE RDSCALL rdsGetBlockLink(
      RDS_BHANDLE Block,    // Идентификатор блока
      RDS_CHANDLE Conn,     // Предыдущая связь
      BOOL Inputs,          // Искать соединенные со входами
      BOOL Outputs,         // Искать соединенные с выходами
      RDS_PPOINTDESCRIPTION pPointDescr);   // Заполняемая структура
                                            // описания точки связи
```

Если параметр Conn равен NULL, функция ищет первую связь, соединенную с блоком Block. Если Conn не NULL, функция ищет следующую после Conn связь. Таким образом, последовательно вызывая rdsGetBlockLink и передавая в параметре Conn результат прошлого вызова, можно перебрать все связи, соединенные с блоком. Если в параметре

Conn передать идентификатор последней связи, функция вернет NULL, сигнализируя о том, что больше у блока связей нет. Параметр Inputs указывает на необходимость искать связи, соединенные со входами блока, Outputs – на необходимость искать связи, соединенные с выходами. Хотя бы один из этих параметров должен быть истинным, иначе функция сразу вернет NULL – бессмысленно искать связь блока, не соединенную ни с входом, ни с выходом. В параметре pPointDescr передается указатель на структуру, которая заполняется описанием точки связи, соответствующей входу или выходу данного блока.

Связи блока перебираются в цикле for, перед которым вспомогательной переменной c присваивается значение NULL. В цикле вызывается rdsGetBlockLink, в которую в качестве идентификатора предыдущей связи передается значение c (сначала c равно NULL, поэтому при первом вызове функция найдет первую связь, соединенную с блоком). Поскольку мы проверяем наличие связей у входа блока, в параметре Inputs передается TRUE, а в параметре Outputs – FALSE. Функция должна вернуть идентификатор найденной связи (присваивается переменной c) и заполнить структуру PtDescr типа RDS_POINTDESCRIPTION описанием точки, которая соединяет эту связь с блоком. Обычно такая точка размещается на границе картинки блока и около нее отображается имя переменной, к которой подсоединена связь. Описание структуры RDS_POINTDESCRIPTION приведено в приложении А, нам потребуется только ее поле VarName, в котором содержится указатель на строку с именем переменной блока, к которой присоединена связь.

Если функция rdsGetBlockLink не нашла очередной связи блока, значение переменной c будет равно NULL – в этом случае цикл будет прерван оператором break. В противном случае имя переменной из структуры описания переменной блока pVarDescr сравнивается с именем переменной из структуры описания точки связи PtDescr при помощи стандартной функции сравнения строк strcmp. При совпадении имен strcmp вернет 0. Это означает, что найденная при помощи rdsGetBlockLink связь соединена с заданной переменной. В этом случае возвращается TRUE – мы проверяли, подключена ли связь к переменной блока с номером num, и только что нашли такую связь. Если же имена не совпали, цикл продолжается, при этом в переменной c содержится идентификатор найденной связи, и при следующем вызове rdsGetBlockLink найдет следующую связь. Так будет продолжаться до тех пор, пока имя переменной в структуре PtDescr не совпадет с именем заданной переменной и не выполнится команда return TRUE, либо пока не будут перебраны все связи и цикл не будет прерван оператором break – в этом случае возвращается FALSE, поскольку все связи блока перебраны, и ни одна из них не соединяется с заданной переменной.

Следует отметить, что функцию CheckBlockInputConnection нельзя применять для проверки наличия связей у входов-структур и входов-массивов – она может работать только с простыми переменными. Дело в том, что у сложных переменных связь может быть присоединена не только ко всей переменной целиком, но и к ее части, например, к полю структуры или элементу массива. Например, если переменная M – массив, то связь может быть присоединена к M[0], M[1] и т.д. Внутри функции имя заданной переменной сравнивается с именем переменной, взятым из точки связи, и, поскольку строки “M” и “M[0]” не совпадают, связь, идущая к элементу массива не будет найдена. Однако, поскольку нам эта функция будет нужна для проверки наличия связи у хранящегося в переменной блока параметра, на это ограничение можно не обращать внимания – параметры обычно хранятся в простых переменных.

Может возникнуть вопрос: почему в функцию CheckBlockInputConnection передается номер переменной, если для поиска связей используется ее имя? Если бы в функцию сразу передавалось имя переменной, можно было бы не вызывать rdsGetBlockVar. Однако, в этом случае при изменении имени переменной пришлось бы переписывать модель блока, поскольку строка имени переменной была бы жестко “зашита” в

вызов CheckBlockInputConnection. Использование номеров переменных позволяет привязываться не к их именам, а к порядку их следования, изменять который в любом случае не следует – большинство моделей рассчитано на жесткую структуру статических переменных, проверяемую в вызове модели с параметром RDS_BFM_VARCHECK.

Теперь напишем функцию, которая, в зависимости от наличия подключенной к вещественному (double) входу связи, будет добавлять в окно настройки, созданное сервисной функцией rdsFORMCreate (см. §2.7.2), поле для ввода значения переменной (если связи нет) или поле для его индикации (если связь есть). Кроме того, при наличии связи к заголовку поля ввода будет добавляться текст "подключена связь". Как и CheckBlockInputConnection, эта функция будет возвращать логическое значение, соответствующее наличию или отсутствию связи у заданного входа.

```
BOOL AddWinEditOrDisplayDouble(
     RDS_HOBJECT window,   // Идентификатор объекта-окна
     RDS_BHANDLE Block,    // Блок
     int varnum,           // Номер переменной в блоке
     int ctrlnum,          // Идентификатор поля ввода в окне
     char *title)          // Заголовок поля или NULL
{ // Структура описания переменной блока
  RDS_VARDESCRIPTION VarDescr;
  // Проверка наличия связи у переменной varnum в блоке Block
  // и заполнение структуры VarDescr описанием переменной
  BOOL conn=CheckBlockInputConnection(Block,varnum,&VarDescr);

  if(conn) // К переменной подключена связь
    { // Вспомогательные переменные
      char *caption;  // Заголовок поля
      double *cur;    // Указатель на данные переменной
      // Заголовок поля формируется из имени переменной и
      // текста "подключена связь"
      caption=rdsDynStrCat(title?title:VarDescr.Name,
                           " (подключена связь)",
                           FALSE);
      // Добавление поля для индикации текущего значения
      rdsFORMAddEdit(window,0,ctrlnum,RDS_FORMCTRL_DISPLAY,
                     caption,80);
      // Освобождение динамически сформированной строки
      // заголовка поля
      rdsFree(caption);
      // Получение указателя на данные переменной
      cur=(double*)rdsGetBlockVarBase(Block,varnum,NULL);
      // Проверка – переменная должна существовать и иметь
      // тип double
      if(cur!=NULL && VarDescr.Type=='D')
        // Занесение текущего значения переменной в поле
        rdsSetObjectDouble(window,ctrlnum,RDS_FORMVAL_VALUE,
                           *cur);
    }
  else // К переменной не подключена связь
    { // Вспомогательная переменная для значения по умолчанию
      char *defval;
      // Получение строки со значением переменной по умолчанию
      // (необходимо потом освободить при помощи rdsFree)
      defval=rdsGetBlockVarDefValueStr(Block,varnum,NULL);
      // Добавление поля для ввода параметра
      rdsFORMAddEdit(window,0,ctrlnum,RDS_FORMCTRL_EDIT,
                     title?title:VarDescr.Name,80);
```

```
            // Занесение в поле ввода значения перменной по умолчанию
            rdsSetObjectStr(window,ctrlnum,RDS_FORMVAL_VALUE,defval);
            // Освобождение динамически сформированной строки
            rdsFree(defval);
          }
        // Возврат: TRUE - есть связь, FALSE - нет связи
        return conn;
      }
      //=======================================
```

Функция принимает пять параметров: идентификатор объекта-окна (`window`), идентификатор блока, значение переменной которого будет редактироваться или отображаться (`Block`), номер переменной в блоке (`varnum`), идентификатор поля, которое должно быть добавлено в окно (`ctrlnum`), и строка-заголовок поля ввода (`title`), вместо которой может быть передано значение `NULL` для использования в качестве заголовка имени переменной. В начале функции описывается структура `VarDescr`, после чего вызывается ранее написанная функция `CheckBlockInputConnection`, которая заносит в нее описание переменной с номером `varnum` в блоке `Block` и возвращает логическое значение, соответствующее наличию связи, присоединенной к этой переменной. Это значение присваивается вспомогательной переменной `conn`.

Если `conn` истинно (`TRUE`), у переменной есть связь, и ее значение нельзя редактировать. В этом случае надо добавить в окно `window` поле для индикации текущего значения переменной и сформировать заголовок этого поля из параметра `title` или имени переменной (его можно получить из поля `Name` структуры `VarDescr`) и текста “подключена связь”. Сначала при помощи функции `rdsDynStrCat` в памяти формируется строка заголовка и указатель на нее присваивается вспомогательной переменной `caption` – позже эту строку нужно будет освободить функцией `rdsFree`. В первых двух параметрах функции передаются объединяемые строки, в третьем – значение `FALSE`, запрещающее функции возвращать `NULL` вместо пустой строки (в данном случае можно было передать и `TRUE`, поскольку второй параметр функции – не пустая строковая константа, и в результате объединения строк не может получиться пустая строка). Затем вызывается функция `rdsFORMAddEdit`, добавляющая в окно `window` поле индикации (`RDS_FORMCTRL_DISPLAY`) с идентификатором `ctrlnum` и шириной в 80 точек экрана. В качестве заголовка поля передается динамически сформированная строка `caption`. После вызова `rdsFORMAddEdit` строка `caption` больше не нужна, и она освобождается при помощи `rdsFree`.

Теперь в добавленное поле необходимо занести текущее значение переменной. Раньше для доступа к переменным в моделях блоков использовались макросы, вычислявшие начальный адрес переменной блока по начальному адресу дерева переменных из структуры данных блока (`RDS_BLOCKDATA`) и фиксированному смещению. В эту функцию не передается указатель на структуру данных блока, поэтому макросы использовать нельзя – неоткуда получить начальный адрес дерева переменных. Для получения начального адреса переменной будет использоваться сервисная функция `rdsGetBlockVarBase` – это медленнее, чем обращение к переменной по фиксированному смещению, но при открытии окна настроек высокие скорости не нужны:

```
LPVOID RDSCALL rdsGetBlockVarBase(
      RDS_BHANDLE Block,    // Идентификатор блока
      int num,              // Номер переменной
      int *pSize);          // Возвращаемый размер переменной
```

В функцию передается идентификатор блока, номер переменной в нем, а также указатель на целую переменную, в которую нужно записать размер переменной блока в дереве. В данном случае нам не нужен размер переменной (мы знаем, что переменная должна иметь тип `double` и ее размер должен равняться восьми байтам), поэтому в параметре `pSize`

передается NULL. Возвращаемое функцией значение – начальный адрес переменной – приводится к типу “указатель на double” и записывается во вспомогательную переменную cur. Перед тем, как обращаться к переменной, нужно на всякий случай проверить, существует ли в блоке переменная с таким номером и имеет ли она тип double. Для проверки наличия переменной значение cur сравнивается с NULL: если переменной с указанным номером нет в блоке, rdsGetBlockVarBase возвращает нулевое значение. Тип переменной можно считать из поля Type структуры VarDescr – как и везде в РДС, типу double соответствует символ “D”. Если оба условия выполнены, текущее значение переменной заносится в поле индикации функцией rdsSetObjectDouble.

Если conn имеет значение FALSE, к переменной не присоединено ни одной связи, и пользователю можно разрешить редактировать ее значение по умолчанию. Для получения значения переменной по умолчанию (оно может отличаться от текущего значения переменной, если ранее к ней была подключена связь и схема некоторое время проработала в режиме расчета), будет использоваться функция rdsGetBlockVarDefValueStr:

```
LPSTR RDSCALL rdsGetBlockVarDefValueStr(
      RDS_BHANDLE Block,    // Идентификатор блока
      int num,              // Номер переменной
      int *pLength);        // Возвращаемая длина строки
```

Как и многие другие функции для работы с переменными блока, эта функция принимает идентификатор блока и номер переменной в этом блоке. Независимо от типа переменной, функция возвращает ее значение по умолчанию в виде указателя на динамически сформированную в памяти строку (ее нужно будет освободить при помощи rdsFree), при этом, если параметр pLength не равен NULL, в целую переменную, на которую указывает pLength, записывается длина этой строки. В данном случае мы работаем с переменными блока типа double, поэтому строка, возвращенная функцией, будет содержать символьное представление вещественного числа двойной точности (число знаков после запятой будет подобрано автоматически). Длина строки нас не интересует, поэтому в параметре pLength передается NULL. Указатель на строку, возвращенный функцией, записывается во вспомогательную переменную defval.

Теперь функцией rdsFORMAddEdit в окно window добавляется поле ввода (RDS_FORMCTRL_EDIT) шириной в 80 точек экрана с идентификатором ctrlnum. В качестве заголовка поля используется значение параметра title, или, если оно равно NULL, имя переменной блока из поля Name структуры описания переменной. При помощи функции rdsSetObjectStr в поле ввода заносится строка со значением переменной по умолчанию defval, после чего память, занятая этой строкой, освобождается при помощи rdsFree.

После того, как обе вспомогательные функции готовы, можно заняться моделью блока с функцией настройки:

```
// Функция настройки K и C для модели блока y=Kx+C
BOOL TestKxCSetup(
      RDS_BHANDLE Block,    // Идентификатор блока
      int numK,             // Номер переменной K в блоке
      int numC)             // Номер переменной C в блоке
{ RDS_HOBJECT window; // Идентификатор вспомогательного объекта
  BOOL ok;            // Пользователь нажал "OK"
  BOOL K_conn,C_conn; // Флаги наличия связей у K и C

  // Создание окна
  window=rdsFORMCreate(FALSE,-1,-1,"Kx+C");

  // Добавление полей для ввода или индикации K и C
  // (в зависимости от наличия связей)
  K_conn=AddWinEditOrDisplayDouble(window,Block,numK,1,NULL);
```

```
  C_conn=AddWinEditOrDisplayDouble(window,Block,numC,2,NULL);

  // Открытие окна
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Нажата кнопка OK - запись параметров в блок
      if(!K_conn) // У K нет связи
        { // Получение строки из поля ввода
          char *str=rdsGetObjectStr(window,1,RDS_FORMVAL_VALUE);
          // Установка значения переменной K по умолчанию
          rdsSetBlockVarDefValueStr(Block,numK,str);
        }
      if(!C_conn) // У C нет связи
        { // Получение строки из поля ввода
          char *str=rdsGetObjectStr(window,2,RDS_FORMVAL_VALUE);
          // Установка значения переменной C по умолчанию
          rdsSetBlockVarDefValueStr(Block,numC,str);
        }
    }
  // Уничтожение окна
  rdsDeleteObject(window);
  // Возвращаемое значение - истина, если нажата "OK"
  return ok;
}
//=======================================

// Функция модели блока
extern "C" __declspec(dllexport)
  int RDSCALL TestKxC(int CallMode,
                      RDS_PBLOCKDATA BlockData,
                      LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))   // 0
#define Ready (*((char *)(pStart+1))) // 1
#define x (*((double *)(pStart+2)))   // 2
#define K (*((double *)(pStart+10)))  // 3
#define C (*((double *)(pStart+18)))  // 4
#define y (*((double *)(pStart+26)))  // 5
  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDDDD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Запуск расчета
      case RDS_BFM_STARTCALC:
        // Если это запуск с начала, взвести Start
        if(((RDS_PSTARTSTOPDATA)ExtParam)->FirstStart)
          Start=1; // Модель запустится в первом же такте
        break;

      // Такт расчета
      case RDS_BFM_MODEL:
        y=K*x+C;
        break;
```

```
        // Функция настройки
        case RDS_BFM_SETUP:
          if(TestKxCSetup(BlockData->Block,3,4))
            { // Нажата "OK"
              Start=1;  // Запустить модель в следующем такте
              return RDS_BFR_MODIFIED;
            }
      }
    return RDS_BFR_DONE;
  // Отмена макроопределений
  #undef y
  #undef C
  #undef K
  #undef x
  #undef Ready
  #undef Start
  #undef pStart
  }
  //=======================================
```

Для улучшения читаемости программы настройка параметров блока вынесена в отдельную функцию `TestKxCSetup`. Она принимает три параметра: идентификатор блока (`Block`) и порядковые номера переменных `K` и `C` (`numK` и `numC` соответственно). Внутри функции при помощи `rdsFORMCreate` создается вспомогательный объект-окно, и его идентификатор присваивается вспомогательной переменной `window`. Затем при помощи ранее созданной функции `AddWinEditOrDisplayDouble` в это окно добавляются поля ввода или индикации для параметров `K` и `C` (в обоих вызовах мы будем использовать в качестве заголовка поля ввода имя соответствующей переменной, поэтому в последнем параметре функции передается значение `NULL`). Для переменной `K` добавляется поле с идентификатором 1, при этом вспомогательной логической переменной `K_conn` присваивается `TRUE`, если к переменной присоединена связь, и `FALSE` в противном случае. Для переменной `C` добавляется поле с идентификатором 2, факт наличия или отсутствия у нее связи заносится в `C_conn`. Теперь можно открывать окно: вызывается функция `rdsFORMShowModalEx`, и ее результат присваивается переменной `ok`. Если пользователь закроет окно кнопкой "OK", значение `ok` будет истинным, при этом необходимо сделать введенные пользователем числа значениями по умолчанию переменных `K` и `C`, но только в том случае, если к этим переменным не подключены связи.

Если `K_conn` ложно, ко входу `K` не присоединено ни одной связи. В этом случае вспомогательной переменной `str` присваивается указатель на строку, введенную пользователем в поле для параметра `K` (это указатель на внутреннюю строку объекта-окна `window`, она будет освобождена автоматически при удалении этого объекта, поэтому `rdsFree` для этой строки вызывать не нужно). Для получения этого указателя используется сервисная функция `rdsGetObjectStr`. Затем при помощи функции `rdsSetBlockVarDefValueStr` полученная строка устанавливается в качестве значения по умолчанию переменной с номером `numK` в блоке `Block`, то есть переменной `K`. Аналогичным образом устанавливается значение по умолчанию переменной `C`, если значение `C_conn` ложно. Затем объект `window` уничтожается и функция возвращает значение вспомогательной переменной `ok`, то есть `TRUE`, если пользователь нажал "OK", или `FALSE`, если он отменил изменения.

За функцией `TestKxCSetup` следует функция модели блока. При ее вызове в режиме `RDS_BFM_VARCHECK`, как обычно, проверяется допустимость структуры переменных блока: в ней должно быть два обязательных сигнала ("S") и четыре переменных типа `double` ("D").

Для того, чтобы в самом первом такте моделирования было вычислено значение выхода блока, соответствующее начальным значениям входа и параметров, в функцию модели введена реакция на запуск расчета (RDS_BFM_STARTCALC). При вызове модели в этом режиме в параметре ExtParam передается указатель на структуру RDS_PSTARTSTOPDATA. Она описана в "RdsDef.h" следующим образом:

```
typedef struct
{ BOOL FirstStart;    // TRUE  - расчет запущен с начала
                      // FALSE - расчет продолжен после остановки
  BOOL Loop;          // TRUE  - запуск непрерывного расчета
                      // FALSE - запуск расчета на один такт
} RDS_STARTSTOPDATA;
// Указатель на структуру
typedef RDS_STARTSTOPDATA *RDS_PSTARTSTOPDATA;
```

Если поле FirstStart этой структуры истинно, расчет запущен с самого начала. В этом случае взводится сигнал Start, чтобы в первом же такте расчета модель сработала и вычислила значение выхода y. Того же результата можно было добиться, установив для переменной Start единичное значение по умолчанию, как в предыдущих примерах. Приведенный здесь способ принудительного запуска модели в первом такте расчета надежнее, поскольку пользователь не сможет вмешаться в работу модели, изменив значение Start по умолчанию.

В реакции на такт расчета (RDS_BFM_MODEL) модель вычисляет значение выхода y по значениям входа x и параметров K и C. При этом не важно, получает модель значения параметров по связям или они введены пользователем в функции настройки. В обоих случаях можно брать текущее значение соответствующей переменной.

В режиме RDS_BFM_SETUP модель вызывает функцию настройки TestKxCSetup, передавая ей идентификатор данного блока из структуры BlockData и порядковые номера переменных K (3) и C (4). Если функция вернула TRUE, взводится сигнал Start, чтобы в ближайшем такте расчета было вычислено новое значение выхода по изменившимся параметрам, после чего возвращается значение RDS_BFR_MODIFIED, информирующее РДС о наличии изменений в схеме.

Чтобы проверить работу этого блока, можно собрать схему, в которой только к одному из двух параметров будет подключено поле ввода – например, к параметру K (рис. 52). В окне параметров блока на вкладке "DLL" нужно установить флаг "блок имеет функцию настройки" (см. рис. 7). Теперь при выборе соответствующего пункта контекстного меню блока (или по двойному щелчку, если на вкладке "Общие" установить флаг "Двойной щелчок в режиме редактирования вызывает функцию настройки") будет открываться окно для ввода значений K и C. В этом окне будут заблокированы значения параметров, к которым подключены связи.

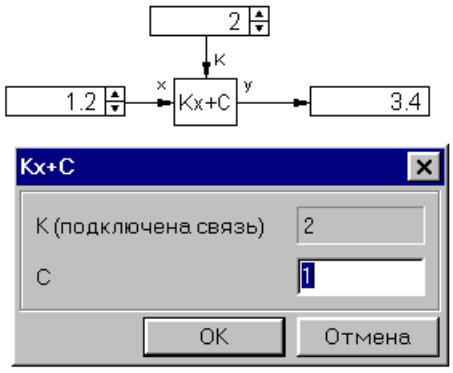


Рис. 52. Блок, хранящий настройки в значениях входов по умолчанию

§2.7.5. Открытие модальных окон средствами Windows API

Рассматривается пример модели блока, позволяющего выбрать произвольный цвет при помощи стандартного диалога Windows и выдающего выбранный цвет на выход в виде целого числа. Пример иллюстрирует открытие модальных окон (в данном случае – стандартного диалога) средствами Windows API.

Вспомогательные объекты-окна РДС могут содержать только набор стандартных элементов (простые поля ввода, выпадающие списки, флаги, области рисования и т.п.), расположенных друг под другом на вкладках окна или на боковой панели. Если модели блока этого недостаточно, она должна организовать диалог с пользователем самостоятельно при помощи функций Windows API или каких-либо специальных библиотек. При этом необходимо соблюдать все правила работы с модальными окнами (см. стр. 29), в частности, информировать РДС об открытии и закрытии модального окна функциями `rdsBlockModalWinOpen` и `rdsBlockModalWinClose`. Рассмотрим пример, в котором блок будет открывать модальное окно средствами Windows. Чтобы не загромождать текст модели блока большим количеством вызовов API, не имеющих прямого отношения к РДС, в качестве модального окна будем использовать стандартный диалог выбора цвета. Это один из простейших диалогов, используемых в Windows, он обладает всеми свойствами модального окна и для него не придется писать собственную функцию обработки событий. Создадим блок, который позволяет выбрать цвет с помощью стандартного диалога Windows, и выдет его на свой выход `Color` типа `int`. Как принято в Windows, младший байт целого числа будет содержать интенсивность красной компоненты цвета, второй байт – интенсивность зеленой, а третий – интенсивность синей (это соответствует структуре стандартного типа `COLORREF` в Windows API). Для наглядности следует задать для блока векторную картинку, связав цвет какой-нибудь геометрической фигуры (например, прямоугольника) с переменной `Color`, в этом случае заданный цвет будет отражаться на внешнем виде блока в режимах моделирования и расчета.

Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	Color	int	4	Выход

Поскольку этот блок не имеет личной области данных, не работает с динамическими переменными и не участвует в расчете, его модель будет достаточно простой:

```
extern "C" __declspec(dllexport)
  int RDSCALL ModalWindowTest(int CallMode,
                              RDS_PBLOCKDATA BlockData,
                              LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Color (*((int *)(pStart+2)))
  // Вспомогательная структура для работы с диалогом цвета
  CHOOSECOLOR cc;
  // Массив дополнительных цветов для диалога
  static COLORREF CustomColors[16]=
                 {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
  // Возвращаемое функцией модели значение
  int result=RDS_BFR_DONE;
```

```
  // Вспомогательная переменная – результат вызова диалога
  BOOL ok;

  switch(CallMode)
    { // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSI}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Функция настройки
      case RDS_BFM_SETUP:
        // Установка параметров структуры для работы с диалогом
        memset(&cc,0,sizeof(cc));
        cc.lStructSize=sizeof(cc);
        cc.hwndOwner=rdsGetAppWindowHandle();
        cc.lpCustColors=CustomColors;
        cc.rgbResult=Color;
        cc.Flags=CC_RGBINIT;
        // Уведомление РДС об открытии модального окна
        rdsBlockModalWinOpen(NULL);
        // Вызов диалога
        ok=ChooseColor(&cc);
        // Уведомление РДС о закрытии модального окна
        rdsBlockModalWinClose(NULL);
        if(ok) // Пользователь выбрал цвет
          { // Запись цвета в Color
            Color=cc.rgbResult;
            // Установка этого значения Color по умолчанию
            // (2 – порядковый номер переменной Color в блоке)
            rdsSetBlockVarDefValueByCur(BlockData->Block,2);
            // Возвращаемое значение должно сигнализировать
            // о наличии изменений в схеме
            result=RDS_BFR_MODIFIED;
          }
        break;
    }
  return result;
// Отмена макроопределений
#undef Color
#undef Ready
#undef Start
#undef pStart
}
//=========================================
```

Для работы с диалогом выбора цвета требуется специальная структура CHOOSECOLOR, поэтому в начале функции модели описана вспомогательная переменная cc этого типа. Перед вызовом диалога в поля этой структуры записываются различные параметры, определяющие вид и поведение диалога, а после закрытия окна из нее можно считать цвет, выбранный пользователем. Кроме этой структуры для работы диалога нужен массив из 16 значений типа COLORREF для хранения цветов, определенных пользователем (подробнее см. описание Windows API). Этот массив также описан в начале функции модели. Он объявлен статическим, чтобы определенные пользователем цвета сохранялись между вызовами диалога и были доступны всем блокам с этой моделью. Дополнительно описывается целая переменная – результат функции модели result, которой исходно присваивается константа RDS_BFR_DONE (если пользователь выберет другой цвет в функции настройки, этой

переменной будет присвоено значение RDS_BFR_MODIFIED), и вспомогательная логическая переменная ok, которой позднее будет присвоен результат вызова функции диалога.

При вызове модели в режиме RDS_BFM_VARCHECK переданная строка типа переменных сравнивается со строкой "{SSI}" – кроме двух обязательных сигналов блок должен иметь целую переменную, в которую будет записываться цвет. При вызове функции настройки блока (RDS_BFM_SETUP) прежде всего инициализируется структура cc типа CHOOSECOLOR, необходимая для стандартного диалога выбора цвета. Сначала при помощи функции memset вся структура заполняется нулевыми байтами, после чего в поле lStructSize записывается размер самой структуры. Это обычная практика при работе с Windows API – функция ChooseColor, в которую будет передан указатель на эту структуру, сможет проверить, соответствует ли размер структуры требованиям функции. В поле hwndOwner записывается дескриптор главного окна приложения, полученный при помощи сервисной функции РДС rdsGetAppWindowHandle. Это нужно для того, чтобы открывающееся модальное окно было привязано к главному окну РДС. В поле lpCustColors записывается указатель на статический массив для шестнадцати пользовательских цветов CustomColors, описанный в начале функции модели. В поле rgbResult записывается исходное значение цвета, которое пользователь будет изменять в диалоге, взятое из переменной Color. Наконец, в поле Flags записывается стандартная константа CC_RGBINIT, приказывающая диалогу взять исходное значение цвета из поля rgbResult этой структуры перед открытием окна. Теперь структура cc готова к вызову диалога.

Перед открытием любого модального окна (в данном случае – перед вызовом диалога) необходимо уведомить об этом РДС при помощи функции rdsBlockModalWinOpen. Эта функция принимает единственный параметр – идентификатор блока, открывающего модальное окно. В данном случае окно открывается изнутри функции модели блока, поэтому РДС в состоянии самостоятельно определить, модель какого блока в данный момент работает, и вместо идентификатора блока можно передать значение NULL (при желании можно было явно указать идентификатор этого блока, вызвав функцию с параметром BlockData->Block). Теперь можно открыть окно диалога, вызвав функцию Windows API ChooseColor и передав в нее указатель на структуру cc. Когда пользователь закроет окно, функция вернет TRUE, если он нажал кнопку "OK" (при этом в поле rgbResult структуры cc будет записан выбранный цвет), или FALSE, если он нажал кнопку "Отмена". Возвращенное функцией значение записывается во вспомогательную переменную ok и будет проанализировано позднее. После вызова функции диалога модель информирует РДС о закрытии модального окна при помощи сервисной функции rdsBlockModalWinClose.

Теперь, когда модальное окно закрыто, и РДС знает об этом, можно разобраться с цветом, выбранным пользователем. Если пользователь закрыл окно кнопкой "OK", значение вспомогательной логической переменной ok будет истинно. При этом выбранный пользователем цвет записывается в переменную блока Color из поля структуры cc.rgbResult. Может показаться, что этого достаточно, однако присвоение нового значения переменной Color изменяет только ее текущее значение, которое не запоминается при сохранении схемы. Кроме того, при первом же сбросе расчета переменной Color будет присвоено значение по умолчанию, и выбранный пользователем цвет будет потерян. Чтобы этого не произошло, нужно кроме текущего значения переменной Color изменить еще и ее значение по умолчанию. Для этого используется сервисная функция РДС rdsSetBlockVarDefValueByCur, запоминающая текущее значение переменной блока в качестве значения по умолчанию. Функция принимает два параметра: идентификатор блока, над переменной которого производится операция (BlockData->Block), и порядковый номер обрабатываемой переменной, начиная с нуля. В данном блоке три переменных: Start

(номер 0), Ready (номер 1) и Color (номер 2), поэтому чтобы изменить значение по умолчанию у переменной Color, нужно передать в функцию число 2. Таким образом, вызов функции rdsSetBlockVarDefValueByCur с параметрами BlockData->Block и 2 делает значение переменной Color по умолчанию равным ее текущему значению, то есть цвету, только что полученному из закрытого пользователем диалога. После этого переменной result присваивается константа RDS_BFR_MODIFIED, чтобы возвращаемое функцией значение сообщило РДС о наличии изменений в схеме.

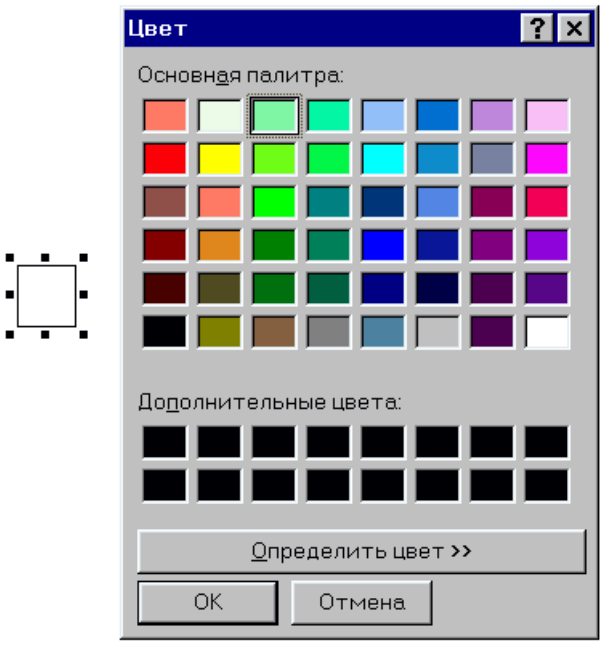


Рис. 53. Вызов модального диалога из функции настройки блока

Чтобы проверить работу этой модели, нужно установить флаг “блок имеет функцию настройки” в окне параметров блока и, для наглядности, задать блоку векторную картинку с какой-нибудь простой геометрической фигурой (например, прямоугольником), цвет которой связан с переменной блока Color. Теперь, если выбрать в контекстном меню блока пункт “Настройка”, задать в открывшемся окне диалога какой-нибудь цвет (рис. 53) и перейти из режима редактирования в режим моделирования, картинка блока должна окраситься в выбранный цвет.

§2.7.6. Открытие модальных окон в режиме расчета

Рассматриваются особенности открытия модальных окон в режиме расчета. Блок из предыдущего примера модифицируется так, чтобы модальное окно в режиме расчета не вызывало проблем. Описывается сервисная функция РДС rdsUnlockAndCall.

Изменим предыдущий пример так, чтобы в режимах моделирования и расчета можно было по щелчку мыши изменить цвет блока (значение переменной Color), но при сбросе расчета значение переменной Color возвращалось бы к исходному, заданному в функции настройки. Может показаться, что для этого достаточно добавить в модель блока реакцию на нажатие кнопки мыши, в которой выполнить ту же последовательность действий, что и в функции настройки, кроме установки значения переменной Color по умолчанию. Попробуем сделать так и посмотрим, к каким проблемам это приведет.

Добавим рядом с оператором case для события RDS_BFM_SETUP еще один case для нажатия кнопки мыши (RDS_BFM_MOUSEDOWN) – в обоих случаях модель будет открывать одно и то же модальное окно. Эта часть модели блока будет выглядеть следующим образом (изменения выделены жирным):

```
// Функция настройки или нажатие кнопки мыши
case RDS_BFM_SETUP:
case RDS_BFM_MOUSEDOWN:
  // Установка параметров структуры для работы с диалогом
  memset(&cc,0,sizeof(cc));
  cc.lStructSize=sizeof(cc);
  cc.hwndOwner=rdsGetAppWindowHandle();
  cc.lpCustColors=CustomColors;
  cc.rgbResult=Color;
```

```
            cc.Flags=CC_RGBINIT;
            // Уведомление РДС об открытии модального окна
            rdsBlockModalWinOpen(NULL);
            // Вызов диалога
            ok=ChooseColor(&cc);
            // Уведомление РДС о закрытии модального окна
            rdsBlockModalWinClose(NULL);
            if(ok) // Пользователь выбрал цвет
              { // Запись цвета в Color
                Color=cc.rgbResult;
                if(CallMode==RDS_BFM_SETUP) // Функция настройки
                  { // Установка этого значения Color по умолчанию
                    // (2 - порядковый номер переменной Color в блоке)
                    rdsSetBlockVarDefValueByCur(BlockData->Block,2);
                    // Возвращаемое значение должно сигнализировать
                    // о наличии изменений в схеме
                    result=RDS_BFR_MODIFIED;
                  }
                else // Нажатие кнопки мыши
                  // Нужно перерисовать окно подсистемы
                  rdsRefreshBlockWindows(BlockData->Parent,FALSE);
              }
            break;
      // .....
```

Если пользователь закрыл окно кнопкой “OK” (значение `ok` истинно), выбранный цвет переписывается в переменную `Color` независимо от того, произошло это в функции настройки в режиме редактирования или в реакции на нажатие кнопки мыши в режимах моделирования и расчета. Однако, дальнейшие действия будут зависеть от режима, в котором находится РДС.

Если окно было открыто в функции настройки блока (параметр `CallMode` равен `RDS_BFM_SETUP`), необходимо выполнить в точности те же действия, что и в предыдущем варианте этого примера: установить значение `Color` по умолчанию и вернуть в РДС константу `RDS_BFR_MODIFIED`. Если же окно было открыто внутри реакции блока на нажатие кнопки мыши (параметр `CallMode` равен `RDS_BFM_MOUSEDOWN`), ничего этого делать не нужно, но необходимо перерисовать окно подсистемы, чтобы изменившееся значение переменной `Color` немедленно отразилось на изображении блока (при условии, что для блока задана векторная картинка, как было указано выше). Если этого не сделать, значение переменной `Color` изменится, но изображение блока в окне подсистемы сохранит свой прежний цвет до тех пор, пока Windows не потребуется обновить это окно. Для перерисовки окна используется сервисная функция `rdsRefreshBlockWindows`:

```
void RDSCALL rdsRefreshBlockWindows(
      RDS_BHANDLE     Block,     // Блок или подсистема
      BOOL            Recursive);// Перерисовывать окна вложенных
```

Первый параметр функции указывает блок или подсистему, чьи окна должны быть перерисованы. Изображение блока находится в окне его родительской подсистемы, поэтому в качестве первого параметра передается поле `Parent` (идентификатор родительской подсистемы) структуры данных блока `BlockData`. Второй параметр функции указывает на необходимость обновить окна всех блоков и подсистем, вложенных в указанную – в данном случае этого не нужно, поэтому этот параметр равен `FALSE`.

Следует отметить, что обычно в реакции на нажатие кнопок мыши не требуется вызывать функцию для обновления окна подсистемы, блок которой получил сообщение о нажатии – РДС делает это автоматически. Однако, в данном случае этого недостаточно: после нажатия кнопки мыши РДС обновит окно тогда, когда снова начнет работать цикл

обработки сообщений приложения (принципы обработки сообщений Windows подробно описаны в литературе по Windows API), то есть сразу после открытия модального окна диалога. Но переменная `Color` изменяется только после закрытия диалога, когда автоматическое обновление окна подсистемы уже выполнено. Поэтому необходимо перерисовать окно еще раз, уже с новым значением переменной `Color`.

Для того, чтобы модель блока начала получать информацию о нажатии кнопок мыши, необходимо включить в окне параметров блока флаг “блок реагирует на мышь” (см. рис. 7). После этого можно перейти в режим моделирования и щелкнуть на изображении блока какой-нибудь кнопкой мыши. Должно открыться такое же окно диалога выбора цвета, как и при вызове функции настройки (рис. 53), и, если выбрать в нем какой-нибудь цвет, картинка блока должна изменить свой цвет на выбранный.

В режиме моделирования все работает, как и планировалось, однако в режиме расчета начнутся проблемы. Поместим в систему блок-планировщик динамического расчета и подключим числовой индикатор к его выходу `Time` – по изменению числа на индикаторе мы сможем проверить, работает ли расчет (рис. 54). Выход `Time` планировщика не отображается в меню при создании связи, поэтому придется сначала выбрать в меню пункт “Список”, а потом уже выбрать выход `Time` в полном списке переменных. Запустим расчет – число на индикаторе начнет увеличиваться, значит, расчет идет. Если теперь щелкнуть какой-нибудь кнопкой мыши на изображении блока выбора цвета, откроется окно диалога, но число на индикаторе перестанет изменяться – открытие модального окна вызвало остановку расчета. Если закрыть окно, расчет продолжится, но пока окно открыто, он будет стоять.

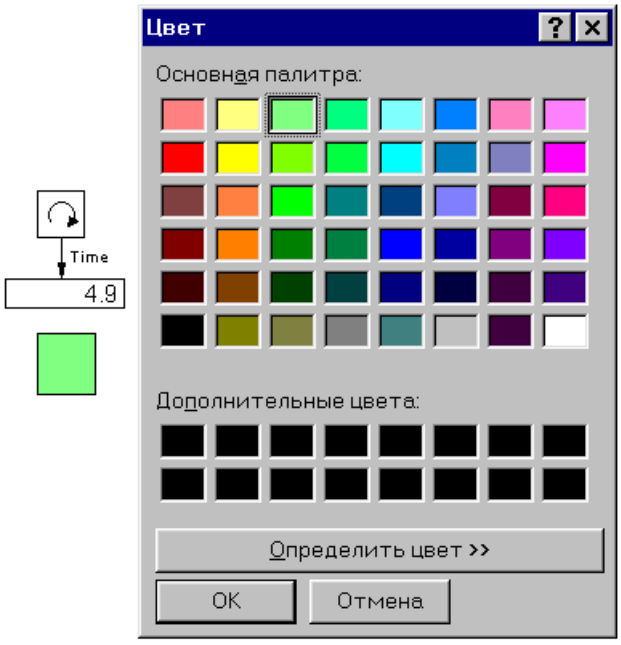


Рис. 54. Пока модальное окно открыто, расчет работать не будет

Все дело в том, что в режиме расчета обычно работает два потока команд. В главном потоке приложения обслуживается пользовательский интерфейс (включая реакции блоков на нажатие кнопок мыши), а в потоке расчета циклически вызываются модели простых блоков и производится передача данных по связям. Чтобы эти два потока не пытались одновременно обратиться к одним и тем же данным, перед вызовом функции модели в любом из потоков данные блоков блокируются. Другой поток, попытавшись получить доступ к этим данным, будет остановлен до тех пор, пока функция модели в первом потоке не завершится и данные не будут разблокированы. Тогда второй поток сможет сам заблокировать данные, и его выполнение продолжится. При щелчке на блоке выбора цвета включается блокировка данных и модель блока вызывается в главном потоке с параметром `RDS_BFM_MOUSEDOWN`. Поскольку внутри реакции на это событие открывается модальное окно, функция модели завершится только после закрытия этого окна, а, значит, и данные останутся заблокированными, пока окно не будет закрыто. Когда поток расчета, выполняя очередной такт, попытается получить доступ к данным блоков, он будет остановлен, пока данные не будут разблокированы, то есть до закрытия модального окна в главном потоке.

Когда окно будет закрыто, функция модели в главном потоке завершится, данные будут разблокированы, и поток расчета сможет продолжить выполнение.

Такая непреднамеренная остановка расчета может привести к неприятным, а главное – неожиданным для пользователя последствиям, особенно если идет моделирование какого-нибудь процесса с синхронизацией с реальным временем. Чтобы расчет не останавливался, необходимо снимать блокировку данных на время открытия модального окна. Окно диалога выбора цвета работает только с локальной переменной `cc` и не обращается ни к каким данным блока, поэтому перед открытием окна можно без опасений снять блокировку, а после его закрытия – восстановить ее, поскольку цвет необходимо записать в переменную блока `Color`, и обращаться к ней без блокировки нельзя.

Для временного снятия блокировки данных с последующим ее восстановлением служит сервисная функция `rdsUnlockAndCall`:

```
BOOL RDSCALL rdsUnlockAndCall(
     RDS_IpV    Callback,  // Функция обратного вызова
     LPVOID     Arg,       // Аргумент функции Callback
     int        *pResult); // Результат функции Callback
```

Эта функция снимает блокировку данных, после чего вызывает функцию пользователя, указатель на которую передан в параметре `Callback`. Функция пользователя должна иметь тип `int RDSCALL func(LPVOID)`, в качестве ее единственного параметра ей передается параметр `Arg`, указанный при вызове `rdsUnlockAndCall`. После завершения пользовательской функции блокировка данных восстанавливается, и возвращенное функцией целое значение записывается по адресу `pResult`, если в этом параметре не было передано значение `NULL`. Таким образом, `rdsUnlockAndCall` позволяет вызвать произвольную функцию пользователя, сняв блокировку данных на время ее выполнения.

Изменим модель блока выбора цвета таким образом, чтобы она не приводила к остановке расчета. Для этого необходимо вынести вызов `ChooseColor` в отдельную функцию, имеющую совместимый с `rdsUnlockAndCall` формат:

```
int RDSCALL ModalWindowTest2Callback(LPVOID data)
{
  return ChooseColor((CHOOSECOLOR*)data);
}
//=======================================
```

Внутри этой функции указатель общего вида `data`, переданный как параметр, приводится к типу “указатель на структуру `CHOOSECOLOR`” и подставляется в вызов `ChooseColor`. Функция обратного вызова имеет тип `int`, а возвращает она результат выполнения `ChooseColor`, то есть `BOOL` – в языке С это допустимо. Функция вернет нулевое значение, если `ChooseColor` вернула `FALSE`, и ненулевое, если `TRUE`. В файлах заголовков Windows API тип `BOOL` определен как `int`, а константы `TRUE` и `FALSE` как 0 и 1, так что на самом деле возвращаемые функциями типы в точности совпадают. Но даже если бы они и не совпадали, такое приведение типов не вызвало бы проблем – в языке С истиной считается любое ненулевое значение.

Теперь необходимо внести изменения в функцию модели блока:

```
// .....
        // Уведомление РДС об открытии модального окна
        rdsBlockModalWinOpen(NULL);
        // Вызов диалога
        if(rdsCalcProcessIsRunning()) // Идет расчет
          { // Вызов со снятием блокировки
            int ret;
            rdsUnlockAndCall(ModalWindowTest2Callback,&cc,&ret);
            ok=ret;
          }
```

```
            else // Режим редактирования или моделирования
              ok=ChooseColor(&cc);
            // Уведомление РДС о закрытии модального окна
            rdsBlockModalWinClose(NULL);
            if(ok) // Пользователь выбрал цвет
              { // Запись цвета в Color
                Color=cc.rgbResult;
// .....
```

Вызывать диалог со снятием блокировки имеет смысл только в режиме расчета, поэтому сначала можно определить режим РДС. Для этого используется функция `rdsCalcProcessIsRunning`, возвращающая `TRUE`, если в данный момент РДС находится в режиме расчета. В этом случае вызывается `rdsUnlockAndCall`, в которую в качестве указателя на функцию обратного вызова передается `ModalWindowTest2Callback`, в качестве аргумента функции – указатель на структуру `cc`, а результат функции обратного вызова будет записан в целую переменную переменную `ret`. Таким образом, вызов

```
rdsUnlockAndCall(ModalWindowTest2Callback,&cc,&ret);
```

эквивалентен вызову

```
ret=ModalWindowTest2Callback(&cc);
```

со снятием блокировки перед вызовом и восстановлением ее после него.

После закрытия диалога выбора цвета блокировка данных восстановится и `rdsUnlockAndCall` завершится. Теперь можно присвоить результат возврата функции обратного вызова (`ret`) переменной `ok`, которая будет анализироваться далее в модели. На самом деле, поскольку в Windows API типы `BOOL` и `int` полностью эквивалентны, можно было бы написать

```
rdsUnlockAndCall(ModalWindowTest2Callback,&cc,&ok);
```

и обойтись без вспомогательной переменной `ret` – здесь она введена для большей ясности примера.

Если же функция `rdsCalcProcessIsRunning` вернет `FALSE`, это будет означать, что РДС находится в режимах редактирования или моделирования. В этом случае диалог можно открыть обычным образом, при помощи непосредственного вызова функции `ChooseColor`.

Теперь, если щелкнуть на блоке выбора цвета в режиме расчета, можно увидеть, что число на индикаторе, подсоединенном к блоку-планировщику (см. рис. 54), продолжает увеличиваться, несмотря на открытое модальное окно диалога, то есть поток расчета продолжает выполняться. Это не мешает работе самого блока – если выбрать в диалоге какой-нибудь цвет, изображение блока окрасится в него, как и планировалось.

Все вышеизложенное справедливо только в том случае, если РДС работает в стандартном режиме, то есть с двумя потоками. Если в настройках РДС на вкладке “Совместимость и быстродействие” будет установлен флаг “производить расчет в главном потоке”, открытие модального окна в любом случае будет останавливать расчет – главный поток будет занят окном и не сможет выполнять расчет до его закрытия. Снятие блокировки на время открытия окна никак не может помочь потоку расчета, потому что потока расчета просто не существует. К счастью, режим с единственным потоком используется крайне редко (в основном, для увеличения скорости расчета на медленных машинах), и в окне настроек содержится предупреждение о проблемах, возникающих при его включении, так что можно считать, что пользователь предупрежден, и знает, что делает, включая этот режим.

§2.8. Сохранение и загрузка параметров блока

Рассматриваются процедуры сохранения и загрузки личных параметров блока в файле схемы. Описывается более простой двоичный способ сохранения и более сложный – текстовый. Для текстового режима приводятся разные способы хранения данных: формат с ключевыми словами и формат, аналогичный стандартным INI-файлам (“имя=значение”). Рассматриваются вспомогательные объекты РДС, упрощающие разбор текста.

§2.8.1. Способы хранения параметров блока

Рассматриваются двоичный и текстовый форматы хранения личных данных блока в файле или буфере обмена, указываются преимущества и недостатки обоих форматов.

Если какие-либо настроечные параметры блока хранятся в его личной области данных, модель должна сама позаботиться об их записи при сохранении схемы, добавлении блока в библиотеку или копировании его в буфер обмена. Можно, конечно, хранить параметры в тексте комментария блока или в значениях по умолчанию статических переменных, как было сделано в примерах на стр. 95 и 135 – эти значения РДС сохраняет самостоятельно. Однако, это не всегда удобно. Комментарий блока может быть изменен пользователем, не знающим, что модель блока использует его для своих целей, а при хранении параметров в значениях переменных по умолчанию придется ограничиваться набором типов, используемых в РДС. Значения по умолчанию хорошо подходят для хранения простых параметров, например, чисел `double` и `int`. Если же параметры имеют сложную структуру, и, к тому же, не предполагается, что пользователь захочет устанавливать значения этих параметров по связям от других блоков, лучше всего хранить их в личной области данных. При этом можно использовать любые типы переменных: структуры, массивы структур, битовые флаги и т.п, и обращаться к ним как к обычным переменным языка С. Конечно, в переменных блока тоже можно было бы организовать, например, массив структур, но хранить в его значениях по умолчанию настроечные параметры не получится – в РДС у всех элементов массива будет одно общее значение по умолчанию.

РДС сохраняет схемы и блоки в одном из двух форматов: в двоичном или в текстовом. Данные в двоичном формате, как правило, занимают меньше места и быстрее загружаются, при этом часть модели блока, отвечающую за загрузку и сохранение данных, можно сделать очень простой – достаточно записать подряд все параметры блока при сохранении схемы или блока, а при загрузке считать их в той же последовательности. Данные в текстовом формате занимают больше места, обрабатываются несколько дольше, и модель блока получается более сложной – необходимо анализировать текст, отыскивая в нем параметры блока (например, по ключевым словам). Однако, в текстовом формате проще обеспечить совместимость новых версий моделей с данными, сохраненными старыми версиями. При изменении структуры параметров блока загрузка старой схемы в двоичном формате, вероятнее всего, вызовет ошибки из-за несоответствия списка параметров, сохраненных предыдущей версией модели, списку, которую попытается загрузить новая версия. Например, если старая версия модели сохраняла два числа `double` и одно число `int` (итого 2x8+4=20 байтов), а новая – три числа `double` и одно число `int` (3x8+4=28 байтов), то данные, записанные старой версией при сохранении схемы, новой версией модели будут загружены неправильно. В текстовом формате такой проблемы обычно не возникает – параметры блока чаще всего опознаются по ключевым словам, и те параметры, которые отсутствовали в старой версии модели, просто не будут загружены и сохранят значения по умолчанию, поскольку соответствующие им ключевые слова при загрузке старой схемы просто не встретятся. Конечно, двоичный формат тоже можно сделать достаточно гибким, например, присвоив каждому параметру блока уникальный код и записывая пары “код-значение”. Однако, при этом теряются основные достоинства двоичного формата – скорость загрузки и простота реализации, поскольку теперь при загрузке придется анализировать код

и тип каждого параметра. Кроме того, схема, сохраненная в текстовом формате, может быть просмотрена и отредактирована в любом текстовом редакторе, что значительно упрощает отладку моделей (всегда можно посмотреть, правильно ли записались параметры блока) и позволяет, при желании, обрабатывать схемы какими-либо внешними программами, которые могут работать с текстовыми файлами (например, искать в файле схемы какое-нибудь слово или сочетание слов при помощи программы “grep”).

В данный момент двоичный формат используется в РДС только при копировании блоков через буфер обмена, поскольку при этом не нужно обеспечивать совместимость с устаревшими версиями библиотек. Начиная с версии 1.0.123 схемы и блоки всегда сохраняются в текстовом формате.

В модель блока, настроечные параметры которого хранятся в личной области данных, достаточно добавить функции сохранения и загрузки этих параметров только в одном из форматов. Формат, в котором сохраняет свои данные модель, не обязательно должен соответствовать формату, в котором РДС сохраняет схему или блок. При сохранении схемы или записи блока в библиотеку модель блока сначала вызывается для сохранения данных в том формате, в котором сохраняется весь файл, то есть в текстовом. Если в результате вызова модели были сохранены какие-либо данные, РДС считает, что модель поддерживает этот формат, и не предпринимает никаких дополнительных действий. Если же данные не были сохранены, РДС снова вызывает модель, но уже для сохранения данных в двоичном формате, после чего записанные моделью данные приводятся в соответствие с требованиями текстового формата. Сохраненный моделью массив двоичных данных переводится в набор шестнадцатеричных чисел и записывается в файл в виде текста “dlldata data <*размер массива*> <*набор чисел*>”. При копировании одного или нескольких блоков в буфер обмена РДС сначала вызывает модель для сохранения данных в двоичном формате. Если модель ничего не сохранила, РДС будет считать, что двоичный формат не поддерживается моделью, и она будет вызвана еще раз для записи в текстовом формате. Текст, который она записала, будет добавлен в буфер обмена как массив байтов.

При загрузке схемы или блока или при вставке блоков из буфера обмена модель всегда вызываются для загрузки данных в том формате, в котором они их сохранила при записи этой схемы или при копировании блока в буфер обмена. Например, если при сохранении схемы модель записала двоичные данные, при загрузке этой схемы модель будет вызвана только для загрузки в двоичном формате, поскольку в схеме присутствуют именно сохраненные двоичные данные.

Чаще всего в моделях реализуют только текстовый формат загрузки и записи параметров как наиболее гибкий. Двоичный формат обычно используют для несложных блоков, которые разрабатываются для какой-либо конкретной схемы и которые не предполагается включать в библиотеки. В принципе, можно сделать в модели сохранение данных в обоих форматах, но на практике так поступают редко – раз более сложный текстовый формат уже поддерживается, нет необходимости добавлять в модель поддержку двоичного.

§2.8.2. Сохранение параметров в двоичном формате

Рассматривается сохранение и загрузка личных данных блока в двоичном формате, описываются соответствующие реакции модели (RDS_BFM_SAVEBIN и RDS_BFM_LOADBIN соответственно). В один из ранее описанных примеров добавляется сохранение и загрузка параметров в двоичном формате.

В качестве самого простого примера рассмотрим сохранение и загрузку параметров блока в двоичном формате для модели `Test1`. Ранее (стр. 121) мы уже добавили в эту модель функцию настройки параметров, теперь сделаем так, чтобы значения этих параметров сохранялись вместе с блоком. Теперь функция модели будет выглядеть следующим образом (изменения выделены жирным):

```
//============= Модель блока ==============
extern "C" __declspec(dllexport)
  int RDSCALL Test1(int CallMode,
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{ TTest1Data *data;
  switch(CallMode)
    { case RDS_BFM_INIT:   // Инициализация
        BlockData->BlockData=new TTest1Data();
        break;
      case RDS_BFM_CLEANUP:// Очистка
        data=(TTest1Data*)(BlockData->BlockData);
        delete data;
        break;
      case RDS_BFM_SETUP:  // Функция настройки
        data=(TTest1Data*)(BlockData->BlockData);
        return data->Setup();
      case RDS_BFM_SAVEBIN:
        // Сохранение параметров в двоичном формате
        data=(TTest1Data*)(BlockData->BlockData);
        rdsWriteBlockData(&(data->IParam),sizeof(data->IParam));
        rdsWriteBlockData(&(data->DParam),sizeof(data->DParam));
        break;
      case RDS_BFM_LOADBIN:
        // Загрузка параметров в двоичном формате
        data=(TTest1Data*)(BlockData->BlockData);
        rdsReadBlockData(&(data->IParam),sizeof(data->IParam));
        rdsReadBlockData(&(data->DParam),sizeof(data->DParam));
        break;
    }
  return RDS_BFR_DONE;
}
//=========================================
```

Как уже отмечалось, основное достоинство двоичного формата – простота реализации. Для каждого из параметров потребовалось всего два вызова функций: один – для сохранения, другой – для загрузки.

При сохранении схемы, записи блока в библиотеку или копировании его в буфер обмена модель будет вызвана с параметром `RDS_BFM_SAVEBIN` для сохранения данных в двоичном формате. Точнее, при сохранении схемы или записи блока в библиотеку модель сначала будет вызвана для сохранения данных в текстовом формате с параметром `RDS_BFM_SAVETXT`, но, поскольку реакция на это событие отсутствует, ничего не будет сохранено, и РДС вызовет модель еще раз с параметром `RDS_BFM_SAVEBIN`. Как и в реакциях на другие события в этой модели, указатель на личную область данных блока, хранящийся в `BlockData->BlockData`, будет приведен к типу “указатель на `TTest1Data`” и присвоен вспомогательной переменной `data`. В личной области данных блока хранятся два параметра: целый `IParam` и вещественный `DParam`. Каждый из них сохраняется при помощи сервисной функции `rdsWriteBlockData`. Эта функция принимает два параметра: указатель на начало области памяти, которую нужно сохранить, и размер этой области в байтах. Для каждого из двух параметров передается указатель на него и размер этого параметра, полученный при помощи оператора `sizeof`. В результате двух последовательных вызовов этой функции в РДС будет передано 12 байтов: 4 байта займет целое число `IParam`, и 8 байтов займет вещественное число с двойной точностью (`double`) `DParam`. Если блок копировался в буфер обмена, эти 12 байтов попадут туда вместе с остальными данными этого блока. Если же запись производилась в текстовом формате, они

будут преобразованы в 24 шестнадцатеричных символа и добавлены к тексту параметров блока отдельной строкой, начинающейся с “dlldata data 12”.

При загрузке схемы или добавлении в нее блока из библиотеки или буфера обмена модель будет вызвана с параметром RDS_BFM_LOADBIN, поскольку данные блока были сохранены в двоичном формате. Параметры IParam и DParam загружаются двумя последовательными вызовами функции rdsReadBlockData, в том же порядке, в котором они сохранялись. Как и парная ей функция rdsWriteBlockData, rdsReadBlockData принимает два параметра – указатель на начало загружаемой области и размер этой области.

Теперь измененные параметры блока не будут теряться при сохранении и последующей загрузке схемы. Для этого потребовалось всего четыре вызова – по два на каждый параметр. Следует, однако, помнить, что любое изменение типа или последовательности сохраняемых параметров приведет к тому, что старые схемы с этим блоком будут загружаться с ошибками.

§2.8.3. Сохранение параметров в текстовом формате

Рассматривается сохранение и загрузка параметров блока в текстовом формате, описываются соответствующие реакции модели (RDS_BFM_SAVETXT и RDS_BFM_LOADTXT). В модель ранее описанного в примерах блока-генератора добавляются процедуры для хранения его параметров в текстовом виде.

Добавим в блок, выдающий на выход синусоиду, косинусоиду или прямоугольные импульсы (стр. 124), загрузку и сохранение параметров в текстовом формате. В личной области данных этого блока хранятся три параметра: целый Type, задающий тип формируемой функции (0 – синус, 1 – косинус, 2 – прямоугольные импульсы), и вещественные Period и Impulse, задающие период функции и длительность прямоугольного импульса соответственно. Поскольку этот блок вполне может использоваться на практике в качестве генератора сигналов, целесообразно хранить его параметры именно в текстовом виде. Это позволит, при необходимости, модифицировать его модель, не опасаясь того, что схемы с этим блоком перестанут читаться.

Личная область данных блока оформлена в виде класса. Добавим в его описание две новых функции-члена: функцию SaveText для сохранения параметров и функцию LoadText для их загрузки. Теперь описание класса будет выглядеть следующим образом (изменения выделены жирным):

```
//====== Класс личной области данных ======
class TTestGenData
{ public:
    int Type;              // Тип (0-sin,1-cos,2-прямоугольные)
    double Period;         // Период
    double Impulse;        // Длительность импульса

    RDS_PDYNVARLINK Time;  // Связь с динамической
                           // переменной времени

    int Setup(void);       // Функция настройки
    void SaveText(void);   // Сохранение параметров
    void LoadText(char *text);// Загрузка параметров
    TTestGenData(void)     // Конструктор класса
      { Type=0; Period=1.0; Impulse=0.5;
    // ... далее без изменений ...
};
```

В функцию модели блока необходимо добавить операторы case для вызова этих функций:

```
//============= Модель блока ==============
extern "C" __declspec(dllexport)
  int RDSCALL TestGen(int CallMode,
```

```
                        RDS_PBLOCKDATA BlockData,
                        LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define y (*((double *)(pStart+2)))
  // Вспомогательная переменная - указатель на личную область
  // данных блока, приведенный к правильному типу
  TTestGenData *data;

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        BlockData->BlockData=new TTestGenData();
        break;
      // Очистка
      case RDS_BFM_CLEANUP:
        data=(TTestGenData*)(BlockData->BlockData);
        delete data;
        break;
      // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;
      // Запись параметров в текстовом формате
      case RDS_BFM_SAVETXT:
        data=(TTestGenData*)(BlockData->BlockData);
        data->SaveText();
        break;
      // Загрузка параметров в текстовом формате
      case RDS_BFM_LOADTXT:
        data=(TTestGenData*)(BlockData->BlockData);
        data->LoadText((char*)ExtParam);
        break;
      // Функция настройки
      case RDS_BFM_SETUP:
        data=(TTestGenData*)(BlockData->BlockData);
        return data->Setup();

      // ... далее без изменений ...

    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef y
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

При сохранении схемы модель блока будет вызвана с параметром `RDS_BFM_SAVETXT`, что приведет к вызову функции-члена `SaveText` класса личной области данных блока `TTestGenData` (перед этим указатель на личную область данных, приведенный к нужному типу, записывается во вспомогательную переменную `data`). Эта функция должна

сформировать текст, который будет записан в схему вместе с остальными данными, относящимися к описанию этого блока. При загрузке схемы модель будет вызвана с параметром RDS_BFM_LOADTXT, при этом в ExtParam будет содержаться указатель на начало текста, считанного из файла схемы. Это тот самый текст, который был сформирован функцией-членом SaveText в момент сохранения. После приведения указателя на текст к типу char* он передается в функцию-член LoadText, задача которой – разобрать его и присвоить нужные значения параметрам блока.

Рассмотрим сначала функцию сохранения параметров SaveText, поскольку от выбранного формата записи будет зависеть способ разбора текста в функции загрузки. Для каждого параметра будем записывать пару "<*ключевое_слово*> <*значение*>": значение параметра Type после слова "type", значение Period после слова "period", значение Impulse – после "impulse". На самом деле, значение параметра Impulse нужно сохранять только тогда, когда выбрано формирование прямоугольных импульсов (значение Type равно 2), поскольку для синуса и косинуса длительность импульса не задается. Тем не менее, пока, для упрощения примера, мы будем всегда сохранять все три параметра. Функция SaveText будет выглядеть следующим образом:

```
// Функция сохранения параметров
void TTestGenData::SaveText(void)
{ char buffer[100]; // Буфер для формирования текста
  // Формирование текста в буфере при помощи функции sprintf
  sprintf(buffer,
          "type %d period %lf impulse %lf",
          Type,Period,Impulse);
  // Передача сформированного текста в РДС
  rdsWriteBlockDataText(buffer,FALSE);
}
//=======================================
```

Прежде всего в функции создается символьный массив buffer размером в 100 символов, в котором будет формироваться текст. Необходимо сохранить целое число, два вещественных числа двойной точности и три ключевых слова – ста символов должно хватить для этого с большим запасом. Далее, при помощи стандартной функции sprintf, в этот массив записывается строка, содержащая ключевые слова и значения параметров. Из-за присутствия функции sprintf для успешной компиляции этого примера необходимо включить в исходный текст стандартный файл заголовка "stdio.h", в котором она описана. Ключевые слова содержатся в строке формата (второй аргумент функции sprintf). В сформированном в массиве buffer тексте вместо спецификаторов формата "%d" и "%lf" появятся значения полей класса Type, Period и Impulse, в том порядке, в котором они были переданы в функцию sprintf. В результате, в массиве buffer окажется строка вида "type 0 period 1.000000 impulse 0.500000" (в данном случае для примера использованы значения параметров блока по умолчанию, заданные в конструкторе).

После того, как строка со значениями параметров блока сформирована, необходимо передать ее в РДС при помощи сервисной функции rdsWriteBlockDataText:

```
void RDSCALL rdsWriteBlockDataText(
     LPSTR      String,    // Передаваемый текст
     BOOL       NewLine);  // Перевод строки перед текстом
```

Эта функция добавляет к тексту, уже записанному для данного блока, фрагмент, указатель на который передается в параметре String. При этом перед добавляемым фрагментом вставляется перевод строки, если значение параметра NewLine истинно, или пробел, если значение NewLine ложно. В данном примере мы одним вызовом передаем в РДС весь сформированный текст, но при необходимости, можно передавать его по частям несколькими вызовами rdsWriteBlockDataText. Например, можно было бы записывать

значение параметра Impulse, только если блок формирует прямоугольные импульсы. В этом случае функция SaveText выглядела бы так:

```
// Функция сохранения параметров - вариант с условием
void TTestGenData::SaveText(void)
{ char buffer[100]; // Буфер для формирования текста
  // Формирование текста для Type и Period
  sprintf(buffer,
          "type %d period %lf",
          Type,Period);
  // Передача сформированного текста в РДС
  rdsWriteBlockDataText(buffer,FALSE);
  // Запись Impulse если Type равно 2
  if(Type==2)
   { // Формирование текста
     sprintf(buffer,"impulse %lf",Impulse);
     // Передача текста в РДС
     rdsWriteBlockDataText(buffer,FALSE);
   }
}
//=======================================
```

Здесь сначала формируется и передается текст для параметров Type и Period, а затем, если значение параметра Type равно двум, в том же массиве buffer формируется и передается в РДС текст для параметра Impulse. Перед ключевым словом “impulse” будет автоматически добавлен пробел, поскольку второй параметр rdsWriteBlockDataText равен FALSE, и это слово не “сольется” со значением параметра Period, которым заканчивается текст, переданный в первом вызове. Можно было бы вместо FALSE указать TRUE, тогда ключевое слово “impulse” и значение параметра разместились бы на отдельной строке. Заметим, что первый вызов функции также производится с параметром FALSE, то есть перед ключевым словом “type” тоже должен был бы быть добавлен пробел, однако РДС автоматически удаляет начальные пробелы и переводы строк в сохраняемом тексте, поэтому этот пробел не будет записан.

Теперь для блока с такой моделью при сохранении в текстовом формате будет записан текст, выглядящий примерно так (результат работы функции SaveText выделен жирным):

```
dllblock name "Block1"
 begin
  pos 40 30
  layer id 1
  vars
   begin
    signal name "Start" in menu run default 0
    signal name "Ready" out menu default 0
    double name "y" out menu default 0
   end
  dll file "$DLL$\\testdll.dll" func "TestGen" setup "" auto
  dlldata text
   type 2 period 1.000000 impulse 0.500000
  enddlldata
 end
```

Текст, сформированный моделью блока, размещается между строками “dlldata text” и “enddlldata”. По ключевому слову “enddlldata” РДС опознает конец текста, поэтому при сохранении параметров блока в текстовом формате ни в коем случае нельзя начинать какую-либо строку с этого слова. При загрузке блока РДС выделяет из файла или буфера обмена текст, заключенный между этими двумя строками, удаляет из каждой строки начальные и

конечные пробелы и символы табуляции, после чего этот текст передается в функцию модели как единый массив символов.

Теперь рассмотрим функцию загрузки параметров `LoadText`, которая будет обрабатывать текст, сформированный при сохранении. Эта функция должна опознавать в переданном ей тексте ключевые слова “type”, “period” и “impulse”, и считывать из следующего слова значение конкретного параметра. Разбор текста и поиск ключевых слов – достаточно часто встречающаяся задача. Чтобы не загромождать пример, будем использовать в нем сервисную функцию РДС, выделяющую первое слово из текста – это стандартная операция, и нет большого смысла расписывать ее полностью. Функция `LoadText` будет извлекать из переданного ей текста слово за словом и сравнивать их с ключевыми, до тех пор, пока текст не закончится:

```
// Функция загрузки параметров
void TTestGenData::LoadText(char *text)
{ char *word,*ptr,c;

  // Установка указателя ptr на начало переданного текста
  ptr=text;
  // Цикл по словам в тексте
  for(;;)
    { // Получить из текста очередное слово
      word=rdsGetTextWord(ptr,&ptr,&c,TRUE);
      if(c==0) // Текст закончился - выход из цикла
        break;
      if(c=='\n') // Перевод строки - пропускаем и продолжаем
        continue;
      if(strcmp(word,"type")==0) // Тип сигнала
        { // Следующее слово - целое число
          word=rdsGetTextWord(ptr,&ptr,NULL,FALSE);
          Type=atoi(word);
        }
      else if(strcmp(word,"period")==0) // Период
        { // Следующее слово - число double
          word=rdsGetTextWord(ptr,&ptr,NULL,FALSE);
          Period=atof(word);
        }
      else if(strcmp(word,"impulse")==0) // Длительность импульса
        { // Следующее слово - число double
          word=rdsGetTextWord(ptr,&ptr,NULL,FALSE);
          Impulse=atof(word);
        }
      else // Неопознанное ключевое слово
        break; // Ошибка - прекращаем обработку
    } // Конец цикла for(;;)
}
//=========================================
```

В начале функции описываются три вспомогательные переменные: `word`, в которую будет помещаться указатель на очередное слово строки; `ptr`, в которой будет храниться указатель на текущую позицию в тексте; и `c`, в которую сервисная функция РДС будет помещать первый символ считанного из строки слова или служебную информацию. Сама функция состоит из “бесконечного” цикла `for(;;)`, перед которым в переменную `ptr` помещается указатель на начало переданного в функцию текста `text`.

Для извлечения слова из текста будет использоваться сервисная функция РДС `rdsGetTextWord`:

```
LPSTR RDSCALL rdsGetTextWord(
     LPSTR      Start,          // Начало текста
```

```
LPSTR      *pNext,          // Указатель на следующее слово
char       *pSym,           // Тип слова или первый символ
BOOL       LowerCase);      // Перевести в нижний регистр
```

В параметре `Start` в эту функцию передается указатель на текст, из которого нужно извлечь первое слово. Функция считает словом любую последовательность символов, ограниченную пробелами, табуляциями или переводами строк, или любую последовательность символов в двойных кавычках. Перевод строки сам по себе тоже считается словом из одного символа с кодом 10 (`0x0A` или '\n' в терминах языка С). Первое слово текста копируется во внутренний буфер РДС и, если параметр `LowerCase` равен `TRUE`, переводится в нижний регистр. Функция возвращает указатель на этот внутренний буфер, при этом указатель на начало следующего слова записывается по адресу, переданному в параметре `pNext`, а однобайтовый тип считанного слова – по адресу, переданному в параметре `pSym`. В качестве типа слова возвращается первый символ этого слова или одна из следующих констант:

- 0 – текст закончился, больше нет слов;
- 10 ('\n') – перевод строки;
- двойная кавычка ('\"') – считанное слово было строкой в кавычках.

Нас здесь будут интересовать только два первых варианта – конец текста, при обнаружении которого нужно выйти из цикла, и перевод строки, который нужно игнорировать.

Поскольку текст с параметрами блока состоит из пар слов вида "<*ключевое_слово*> <*значение*>", в цикле необходимо считывать ключевое слово, опознавать его, после чего брать из следующего за ключевым слова значение соответствующего параметра. В самом первом операторе цикла производится чтение из текста очередного слова, которое будет потом сравниваться с ключевыми – указатель на внутренний буфер, в котором находится слово, присваивается переменной `word`:

```
word=rdsGetTextWord(ptr,&ptr,&c,TRUE);
```

В качестве начала текста в функцию передается указатель на текущую позицию в тексте `ptr`, в эту же переменную будет записан указатель на следующее слово текста. Тип слова будет записан в переменную `c`. В параметре `LowerCase` в функцию передано значение `TRUE`, поэтому слово, извлеченное из текста, будет переведено в нижний регистр. Может показаться, что в данном случае перевод в нижний регистр не нужен – при сохранении параметров блока функцией `SaveText` все ключевые слова записываются в нижнем регистре, поэтому при чтении верхнему регистру будет просто неоткуда взяться. Однако, следует помнить, что файл схемы или блока в текстовом формате может быть отредактирован пользователем вручную, и, если он напишет "TYPE" или "Type" вместо "type", ключевое слово может быть не опознано функцией `LoadText`. Поэтому, желательно либо переводить все считанные слова в нижний регистр, либо сравнивать их с ключевыми без учета регистра.

После того, как слово считано, нужно проверить его тип. Если достигнут конец текста, и больше слов в нем нет (`c==0`), выполняется выход из цикла. Если вместо слова считан перевод строки (`c=='\n'`), его нужно пропустить и считать следующее слово. В противном случае следует сравнить считанное слово с одним из трех ключевых.

Сначала, при помощи стандартной функции `strcmp` (описана в файле заголовков "string.h"), считанное слово сравнивается со строкой "type". Если функция вернула 0, значит, строки совпали, и следующее слово представляет собой символьное представление целого числа, которое нужно занести в переменную `Type`. После первого вызова `rdsGetTextWord` указатель на начало следующего слова был записан в переменную `ptr`, поэтому еще один вызов вида `word=rdsGetTextWord(ptr,&ptr, ... )` считает из текста следующее слово. Вообще, каждый такой вызов продвигает указатель `ptr` на одно слово вперед и возвращает очередное считанное слово через переменную `word`. Считанное таким образом значение типа формируемого сигнала переводится в целое число при помощи стандартной функции `atoi` и присваивается переменной `Type`.

Если strcmp вернула ненулевое значение, строки не совпадают, и word необходимо сравнить с другими ключевыми словами аналогичным образом. Проверка на ключевые слова “period” и “impulse” отличается от уже рассмотренного фрагмента только тем, что за этими ключевыми словами следуют вещественные значения, поэтому для преобразования их в число следует использовать функцию atof вместо atoi.

Следует отметить, что если между ключевым словом и значением попадется перевод строки, функция сработает неправильно, поскольку при вызове rdsGetTextWord для чтения значения мы не проверяем тип считанного слова – в параметре pSym передается NULL. Однако, это не является большим недостатком, поскольку перевод строки может оказаться там только после редактирования файла пользователем, а пользователю вряд ли придет в голову располагать название параметра и его значение на разных строках. В конце концов, при необходимости, можно запретить делать это в описании пользователя для разрабатываемого блока.

Если считанное из текста ключевое слово не совпало ни с одним из трех используемых в этом блоке, выполнение цикла прерывается. Это может произойти либо если файл схемы испорчен, либо если мы пытаемся загрузить данные, сохраненные более новой версией модели блока (в которой могли добавиться новые ключевые слова) при помощи более старой модели. В любом случае, дальнейшая загрузка данных блока при этом бессмысленна.

При использовании функции rdsGetTextWord следует всегда помнить, что извлеченное из текста слово хранится во внутреннем буфере РДС только до тех пор, пока функция не будет вызвана в следующий раз. В приведенном примере слова в тексте идут парами – сначала ключевое слово, затем значение. Может показаться, что можно сразу считать два слова, а затем, в зависимости от первого, преобразовать второе в целое или вещественное число и присвоить это число соответствующему параметру:

```
for(;;)
  { // Получить из текста первое слово
    word1=rdsGetTextWord(ptr,&ptr,&c,TRUE);
    if(c==0) // Текст закончился – выход из цикла
      break;
    if(c=='\n') // Перевод строки – пропускаем и продолжаем
      continue;
    // Получить из текста второе слово (word1 будет утеряно!)
    word2=rdsGetTextWord(ptr,&ptr,NULL,FALSE);
    // Анализ
    if(strcmp(word1,"type")==0) // Тип сигнала
      Type=atoi(word2);
    else if(strcmp(word1,"period")==0) // Период
      Period=atof(word2);
    // ...
```

Однако, приведенный фрагмент программы не будет работать. Первый в цикле вызов rdsGetTextWord считает ключевое слово во внутренний буфер и присвоит указатель на этот буфер переменной word1. Второй вызов считает следующее слово в этот же буфер, присвоив указатель на него word2. В результате, в лучшем случае первое считанное слово будет просто потеряно, в худшем – указатель, содержащийся в переменной word1, будет ссылаться на уже освобожденную область памяти (РДС может заново отвести память под буфер другого размера при очередном вызове rdsGetTextWord). В любом случае, после второго вызова rdsGetTextWord переменную word1 уже нельзя использовать, и вызов strcmp(word1,"type") приведет к непредсказуемым последствиям. Если необходимо считать все слова из текста до их анализа, можно использовать функцию rdsGetTextWordDyn, которая работает с текстом точно так же, как rdsGetTextWord, и имеет те же самые параметры, но вместо того, чтобы возвращать указатель на слово во

внутреннем буфере, создает и возвращает динамическую строку с этим словом. Разумеется, строка, возвращенная `rdsGetTextWordDyn`, должна быть потом обязательно освобождена вызовом `rdsFree`.

§2.8.4. Поиск ключевых слов с помощью объекта РДС

Рассматривается вспомогательный объект РДС, облегчающий поиск ключевых слов в текстовом формате хранения личных данных блока. Модель из предыдущего примера переписывается с использованием этого объекта.

Рассмотренный пример имеет небольшой недостаток – при анализе загруженного текста опознание ключевого слова производится последовательными сравнением в конструкции `if ... else if ... else ...` Чем дальше в этой конструкции находится оператор сравнения с конкретным ключевым словом, тем больше проверок будет выполнено перед тем, как он сработает. Например, если считано слово “impulse”, сначала оно будет сравниваться со словами “type” и “period”, и только после этого будет опознано. Конечно, при трех ключевых словах это не вызовет сильного замедления загрузки. Тем не менее, в сложных блоках может быть несколько десятков параметров, и каждому из них может соответствовать ключевое слово в текстовом формате. В этом случае сравнение последовательным перебором может занять заметное время, особенно если таких блоков в схеме будет очень и очень много. Здесь логичнее использовать какой-либо другой, более быстрый, способ поиска ключевого слова. Например, можно отсортировать все ключевые слова по алфавиту и искать среди них считанное из текста слово методом деления пополам. В РДС есть механизм, позволяющий организовать такой поиск при помощи вспомогательного объекта. Ранее (стр. 121) уже рассматривался вспомогательный объект РДС, открывающий модальные окна. Теперь воспользуемся другим объектом, позволяющим производить быстрый поиск слов в переданном ему массиве.

Попутно внесем в модель блока еще одно усовершенствование. В предыдущем примере каждое ключевое слово фигурировало два раза: первый раз – в форматной строке функции `sprintf` при сохранении параметров, второй раз – в параметре функции `strcmp` при загрузке. Необходимость указывать каждое ключевое слово два раза в разных местах программы повышает вероятность ошибки при ее написании. Конечно, можно было бы сделать ключевые слова `define`-константами, и использовать при загрузке и записи параметров их символические имена – это решило бы проблему с возможными опечатками. Однако, есть более удачное решение – создать глобальный массив ключевых слов. Такой массив все равно понадобится нам для вспомогательного объекта, разбирающего текст. При сохранении параметров мы будем обращаться к элементам этого массива по индексам, таким образом, один и тот же массив слов будет использоваться и при сохранении параметров, и при загрузке.

С глобальным массивом ключевых слов и указанными изменениями функция `SaveText` примет следующий вид:

```
// Глобальный массив ключевых слов
// Индексы                    0       1        2
char *TestGen_Keywords[]={"type","period","impulse",NULL};
// define-константы для индексов
#define TESTGEN_KW_TYPE     0
#define TESTGEN_KW_PERIOD   1
#define TESTGEN_KW_IMPULSE  2
//=========================================
```

```
// Функция сохранения параметров
void TTestGenData::SaveText(void)
{ // Запись "type" и целого значения
  rdsWriteWordValueText(TestGen_Keywords[TESTGEN_KW_TYPE],Type);
  // Запись "period" и вещественного значения
  rdsWriteWordDoubleText(TestGen_Keywords[TESTGEN_KW_PERIOD],
                         Period);
  // Запись "impulse" и вещественного значения
  if(Type==2)
    rdsWriteWordDoubleText(TestGen_Keywords[TESTGEN_KW_IMPULSE],
                           Impulse);
}
//=======================================
```

Перед функцией описывается массив ключевых слов `TestGen_Keywords`. Технически, его можно было бы сделать статическим членом класса `TTestGenData`, но, для простоты примера, он описан как глобальный. Каждое ключевое слово – это строка (`char*`), поэтому массив описан как `char*<имя_массива>[]`. Он состоит из четырех элементов – трех ключевых слов и нулевого указателя (`NULL`), который будет использоваться в качестве маркера конца массива. После массива описываются `define`-константы для индексов каждого из трех ключевых слов. Это сделано для лучшей читаемости текста: запись `TestGen_Keywords[TESTGEN_KW_PERIOD]` выглядит гораздо более информативно, чем `TestGen_Keywords[1]` – сразу становится понятно, что этот элемент имеет отношение к периоду.

В новой версии функции `SaveText` больше не используется сервисная функция `rdsWriteBlockDataText`. Ее заменили вызовы `rdsWriteWordValueText` и `rdsWriteWordDoubleText`, специально предназначенные для передачи в РДС целого и вещественного значений для записи вместе с ключевыми словами. Целое значение параметра `Type` записывается при помощи функции `rdsWriteWordValueText`, которая добавляет к уже сформированному на данный момент тексту параметров блока ключевое слово, переданное в первом параметре (`TestGen_Keywords[TESTGEN_KW_TYPE]`, то есть “type”), и целое число `Type`, переданное во втором. При этом перед ключевым словом и после него автоматически добавляется один пробел. Преобразование целого числа в символьную форму производится внутри сервисной функции, поэтому в новой версии `SaveText` теперь не нужен массив символов для формирования текста. Вещественные значения `Period` и `Impulse` сохраняются функцией `rdsWriteWordDoubleText`, которая отличается от `rdsWriteWordValueText` только тем, что второй параметр у нее вещественный, а не целый. В результате трех вызовов этих функций в РДС будет передан точно такой же текст с параметрами блока, как и в первом варианте `SaveText` (стр. 158), в котором использовалась функция `sprintf`. Новая версия функции стала короче и выглядит теперь понятнее – на каждый сохраняемый параметр приходится только один вызов сервисной функции.

Теперь перепишем функцию `LoadText`, использовав в ней объект для разбора текста:

```
// Функция загрузки параметров
void TTestGenData::LoadText(char *text)
{ RDS_HOBJECT Parser; // Вспомогательный объект
  BOOL work=TRUE;     // Флаг цикла

  // Создание объекта для разбора текста
  Parser=rdsSTRCreateTextReader(TRUE);
  // Передача объекту массива ключевых слов
  rdsSTRAddKeywordsArray(Parser,TestGen_Keywords,-1,0);
```

```
    // Передача объекту разбираемого текста
    rdsSetObjectStr(Parser,RDS_HSTR_SETTEXT,0,text);

    // Цикл до тех пор, пока в тексте не кончатся слова
    while(work)
      { int id;         // Уникальный идентификатор слова
        // Извлечь из текста и опознать слово
        id=rdsSTRGetWord(Parser,NULL,NULL,NULL,TRUE);
        // Действия в зависимости от слова
        switch(id)
          { // Нет слова - конец текста
            case RDS_HSTR_DEFENDOFTEXT:
              work=FALSE;    // Выйти из цикла
              break;

            // Перевод строки - пропускаем
            case RDS_HSTR_DEFENDOFLINE:
              break;

            // Слово "type"
            case TESTGEN_KW_TYPE:
              // Извлекаем следующее слово и переводим в целое
              Type=rdsGetObjectInt(Parser,RDS_HSTR_READINT,TRUE);
              break;

            // Слово "period"
            case TESTGEN_KW_PERIOD:
              // Извлекаем следующее слово и переводим в double
              Period=rdsGetObjectDouble(Parser,RDS_HSTR_READDOUBLE,
                                        TRUE);
              break;

            // Слово "impulse"
            case TESTGEN_KW_IMPULSE:
              // Извлекаем следующее слово и переводим в double
              Impulse=rdsGetObjectDouble(Parser,RDS_HSTR_READDOUBLE,
                                         TRUE);
              break;

            default: // Слово не опознано - ошибка
              work=FALSE;    // Выйти из цикла
          } // Конец switch(...)
      } // Конец while(work)

    // Удаление вспомогательного объекта
    rdsDeleteObject(Parser);
  }
  //=======================================
```

В начале функции описываются две вспомогательных переменных: `Parser` (типа `RDS_HOBJECT`), для хранения идентификатора объекта, разбирающего текст, и логическая переменная `work`, которая будет использоваться как условие цикла `while`. Чтобы разбирать текст при помощи вспомогательного объекта РДС, нужно сначала создать этот объект и передать ему массив ключевых слов и текст, после чего можно будет вызывать его в цикле и запрашивать идентификаторы считанных из текста слов.

Для создания объекта используется сервисная функция РДС `rdsSTRCreateTextReader`, принимающая единственный логический параметр: `TRUE`,

если слова текста нужно сравнивать с ключевыми без учета регистра, и FALSE в противном случае. В данном случае регистр слов нас не интересует, поэтому в функцию передается значение TRUE. Возвращенный идентификатор созданного функцией объекта записывается в переменную Parser.

После того, как объект создан, необходимо передать ему массив ключевых слов, описанный ранее перед функцией SaveText, и сам текст, который этот объект будет разбирать на слова. Массив передается объекту при помощи функции rdsSTRAddKeywordsArray:

```
BOOL RDSCALL rdsSTRAddKeywordsArray(
      RDS_HOBJECT Parser,   // Объект
      LPSTR *pWords,        // Массив слов
      int WordCount,        // Число слов в массиве или -1
      int StartId);         // Начальный идентификатор
```

В первом параметре функции передается идентификатор объекта, разбирающего текст – в нашем случае он хранится в переменной Parser. Во втором параметре передается указатель на начало массива слов TestGen_Keywords, имеющий тип LPSTR*. Тип LPSTR в Windows API соответствует стандартному типу char* языка C, поэтому LPSTR* будет соответствовать char**, то есть типу "указатель на указатель на char", что соответствует описанию массива (char *TestGen_Keywords[]). В параметре WordCount передается число ключевых слов, которое нужно считать из массива, или –1, если нужно считать все слова до значения NULL. В данном случае массив ключевых слов завершается нулевым значением, и из него нужно считать все слова, поэтому этот параметр в примере равен –1. Наконец, в последнем параметре StartId передается идентификатор, который получит первое ключевое слово массива. Следующее за ним слово получит идентификатор StartId+1, следующее – StartId+2 и т.д. Эти идентификаторы будут возвращаться объектом при совпадении слова, считанного из текста, с одним из слов массива. В данном примере в StartId передается 0, поэтому идентификаторы ключевых слов будут совпадать с их индексами в массиве и, таким образом, с define-константами, описанными для этих индексов сразу после массива.

Далее нужно указать созданному объекту на текст, который он будет разбирать. Для этого используется универсальная функция передачи строки объекту rdsSetObjectStr. Эта функция уже использовалась при создании окна настройки параметров этого же блока (стр. 124). Как и парная ей функция получения строки rdsGetObjectStr, а также другие универсальные функции получения и установки параметров объекта (rdsGetObject... и rdsSetObject...), эта функция может использоваться для взаимодействия с любым вспомогательным объектом РДС. Результат ее действия зависит от типа объекта, идентификатор которого передан в первом параметре, и целых констант, переданных в втором и третьем. При вызове этой функции для объекта, созданного при помощи rdsSTRCreateTextReader, с константой RDS_HSTR_SETTEXT (вторая целая константа при этом не используется, поэтому в третьем параметре функции передается 0), в объект передается указатель на разбираемый текст из четвертого параметра функции, то есть значение переменной text.

Теперь, когда объект подготовлен к работе, можно начинать разбор текста. В цикле while из текста будет извлекаться слово за словом до тех пор, пока текст не закончится. Чтением слова из текста с одновременным поиском соответствующего ему идентификатора в массиве ключевых слов занимается функция rdsSTRGetWord:

```
int RDSCALL rdsSTRGetWord(
      RDS_HOBJECT Parser,   // Объект, разбирающий текст
      LPSTR      *pWord,    // Возвращаемый указатель на слово
      LPSTR      *pNext,    // Указатель на следующее слово
```

```
        char        *pSym,      // Тип слова или первый символ
        BOOL        Analyse);   // Сравнивать ли с ключевыми словами
```

Функция rdsSTRGetWord очень похожа на уже рассмотренную ранее rdsGetTextWord, но, в отличие от последней, эта функция возвращает не указатель на считанное во внутренний буфер слово (теперь он возвращается через параметр pWord), а целый идентификатор опознанного ключевого слова, если параметр Analyse равен TRUE. Если считанное слово не совпадает ни с одним из ключевых, или параметр Analyse равен FALSE, возвращается специальный идентификатор неопознанного слова. По умолчанию это константа RDS_HSTR_DEFUNKNOWNWORD, равная –1, но, при необходимости, для неопознанного слова можно установить другое значение, чтобы оно не пересекалось с используемыми для ключевых слов. В нашем примере ключевые слова нумеруются начиная с 0, поэтому можно оставить стандартное значение. При обнаружении конца строки и конца текста также возвращаются специальные идентификаторы, по умолчанию – RDS_HSTR_DEFENDOFLINE (–2) и RDS_HSTR_DEFENDOFTEXT (–3) соответственно.

Идентификатор считанного из текста слова присваивается вспомогательной переменной id. Параметры pWord, pNext и pSym в вызове rdsSTRGetWord имеют значение NULL, поскольку в данном случае нас не интересует ни само считанное слово, ни его тип, ни указатель на следующее – нам нужен только идентификатор. Поскольку идентификатор – целое число, больше не нужна конструкция if … else if …, как в прошлом примере, вместо нее используется более наглядный оператор switch(id).

Мы не меняли у объекта значения стандартных идентификаторов, поэтому при обнаружении конца текста в id будет записана константа RDS_HSTR_DEFENDOFTEXT. В этом случае необходимо выйти из цикла разбора текста, для этого условию цикла (переменной work) присваивается значение FALSE. При обнаружении конца строки (id равно RDS_HSTR_DEFENDOFLINE) необходимо просто пропустить его и продолжить разбор текста – после соответствующего оператора case внутри switch записан оператор break без каких-либо действий. Далее следуют три оператора case для каждого из трех ключевых слов, которые используются в этом блоке.

Если из текста было считано слово “type”, функция rdsSTRGetWord вернет число 0, поскольку “type” – первое слово, то есть нулевой индекс, в переданном объекту массиве ключевых слов TestGen_Keywords, и в качестве начального идентификатора для слов из этого массива было указано нулевое значение: 0 + 0 = 0. Сразу после описания массива TestGen_Keywords перед функцией SaveText для нулевого идентификатора, соответствующего этому слову, была определена define-константа TESTGEN_KW_TYPE, поэтому действия, выполняемые при обнаружение слова “type”, записаны внутри switch(id) после оператора “case TESTGEN_KW_TYPE:”.

В принципе, по аналогии с предыдущей версией функции LoadText, для чтения типа формируемого сигнала можно было бы ввести вспомогательную переменную word типа char* и записать такую конструкцию:

```
rdsSTRGetWord(Parser,&word,NULL,NULL,FALSE);
Type=atoi(word);
```

В первой строке из текста считывается очередное слово, и указатель на него записывается в переменную word (распознавание ключевых слов при этом не производится – параметр Analyse функции rdsSTRGetWord равен FALSE), во второй – это слово преобразуется в целое число и присваивается параметру блока Type. Однако, есть более простой способ – можно сразу запросить у объекта Parser преобразование следующего слова в целое число при помощи стандартной функции получения данных объекта rdsGetObjectInt с параметром RDS_HSTR_READINT. Третий параметр функции (TRUE) в данном случае указывает на необходимость пропускать переводы строк, если они встретятся перед следующим словом текста. Таким образом, для чтения очередного слова текста и

преобразования его в целое число с пропуском всех “лишних” переводов строк теперь требуется всего один вызов сервисной функции.

Аналогичным образом, при обнаружении в тексте слов “period” и “impulse” (оператор `case` с константами `TESTGEN_KW_PERIOD` и `TESTGEN_KW_IMPULSE` соответственно), следующее за ними слово преобразуется в вещественное число при помощи вызова `rdsGetObjectDouble` с параметром `RDS_HSTR_READDOUBLE` и присваивается соответствующему параметру блока. Если же считанное из текста ключевое слово не было опознано, будет выполнен оператор, следующий за `default` внутри `switch(id)`, и переменной `work` будет присвоено значение `FALSE`, что приведет к выходу из цикла `while(work)`.

Перед завершением функции загрузки параметров необходимо уничтожить вспомогательный объект, созданный функцией `rdsSTRCreateTextReader`. Объект уничтожается при помощи универсальной сервисной функции `rdsDeleteObject`, которая уже использовалась в других примерах для уничтожения объектов-окон. Эта функция может корректно удалить любой вспомогательный объект РДС.

Текст новой функции `LoadText` получился несколько длиннее старого варианта, но зато приобрел более четкую структуру. При этом новый вариант функции будет выполняться несколько быстрее. Какой из способов разбора текста выбрать – решать программисту.

§2.8.5. Сохранение параметров блока в формате INI-файла

Рассматривается вспомогательный объект РДС, позволяющий организовать хранение личных данных блока в текстовом формате, похожем на стандартные INI-файлы Windows (“имя=значение”). В один из предыдущих примеров добавляются процедуры сохранения и загрузки параметров с использованием этого объекта.

Выше уже приводились причины, по которым текстовый формат хранения параметров блока предпочтительнее двоичного: текстовый формат более универсален, при его использовании менее вероятны проблемы с совместимостью разных версий моделей, данные в текстовом формате могут быть, при необходимости, просмотрены и отредактированы пользователем без использования РДС. Однако, поддержка полноценного текстового формата требует от программиста достаточно больших усилий, даже при использовании вспомогательного объекта для разбора текста. Требуется организовать цикл для считывания текста по словам, сравнивать считанные слова с ключевыми тем или иным способом и т.д. При желании, разработчик модели блока может упростить себе жизнь, используя для записи параметров блока не пары “<*ключевое_слово*> <*значение*>”, а формируя текст в формате INI-файлов Windows, то есть в виде отдельных строк “<*ключевое_слово*>=<*значение*>”, разбитых на секции, названия которых указываются в квадратных скобках. В РДС есть вспомогательный объект для работы с таким форматом данных, причем его можно использовать как при формировании текста с параметрами, так и при его разборе, что позволит сделать функции записи и чтения параметров очень похожими. Надо заметить, что скорость работы у него несколько ниже, чем у описанного выше объекта для поиска ключевых слов, и это нужно учитывать при создании моделей блоков с большим числом параметров.

При рассмотрении записи параметров в двоичном формате мы использовали в качестве примера модель `Test1` (стр. 154). Хотя эта модель и не выполняет никаких полезных действий, для иллюстрации различных способов сохранения параметров она довольно удобна, поскольку у нее есть личная область данных и два параметра разных типов. Добавим в нее поддержку текстового формата хранения параметров с использованием нового вспомогательного объекта, выделив для удобства загрузку и сохранение параметров в отдельные функции-члены класса `TTest1Data`. Описание класса личной области данных блока с двумя новыми функциями будет выглядеть следующим образом:

```
//====== Класс личной области данных ======
class TTest1Data
{ public:
    int IParam;       // Целый параметр
    double DParam;    // Вещественный параметр
    void SaveText(void);   // Функция записи параметров
    void LoadText(char *text); // Функция загрузки параметров
    int Setup(void);  // Функция настройки параметров
    TTest1Data(void)  // Конструктор класса
      { IParam=0; DParam=0.0;
        rdsMessageBox("Область создана","TTest1Data",MB_OK);
      };
    ~TTest1Data()     // Деструктор класса
      { rdsMessageBox("Область удалена","TTest1Data",MB_OK);};
};
//=========================================
```

Модель блока будет вызывать функции `SaveText` и `LoadText` при сохранении и загрузке параметров точно так же, как и модели в предыдущих примерах. В оператор `switch` внутри функции модели (см. стр. 154) необходимо добавить два новых оператора `case`:

```
  // Запись параметров в текстовом формате
  case RDS_BFM_SAVETXT:
    data=(TTest1Data*)(BlockData->BlockData);
    data->SaveText();
    break;
  // Загрузка параметров в текстовом формате
  case RDS_BFM_LOADTXT:
    data=(TTest1Data*)(BlockData->BlockData);
    data->LoadText((char*)ExtParam);
    break;
```

В функции `SaveText` мы будем формировать текст с параметрами блока при помощи вспомогательного объекта для работы с INI-файлами:

```
// Функция сохранения параметров
void TTest1Data::SaveText(void)
{ RDS_HOBJECT ini; // Вспомогательный объект
  // Создание объекта для работы с образом INI-файла
  ini=rdsINICreateTextHolder(TRUE);
  // Создание новой секции "General"
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"General");
  // Запись двух параметров блока
  rdsINIWriteInt(ini,"IParam",IParam);
  rdsINIWriteDouble(ini,"DParam",DParam);
  // Передача сформированного в объекте текста в РДС для записи
  rdsCommandObject(ini,RDS_HINI_SAVEBLOCKTEXT);
  // Удаление объекта
  rdsDeleteObject(ini);
}
//=========================================
```

Вспомогательный объект, работающий с образом INI-файла в памяти, создается сервисной функцией `rdsINICreateTextHolder`. Единственный логический параметр функции указывает на необходимость учитывать (`FALSE`) или игнорировать (`TRUE`) регистр символов в именах параметров и секций файла. В данном случае регистр нам не важен, поэтому в функцию передается значение `TRUE`. Идентификатор созданного объекта присваивается переменной `ini`.

Сразу после создания объекта он не содержит ни одной секции. Несмотря на то, что в данном примере в блоке всего два параметра, и в разбиении их на секции нет

необходимости, нам все равно придется записывать параметры в какую-либо секцию образа INI-файла – таковы требования формата. Будем хранить оба параметра в секции с названием “General”. Прежде чем записывать значения параметров блока, необходимо создать эту секцию при помощи универсальной функции передачи строки объекту `rdsSetObjectStr` с параметром `RDS_HINI_CREATESECTION`. Начиная с этого момента, все команды записи и чтения параметров будут работать с этой секцией.

Теперь, когда в образе INI-файла создана секция, можно записывать в нее параметры блока. Для этого используются сервисные функции `rdsINIWriteInt` (для целого параметра) и `rdsINIWriteDouble` (для вещественного), в которые, помимо идентификатора объекта, передается имя параметра и его значение. В данном случае имена параметров в тексте будут совпадать с именами параметров в классе блока: “IParam” для `IParam` и “DParam” для `DParam`. Имя секции в эти функции не передается, они всегда работают с текущей секцией образа файла. В данном случае текущей будет последняя созданная секция, то есть “General”.

Теперь, когда образ INI-файла сформирован в объекте, необходимо передать его в РДС для записи вместе с другими параметрами блока. Самый простой способ сделать это – вызвать функцию `rdsCommandObject` с параметром `RDS_HINI_SAVEBLOCKTEXT`. После этого вспомогательный объект больше не нужен – его следует уничтожить при помощи функции `rdsDeleteObject`.

При сохранении блока в текстовом формате для блока с измененной моделью будет записан текст следующего вида (результат работы функции `SaveText` выделен жирным):

```
dllblock name "Block1"
 begin
  pos 10 10
  layer id 1
  vars
   begin
    signal name "Start" in menu run default 0
    signal name "Ready" out menu default 0
   end
  dll file "$DLL$\\testdll.dll" func "Test1" cycle setup "" auto
  dlldata text
   [General]
   IParam=1
   DParam=5.5
  enddlldata
 end
```

Как обычно, текст, сформированный моделью блока, размещается между строками “dlldata text” и “enddlldata”.

Функция `SaveText` в этом примере получилась сложнее аналогичных функций из прошлых примеров, в которых не использовались вспомогательные объекты. В принципе, эту функцию тоже можно было бы переписать так, чтобы точно такой же текст формировался без участия объекта. В этом случае функция примет следующий вид:

```
// Функция сохранения параметров – упрощенный вариант
void TTest1Data::SaveText(void)
{ char buffer[1024];  // Буфер для формирования текста
  // Формирование текста в буфере при помощи функции sprintf
  sprintf(buffer,
          "[General]\nIParam=%d\nDParam=%lf",
          IParam,DParam);
  // Передача сформированного текста в РДС
  rdsWriteBlockDataText(buffer,FALSE);
}
//=========================================
```

Функция получится значительно короче, однако, это будет ее единственным достоинством. При увеличении числа параметров блока строка формата функции sprintf очень скоро разрастется до огромных размеров, что существенно затруднит отладку и внесение изменений в нее. Чтобы найти спецификатор формата для какого-либо параметра блока, придется определять его номер в списке аргументов sprintf, а затем отсчитывать в форматной строке такое же число спецификаторов. Строка формата не отличается хорошей читаемостью, поэтому здесь легко допустить ошибку, что приведет к тому, что параметры блока будут записываться в текст не на своих местах или в неправильном формате. Кроме того, необходимо постоянно следить за тем, чтобы формируемый текст гарантированно уместился в буфер, независимо от фактических значений параметров. Можно, конечно, записывать в буфер по одному значению параметра за раз, и каждый раз после этого вызывать rdsWriteBlockDataText. Это позволит использовать меньший буфер и улучшит читаемость текста, однако, при этом потеряется единственное достоинство упрощенной функции SaveText – ее длина станет больше, чем у исходной версии, использующей вспомогательный объект. По этой причине первый вариант SaveText предпочтительнее: объект сам следит за необходимым размером своих внутренних буферов.

Рассмотрим теперь функцию LoadText, которая будет извлекать значения параметров блока из текста в формате INI-файла:

```
// Функция загрузки параметров
void TTest1Data::LoadText(char *text)
{ RDS_HOBJECT ini; // Вспомогательный объект
  // Создание объекта для работы с образом INI-файла
  ini=rdsINICreateTextHolder(TRUE);
  // Передача в объект текста, полученного из РДС
  rdsSetObjectStr(ini,RDS_HINI_SETTEXT,0,text);
  // Установить текущую секцию
  if(rdsINIOpenSection(ini,"General"))
    { // Такая секция есть в тексте - считать из нее параметры
      IParam=rdsINIReadInt(ini,"IParam",0);
      DParam=rdsINIReadDouble(ini,"DParam",0.0);
    }
  // Удаление объекта
  rdsDeleteObject(ini);
}
//=======================================
```

Прежде всего, как и в SaveText, в этой функции вызывается rdsINICreateTextHolder для создания вспомогательного объекта. Затем в этот объект передается текст параметров блока, полученный из РДС. Для этого используется вызов rdsSetObjectStr с параметром RDS_HINI_SETTEXT. В этот момент объект разбирает переданный ему текст на секции, имена параметров и их значения, и переводит его во внутренний формат для ускорения поиска. Теперь можно получить из объекта значения параметров блока, но прежде необходимо указать, с какой секцией текста мы собираемся работать. Установка текущей секции производится с помощью функции rdsINIOpenSection, в которую, кроме идентификатора объекта, передается имя секции (в данном случае, “General”). После этого все команды чтения параметров (как и команды записи, хотя мы и не используем их в этой функции) будут обращаться к указанной секции. Функция возвращает TRUE, если такая секция есть в тексте, и FALSE, если ее нет. Читать параметры блока имеет смысл только в том случае, если секция существует, поэтому вызов rdsINIOpenSection производится в условии оператора if. Если функция вернула TRUE, значения параметров блока извлекаются из текущей секции при помощи функций rdsINIReadInt и rdsINIReadDouble, в которые передается идентификатор объекта, имя параметра и значение по умолчанию, которое будет возвращено в случае отсутствия в секции параметра с таким именем. В конце

функции созданный вспомогательный объект уничтожается при помощи `rdsDeleteObject`.

Можно заметить, что функция `LoadText` по структуре похожа на `SaveText`. Обе они начинаются созданием объекта и заканчиваются его уничтожением, созданию секции в `SaveText` соответствует установка текущей секции в `LoadText`, вызовам `rdsINIWrite...` при записи параметров соответствуют аналогичные вызовы `rdsINIRead...` для их чтения. Таким образом, написав и отладив функцию записи параметров, можно легко сделать на ее основе функцию чтения, заменив одни вызовы на другие с практически идентичными параметрами, и вставив в начало функции загрузки команду передачи текста в объект вместо команды передачи текста из объекта в РДС в конце функции записи. Из-за похожести этих функций значительно упрощается добавление в блок новых хранимых параметров: команды их чтения и записи будут вставляться примерно в одни и те же места двух похожих функций.

Функция загрузки параметров, работающая с форматом INI-файла, устроена значительно проще функции, поддерживающей разбор произвольного текста (см. функцию на стр. 164) – в ней не нужно организовывать цикл для пословного чтения. Если при произвольном разборе считанное из текста слово ищется в массиве ключевых слов, то в этом формате, наоборот, указанное ключевое слово (имя параметра) ищется в тексте. Из-за этого вспомогательный объект, работающий с образом INI-файла, требует для работы гораздо больше памяти, чем объект для поиска ключевых слов. В отличие от последнего, хранящего в памяти сравнительно небольшой массив с ключевыми словами и их целыми идентификаторами, он переводит во внутренний формат весь предоставленный ему текст, разбивая его на секции и выделяя имена параметров. Это несколько замедляет работу, но без этого объекту пришлось бы каждый раз просматривать весь текст в поисках нужной секции и нужного параметра, что привело бы к гораздо большему снижению скорости. Другой недостаток этого формата – его жесткость. В нем нельзя организовывать иерархические конструкции, вроде блоков `begin ... end`, которые широко используются в текстовом формате РДС и могут быть реализованы при разборе произвольного текста (хотя, следует отметить, что для этого придется затратить некоторые усилия – встроенного механизма для поддержки такой иерархии в объекте нет). Однако, для большинства блоков такой формат вполне подходит.

§2.9. Использование таймеров

Рассматривается использование таймеров – объектов РДС, позволяющих организовать вызов модели блока через определенные промежутки времени. Описываются циклические таймеры, вызывающие модель постоянно с заданным интервалом, и однократные, вызывающие модель один раз через указанный промежуток времени и требующие явного перезапуска. Отдельно рассматриваются особенности работы модели блока с несколькими таймерами одновременно.

§2.9.1. Таймеры в РДС

Описывается создание таймеров в РДС, режимы их работы и возможные способы вызова модели блока при срабатывании таймера.

Достаточно часто модели блока приходится выполнять различные действия через заданный интервал времени. Это может быть опрос каких-либо датчиков, обновление информации на экране и т.п. РДС включает в себя набор сервисных функций, позволяющих модели создавать произвольное число *таймеров* и получать информацию об их срабатывании. Таймеры могут быть как однократными, срабатывающими один раз по истечении заданного интервала времени, так и циклическими, вызывающими функцию модели с заданной периодичностью.

Для создания нового таймера или изменения параметров существующего используется сервисная функция rdsSetBlockTimer, которая возвращает идентификатор созданного таймера:

```
RDS_TIMERID RDSCALL rdsSetBlockTimer(
      RDS_TIMERID timer,    // Идентификатор таймера
      DWORD delay,          // Интервал срабатывания, мс
      DWORD mode,           // Режим и флаги
      BOOL start);          // Запустить таймер немедленно
```

Первый параметр функции timer – это идентификатор таймера, параметры которого необходимо изменить, или NULL, если необходимо создать новый таймер. Параметр delay указывает интервал работы таймера в миллисекундах. Если логический параметр start – истина, созданный или измененный таймер будет немедленно запущен и начнет отсчитывать время до интервала delay, в противном случае он будет ждать команды запуска. И, наконец, в параметре mode передается режим работы таймера, способ информирования модели о его срабатывании, а также дополнительные флаги, объединенные побитовой операцией “ИЛИ” (или простым сложением) в одно целое число типа DWORD.

Таймер может работать в одном из трех режимов:

- RDS_TIMERM_LOOP – циклическая работа. Отсчитав заданный интервал времени, таймер уведомляет об этом модель и автоматически перезапускается. Этот режим используется в тех случаях, когда модели необходимо получать сигнал от таймера через равные промежутки времени.
- RDS_TIMERM_STOP – однократное срабатывание. Запущенный таймер отсчитывает заданный интервал, уведомляет модель и останавливается. После этого его, при необходимости, можно запустить снова. Этот режим используется для создания задержек, или в тех случаях, когда нужно динамически менять интервал срабатывания у постоянно работающего таймера.
- RDS_TIMERM_DELETE – однократное срабатывание с автоматическим удалением таймера. Запущенный таймер отсчитывает заданный интервал, сообщает об этом модели и удаляется. Этот режим удобен для блоков, достаточно редко выполняющих какое-либо действие с задержкой.

Таймер уведомляет модель о своем срабатывании несколькими разными способами:

- RDS_TIMERS_SIGNAL – при срабатывании таймера первой сигнальной переменной блока (то есть сигналу запуска, по умолчанию он называется Start) присваивается значение 1. Таким образом, если блок в режиме расчета работает по сигналу, модель блока будет автоматически вызвана в режиме RDS_BFM_MODEL в первом после срабатывания таймера такте расчета. Никаких других вызовов модели при срабатывании таймера не производится. Такое уведомление удобно тем, что в текст модели блока не нужно добавлять каких-либо новых реакций, однако, модель в данном случае не сможет определить, вызвана она из-за срабатывания таймера или из-за срабатывания связи, подключенной к сигнальному входу запуска.
- RDS_TIMERS_TIMER – при срабатывании таймера модель вызывается в потоке расчета в режиме RDS_BFM_TIMER. При этом в третьем параметре функции модели (LPVOID ExtParam) передается идентификатор сработавшего таймера.
- RDS_TIMERS_WINREF – при срабатывании таймера модель блока вызывается в главном потоке в режиме RDS_BFM_WINREFRESH. Этот способ используется для обновления окон, принадлежащих блокам. В третьем параметре функции модели при этом передается указатель на структуру RDS_WINREFRESHDATA, по полям которой модель может понять, вызвана ли необходимость обновлять окна таймером или вызовом сервисной функции, и если таймером, то каким именно.
- RDS_TIMERS_SYSTIMER – при срабатывании таймера модель вызывается в главном потоке в режиме RDS_BFM_TIMER. От всех остальных способов срабатывания, включая

RDS_TIMERS_TIMER (при котором модель вызывается в этом же режиме, но в потоке расчета) этот отличается тем, что таймер работает не только в режиме расчета, но и в режимах редактирования и моделирования. Таймеры, для которых указаны способы срабатывания RDS_TIMERS_SIGNAL, RDS_TIMERS_TIMER и RDS_TIMERS_WINREF, останавливаются в момент остановки расчета и продолжают работу при его повторном запуске. Таймер RDS_TIMERS_SYSTIMER работает все время, вне зависимости от режима работы РДС.

Вместе со способом уведомления модели RDS_TIMERS_WINREF можно, при необходимости, указать флаг RDS_TIMERF_FIXFREQ – это исключит данный таймер из автоподстройки частоты обновления окон. При включенной автоподстройке РДС постоянно вычисляет суммарное время обновления всех окон системы, и, если это время будет слишком большим, частота обновления будет автоматически понижаться. Если в параметрах таймера указать флаг RDS_TIMERF_FIXFREQ, его частота изменяться не будет. При всех остальных способах уведомления модели о срабатывании таймера никакой автоподстройки не происходит и этот флаг игнорируется.

§2.9.2. Циклический таймер

Описывается циклический таймер, постоянно вызывающий модель блока через заданный интервал времени. Рассматривается пример блока, изображающего мигающую индикаторную лампочку.

Рассмотрим для примера блок, изображающий мигающую индикаторную лампочку на каком-либо пульте. Внешний вид этой лампочки может быть любым – мигание будет реализовано при помощи векторной картинки блока. Блок будет иметь логический вход Flash, при подаче на который единицы изображение блока должно мигать с частотой 1 Гц, привлекая внимание оператора. Для реализации самого мигания будет использоваться внутренняя логическая переменная State, значение которой будет инвертироваться по таймеру с задержкой в половину секунды, что и даст частоту в 1 Гц. В векторной картинке блока с этой переменной будет связан цвет изображающего индикатор прямоугольника (рис. 55). При нулевом значении переменной прямоугольник будет иметь цвет, заданные при его создании (в данном случае – белый), а при единичном – альтернативный (красный). Таким образом, меняя значение State, можно выбирать один из двух цветов прямоугольника.

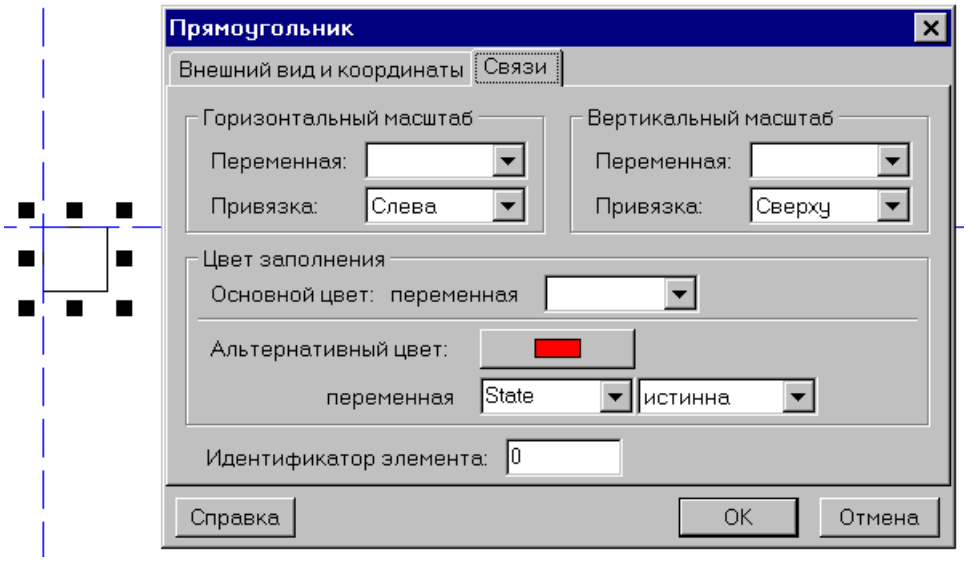


Рис. 55. Связь логической переменной с цветом в векторной картинке блока

Вместо прямоугольника можно было бы использовать круг или произвольный многоугольник, можно было бы вместо цвета фигуры связать с переменной `State` видимость какого-либо набора объектов (и сделать, например, мигающую надпись) и т.п. – в данном случае внешний вид блока ограничен только фантазией создателя векторной картинки, функция модели от этого не изменится.
Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Пуск*	*Вход/выход*
0	Start	Сигнал	1	✓	Вход
1	Ready	Сигнал	1		Выход
2	Flash	Логический	1	✓	Вход
3	State	Логический	1		Внутренняя

Чтобы не тратить зря системные ресурсы, как обычно, включим для блока запуск только по сигналу, и у входа `Flash` установим флаг “Пуск”, чтобы модель запустилась при срабатывании связи, подключенной к этому входу.

Модель блока с обычными макроопределениями для переменных будет иметь следующий вид:

```
//====== Модель простого мигающего блока ======
// Функция модели блока
extern "C" __declspec(dllexport) int RDSCALL SimpleFlash(
                    int CallMode,
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Flash (*((char *)(pStart+2)))
#define State (*((char *)(pStart+3)))
  // Вспомогательная переменная – указатель на личную область
  // В личной области хранится только идентификатор таймера
  RDS_TIMERID *data=(RDS_TIMERID*)(BlockData->BlockData);

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Отводим место для хранения идентификатора таймера
        BlockData->BlockData=data=new RDS_TIMERID;
        // Создаем таймер, интервал пока не устанавливаем
        *data=rdsSetBlockTimer(NULL,0,
                RDS_TIMERM_LOOP|RDS_TIMERS_TIMER,FALSE);
        break;

      // Очистка данных блока
      case RDS_BFM_CLEANUP:
        // Уничтожаем таймер
        rdsDeleteBlockTimer(*data);
        // Освобождаем память, где он хранился
        delete data;
        break;

      // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
```

```
        if(strcmp((char*)ExtParam,"{SSLL}"))
          return RDS_BFR_BADVARSMSG;
        return RDS_BFR_DONE;

      // Такт расчета
      case RDS_BFM_MODEL:
        if(Flash) // Включаем мигание
          { // Запускаем таймер с интервалом 500 мс
            rdsRestartBlockTimer(*data,500);
            // Сразу "зажигаем" изображение блока
            State=1;
          }
        else // Выключаем мигание
          { // Останавливаем таймер
            rdsStopBlockTimer(*data);
            // "Гасим" изображение блока
            State=0;
          }
        break;

      // Срабатывание таймера
      case RDS_BFM_TIMER:
        // Инвертируем состояние
        State=!State;
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений для переменных
#undef State
#undef Flash
#undef Ready
#undef Start
#undef pStart
}
//================================================
```

В этой модели в личной области данных блока хранится идентификатор таймера, с которым эта модель работает. Для удобства доступа к хранимому идентификатору в начале модели вводится вспомогательная переменная `data`, которой сразу присваивается указатель на личную область данных, приведенный к типу `RDS_TIMERID*`. При вызове в режиме `RDS_BFM_INIT` модель отводит память для хранения идентификатора таймера и, как обычно, запоминает указатель на нее в структуре данных блока. Затем функцией `rdsSetBlockTimer` создается сам таймер. Первый параметр функции – `NULL`, поскольку мы создаем новый таймер, а не изменяем параметры существующего. Интервал срабатывания таймера мы зададим позднее, при запуске таймера, поэтому в `rdsSetBlockTimer` он не задается – второй параметр функции равен нулю. Создаваемый таймер будет циклическим (константа `RDS_TIMERM_LOOP`), при его срабатывании модель будет вызываться в режиме `RDS_BFM_TIMER` (константа `RDS_TIMERS_TIMER`), таймер не запускается сразу после создания (последний параметр функции – `FALSE`).

При вызове модели в режиме `RDS_BFM_CLEANUP` (при удалении блока, закрытии РДС, отключении модели от блока и т.п.) созданный таймер удаляется функцией `rdsDeleteBlockTimer`. Технически, можно было бы этого не делать – при отключении модели от блока РДС автоматически удаляет все созданные блоком объекты, включая таймеры, однако, хороший стиль программирования обычно подразумевает явное удаление созданных объектов (если все время полагаться на автоматику, можно по привычке забыть уничтожить какой-либо объект, автоматическое удаление которого не предусмотрено).

В такте расчета (режим RDS_BFM_MODEL) модель анализирует значение логического входа Flash. Если оно равно единице (“истина”), таймер запускается с интервалом 500 мс. Начиная с этого момента модель будет вызываться в режиме RDS_BFM_TIMER два раза в секунду по системным часам. Кроме того, переменной State при запуске таймера присваивается значение 1, чтобы изображение блока “зажглось” сразу, а не через 500 миллисекунд, когда сработает таймер. Если же значение Flash равно нулю (“ложь”), таймер останавливается функцией rdsStopBlockTimer и его изображение “гасится” присваиванием нуля переменной State.

Наконец, при вызове модели в режиме RDS_BFM_TIMER, значение State инвертируется, что приведет к смене цвета изображения блока. Таким образом, цвет блока будет меняться два раза в секунду, что и создаст эффект мигания индикатора.

Для проверки работы блока можно подключить к его входу Flash стандартную переключающуюся кнопку (рис. 56). В режиме расчета при нажатой кнопке изображение блока должно мигать.

Рис. 56. Проверка работы мигающего блока

Приведенная выше модель имеет один существенный недостаток. Представим себе, что ко входу Flash подключен выход какого-либо блока, работающего каждый такт. Выход этого блока будет передаваться на вход Flash в каждом такте расчета, что будет каждый раз приводить к запуску модели нашего мигающего блока. В реакции на такт расчета наш блок анализирует значение Flash и запускает или останавливает таймер. Если единичное значение будет передаваться на вход модели в каждом такте, значит, таймер также будет перезапускаться в каждом такте. Фактически, этот постоянный перезапуск не даст ему считать, модель не будет вызываться в режиме RDS_BFM_TIMER, переменная State не будет инвертироваться, и блок не будет мигать.

Чтобы устранить этот эффект, необходимо запускать таймер при единичном значении Flash только в том случае, если он еще не запущен. Таким образом повторные срабатывания модели не приведут к перезапуску таймера. При этом останавливать таймер нужно при совпадении двух условий: значение Flash равно нулю и таймер в данный момент работает.

Для реализации такого поведения модели можно добавить в блок новую логическую переменную-флаг, взводить ее при запуске таймера и сбрасывать при остановке, а в такте расчета сравнивать ее со значением Flash и управлять таймером, если они не совпадают. Мы сделаем по-другому – при вызове модели в такте расчета будем проверять, запущен ли таймер, при помощи сервисной функции чтения параметров таймера rdsGetBlockTimerDescr. Для этого нужно ввести в функцию модели дополнительную переменную – структуру, в которую будут записаны запрошенные параметры таймера (выделена жирным шрифтом):

```
//====== Модель простого мигающего блока ======
// Функция модели блока
extern "C" __declspec(dllexport) int RDSCALL SimpleFlash(
                    int CallMode,
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{
  RDS_TIMERDESCRIPTION descr;
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
// (...)
```

В этой структуре нас будет интересовать единственное поле On, имеющее значение TRUE для работающего таймера и FALSE для остановленного. Реакцию модели на такт расчета перепишем следующим образом (изменения выделены жирным):

```
case RDS_BFM_MODEL:
  // Запрашиваем параметры таймера
  descr.servSize=sizeof(descr);
  rdsGetBlockTimerDescr(*data,&descr);
  if(Flash) // Включаем мигание, если таймер остановлен
    { if(!descr.On) // Таймер остановлен
        { rdsRestartBlockTimer(*data,500);
          State=1;
        }
    }
  else // Выключаем мигание, если таймер работает
    { if(descr.On) // Таймер работает
        { rdsStopBlockTimer(*data);
          State=0;
        }
    }
  break;
```

Прежде всего мы присваиваем полю servSize структуры описания таймера descr размер этой структуры в байтах – sizeof(descr). Без этого функция чтения параметров таймера, как и многие другие сервисные функции РДС, откажется работать. Далее мы вызываем функцию rdsGetBlockTimerDescr, которая заполнит структуру descr описанием таймера, идентификатор которого находится в *data. Затем мы запускаем таймер в том случае, если Flash – истина, и таймер не работает в данный момент (descr.On имеет значение FALSE), и останавливаем его, если Flash – ложь, и таймер работает (descr.On имеет значение TRUE). Теперь многократные повторные передачи значения 1 на вход Flash никак не повлияют на мигание нашего блока.

Можно заметить, что блок будет мигать только в режиме расчета, т.к. таймеры с параметром RDS_TIMERS_TIMER останавливаются вместе с расчетом. Если требуется, чтобы при остановленном расчете блок продолжал мигать, следует заменить константу RDS_TIMERS_TIMER на RDS_TIMERS_SYSTIMER. В этом случае таймер продолжит считать и при остановке расчета, и изображение блока будет мигать и в режиме моделирования. В режиме редактирования изображение мигать не будет – хотя модель будет вызываться по таймеру и значение переменной State будет инвертироваться, в режиме редактирования векторная картинка блока не отражает состояния его переменных, то есть значение переменной State не будет влиять на внешний виде блока. Если бы вместо векторной картинки мы задали для блока программное рисование (см. §2.10), добавив в его модель соответствующую реакцию, он мигал бы и в режиме редактирования.

§2.9.3. Однократно срабатывающий таймер

Описывается таймер, однократно вызывающий модель блока по истечении заданного интервала. Рассматривается пример блока, изображающего мигающую индикаторную лампочку, длительность горения которой не равна длительности паузы (после каждого срабатывания таймер перезапускается с новой задержкой).

В приведенном выше примере использовался циклический таймер – рассмотрим теперь работу с однократно срабатывающим. Допустим, нам нужен похожий индикатор, но скважность его мигания не должна равняться двум, то есть время “горения” и время “паузы” индикатора должно быть разным. Создадим блок, который при единичном значении входа Flash будет зажигаться на 500 миллисекунд, затем гаснуть на одну секунду, затем снова зажигаться на 500 мс и т.д. Циклический таймер нам здесь не поможет – интервал между его

срабатываниями постоянен. Воспользуемся однократно срабатывающим таймером, запуская его то на 500 мс, то на тысячу. При той же структуре переменных модель нового блока будет очень похожа на модель из предыдущего примера с некоторыми отличиями (выделены жирным):

```
//====== Модель неравномерно мигающего блока ======
// Функция модели блока
extern "C" __declspec(dllexport) int RDSCALL SimpleFlash1(
                    int CallMode,
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Flash (*((char *)(pStart+2)))
#define State (*((char *)(pStart+3)))
  // Вспомогательная переменная - указатель на личную область
  // В личной области хранится только идентификатор таймера
  RDS_TIMERID *data=(RDS_TIMERID*)(BlockData->BlockData);
  RDS_TIMERDESCRIPTION descr;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Отводим место для хранения идентификатора таймера
        BlockData->BlockData=data=new RDS_TIMERID;
        // Создаем таймер, интервал пока не устанавливаем
        *data=rdsSetBlockTimer(NULL,0,
                **RDS_TIMERM_STOP**|RDS_TIMERS_TIMER,FALSE);
        break;

      // Очистка данных блока
      case RDS_BFM_CLEANUP:
        // Уничтожаем таймер
        rdsDeleteBlockTimer(*data);
        // Освобождаем память, где он хранился
        delete data;
        break;

      // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSLL}"))
          return RDS_BFR_BADVARSMSG;
        return RDS_BFR_DONE;

      // Такт расчета
      case RDS_BFM_MODEL:
        // Запрашиваем параметры таймера
        descr.servSize=sizeof(descr);
        rdsGetBlockTimerDescr(*data,&descr);
        if(Flash) // Включаем мигание, если таймер остановлен
          { if(!descr.On) // Таймер остановлен
              { // Запускаем таймер на 500 мс
                rdsRestartBlockTimer(*data,500);
```

```
                State=1;
              }
          }
        else // Выключаем мигание, если таймер работает
          { if(descr.On) // Таймер работает
              { // Останавливаем таймер
                rdsStopBlockTimer(*data);
                State=0;
              }
          }
        break;

      // Срабатывание таймера
      case RDS_BFM_TIMER:
        if(State) // Индикатор был "зажжен"
          { // Гасим и запускаем таймер на 1 сек
            State=0;
            rdsRestartBlockTimer(*data,1000);
          }
        else // Индикатор был "погашен"
          { // Зажигаем и запускаем таймер на 500 мс
            State=1;
            rdsRestartBlockTimer(*data,500);
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений для переменных
#undef State
#undef Flash
#undef Ready
#undef Start
#undef pStart
}
//================================================
```

Фактически, в модели изменились только параметры функции создания таймера и реакция на срабатывание таймера. Таймер теперь создается с параметром `RDS_TIMERM_STOP` вместо `RDS_TIMERM_LOOP` (однократное срабатывание вместо циклического). В реакции на срабатывание таймера (`RDS_BFM_TIMER`) теперь анализируется значение переменной `State`. Если оно равно 1 (индикатор был зажжен), ей присваивается 0 и таймер запускается на одну секунду для отработки паузы. Если же оно равно нулю (индикатор погашен), ей присваивается единица и таймер запускается на 500 мс для отработки "горения". Таким образом, остановившийся таймер немедленно перезапускается с новым значением задержки.

Реакцию на срабатывание таймера можно переписать компактнее – в тексте модели выше она просто расписана подробно с комментариями для большей ясности программы. Те же действия могут быть выполнены и так:

```
      case RDS_BFM_TIMER:
        rdsRestartBlockTimer(*data,State?1000:500);
        State=!State;
        break;
```

Мы в любом случае заново запускаем таймер, просто его интервал зависит от значения `State`. Само значение `State` инвертируется, как и в предыдущей модели с циклическим таймером.

§2.9.4. Несколько таймеров в одной модели

Рассматриваются особенности работы модели блока с несколькими таймерами одновременно. Приводится пример блока, изображение которого вращается (для чего используется первый таймер) и мигает (по сигналу от второго таймера).

В двух предыдущих примерах модель блока создавала единственный таймер, поэтому в реакции на его срабатывание не проверялось, какой именно таймер сработал. Если блок работает с несколькими таймерами одновременно, такая проверка необходима.

Создадим новый блок-индикатор, изображение которого может не только мигать, изменяя свой цвет, при подаче единицы на вход `Flash`, но и вращаться вокруг своей оси при подаче единицы на вход `Rotate`. Мигание будет реализовано как и в предыдущих моделях, а для вращения будет использована новая вещественная переменная `Angle`, которая будет связана с углом поворота одного из векторных элементов картинки. Каждый раз при срабатывании таймера к этой переменной будет прибавляться небольшая постоянная величина, что приведет к вращению изображения с постоянной скоростью.

Чтобы изображение вращалось плавно, переменную `Angle` нужно изменять достаточно часто – например, десять раз в секунду (чаще менять ее не имеет смысла, максимальная частота обновления окна подсистемы в РДС равна 10 Гц). Мигать изображение должно значительно реже, например, один раз в две секунды. Таким образом, нам понадобятся два циклических таймера – один с интервалом в 100 миллисекунд (для увеличения `Angle`), другой – в одну секунду (для инвертирования `State`). В принципе, такой блок может обойтись и одним таймером в 100 мс, если считать число его срабатываний, и инвертировать переменную `State` после каждого десятого. Однако, модель в этом случае получается более сложной, поэтому проще всего создать два таймера и работать с ними независимо.

Для работы блоку нужны будут следующие переменные:

Смещение	*Имя*	*Тип*	*Размер*	*Пуск*	*Вход/выход*
0	Start	Сигнал	1	✓	Вход
1	Ready	Сигнал	1		Выход
2	Flash	Логический	1	✓	Вход
3	Rotate	Логический	1	✓	Вход
4	State	Логический	1		Внутренняя
5	Angle	double	8		Внутренняя

Рассмотрим функцию модели блока:

```
// Мигающий и вращающийся блок
// Структура личной области данных
struct TRotateFlashData
{ RDS_TIMERID FlashTimer;  // Таймер мигания
  RDS_TIMERID RotTimer;    // Таймер вращения
};
extern "C" __declspec(dllexport) int RDSCALL RotateFlash(
                     int CallMode,
                     RDS_PBLOCKDATA BlockData,
                     LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Flash (*((char *)(pStart+2)))
```

```
#define Rotate (*((char *)(pStart+3)))
#define State (*((char *)(pStart+4)))
#define Angle (*((double *)(pStart+5)))
  // Вспомогательная переменная – указатель на личную область
  TRotateFlashData *data=(TRotateFlashData*)(BlockData->BlockData);
  // Структура для получения параметров таймера
  RDS_TIMERDESCRIPTION descr;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Отведение памяти под личную область
        BlockData->BlockData=data=new TRotateFlashData;
        // Создание двух одинаковых таймеров
        data->FlashTimer=rdsSetBlockTimer(NULL,0,
                          RDS_TIMERM_LOOP|RDS_TIMERS_TIMER,FALSE);
        data->RotTimer=rdsSetBlockTimer(NULL,0,
                          RDS_TIMERM_LOOP|RDS_TIMERS_TIMER,FALSE);
        break;

      // Очистка данных блока
      case RDS_BFM_CLEANUP:
        // Уничтожение таймеров
        rdsDeleteBlockTimer(data->FlashTimer);
        rdsDeleteBlockTimer(data->RotTimer);
        // Освобождение памяти, занятой под личную область
        delete data;
        break;

      // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSLLLD}"))
          return RDS_BFR_BADVARSMSG;
        return RDS_BFR_DONE;

      // Такт расчета
      case RDS_BFM_MODEL:
        // Подготовка структуры descr к чтению данных таймера
        descr.servSize=sizeof(descr);
        // Управление миганием
        rdsGetBlockTimerDescr(data->FlashTimer,&descr);
        if(Flash) // Включить мигание
          { if(!descr.On) // Таймер остановлен
              { // Запускаем таймер и "зажигаем" индикатор
                rdsRestartBlockTimer(data->FlashTimer,1000);
                State=1;
              }
          }
        else // Выключить мигание
          { if(descr.On) // Таймер работает
              { // Останавливаем и "гасим" индикатор
                rdsStopBlockTimer(data->FlashTimer);
                State=0;
              }
          }
        // Управление вращением
        rdsGetBlockTimerDescr(data->RotTimer,&descr);
```

```
          if(Rotate) // Включить вращение
            { if(!descr.On) // Таймер остановлен - запускаем
                rdsRestartBlockTimer(data->RotTimer,100);
            }
          else // Выключить вращение
            { if(descr.On) // Таймер работает - останавливаем
                rdsStopBlockTimer(data->RotTimer);
            }
          break;

        // Срабатывание таймера
        case RDS_BFM_TIMER:
          // Сравниваем с имеющимися таймерами
          if(ExtParam==((LPVOID)data->FlashTimer))
            // Это - таймер мигания
            State=!State;
          else // Если не таймер мигания, значит, вращения
            Angle=fmod(Angle+0.4,2*M_PI);
          break;
      }
    return RDS_BFR_DONE;

  // Отмена макроопределений для переменных
  #undef Angle
  #undef State
  #undef Rotate
  #undef Flash
  #undef Ready
  #undef Start
  #undef pStart
  }
  //================================================
```

Поскольку модель теперь использует два таймера, в личной области данных блока необходимо предусмотреть хранение идентификаторов обоих. Для этого перед функцией модели описывается структура `TRotateFlashData` с двумя полями: `FlashTimer` для идентификатора таймера, который будет управлять миганием изображения блока, и `RotTimer` для идентификатора таймера, управляющего вращением. При инициализации блока (режим `RDS_BFM_INIT`) модель отводит память под эту структуру и создает два одинаковых циклических таймера, не задавая пока им интервал срабатывания. При очистке данных блока (режим `RDS_BFM_CLEANUP`) оба таймера уничтожаются и занятая память освобождается. В этом отношении данная модель отличается от пары предыдущих только наличием двух таймеров вместо одного.

Реакция на такт расчета этой модели также похожа на дважды повторенную реакцию предыдущих. Сначала выполняются действия по запуску или остановке таймера мигания `FlashTimer`. В поле `servSize` структуры описания таймера `descr` записывается размер этой структуры, после чего вызывается функция `rdsGetBlockTimerDescr`. Если вход `Flash` получил значение 1 и таймер `FlashTimer` в данный момент не работает, этот таймер запускается с частотой 1 Гц (задержка в одну секунду), и переменной `State` присваивается единица, чтобы индикатор немедленно зажегся. Если значение `Flash` – нулевое, и таймер работает, модель останавливает его и присваивает переменой `State` ноль, гася индикатор.

Затем производится запуск или остановка вращения изображения блока в зависимости от значения переменной `Rotate`. Снова вызывается функция `rdsGetBlockTimerDescr`, но теперь уже для таймера `RotTimer` (в поле `servSize` структуры `descr` не записывается

размер этой структуры, т.к. он там сохранился с предыдущего вызова). Если значение Rotate равно 1, и таймер не работает, он запускается с интервалом 100 мс. Если же значение Rotate – нулевое, и таймер работает, он останавливается.

Реакция на срабатывание таймера этой модели (режим RDS_BFM_TIMER) уже сильно отличается от предыдущих примеров. В модели теперь два таймера, и для них должны выполняться разные действия. Чтобы модель могла опознать таймер, срабатывание которого запустило ее в режиме RDS_BFM_TIMER, РДС передает функции модели в параметре ExtParam идентификатор этого таймера, приведенный к типу LPVOID. Этот идентификатор можно сравнить с идентификаторами таймеров, хранящихся в личной области данных блока, и определить таким образом, какой из них сработал.

Может возникнуть вопрос: почему приведение идентификатора таймера RDS_TIMERID к типу LPVOID (то есть void*, произвольному указателю) допустимо? Дело в том, что идентификатор таймера в РДС на самом деле представляет собой указатель на некую внутреннюю структуру, в которой хранятся параметры этого таймера. По совпадению идентификатора с указателем на одну из структур РДС и опознает таймер. Описание типа RDS_TIMERID в файле RdsDef.h выглядит следующим образом:

```
typedef LPVOID RDS_TIMERID;
```

Таким образом, идентификаторы таймеров можно свободно приводить к типу LPVOID.

При вызове нашей модели в режиме RDS_BFM_TIMER переданный в ExtParam указатель сравнивается с приведенным к типу LPVOID идентификатором таймера мигания data->FlashTimer. Если они совпали, значит, сработал таймер мигания, и нужно инвертировать переменную State. Если они не совпали, значит, сработавший таймер – таймер вращения, поскольку других таймеров в этой модели нет. В этом случае к переменной Angle добавляется константа 0.4 и при помощи функции взятия остатка от деления fmod из получившегося результата выбрасывается целое число полных оборотов (2π), чтобы избежать бесконечного роста значения Angle по мере вращения блока. Значение 0.4 радиана (23°) выбрано опытным путем – добавляя это значение к углу поворота изображения каждые 100 мс, мы будем наблюдать вращение со скоростью примерно 2 градуса в секунду, что выглядит достаточно убедительно. При желании, константу можно изменить, увеличив или уменьшив скорость вращения.

Для проверки работы блока к его входам Flash и Rotate следует присоединить переключающиеся кнопки (рис. 57). В зависимости от нажатых кнопок, блок должен вращаться, мигать, или делать и то, и другое. Разумеется, он будет делать это только в режиме расчета, поскольку для обоих таймеров задана константа RDS_TIMERS_TIMER. Чтобы заставить его вращаться и мигать еще и в режиме моделирования, следует заменить эту константу на RDS_TIMERS_SYSTIMER.

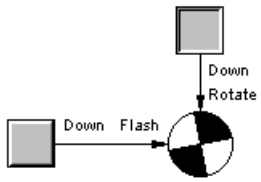


Рис. 57. Проверка работы блока с двумя таймерами

§2.10. Программное рисование внешнего вида блока

Рассматриваются способы программного рисования внешнего вида блоков в окне подсистемы и вывод дополнительных изображений поверх блоков. Также рассматривается создание в окнах подсистем специальных панелей, на которых блоки блоки могут размещать элементы управления Windows или использовать их для вывода изображений, формируемых сторонними библиотеками. В частности, приводится пример построения на панели трехмерного изображения при помощи библиотеки OpenGL.

§2.10.1. Рисование изображения блока в окне подсистемы

Рассматривается программное рисование внешнего вида блоков на примере простого вертикального индикатора уровня и построения графика зависимости значения входа блока от системной переменной времени.

Внешний вид блока в РДС задается одним из трех способов: прямоугольником с текстом внутри, векторной картинкой с возможностью анимации, или программой рисования в функции модели блока. Прямоугольник с текстом – самое простое изображение блока. Цвет прямоугольника, текст внутри него и шрифт, которым выводится этот текст, задаются жестко и не могут отображать состояние блока или значения каких-либо его переменных. Этого вполне достаточно для большинства блоков, занимающихся исключительно расчетом, и ничего не индицирующих.

Использование векторной картинки дает больше свободы в выборе внешнего вида блока. Она может состоять из геометрических фигур, положение, относительные размеры, угол поворота, цвет и видимость которых могут быть связаны с различными переменными. Кроме того, картинка может включать текст, который также может быть связан со строковыми или числовыми переменными и отображать их значения. С помощью векторных картинок можно создавать как простые индикаторы, так и довольно сложные анимированные изображения. Даже если изображение блока должно быть статичным, ему иногда задают векторную картинку, чтобы иметь возможность включить в изображение символы разного цвета и начертания, стрелки, геометрические фигуры и т.п.

Самый сложный, но, при этом, самый богатый возможностями способ создания изображения блока – программное рисование из его модели. Используя стандартные функции рисования API Windows или сервисные графические функции РДС (фактически, представляющие собой оболочки функций API) модель блока может формировать произвольное изображение на рабочем поле окна подсистемы. Каждый раз, когда окно подсистемы, в которой находится блок, обновляется, функция модели блока вызывается с константой `RDS_BFM_DRAW`, при этом ей передается положение и размер прямоугольной области, которую занимает блок, контекст устройства Windows, на котором необходимо построить изображение, и некоторые другие параметры, которые могут потребоваться для рисования. Многие сложные изображения (графики, диаграммы и т.п.) могут быть построены только таким образом.

В качестве первого примера рассмотрим один из простейших программно рисуемых индикаторов – индикатор уровня. Прямоугольное изображение блока будет разделено на две части по вертикали: нижняя часть, высота которой пропорциональна значению входа, будет закрашена одним цветом (например, синим), верхняя – другим (например, белым). Для определенности будем считать, что при нулевом значении входа раздел будет проходить по нижней границе прямоугольника блока, а при значении, равном 100 – по верхней. Таким образом, блок будет рисовать вертикальный столбик, высота которого в процентах относительно полной высоты блока равна значению входа. Такие индикаторы применяются довольно часто, причем обычно максимальное и минимальное отображаемое значение у них настраивается пользователем, но, для упрощения примера, мы будем считать их константами 0 и 100 соответственно. По этой же причине мы не будем делать настраиваемыми цвета блока.

Для такого индикатора нужен единственный вещественный вход (назовем его x), поэтому структура переменных блока будет выглядеть следующим образом:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	x	double	8	Вход

Реакции на такт расчета у этого блока не будет (он ничего не считает, только индицирует), поэтому состояние флага “Пуск” для его входа не важно, но лучше его сбросить, и установить для блока запуск по сигналу, чтобы модель вообще не запускалась в тактах расчета.

Для большей ясности примера, вынесем рисование в отдельную функцию, но писать ее пока не будем – ограничимся ее прототипом перед функцией модели. Параметры передаваемые в функцию рисования, можно описать уже сейчас: это указатель на структуру RDS_DRAWDATA, передаваемую в функцию модели блока при ее вызовах для рисования изображений, и вещественное значение входа блока (функция рисования, в отличие от функции модели, не будет иметь доступа к переменным блока, поэтому отображаемое значение нужно передать ей явно).

Модель блока с прототипом функции рисования будет иметь следующий вид:

```
// Прототип функции рисования
void SimpleLevelIndicatorDraw(RDS_PDRAWDATA draw,double val);
// Модель простого индикатора уровня
extern "C" __declspec(dllexport) int RDSCALL SimpleLevelIndicator(
                int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define x (*((double *)(pStart+2)))

  switch(CallMode)
    { // Проверка допустимости типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSD}"))
          return RDS_BFR_BADVARSMSG;
        return RDS_BFR_DONE;

      // Рисование внешнего вида блока
      case RDS_BFM_DRAW:
        SimpleLevelIndicatorDraw((RDS_PDRAWDATA)ExtParam,x);
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef x
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Модель содержит всего две реакции – обычную проверку допустимости типа статических переменных (RDS_BFM_VARCHECK) и реакцию на вызов в режиме рисования изображения

блока RDS_BFM_DRAW. В этом режиме в параметре ExtParam передается указатель на структуру RDS_DRAWDATA, содержащую все необходимые для рисования данные: контекст устройства Windows типа HDC, на котором нужно рисовать, размеры описывающего прямоугольника блока, текущий масштаб и т.п. В “RdsDef.h” эта структура описана следующим образом:

```
typedef struct
{ HDC dc;        // Контекст устройства Windows (где рисовать)
  BOOL CalcMode;// Режим моделирования/расчета (TRUE) или
                // редактирования (FALSE)
  int BlockX,BlockY;  // Координаты точки привязки блока
                      // с учетом связи с переменными
  double DoubleZoom;  // Масштаб в долях единицы

  // Данные описывающего прямоугольника блока в текущем масштабе
  BOOL RectValid;     // TRUE если по результатам рисования
                      // описывающий прямоугольник нужно изменить,
                      // FALSE в противном случае (по умолчанию)
  int Left,Top;       // Левый верхний угол прямоугольника
  int Width,Height;   // Размеры прямоугольника
  //---------

  RECT *VisibleRect;  // Видимая в окне часть рабочего поля
                      // подсистемы (только чтение)
  BOOL FullDraw;      // TRUE - необходимо нарисовать все,
                      // FALSE - только изменения с прошлого
                      // рисования
} RDS_DRAWDATA;
typedef RDS_DRAWDATA *RDS_PDRAWDATA;
```

Сейчас нам из этой структуры понадобятся только поля Left, Top, Width и Height, в которых содержатся координаты описывающего прямоугольника блока в текущем масштабе, с учетом положения полос прокрутки окна, а также связи координат блока с какими-либо переменными, если таковая имеется (в нашем блоке такой связи нет). Именно эти координаты определяют прямоугольную область в окне подсистемы, занимаемую нашим блоком, внутри нее мы и будем рисовать индикатор уровня.

Указатель на структуру RDS_DRAWDATA (предварительно приводя его к правильному типу, поскольку ExtParam имеет тип void*) мы передаем в функцию рисования SimpleLevelIndicatorDraw. Вторым параметром в эту функцию передается текущее значение входа x.

Теперь напишем функцию рисования:

```
// Функция рисования простого индикатора уровня
void SimpleLevelIndicatorDraw(RDS_PDRAWDATA draw,double val)
{ // Диапазон допустимых значений входа
  const double Min=0.0,Max=100.0;
  // Цвета индикатора
  const COLORREF empty=0xffffff,      // Верхняя часть (белый)
                 fill=0xff0000,       // Нижняя часть (синий)
                 border=0;            // Рамка вокруг (черный)
  // Вспомогательные переменные
  int height,fullheight,x1,y1,x2,y2;

  // Координаты прямоугольника внутри рамки (отступ в 1 точку)
  x1=draw->Left+1;
  y1=draw->Top+1;
  x2=draw->Left+draw->Width-1;
  y2=draw->Top+draw->Height-1;
```

```
    // Высотота блока без толщины рамки (==draw->Height-2)
    fullheight=(y2-y1);

    // Высота столбика (от нижней границы до линии раздела)
    height=(val-Min)*fullheight/(Max-Min);
    // Ограничения сверху и снизу
    if(height>fullheight)
      height=fullheight;
    else if(height<0)
      height=0;

    // Рисование рамки
    rdsXGSetPenStyle(0,PS_SOLID,1,border,R2_COPYPEN);
    rdsXGSetBrushStyle(0,RDS_GFS_EMPTY,0);
    rdsXGRectangle(draw->Left,draw->Top,
                   draw->Left+draw->Width,draw->Top+draw->Height);
    // Закраска верхней части цветом empty
    if(height!=fullheight)
      { rdsXGSetBrushStyle(0,RDS_GFS_SOLID,empty);
        rdsXGFillRect(x1,y1,x2,y2-height);
      }
    // Закраска нижней части цветом fill
    if(height!=0)
      { rdsXGSetBrushStyle(0,RDS_GFS_SOLID,fill);
        rdsXGFillRect(x1,y2-height,x2,y2);
      }
  }
  //=========================================
```

Первым параметром в функцию передается указатель на структуру `RDS_DRAWDATA`, полученный от РДС. Хотя в одном из полей этой структуры и передается контекст устройства, на котором блок должен нарисовать свое изображение, он нам не понадобится – вместо стандартных функций Windows API мы будем пользоваться сервисными функциями рисования РДС, поскольку работать с ними несколько проще. Точно так же мы поступили в примере со сложной функцией настройки на стр.129. Рисование выполняется в три приема: сначала мы рисуем черную рамку толщиной в одну точку по размеру блока. Затем, отступив одну точку от границ блока внутрь, чтобы не перекрыть рамку, закрашиваем верхнюю часть прямоугольника белым цветом, а нижнюю – синим. Координаты границы раздела мы вычисляем, зная значение входа, переданное в параметре `val`, и высоту прямоугольника блока.

В начале функции описаны константы, определяющие диапазон возможных значений входа блока (`Min` и `Max`), а также цвета рамки, верхней и нижней части индикатора (`border`, `empty` и `fill` соответственно). Все цвета, как и везде в Windows API, задаются целыми числами в формате `0x00bbggrr`, где `bb` – байт интенсивности синего канала цвета, `gg` – зеленого, а `rr` – красного (константы заданы в шестнадцатеричном виде). Мы уже решили не делать функцию настройки для этих параметров, чтобы не усложнять пример, поэтому эти цвета и объявлены как константы. При необходимости, можно хранить цвета в личной области данных или в статических переменных блока и разрешить пользователю изменять их (примеры функций настройки приведены в §2.7).

Далее вспомогательным переменным `x1` и `y1` присваиваются координаты левого верхнего угла закрашиваемой области индикатора – они отстоят от левого верхнего угла всего изображения на одну точку, поскольку по краю блока пройдет рамка толщиной в одну точку, и закрашивать эту границу не нужно. Координаты левого верхнего угла изображения блока берутся из полей `Left` и `Top` структуры `RDS_DRAWDATA`, указатель на которую передан в функцию в параметре `draw`. В этих параметрах уже учтен текущий масштаб окна

подсистемы и положение его полос прокрутки. Например, если координаты левого верхнего угла блока – (15, 40), масштаб окна установлен в 200%, горизонтальная полоса прокрутки сдвинута до упора влево, а вертикальная – до упора вверх (то есть в окне видна левая верхняя часть рабочего поля), `Left` будет равно 30, а `Top` – 80. Если при этом начать двигать горизонтальную полосу прокрутки вправо (рабочее поле начнет "смещаться" влево), `Left` начнет уменьшаться. В общем, при любых изменениях масштаба и прокрутке рабочей области точка (`Left`, `Top`) будет соответствовать левому верхнему углу прямоугольной области, занимаемой блоком в окна в данный момент.

Вспомогательным переменным `x2` и `y2` в функции присваиваются координаты правого нижнего угла закрашиваемой области – они тоже отстоят от правого нижнего угла всего изображения на одну точку. В структуре `RDS_PDRAWDATA` передаются ширина и высота изображения в текущем масштабе, поэтому для получения правого нижнего угла к левому верхнему добавляют ширину и высоту соответственно (и вычитают 1, чтобы отступить внутрь блока на одну точку, необходимую для рисования рамки). Затем, когда все четыре координаты закрашиваемой области индикатора вычислены, дополнительно вычисляется высота этой области `fullheight` – она понадобится для вычисления координаты границы раздела цветов.

Граница раздела цветов, а точнее, высота нижней закрашиваемой части `height`, вычисляется из следующих соображений: вещественное значение `val` может изменяться от `Min` до `Max`, при этом высота нижней части линейно меняется от 0 до `fullheight` соответственно. Таким образом,

```
height=(val-Min)*fullheight/(Max-Min);
```

Теперь важно ограничить значение `height` так, чтобы граница раздела всегда оставалась внутри изображения блока. Если значение `val` будет больше выбранного нами ограничения `Max`, `height` будет больше `fullheight`, что недопустимо – закрашенная часть при этом будет выходить за границы блока сверху. Поэтому, если переменная `height` превышает `fullheight`, мы ограничиваем ее значением `fullheight`. Точно так же при отрицательных значениях переменной `height` мы принудительно присваиваем ей ноль.

Теперь можно приступать к рисованию. Сначала мы рисуем рамку – для этого предварительно функцией `rdsXGSetPenStyle` устанавливается стиль линии и функцией `rdsXGSetBrushStyle` отключается заливка (эти функции уже знакомы нам по примеру функции настройки блока-генератора на стр.129). Далее функцией `rdsXGRectangle` рисуется прямоугольная рамка. Функция принимает четыре целых параметра – левый верхний и правый нижний углы прямоугольника – и рисует прямоугольник с использованием текущего стиля линии и заливки. В данном случае мы установили черную линию толщиной в одну точку и отключили заливку, поэтому будет нарисована пустая внутри прямоугольная рамка вокруг блока.

Осталось закрасить верхнюю часть прямоугольника цветом `empty` (белым), а нижнюю – цветом `fill` (синим). Отступ границы раздела от нижней части блока у нас уже вычислен и находится в переменной `height`, поэтому верхняя закрашенная часть будет располагаться между вертикальными координатами `y1` и `y2-height` (нужно всегда помнить, что в окнах вертикальная координатная ось направлена сверху вниз, а не снизу вверх). Закрашивать верхнюю часть мы будем только тогда, когда `height` не равно `fullheight`, так как при их равенстве весь прямоугольник блока нужно закрасить синим (столбик индикатора имеет максимальную высоту). Для закраски используется уже знакомая нам функция заливки прямоугольника `rdsXGFillRect` – она не использует стиль линии, поэтому перед ее вызовом мы устанавливаем только стиль заливки (сплошная, цвет `empty`).

Точно так же мы закрашиваем нижнюю часть прямоугольника (между `y2-height` и `y2`) цветом `fill` в том случае, если `height` не равно нулю (столбик имеет ненулевую высоту).

Для того, чтобы проверить эту модель, следует подключить ее к блоку, в параметрах которого на вкладке “Внешний вид” задано “определяется функцией DLL”. Также желательно разрешить масштабирование блока на той же вкладке, чтобы пользователь мог менять его размер мышью, перетаскивая один из восьми прямоугольных маркеров (см. рис. 58). Ко входу блока x можно подключить поле ввода, при изменении его значения от 0 до 100 в режиме расчета высота столбика в блоке будет изменяться от нулевой до максимальной. При подаче на вход отрицательных значений весь блок будет залит белым, при подаче значений, больших 100 – синим.

Можно заметить, что модель индикатора уровня получилась довольно простой – в том числе, и за счет использования графических сервисных функций РДС. В большинстве случаев удается обойтись ими и не использовать функции Windows API для рисования изображений блоков – это делает модели несколько более наглядными и гарантирует их совместимость с разными версиями РДС.

На самом деле, точно такой же индикатор можно было бы сделать и не прибегая к программному рисованию внешнего вида блоков – в редакторе векторной картинки можно связать вертикальный масштаб прямоугольника с какой-либо переменной блока и, меняя эту переменную, изменять высоту прямоугольника. Останется только ввести в модель блока приведение входа к диапазону 0...1 и запись получившегося значения в переменную, связанную с картинкой, и индикатор уровня готов. Тем не менее, в некоторых случая без программного рисования не обойтись.

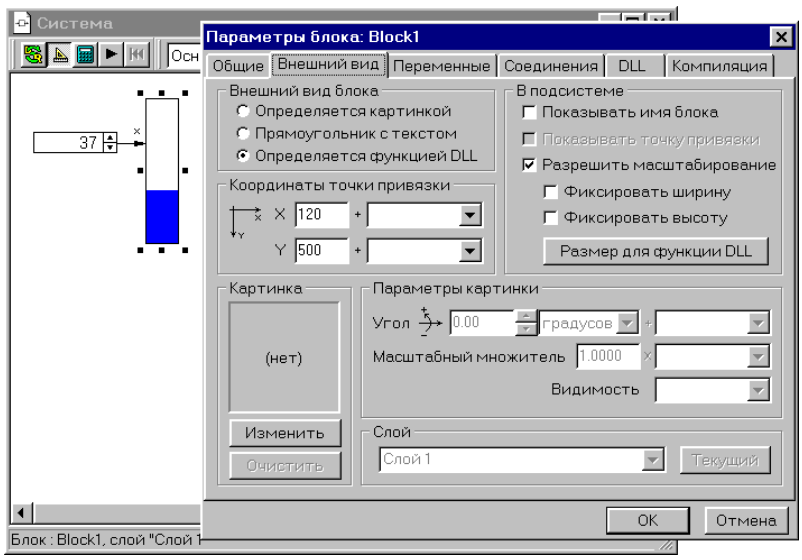


Рис. 58. Индикатор уровня и параметры его внешнего вида

Создадим блок, который будет строить график зависимости значения входа от времени. Блок будет довольно сложным – мы будем отображать не только сам график (для чего блоку потребуются динамически отводимые массивы для хранения запомненных отсчетов), но и координатные оси с разметкой и числами на них. Кроме того, в этом блоке мы предусмотрим настройку диапазонов обеих осей, цвета и толщины линии графика, цвета осей и фона, шрифта чисел на осях и т.д. Такой блок можно создать только с использованием программного рисования внешнего вида.

Чтобы совсем уж не усложнять пример, мы не будем делать в блоке автоматическую настройку диапазонов горизонтальной и вертикальной осей, автоматическое увеличение массива, в котором хранятся отсчеты графика, при его переполнении и т.д. Разумеется, серьезный, удобный в использовании блок-график должен иметь эти возможности, но цель этого примера – демонстрация возможностей программного рисования, поэтому диапазоны осей будут задаваться пользователем в настройках блока и останутся неизменными в

процессе расчета. Вместо размера массива мы дадим пользователю задать шаг записи графика, то есть интервал времени между записью в массив отсчетов. Размер массива при этом можно вычислить автоматически: максимально возможное число отсчетов в массиве будет равно диапазону горизонтальной оси, деленному на шаг записи графика. Значение времени наш блок будет брать из стандартной динамической переменной “DynTime” (как и многие другие уже рассмотренные блоки, например, блок-генератор со стр. 124), что сделает его совместимым с блоками, входящими в стандартную библиотеку РДС.

Для вывода графика будем рисовать внутри прямоугольника блока прямоугольник меньшего размера, на который будет наложена пунктирная сетка разметки (рис. 59). Слева и снизу от этого прямоугольника будем выводить числа, соответствующие делениям горизонтальной и вертикальной осей. Размер внутреннего прямоугольника модель блока будет автоматически подбирать таким образом, чтобы числа на осях, выведенные выбранным пользователем шрифтом, уместились внутрь прямоугольника блока. Из рисунка видно, что расстояние между левой границей внутреннего прямоугольника и левой границей внешнего прямоугольника блока (на рисунке это расстояние обозначено как Gr_x1) должно равняться ширине самого большого числа на вертикальной оси (на рисунке – “400”), иначе числа вертикальной оси не уместятся внутрь внешнего прямоугольника. Gr_x1 также не должно быть меньше половины ширины минимального числа горизонтальной оси, иначе первое число этой оси также не уместится во внешний прямоугольник. Правая граница внутреннего прямоугольника Gr_x2 должна вычисляться так, чтобы во внешний прямоугольник уместилось максимальное число горизонтальной оси (на рисунке – “40”), таким образом, расстояние между правыми границами внешнего и внутреннего прямоугольников должно равняться половине ширины максимального числа горизонтальной оси.

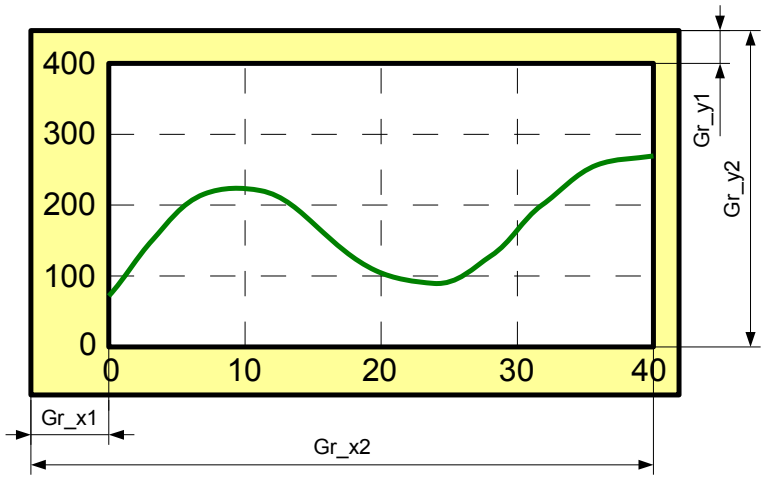


Рис. 59. Предполагаемый внешний вид блока-графика

Расстояние между верхними границами прямоугольников (Gr_y1) должно равняться половине высоты числа, чтобы во внешний прямоугольник уместилось самое верхнее число вертикальной оси. Расстояние между нижними границами должно равняться полной высоте числа, чтобы уместились числа горизонтальной оси. Интервал следования чисел на горизонтальной и вертикальной осях, а также дробная часть этих чисел будут задаваться пользователем в настройках блока.

Теперь, когда мы представляем себе, что и как должна рисовать модель блока, можно приступать к ее написанию. Начнем с личной обрасти данных блока, в которой мы будем хранить все задаваемые пользователем цвета, диапазоны и другие настроечные параметры. Также в ней будут находиться указатели на динамически отводимые массивы отсчетов

графика и указатель на структуру подписки, с помощью которой блок будет обращаться к динамической переменной времени “DynTime” (см. §2.6.2). Оформим личную область данных как класс C++:

```
//=======================================
// Простой график – личная область данных
//=======================================
class TSimplePlotData
{ private:
    // Настроечные параметры графика (цвета, шаг и т.п.)
    double TimeStep;             // Шаг записи отсчетов
    RDS_SERVFONTPARAMS Font;     // Шрифт чисел на осях
    COLORREF BorderColor;        // Цвет рамки вокруг блока
    COLORREF FillColor;          // Цвет фона блока
    COLORREF PlotBorderColor;    // Цвет рамки поля графика
    COLORREF PlotFillColor;      // Цвет фона поля графика
    COLORREF LineColor;          // Цвет лини графика
    int LineWidth;               // Толщина линии графика

    // Ось X
    double Xmin,Xmax;       // Диапазон
    double XGridStep;       // Шаг чисел на осях
    int XNumDecimal;        // Дробная часть чисел на осях

    // Ось Y
    double Ymin,Ymax;       // Диапазон
    double YGridStep;       // Шаг чисел на осях
    int YNumDecimal;        // Дробная часть чисел на осях

    // Массивы для хранения отсчетов графика
    double *Times;    // Массив отсчетов времени
    double *Values;   // Массив значений
    int Count;        // Размер массивов
    int NextIndex;    // Индекс для записи следующего значения
    double NextTime;  // Время записи следующего значения

    RDS_PDYNVARLINK Time;  // Связь с динамической переменной
                           // времени (“DynTime”)
  public:
    // Функция отведения массивов отсчетов
    void AllocateArrays(void);
    // Функция освобождения массивов отсчетов
    void ClearArrays(void);
    // Добавление очередной точки в массив отсчетов графика
    void AddPoint(double v);

    int Setup(void);  // Функция настройки параметров
    void SaveText(void);   // Функция сохранения параметров
    void LoadText(char *text);  // Функция загрузки параметров
    void Draw(RDS_PDRAWDATA DrawData);     // Функция рисования

    TSimplePlotData(void); // Конструктор класса
    ~TSimplePlotData();    // Деструктор класса
};
//=======================================
```

Все параметры блока описаны в закрытой области (`private`) – обращение к ним будет вестись только из функций-членов класса. В начале области располагаются настроечные

параметры: цвета, шаг записи, диапазоны осей и т.д. В тексте описания класса содержатся комментарии, поясняющие назначение каждого параметра, поэтому нет смысла подробно описывать каждый из них. Остановимся только на параметрах шрифта чисел на осях, поскольку описания шрифтов ранее в примерах моделей блоков не встречалось. Описание выглядит так:

```
RDS_SERVFONTPARAMS Font;
```

RDS_SERVFONTPARAMS – это структура, используемая некоторыми сервисными функциями РДС при получении или установке параметров шрифта: начертания, размера, жирности и т.п. Она описана в “RdsDef.h” следующим образом:

```
#define RDS_SERVFONTPARAMSNAMESIZE    256   // Размер массива
                                            // имени шрифта
typedef struct
{ DWORD servSize;     // Размер структуры в байтах
  char Name[RDS_SERVFONTPARAMSNAMESIZE];   // Имя шрифта
  int CharSet;        // Набор символов
  int Height;         // Высота шрифта в точках экрана
  int Size;           // Кегль шрифта
  BOOL SizePriority;  // При установке шрифта использовать
                      // кегль (TRUE), или высоту в точках (FALSE)
  COLORREF Color;     // Цвет шрифта
  BOOL Bold;          // Жирный
  BOOL Italic;        // Курсив
  BOOL Underline;     // Подчеркнутый
  BOOL StrikeOut;     // Зачеркнутый
} RDS_SERVFONTPARAMS;
```

В этой структуре содержатся основные параметры шрифта, которые могут потребоваться для его установки. Кроме того, как и в большинстве структур, с которыми работают сервисные функции РДС, в ней есть дополнительное служебное поле servSize, которому необходимо присвоить размер этой структуры в байтах, то есть sizeof(RDS_SERVFONTPARAMS). Это поле используется для контроля правильности переданной в какую-либо функцию структуры: если его значение, то есть размер структуры, будет неверным, значит, модель использует неправильные описания, и РДС не будет с ней работать.

Для задания высоты шрифта в структуре предусмотрено два поля: Height, задающее высоту шрифта в точках экрана, и Size, задающее высоту в типографских единицах (кегль), более привычных для пользователя. Если структура используется для установки параметров шрифта (например, при рисовании), значение берется только из одного из этих полей: при SizePriority равном TRUE – из поля Size, при SizePriority равном FALSE – из поля Height. Если же в структуру записываются параметры какого-либо шрифта при вызове одной из сервисных функций получения параметров, эта функция заполняет оба этих поля независимо от значения SizePriority. В нашей модели эта структура будет использоваться как при установке параметров шрифта (при рисовании осей), так и при их получении (из диалога выбора шрифта при настройке параметров пользователем, а также при загрузке параметров блока).

После настроечных параметров блока в классе описываются массивы отсчетов графика. Отсчеты хранятся в двух вещественных массивах, каждый из которых содержит Count элементов: в массиве Times хранятся значения времени, а в массиве Values – значения входа блока, соответствующие этим моментам. Таким образом, целому индексу i, находящемуся в диапазоне [0...Count-1], соответствует точка графика (Times[i],Values[i]).

В процессе работы системы массивы Times и Values будут постепенно заполняться отсчетами. Заполнение массивов управляется полем NextIndex, в котором находится индекс первой свободной ячейки. Таким образом, в любой момент времени элементы

массивов [0...NextIndex-1] заполнены, а [NextIndex...Count-1] – свободны. Значение времени, по достижении которого в массив будет записан очередной отсчет, задается полем NextTime.

В общих чертах, запись отсчетов в массивы для последующего построения графика будет работать следующим образом:

- Исходно массивы Times и Values не отведены – оба поля содержат значения NULL.
- При первом запуске расчета вычисляется требуемый размер массивов (диапазон оси времени, деленный на шаг записи) и записывается в поле Count. Отводится место под массивы Times и Values – оба будут содержать по Count вещественных чисел двойной точности. Полю NextTime присваивается значение начала оси времени (Xmin), полю NextIndex – значение 0. Теперь блок ждет наступления времени NextTime, чтобы записать первый отсчет в массивы.
- Как только текущее время, получаемое из динамической переменной “DynTime”, станет большим или равным NextTime, значение входа блока запишется в Values[NextIndex], а значение времени – в Times[NextIndex]. После этого NextIndex увеличится на 1, а к NextTime будет прибавлен шаг записи TimeStep, и блок снова будет ждать наступления времени NextTime для записи очередного отсчета.
- Последний пункт будет повторяться до тех пор, пока NextIndex не станет равным Count, что укажет на заполнение всего массива. После этого новые отсчеты никуда писаться не будут. В настоящем блоке-графике в этот момент следовало бы увеличить размер массива или сдвинуть диапазон, но, для упрощения примера, мы решили этого не делать.

Разобравшись с принципом записи отсчетов в график и необходимыми для этого полями, вернемся к классу личной области данных. В открытой области (public) находятся описания функций-членов, которые будут вызываться из модели блока. Это пара функций для отведения и освобождения памяти под описанные выше массивы отсчетов, функция вызова окна настроек блока, в котором пользователь сможет вводить значения параметров, функции сохранения и загрузки параметров блока (раз мы сделали функцию настройки, необходимо записывать введенные пользователем значения при сохранении схемы и считывать их при ее загрузке), функция рисования внешнего вида блока (ради которой и рассматривается этот пример), а также функция записи очередного отсчета графика в массив, реализующая описанный выше алгоритм работы блока. И, разумеется, у класса будет конструктор, в котором будут устанавливаться начальные значения параметров и запрашиваться подписка на динамическую переменную “DynTime”, и деструктор, в котором будет освобождаться память, занятая массивами отсчетов, и прекращаться подписка на “DynTime”. Далее мы по очереди рассмотрим все эти функции, но сначала запишем собственно функцию модели, которая будет создавать и уничтожать объект описанного класса и вызывать его функции-члены.

Для работы блоку будут нужны будут следующие переменные:

Смещение	*Имя*	*Тип*	*Размер*	*Пуск*	*Вход/выход*
0	Start	Сигнал	1	✓	Вход
1	Ready	Сигнал	1		Выход
2	x	double	8		Вход

Заметим, что для входа блока x, значение которого будет строиться на графике, не задан флаг “пуск”, то есть при поступлении на вход блока нового значения модель не будет запускаться автоматически. Согласно описанному выше алгоритму работы, блок записывает новый отсчет в массивы при изменении времени. Время блок получает через стандартную динамическую переменную, поэтому в модели необходима реакция на ее изменение

(RDS_BFM_DYNVARCHANGE), а на такт расчета (RDS_BFM_MODEL) реакция не нужна. таким образом, модели блока не нужен ни запуск при срабатывании связи, подключенной ко входу x, ни запуск каждый такт (в параметрах блока следует выбрать “запуск по сигналу”).

Модель блока выглядит следующим образом:

```
// Простой график
extern "C" __declspec(dllexport) int RDSCALL SimplePlot(
                int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define x (*((double *)(pStart+2)))
  // Указатель на личную область, приведенный к правильному типу
  TSimplePlotData *data=(TSimplePlotData*)(BlockData->BlockData);

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        BlockData->BlockData=new TSimplePlotData();
        break;

      // Очистка данных блока
      case RDS_BFM_CLEANUP:
        delete data;
        break;

      // Проверка типов статических переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSD}")==0)
          return RDS_BFR_DONE;
        return RDS_BFR_BADVARSMSG;

      // Функция настройки параметров
      case RDS_BFM_SETUP:
        return data->Setup();

      // Загрузка параметров в текстовом формате
      case RDS_BFM_LOADTXT:
        data->LoadText((char*)ExtParam);
        break;

      // Созранение параметров в текстовом формате
      case RDS_BFM_SAVETXT:
        data->SaveText();
        break;

      // Рисование внешнего вида блока
      case RDS_BFM_DRAW:
        data->Draw((RDS_PDRAWDATA)ExtParam);
        break;

      // Запуск расчета
      case RDS_BFM_STARTCALC:
```

```
            if(((RDS_PSTARTSTOPDATA)ExtParam)->FirstStart)
              data->AllocateArrays(); // Первый запуск
            break;

          // Сброс расчета
          case RDS_BFM_RESETCALC:
            data->ClearArrays();
            break;

          // Реакция на изменение динамической переменной
          case RDS_BFM_DYNVARCHANGE:
            data->AddPoint(x);
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений для переменных
    #undef x
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Большинство реакций в этой модели уже неоднократно описывалось: при инициализации (RDS_BFM_INIT) модель создает объект класса TSimplePlotData, при очистке (RDS_BFM_CLEANUP) – уничтожает его, при проверке типа статических переменных (RDS_BFM_VARCHECK) – сравнивает переданную строку с правильной строкой типа. В модели также присутствуют реакции на сохранение данных блока в текстовом формате (RDS_BFM_SAVETXT), загрузку этих данных (RDS_BFM_LOADTXT) и вызов пользователем окна настройки (RDS_BFM_SETUP). Все они вызывают соответствующие функции-члены класса, которые будут рассмотрены позднее. Для рисования блока в модель, как и в предыдущем примере, введена реакция RDS_BFM_DRAW, только теперь функция рисования Draw является членом класса TSimplePlotData. Следует помнить, что в режиме RDS_BFM_DRAW вызываются только модели тех блоков, для которых в параметрах задано рисование с помощью функции DLL (см. рис. 58). И, наконец, для записи данных в массивы, из которых строится график, и для обеспечения работы этих массивов в модель введены реакции на запуск расчета RDS_BFM_STARTCALC, сброс расчета RDS_BFM_RESETCALC и изменение динамической переменной RDS_BFM_DYNVARCHANGE (в данном случае у блока есть единственная динамическая переменная – “DynTime”, то есть время). Остановимся на этих реакциях подробнее.

Согласно алгоритму записи отсчетов в массивы, описанному немного выше, память под массивы Times и Values должна отводиться при запуске расчета. Однако, не следует делать это при каждом запуске. Представим себе, что пользователь запустил расчет, после чего остановил его в тот момент, когда была построена только половина графика. Изменив какие-либо переменные системы (введя новые значения в поля ввода, передвинув рукоятки и т.п.) он решил продолжить работу и, не сбрасывая расчет, снова запустил его. Естественно, он будет ожидать, что график будет достроен до конца, и его первая половина, построенная до промежуточной остановки расчета, не сотрется. Если же отводить память под массивы при каждом запуске расчета, запомненные до остановки данные будут потеряны. Чтобы избежать этого, память под массивы нужно отводить только при первом после сброса (или после загрузки схемы) запуске расчета.

Для того, чтобы отличить первый запуск расчета от повторного, можно обратиться к полю FirstStart структуры RDS_STARTSTOPDATA, указатель на которую передается в функцию модели в параметре ExtParam при вызове ее в режиме RDS_BFM_STARTCALC.

Если это поле равно TRUE, значит, расчет запущен в первый раз, и нужно вызвать функцию отведения массивов AllocateArrays (при этом ExtParam, как всегда, приходится предварительно приводить к правильному типу, в данном случае, к RDS_PSTARTSTOPDATA, то есть к указателю на RDS_STARTSTOPDATA).

Освобождение памяти (то есть стирание массивов) логично выполнять в момент сброса расчета. Это будет соответствовать ожиданиям пользователя – после сброса поле графика будет очищаться. Поэтому вызов функции освобождения массивов ClearArrays производится в реакции модели на событие RDS_BFM_RESETCALC. Необходимо будет вставить вызов этой функции еще и в деструктор класса, поскольку пользователь может стереть блок или закрыть РДС не сбрасывая расчет, и в этом случае отведенную память также нужно освободить.

Наконец, при любом изменении единственной динамической переменной блока (событие RDS_BFM_DYNVARCHANGE) будет вызываться функция AddPoint, в которую передается текущее значение входа блока x. Внутри этой функции, при необходимости, пара “время-значение” будет добавлена в массивы отсчетов графика.

Рассмотрение функций-членов класса TSimplePlotData начнем с конструктора класса. Он выглядит следующим образом:

```
// Конструктор класса личной области данных графика
TSimplePlotData::TSimplePlotData(void)
{ // Присвоение начальных значений параметрам
  TimeStep=0.1;             // Шаг записи
  BorderColor=0;            // Цвет рамки вокруг блока
  FillColor=0xffffff;       // Цвет фона блока
  PlotBorderColor=0;        // Цвет рамки окна графика и сетки
  PlotFillColor=0xffffff;   // Цвет окна графика
  LineColor=0;              // Цвет линии графика
  LineWidth=1;              // Толщина линии графика

  // Параметры шрифта
  Font.servSize=sizeof(Font);
  strcpy(Font.Name,"Arial");
  Font.SizePriority=FALSE;
  Font.Height=15;
  Font.Color=0;
  Font.Bold=Font.Italic=Font.Underline=Font.StrikeOut=FALSE;
  Font.CharSet=DEFAULT_CHARSET;

  // Диапазоны осей, шаг сетки, число десятичных знаков
  // в числах на осях
  Xmin=0.0; Xmax=10.0;
  XGridStep=5.0;
  XNumDecimal=0;
  Ymin=-1.0; Ymax=1.0;
  YGridStep=0.5;
  YNumDecimal=1;

  // Обнуление указателей на массивы и их размера
  // (массивы еще не отведены)
  Times=Values=NULL;
  Count=NextIndex=0;
  NextTime=Xmin;

  // Подписка на динамическую переменную времени
  Time=rdsSubscribeToDynamicVar(
    RDS_DVPARENT,     // В родительской подсистеме
```

```
    "DynTime",        // Имя переменной
    "D",              // Тип переменной (double)
    TRUE);            // Искать по иерархии
}
//=======================================
```

В конструкторе присваиваются начальные значения всем полям класса, описывающим внешний вид графика, обнуляются указатели массивов (массивы будут отведены позже) и осуществляется подписка на стандартную переменную времени “DynTime” – она понадобится блоку для записи моментов времени в массив `Times`.

В деструкторе класса необходимо освободить память, занятую массивами (для этого будет использоваться функция `ClearArrays`, которую мы напишем позже) и прекратить подписку на переменную “DynTime”:

```
// Деструктор класса
TSimplePlotData::~TSimplePlotData()
{ rdsUnsubscribeFromDynamicVar(Time); // Прекратить подписку
  ClearArrays();                      // Освободить массивы
}
//=======================================
```

Следующей рассмотрим функцию рисования `Draw` – в конце концов, этот пример посвящен программному рисованию, и все остальные функции играют вспомогательные роли. Она выглядит следующим образом:

```
// Рисование внешнего вида блока
void TSimplePlotData::Draw(RDS_PDRAWDATA DrawData)
{ // Вспомогательные переменные
  int Gr_x1,Gr_x2,Gr_y1,Gr_y2;
  int x1,y1,x2,y2,textheight,w1,w2;
  char buf[80];

  // Рамка графика
  rdsXGSetPenStyle(0,PS_SOLID,1,BorderColor,R2_COPYPEN);
  rdsXGSetBrushStyle(0,RDS_GFS_SOLID,FillColor);
  rdsXGRectangle(DrawData->Left,DrawData->Top,
                 DrawData->Left+DrawData->Width,
                 DrawData->Top+DrawData->Height);

  // Необходимо вычислить координаты поля графика относительно
  // верхнего левого угла блока

  // Установка параметров шрифта с учетом масштаба
  rdsXGSetFontByParStr(&Font,DrawData->DoubleZoom);
  // Зазор сверху - половина высоты цифры + 1 точка
  rdsXGGetTextSize("0",NULL,&textheight);
  Gr_y1=textheight/2+1;
  // Зазор снизу - полная высота цифры + 1 точка
  Gr_y2=DrawData->Height-textheight-1;
  // Зазор слева - ширина самого длинного числа вертикальной
  // оси или половина ширины Xmin
  sprintf(buf," %.*lf ",YNumDecimal,Ymin);
  rdsXGGetTextSize(buf,&w1,NULL);     // Ширина Ymin
  sprintf(buf," %.*lf ",YNumDecimal,Ymax);
  rdsXGGetTextSize(buf,&w2,NULL);     // Ширина Ymax
  if(w2>w1) w1=w2;
  sprintf(buf," %.*lf ",XNumDecimal,Xmin);
  rdsXGGetTextSize(buf,&w2,NULL);     // Ширина Xmin
  w2/=2;
  if(w2>w1) w1=w2;
```

```
Gr_x1=w1;
// Зазор справа - половина ширины Xmax
sprintf(buf," %.*lf ",XNumDecimal,Xmax);
rdsXGGetTextSize(buf,&w2,NULL);     // Ширина Xmax
w2/=2;
Gr_x2=DrawData->Width-w2;

// Абсолютные (на рабочем поле) координаты поля графика
x1=DrawData->Left+Gr_x1;
x2=DrawData->Left+Gr_x2;
y1=DrawData->Top+Gr_y1;
y2=DrawData->Top+Gr_y2;

if(x1>=x2 || y1>=y2) // Негде рисовать
  return;

// Прямоугольник поля графика
rdsXGSetPenStyle(0,PS_SOLID,1,PlotBorderColor,R2_COPYPEN);
rdsXGSetBrushStyle(0,RDS_GFS_SOLID,PlotFillColor);
rdsXGRectangle(x1,y1,x2,y2);

// Установка пунктирного стиля линии
rdsXGSetPenStyle(0,PS_DOT,1,PlotBorderColor,R2_COPYPEN);
rdsXGSetBrushStyle(0,RDS_GFS_EMPTY,0);   // Без заливки

// Горизонтальная ось с сеткой
for(double x=Xmin;x<=Xmax+XGridStep*0.5;x+=XGridStep)
  { // ix - координата линии на рабочем поле
    int ix=x1+(x-Xmin)*(x2-x1)/(Xmax-Xmin);
    if(ix>x1 && ix<x2)   // Чертим вертикальную линию
      { rdsXGMoveTo(ix,y1);
        rdsXGLineTo(ix,y2);
      }
    // Вывод числа на оси под полем
    sprintf(buf,"%.*lf",XNumDecimal,x);
    rdsXGGetTextSize(buf,&w1,NULL);
    rdsXGTextOut(ix-w1/2,y2,buf);
  }
// Вертикальная ось с сеткой
for(double y=Ymin;y<=Ymax+YGridStep*0.5;y+=YGridStep)
  { // iy - координата линии на рабочем поле
    int iy=y2-(y-Ymin)*(y2-y1)/(Ymax-Ymin);
    if(iy>y1 && iy<y2)   // Чертим горизонтальную линию
      { rdsXGMoveTo(x1,iy);
        rdsXGLineTo(x2,iy);
      }
    // Вывод числа на оси слева от поля
    sprintf(buf,"%.*lf ",YNumDecimal,y);
    rdsXGGetTextSize(buf,&w1,&textheight);
    rdsXGTextOut(x1-w1-2,iy-textheight/2,buf);
  }

// Если массивы не пустые - рисовать график
if(Count)
  { RECT r;
    // Установить область отсечения рисования по полю графика
    r.left=x1+1;
    r.top=y1+1;
```

```
        r.right=x2-1;
        r.bottom=y2-1;
        rdsXGSetClipRect(&r);

        // Установить сплошной стиль линии, заданный для
        // графика цвет и толщину линии с учетом масштаба
        rdsXGSetPenStyle(0,PS_SOLID,
             LineWidth*DrawData->DoubleZoom,
             LineColor,R2_COPYPEN);

        // Строим ломанную линию по отсчетам из массивов
        for(int i=0;i<NextIndex;i++)
          { // Преобразуем вещественные отсчеты в целочисленные
            // координаты на рабочем поле
            int ix=x1+(Times[i]-Xmin)*(x2-x1)/(Xmax-Xmin),
                iy=y2-(Values[i]-Ymin)*(y2-y1)/(Ymax-Ymin);
            if(i) // Не первая точка - строим линию от предыдущей
              rdsXGLineTo(ix,iy);
            else // Первая точка графика - делаем ее текущей
              rdsXGMoveTo(ix,iy);
          }

        // Отмена отсечения
        rdsXGSetClipRect(NULL);
      }
  }
  //=========================================
```

Примерный вид изображения, которое строит эта функция, представлен на рис. 59, там же объясняется, как вычислить координаты поля графика таким образом, чтобы числа на осях уместились между этим полем и внешними границами блока. Этим координатам соответствуют переменные функции `Gr_x1`, `Gr_x2`, `Gr_y1` и `Gr_y2`. В функции объявлены и другие локальные переменные, которые потребуются в процессе работы.

Рисование внешнего вида блока начинается с рамки. Функцией `rdsXGSetPenStyle` устанавливается стиль линии (сплошная, толщиной в одну точку, цвет – поле класса `BorderColor`), функцией `rdsXGSetBrushStyle` – тип заливки фигур (сплошная, цвет – поле класса `FillColor`), после чего функцией `rdsXGRectangle` рисуется прямоугольник во весь размер блока. В дальнейшем на этот прямоугольник будет наложено поле графика - другой прямоугольник меньшего размера, на котором будет рисоваться сетка и линия графика. Слева и снизу от этого меньшего прямоугольника будут выведены числовые метки осей.

Перед рисованием поля графика необходимо вычислить его координаты внутри большого прямоугольника. Для этого необходимо знать линейные размеры чисел на осях, выраженные в точках экрана. Эти размеры зависят от диапазонов осей (число “100” займет больше места, чем число “10”), числа знаков после десятичной точки (“10.0” длиннее “10”) и параметров шрифта, которым изображаются числа на осях. Для определения размеров чисел используется сервисная функция РДС `rdsXGGetTextSize`. В первом параметре этой функции передается указатель на произвольную строку текста (типа `char*`), а во втором и третьем – указатели на целые переменные, в которые функция запишет ширину и высоту прямоугольной области экрана, которую займет переданная строка, если ее вывести текущим шрифтом. Таким образом, перед вызовом этой функции нужно, во-первых, установить параметры шрифта согласно полю класса `Font`, и, во-вторых, преобразовать число, размеры которого мы хотим получить, в строку.

Для установки параметров шрифта по структуре `Font` используется сервисная функция РДС `rdsXGSetFontByParStr`, в которую передаются два параметра: указатель

на структуру описания шрифта (&Font) и масштабный коэффициент (DrawData->DoubleZoom), на который умножается размер шрифта при установке. Таким образом, размер шрифта, которым будут выводиться числа на осях графика, будет зависеть от выбранного пользователем масштаба схемы. Это логично, поскольку в противном случае на мелких масштабах метки осей заняли бы большую часть площади блока, и на сам график места бы не осталось. Шрифт устанавливается в функции один раз – все числа на осях выводятся одним и тем же шрифтом, поэтому изменять его не придется.

Далее, в соответствие с описанными выше вычислениями (см. стр. 191), определяются размеры интересующих нас чисел, и по этим размерам вычисляются относительные координаты поля графика Gr_x1, Gr_x2, Gr_y1 и Gr_y2. Каждое число предварительно преобразуется в строку при помощи функции sprintf из стандартной библиотеки C (для того, чтобы можно было использовать эту функцию, в исходный текст программы должен быть включен файл заголовков “stdio.h”). Описание этой функции есть в каждом руководстве по языку C. Следует обратить внимание на то, что в строке формата функции sprintf “ %.*lf ” не указано число знаков числа после десятичной точки – вместо него стоит символ “*” (“звездочка”). Этот символ указывает на то, что число знаков необходимо взять из следующего аргумента функции. То есть, вызов функции

```
sprintf(buf," %.*lf ",YNumDecimal,Ymin);
```

сформирует в массиве buf символьное представление вещественного числа Ymin, при этом в этом представлении будет YNumDecimal знаков после десятичной точки. Кроме того, поскольку строка начинается и заканчивается пробелом, сформированное число также будет окружено пробелами. Эти пробелы нужны для того, чтобы между выводимыми числами и рамкой графика и рабочего поля оставался небольшой зазор. Массив buf размером в 80 символов, в котором формируются все строки, объявлен в начале функции, его с большим запасом хватит для представления любого вещественного числа двойной точности.

Для преобразованных в строки чисел вызывается функция rdsXGGetTextSize, и полученные с ее помощью размеры прямоугольной области используются для вычисления координат поля графика. Эти вычисления занимают довольно большую часть функции. Их результатом будут координаты левого верхнего (Gr_x1, Gr_y1) и правого нижнего (Gr_x2, Gr_y2) углов поля графика относительно левого верхнего угла всей прямоугольной области, занятой блоком. Затем эти относительные координаты переводятся в абсолютные координаты на рабочем поле подсистемы (x1, y1) и (x2, y2). Если окажется, что x2 меньше x1 или y2 меньше y1, значит, поле графика не умещается в текущие размеры блока. Такое может произойти, если выбрать слишком большой размер шрифта для чисел на осях и слишком маленький размер самого блока. Например, очевидно, что если высота шрифта чисел больше высоты блока, то график просто негде рисовать. В этом случае функция завершается, не нарисовав ничего, кроме рамки графика. В принципе, вместо завершения функции можно было бы все равно нарисовать график, не выводя числа на осях, но мы не будем этого делать, чтобы не усложнять пример дополнительными условными операторами.

После того, как координаты поля графика вычислены, и мы убедились, что это поле вместе с числами осей умещается в прямоугольник блока, можно приступать к рисованию поля и его оформления. Сначала рисуется прямоугольник поля (x1, y1) – (x2, y2), залитый цветом PlotFillColor со сплошной рамкой цвета PlotBorderColor. Затем устанавливается пунктирный стиль линии и отключается заливка, после чего в двух циклах рисуется сначала горизонтальная сетка (вертикальные пунктирные линии с шагом XGridStep и числа под полем графика рядом с каждой из этих линий), а затем – вертикальная (горизонтальные пунктирные линии с шагом YGridStep и числа слева от поля графика рядом с ними). Подробно рассмотрим цикл рисования горизонтальной сетки – цикл вертикальной сетки будет аналогичен ему.

Вещественная переменная цикла x изменяется от начала диапазона горизонтальной оси Xmin до конца диапазона Xmax с шагом сетки XGridStep. Можно заметить, что проверкой выполнения цикла является не выражение x<=Xmax, как можно было бы ожидать, а x<=Xmax+XGridStep*0.5. То есть, к концу диапазона добавлено значение, заведомо меньшее шага изменения x. Это сделано из-за того, что x – вещественная переменная, а точное сравнение вещественных чисел крайне нежелательно. Допустим, мы хотим рисовать график с горизонтальной осью от 0 до 10 и шагом 2.5, таким образом, на горизонтальной оси должно быть пять меток: 0.0, 2.5, 5.0, 7.5 и 10.0. Однако, когда в процессе рисования оси мы прибавим шаг 2.5 к метке 7.5, чтобы получить последнюю метку, из-за погрешностей вычисления мы можем получить не 10.0, а 10.00...001. В результате последнее число на оси выведено не будет, поскольку оно окажется больше Xmax, и цикл завершится слишком рано. Чтобы избежать этого, нужно добавить к верхней границе цикла число, заведомо большее, чем возможная погрешность, но меньшее шага цикла. Половина шага цикла, в данном случае, вполне подходит.

Внутри цикла вещественное значение x, соответствующее очередной метке на оси, преобразуется в целую координату этой метки на рабочем поле ix по формуле:

```
ix=x1+(x-Xmin)*(x2-x1)/(Xmax-Xmin);
```

Это стандартная формула преобразования диапазона: разница между вещественным значением и началом его диапазона (x-Xmin) делится на весь диапазон изменения (Xmax-Xmin) и умножается на новый диапазон (x2-x1), после чего к получившемуся значению прибавляется начало нового диапазона x1. Таким образом, значению x=Xmin будет соответствовать ix=x1, значению x=Xmax будет соответствовать ix=x2, а между ними ix будет изменяться пропорционально изменению x.

После вычисления целой координаты ix ее значение сравнивается с допустимым диапазоном x1...x2, и, если координата попадает в этот диапазон, рисуется вертикальная пунктирная линия из конца в конец поля графика. Проверка на диапазон нужна из-за возможной погрешности вычисления, описанной выше: к концу диапазона эта погрешность может накопиться, и последняя вертикальная линия может оказаться на одну точку экрана правее границы поля графика, что будет выглядеть не очень хорошо. Для рисования линии используется пара сервисных функций rdsXGMoveTo – rdsXGLineTo, которые уже рассматривались ранее в примере на стр. 129.

Независимо от того, попала ли координата ix в допустимый диапазон, ниже поля графика выводится значение x. Оно преобразуется в строку в массиве buf при помощи функции sprintf, а затем функция rdsXGGetTextSize записывает ширину получившейся строки, выведенной текущим шрифтом, в переменную w1. Ширина строки нужна для того, чтобы выровнять выводимое число по горизонтали так, чтобы его середина пришлась на нарисованную пунктирную линию, то есть на координату ix. Вывод строки осуществляется сервисной функцией РДС rdsXGTextOut, в первых двух параметрах которой указываются координаты верхнего левого угла выводимой строки, а в третьем – сама строка. Таким образом, если нужно вывести строку, имеющую на экране ширину w1 точек, так, чтобы ее центр пришелся на координату ix, и она размещалась ниже нижней границы поля графика y2, координаты левого верхнего угла этой строки должны быть (ix-w1/2, y2). Именно эти значения передаются в функцию rdsXGTextOut.

Мы не будем подробно рассматривать цикл рисования меток вертикальной оси, поскольку он похож на уже описанный. В нем по вещественной переменной цикла y вычисляется целая координата iy, рисуются горизонтальные пунктирные линии, и выводятся числовые метки слева от левой границы поля графика.

После того, как все оформление поля графика нарисовано, можно приступать к рисованию самого графика. Разумеется, рисовать график нужно только в том случае, если массивы времени и отсчетов отведены – это проверяется оператором if(Count). Можно

было бы проверить указатели Times и Values на значение NULL, но проще сравнить с нулем поле Count, в котором должен храниться размер обоих массивов. В конструкторе мы присвоили Count значение 0, так же будем поступать и в еще не написанной функции очистки массивов ClearArrays. Таким образом, отличие Count от нуля можно использовать как признак существования массивов.

Если массивы отведены, первое, что необходимо сделать – это установить область отсечения рисования. В параметрах блока мы задаем диапазоны горизонтальной (Xmin...Xmax) и вертикальной (Ymin...Ymax) осей, точки вне этих диапазонов будут находиться за пределами изображаемого поля графика. Если не принять мер, выход значения времени за диапазон горизонтальной оси, или выход значения входа блока за диапазон вертикальной, приведет к тому, что рисуемая линия выйдет за пределы поля графика и затронет рамку графика, или даже другие блоки на рабочем поле. Чтобы не допустить этого, проще всего временно ограничить область экрана, в которой можно рисовать. Любые изображения, вышедшие за пределы этой области, будут автоматически отсекаться, причем отсекаться корректно: если, например, одна точка отрезка линии находится внутри области отсечения, а другая – снаружи, будет нарисована только часть отрезка вплоть до границы области. Чтобы программно реализовать такую возможность в функции рисования, пришлось бы вычислять точку пересечения отрезка с границей области. Использование областей отсечения позволяет переложить эти вычисления на Windows API.

Для задания прямоугольной области отсечения используется сервисная функция РДС rdsXGSetClipRect, которая принимает единственный параметр – указатель на структуру типа RECT, описывающую прямоугольник. RECT – стандартная структура Windows API, часто используемая в различных графических функциях – имеет четыре целых поля, задающих левый верхний (left, top) и правый нижний (right, bottom) углы прямоугольника. Мы будем задавать отсечение по прямоугольнику поля графика с отступом на одну точку внутрь поля, чтобы рисуемая линия не наложилась на его рамку. Таким образом, левым верхним углом области, в которой разрешено рисование, будет (x1+1, y1+1), а правым нижним – (x2-1, y2-1). Теперь можно рисовать линию графика, предварительно установив цвет линии (LineColor) и ее толщину. Толщина линии устанавливается с учетом текущего масштабного коэффициента, таким образом, в функцию rdsXGSetPenStyle в качестве толщины линии передается произведение заданной в параметрах блока толщины LineWidth и масштаба DrawData->DoubleZoom. Это приводит к тому, что при увеличении масштаба окна подсистемы линия графика будет становиться толще, а при уменьшении – тоньше. Если бы толщина линии не зависела от масштаба, в мелких масштабах графики с толстыми линиями становились бы нечитаемыми. Следует отметить, что, несмотря на то, что в мелких масштабах произведение LineWidth* DrawData->DoubleZoom может принимать значения, меньшие единицы, и при округлении этих значений до целого внутри функции rdsXGSetPenStyle будет получаться нулевое значение толщины линии, это не вызовет проблем. Установка нулевой толщины линии в сервисных функциях РДС приводит к рисованию линий толщиной в одну точку, то есть наиболее тонкой линии из возможных.

Согласно описанной выше логике работы блока, в любой момент времени массивы отсчетов графика будут заполнены данными до индекса NextIndex-1 включительно. Необходимо построить ломаную линию (Times[0], Values[0]) – (Times[1], Values[1]) – ... – (Times[NextIndex-1], Values[NextIndex-1]), переводя вещественные значения из массивов Times и Values в целые координаты поля графика согласно диапазонам осей. Для этого используется цикл по целой переменной i, принимающей значения от 0 до NextIndex-1, внутри которого вычисляются целые координаты ix и iy очередной точки ломаной по формулам, аналогичным использованным при построении горизонтальной и вертикальной сеток графика. Для самой первой точки

ломаной (при i равном 0) вызывается функция rdsXGMoveTo, для всех остальных – rdsXGLineTo. Таким образом, начиная со второй точки массива, каждая очередная точка будет соединяться линией с предыдущей.

После того, как цикл завершится, необходимо отменить использование области отсечения, вызвав функцию rdsXGSetClipRect с параметром NULL. На этом рисование графика заканчивается.

Следующие по важности после рисования – функции работы с массивами отсчетов. Рассмотрим сначала функцию отведения памяти под массивы, которая вызывается из функции модели блока SimplePlot при первом запуске расчета:

```
// Отведение памяти под массивы
void TSimplePlotData::AllocateArrays(void)
{ // Сначала нужно очистить массивы, если они были отведены ранее
  ClearArrays();
  // При нулевом или отрицательном шаге записи отсчетов
  // работа блока невозможна
  if(TimeStep<=0.0) return;
  // Вычисление требуемого числа отсчетов по диапазону оси
  // времени и шагу записи
  Count=(Xmax-Xmin)/TimeStep+1;
  // Число отсчетов должно быть положительным
  if(Count<=0) {Count=0; return; }
  // Отведение памяти - по Count чисел double
  Times=new double[Count];
  Values=new double[Count];
  // Первый свободный индкс массива - 0
  NextIndex=0;
  // Момент записи отсчета – начало диапазона оси времени
  NextTime=Xmin;
}
//=========================================
```

Первое, что делает эта функция – освобождает память, занятую массивами, если она уже отведена. Это позволяет избежать утечек памяти если, по ошибке, мы вызовем функцию AllocateArrays два раза подряд. Затем вычисляется размер массива, необходимый для записи всего графика. Для этого диапазон оси времени графика (Xmax-Xmin) делится на шаг записи, и к получившемуся результату добавляется единица, чтобы в графике был один лишний отсчет, так как отсчетов на один больше, чем интервалов. Например, если мы хотим строить график с горизонтальным диапазоном 0..5 и шагом записи 1, нам потребуется массив на 6 отсчетов: 0, 1, 2, 3, 4, 5. Если вычисленный размер массива отрицателен или равен нулю, поле Count обнуляется и функция завершается без отведения массивов. Если же он положителен, он записывается в поле Count, после чего при помощи оператора C++ new отводится память под массивы Times и Values размером в Count вещественных чисел двойной точности. Затем инициализируются переменные NewIndex (следующий свободный индекс массивов) и NextTime (момент времени, после которого будет записан очередной отсчет). Поскольку оба массива полностью пусты, в NewIndex записывается 0 (запись будет производиться с начала массива), а в NewtTime – начало диапазона оси времени Xmin.

Теперь рассмотрим функцию освобождения памяти ClearArrays. Мы уже вызывали ее в двух местах (в деструкторе класса и в функции AllocateArrays), пришло время записать ее код. Он будет простым:

```
// Освобождение массивов
void TSimplePlotData::ClearArrays(void)
{ if(Count) // Массивы были отведены
    { delete[] Times;
```

```
      delete[] Values;
    }
  // Обнуление указателей и Count
  Times=Values=NULL;
  Count=NextIndex=0;
}
//=======================================
```

Эта функция довольно проста: если значение поля `Count` не нулевое (мы договорились использовать его в качестве признака наличия массивов), память, на которую ссылаются указатели `Times` и `Values`, освобождается оператором `delete[]`. Затем обнуляются все поля класса, управляющие массивами отсчетов, включая `Count`.

Наконец, запишем функцию `AddPoint`, которая будет добавлять к массивам очередной отсчет, если пришло его время (общий принцип ее работы уже описан на стр. 194). Функция принимает единственный параметр – значение, которое, возможно, нужно добавить в массив `Values` – и вызывается из модели блока при любом изменении динамической переменной. Значение времени функция берет из динамической переменной “DynTime”.

```
// Добавление отсчета в массив
void TSimplePlotData::AddPoint(double v)
{ double t;
  if(NextIndex>=Count) // Весь массив заполнен
    return;
  if(Time==NULL || Time->Data==NULL) // Нет доступа к "DynTime"
    return;
  // Получение значения времени из "DynTime"
  t=*((double*)Time->Data);
  if(t<NextTime) // Еще не пришло время писать отсчет
    return;
  // Достигнуто время записи
  Values[NextIndex]=v;
  Times[NextIndex]=t;
  NextIndex++;        // Следующий отсчет - в следующий индекс
  NextTime+=TimeStep; // Время записи следующего отсчета
}
//=======================================
```

Сначала в функции проверяется, не заполнен ли весь массив (эта же проверка сработает, если массивы отсчетов на отведены) и есть ли доступ к динамической переменной времени. Если доступ есть и в массиве еще есть место, значение текущего времени считывается во вспомогательную переменную `t` и сравнивается со временем записи следующего отсчета `NextTime`. Если `t` окажется меньше `NextTime`, значит, время записи очередного отсчета еще не пришло, и функция завершается. В противном случае значение, переданное функции, записывается в текущий элемент массива `Values`, а время – в текущий элемент массива `Times`. После этого текущий индекс массива увеличивается на 1, а время записи – на шаг записи `TimeStep`. Теперь функция готова к записи следующего отсчета, которая произойдет, как только значение времени превысит новое время записи.

Из функций-членов, объявленных в классе `TSimplePlotData`, остались не написанными только функции сохранения, загрузки и настройки параметров блока. Эти функции достаточно громоздки, поскольку у блока много параметров, но в них не будет ничего принципиально нового – подобные функции рассматривались в §2.7 и §2.8. Мы не будем разбирать их подробно по выполняемым действиям, обратим внимание только на некоторые ранее не встречавшиеся сервисные функции РДС, которые в них используются.

Функция сохранения параметров блока будет записывать их в текстовом виде в формате INI-файлов Windows:

```
// Сохранение параметров в текстовом виде
void TSimplePlotData::SaveText(void)
{ RDS_HOBJECT ini;
  char *str;

  // Создание объекта для работы с текстом
  ini=rdsINICreateTextHolder(TRUE);

  // Создание в тексте секции "[General]"
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"General");
  // Запись в секцию различных параметров
  rdsINIWriteDouble(ini,"TimeStep",TimeStep);
  rdsINIWriteDouble(ini,"Xmin",Xmin);
  rdsINIWriteDouble(ini,"Xmax",Xmax);
  rdsINIWriteDouble(ini,"XGridStep",XGridStep);
  rdsINIWriteInt(ini,"XNumDecimal",XNumDecimal);
  rdsINIWriteDouble(ini,"Ymin",Ymin);
  rdsINIWriteDouble(ini,"Ymax",Ymax);
  rdsINIWriteDouble(ini,"YGridStep",YGridStep);
  rdsINIWriteInt(ini,"YNumDecimal",YNumDecimal);

  // Создание в тексте секции "[Visuals]"
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"Visuals");
  // Запись в секцию различных параметров
  rdsINIWriteInt(ini,"BorderColor",(int)BorderColor);
  rdsINIWriteInt(ini,"FillColor",(int)FillColor);
  rdsINIWriteInt(ini,"PlotBorderColor",(int)PlotBorderColor);
  rdsINIWriteInt(ini,"PlotFillColor",(int)PlotFillColor);
  rdsINIWriteInt(ini,"LineColor",(int)LineColor);
  rdsINIWriteInt(ini,"LineWidth",LineWidth);

  // Преобразование описания шрифта в строку для сохранения
  str=rdsStructToFontText(&Font,NULL);
  // Запись строки с описанием шрифта
  rdsINIWriteString(ini,"Font",str);
  rdsFree(str); // Освобождение памяти, занятой строкой

  // Запись сформированного текста в файл схемы или буфер обмена
  rdsCommandObject(ini,RDS_HINI_SAVEBLOCKTEXT);

  // Уничтожение вспомогательного объекта
  rdsDeleteObject(ini);
}
//=======================================
```

Эта функция устроена так же, как и функция сохранения параметров в примере на стр. 168: создается вспомогательный объект для работы с текстом, в нем создаются секции, в них записываются параметры, после чего сформированный текст передается в РДС для записи и вспомогательный объект уничтожается. Следует отметить только два новых момента: во-первых, все параметры, описывающие цвета различных элементов изображения, приводятся к типу `int` и заносятся в текст как целые числа функцией `rdsINIWriteInt`. Тип `COLORREF`, используемый в Windows API для хранения цветов, допускает такое преобразование. Во-вторых, параметры шрифта, которые хранятся в структуре `Font`, сохраняются не по отдельности, а преобразуются в одну строку описания шрифта при помощи сервисной функции `rdsStructToFontText`. Эта функция формирует динамическую строку, в которой параметры шрифта перечислены после стандартных

ключевых слов. Например, для шрифта, заданного в конструкторе класса, будет сформирована строка

```
font "Arial" height 15 charset 1 color 0
```

Поскольку строка, которую возвращает функция, отводится в динамической памяти, после использования ее необходимо освободить функцией `rdsFree`, как и все динамические строки, используемые в РДС.

Функция загрузки параметров блока тоже выглядит знакомо:

```
// Загрузка параметров в текстовом виде из строки text
void TSimplePlotData::LoadText(char *text)
{ RDS_HOBJECT ini;
  char *str;

  // Создание объекта для работы с текстом
  ini=rdsINICreateTextHolder(TRUE);

  // Загрузка текста в объект
  rdsSetObjectStr(ini,RDS_HINI_SETTEXT,0,text);

  // Если в тексте есть секция "General", загрузить из нее данные
  if(rdsINIOpenSection(ini,"General"))
    { TimeStep=rdsINIReadDouble(ini,"TimeStep",TimeStep);
      Xmin=rdsINIReadDouble(ini,"Xmin",Xmin);
      Xmax=rdsINIReadDouble(ini,"Xmax",Xmax);
      XGridStep=rdsINIReadDouble(ini,"XGridStep",XGridStep);
      XNumDecimal=rdsINIReadInt(ini,"XNumDecimal",XNumDecimal);
      Ymin=rdsINIReadDouble(ini,"Ymin",Ymin);
      Ymax=rdsINIReadDouble(ini,"Ymax",Ymax);
      YGridStep=rdsINIReadDouble(ini,"YGridStep",YGridStep);
      YNumDecimal=rdsINIReadInt(ini,"YNumDecimal",YNumDecimal);
    }
  // Если в тексте есть секция "Visuals", загрузить из нее данные
  if(rdsINIOpenSection(ini,"Visuals"))
    { BorderColor=(COLORREF)rdsINIReadInt(ini,"BorderColor",
                    (int)BorderColor);
      FillColor=(COLORREF)rdsINIReadInt(ini,"FillColor",
                    (int)FillColor);
      PlotBorderColor=(COLORREF)rdsINIReadInt(ini,
                    "PlotBorderColor",(int)PlotBorderColor);
      PlotFillColor=(COLORREF)rdsINIReadInt(ini,"PlotFillColor",
                    (int)PlotFillColor);
      LineColor=(COLORREF)rdsINIReadInt(ini,"LineColor",
                    (int)LineColor);
      LineWidth=(COLORREF)rdsINIReadInt(ini,"LineWidth",LineWidth);
      str=rdsINIReadString(ini,"Font","",NULL);
      if(str) rdsFontTextToStruct(str,NULL,&Font);
    }

  // Уничтожение вспомогательного объекта
  rdsDeleteObject(ini);
}
//=======================================
```

Точно так же, как и функция сохранения параметров, эта функция приводит все данные типа `COLORREF` к типу `int` и работает с цветами как с целыми числами. Для разбора сохраненной строки параметров шрифта используется сервисная функция `rdsFontTextToStruct`, которая по этой строке заполняет отдельные поля структуры `Font`. Функция `rdsINIReadString`, которая считывает строку из вспомогательного

объекта, не отводит память под новую строку, а возвращает указатель на строку из своего внутреннего буфера, поэтому здесь не требуется освобождать какую-либо память функцией `rdsFree`.

Функция настройки параметров будет создавать окно с двумя вкладками: “Оси” и “Внешний вид”, на которых встретится несколько не использовавшихся ранее типов полей ввода:

```
// Функция настройки параметров блока
int TSimplePlotData::Setup(void)
{ RDS_HOBJECT window;
  BOOL ok;
  char *str;

  // Создание окна
  window=rdsFORMCreate(TRUE,-1,-1,"Простой график");

  // Вкладка "Оси"
  rdsFORMAddTab(window,1,"Оси");

  rdsFORMAddEdit(window,1,100,
     RDS_FORMCTRL_EDIT | RDS_FORMFLAG_LINE,"Шаг записи",50);
  rdsSetObjectDouble(window,100,RDS_FORMVAL_VALUE,TimeStep);

  // Текстовая метка без возможности ввода
  rdsFORMAddEdit(window,1,1,RDS_FORMCTRL_LABEL,"Ось X:",0);

  // Диапазон (два поля ввода в одной строке)
  rdsFORMAddEdit(window,1,2,RDS_FORMCTRL_RANGEEDIT,"Диапазон",90);
  rdsSetObjectDouble(window,2,RDS_FORMVAL_VALUE,Xmin);
  rdsSetObjectDouble(window,2,RDS_FORMVAL_RANGEMAX,Xmax);

  rdsFORMAddEdit(window,1,3,RDS_FORMCTRL_EDIT,"Шаг сетки",50);
  rdsSetObjectDouble(window,3,RDS_FORMVAL_VALUE,XGridStep);

  // Поле ввода со стрелками увеличения/уменьшения
  rdsFORMAddEdit(window,1,4,
          RDS_FORMCTRL_UPDOWN | RDS_FORMFLAG_LINE,
          "Дробная часть чисел",50);
  rdsSetObjectInt(window,4,RDS_FORMVAL_VALUE,XNumDecimal);
  rdsSetObjectInt(window,4,RDS_FORMVAL_UPDOWNMIN,0);
  rdsSetObjectInt(window,4,RDS_FORMVAL_UPDOWNMAX,5);
  rdsSetObjectInt(window,4,RDS_FORMVAL_UPDOWNINC,1);

  // Текстовая метка без возможности ввода
  rdsFORMAddEdit(window,1,5,RDS_FORMCTRL_LABEL,"Ось Y:",0);

  rdsFORMAddEdit(window,1,6,RDS_FORMCTRL_RANGEEDIT,"Диапазон",90);
  rdsSetObjectDouble(window,6,RDS_FORMVAL_VALUE,Ymin);
  rdsSetObjectDouble(window,6,RDS_FORMVAL_RANGEMAX,Ymax);

  rdsFORMAddEdit(window,1,7,RDS_FORMCTRL_EDIT,"Шаг сетки",50);
  rdsSetObjectDouble(window,7,RDS_FORMVAL_VALUE,YGridStep);

  rdsFORMAddEdit(window,1,8,RDS_FORMCTRL_UPDOWN,
          "Дробная часть чисел",50);
  rdsSetObjectInt(window,8,RDS_FORMVAL_VALUE,YNumDecimal);
  rdsSetObjectInt(window,8,RDS_FORMVAL_UPDOWNMIN,0);
```

```
rdsSetObjectInt(window,8,RDS_FORMVAL_UPDOWNMAX,5);
rdsSetObjectInt(window,8,RDS_FORMVAL_UPDOWNINC,1);

// Вкладка "Внешний вид"
rdsFORMAddTab(window,2,"Внешний вид");

rdsFORMAddEdit(window,2,9,RDS_FORMCTRL_COLOR,
   "Цвет рамки блока",50);
rdsSetObjectInt(window,9,RDS_FORMVAL_VALUE,(int)BorderColor);
rdsFORMAddEdit(window,2,10,
   RDS_FORMCTRL_COLOR | RDS_FORMFLAG_LINE,"Цвет фона блока",50);
rdsSetObjectInt(window,10,RDS_FORMVAL_VALUE,(int)FillColor);

rdsFORMAddEdit(window,2,11,RDS_FORMCTRL_COLOR,
   "Цвет рамки графика и сетки",50);
rdsSetObjectInt(window,11,RDS_FORMVAL_VALUE,
   (int)PlotBorderColor);
rdsFORMAddEdit(window,2,12,RDS_FORMCTRL_COLOR,
   "Цвет фона графика",50);
rdsSetObjectInt(window,12,RDS_FORMVAL_VALUE,(int)PlotFillColor);
rdsFORMAddEdit(window,2,13,RDS_FORMCTRL_COLOR,
   "Цвет линии графика",50);
rdsSetObjectInt(window,13,RDS_FORMVAL_VALUE,(int)LineColor);
rdsFORMAddEdit(window,2,14,
   RDS_FORMCTRL_UPDOWN | RDS_FORMFLAG_LINE,
   "Толщина линии графика",50);
rdsSetObjectInt(window,14,RDS_FORMVAL_VALUE,LineWidth);
rdsSetObjectInt(window,14,RDS_FORMVAL_UPDOWNMIN,0);
rdsSetObjectInt(window,14,RDS_FORMVAL_UPDOWNMAX,5);
rdsSetObjectInt(window,14,RDS_FORMVAL_UPDOWNINC,1);

// Кнопка открытия диалога выбора шрифта
rdsFORMAddEdit(window,2,15,RDS_FORMCTRL_FONTSELECT,
   "Шрифт чисел",0);
// Преобразование шрифта в строку и занесение в поле ввода
str=rdsStructToFontText(&Font,NULL);
rdsSetObjectStr(window,15,RDS_FORMVAL_VALUE,str);
rdsFree(str);

// Открытие окна
ok=rdsFORMShowModalEx(window,NULL);
if(ok)
  { // Нажата кнопка OK - запись параметров обратно в блок
    Xmin=rdsGetObjectDouble(window,2,RDS_FORMVAL_VALUE);
    Xmax=rdsGetObjectDouble(window,2,RDS_FORMVAL_RANGEMAX);
    XGridStep=rdsGetObjectDouble(window,3,RDS_FORMVAL_VALUE);
    XNumDecimal=rdsGetObjectInt(window,4,RDS_FORMVAL_VALUE);

    TimeStep=rdsGetObjectDouble(window,100,RDS_FORMVAL_VALUE);

    Ymin=rdsGetObjectDouble(window,6,RDS_FORMVAL_VALUE);
    Ymax=rdsGetObjectDouble(window,6,RDS_FORMVAL_RANGEMAX);
    YGridStep=rdsGetObjectDouble(window,7,RDS_FORMVAL_VALUE);
    YNumDecimal=rdsGetObjectInt(window,8,RDS_FORMVAL_VALUE);

    BorderColor=(COLORREF)rdsGetObjectInt(window,9,
         RDS_FORMVAL_VALUE);
```

```
            FillColor=(COLORREF)rdsGetObjectInt(window,10,
                 RDS_FORMVAL_VALUE);
            PlotBorderColor=(COLORREF)rdsGetObjectInt(window,11,
                 RDS_FORMVAL_VALUE);
            PlotFillColor=(COLORREF)rdsGetObjectInt(window,12,
                 RDS_FORMVAL_VALUE);
            LineColor=(COLORREF)rdsGetObjectInt(window,13,
                 RDS_FORMVAL_VALUE);
            LineWidth=rdsGetObjectInt(window,14,RDS_FORMVAL_VALUE);

            // Получение параметров шрифта из строки
            str=rdsGetObjectStr(window,15,RDS_FORMVAL_VALUE);
            rdsFontTextToStruct(str,NULL,&Font);
          }
        // Уничтожение окна
        rdsDeleteObject(window);
        // Возвращаемое значение
        return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
      }
      //=======================================
```

Как и функции сохранения и загрузки параметров, эта функция работает с цветами как с целыми числами. Для задания цвета используется специальный тип поля ввода `RDS_FORMCTRL_COLOR`, выглядящий как кнопка с цветным прямоугольником. При нажатии на эту кнопку открывается стандартный диалог Windows для выбора цвета. Для визуального отделения параметров горизонтальной и вертикальной осей друг от друга на вкладке “Оси” использованы текстовые метки (`RDS_FORMCTRL_LABEL`), которые отображают названия осей и никак не реагируют на действия пользователя. Также для большей наглядности и удобства ввода диапазоны осей `Xmin...Xmax` и `Ymin...Ymax` задаются в специальных двойных полях ввода (тип `RDS_FORMCTRL_RANGEEDIT`), в которых в одной строке задаются начало и конец диапазона. Толщина линии графика вводится в поле ввода со стрелками для увеличения и уменьшения значения (`RDS_FORMCTRL_UPDOWN`) с заданием максимального и минимального возможного значения. Наконец, для задания шрифта используется специальная кнопка (`RDS_FORMCTRL_FONTSELECT`), нажатие на которую открывает стандартный диалог выбора шрифта Windows. Для работы с этим полем-кнопкой параметры шрифта переводятся в строку уже знакомой нам функцией `rdsStructToFontText`, а при закрытии окна кнопкой “OK” заносятся обратно в структуру `Font` функцией `rdsFontTextToStruct`. Внешний вид обеих вкладок окна настройки блока приведен на рис. 60.

Для того, чтобы этот блок мог отображать график, в его параметрах следует включить рисование функцией DLL и разрешить масштабирование (см. рис. 58). Для проверки его работоспособности следует подключить ко входу блока `x` какое-нибудь изменяющееся от времени значение (например, выход генератора), добавить в схему блок-планировщик динамического расчета, если его там еще нет, и запустить расчет. На рис. 61 показан внешний вид графика, подключенного к генератору синусоидального сигнала.

Рассмотренному примеру, конечно, далеко до полнофункционального графика. Для удобства пользователя график должен иметь возможности автоматической подстройки горизонтального и вертикального диапазонов, автоматического увеличения числа отсчетов при переполнении массива, индикации текущего и произвольно выбранного на графике значений и т.п. Однако, данный пример хорошо иллюстрирует большие возможности программного рисования внешнего вида блоков, и все эти функции могут быть, при желании, к нему добавлены.

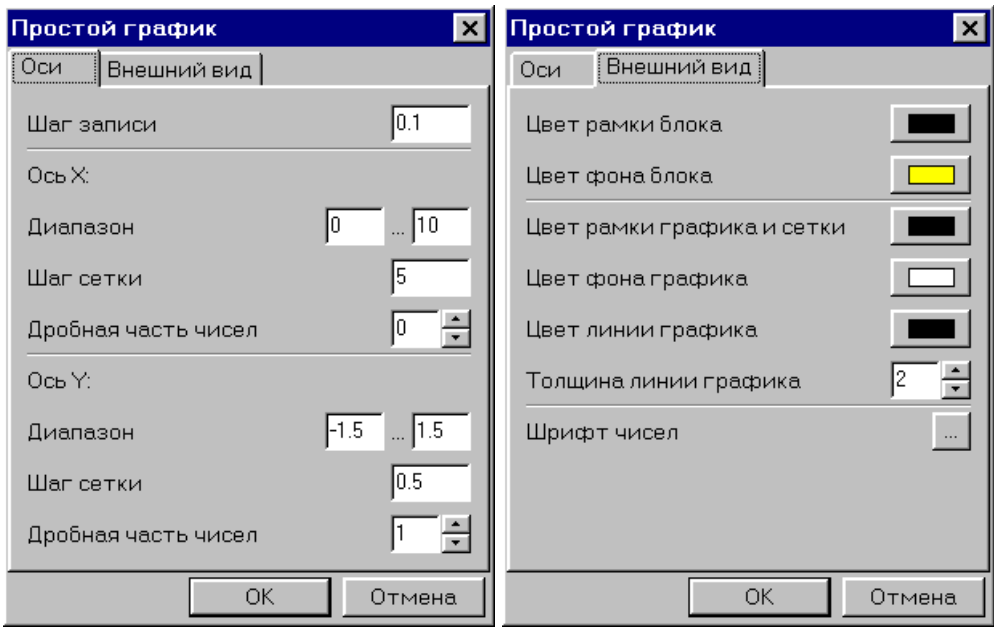


Рис. 60. Окно настройки простого графика

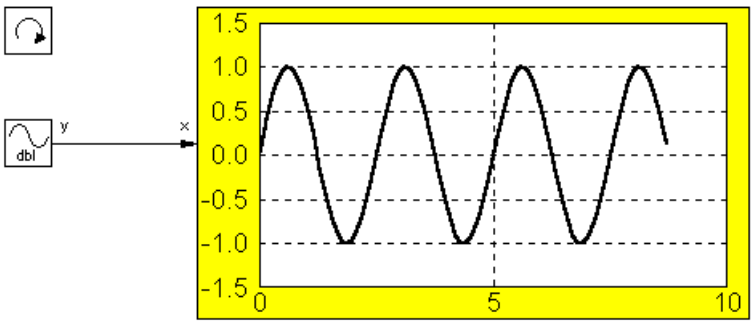


Рис. 61. Простой график в процессе работы

§2.10.2. Оптимизация рисования

Рассматриваются различные способы ускорения программного рисования внешнего вида блоков. Два из этих способов используются для улучшения работы графика из примера, описанного в предыдущем параграфе.

Программное рисование внешнего вида блока – одна из самых затратных по времени функций. В режимах редактирования и моделирования это не так страшно – в них перерисовка окон подсистемы производится достаточно редко: при изменении их размера, при закрытии других окон, которые ранее перекрывали окно подсистемы и т.п. В режиме расчета же окна обновляются по таймеру с заданной частотой, поэтому задержки в функции обновления становятся критичными, особенно если блоков в окне много и внутреннее время работающей системы должно быть синхронизировано с реальным. Хотя в РДС предприняты некоторые меры по уменьшению задержек (например, при включенном в настройках адаптивном алгоритме обновления окон, частота обновления автоматически снижается, если доля задержек от рисования блоков подсистемы превысит заданное значение), разработчику модели блока следует писать функции рисования так, чтобы они работали как можно быстрее. При этом он может пользоваться некоторыми сервисными возможностями РДС, позволяющими не перерисовывать изображение блока, если в этом нет необходимости.

Самый простой способ избежать ненужных обновлений окна – это информирование РДС о том, что в блоке ничего не изменилось. Каждый раз перед вызовом функции модели блока РДС взводит битовый флаг RDS_NEEDSDLLREDRAW в поле Flags структуры RDS_BLOCKDATA (указатель на нее передается в функцию модели при каждом ее вызове, см. стр. 36). Если в результате работы функции модели не произошло никаких изменений, которые должны быть отражены на внешнем виде блока, функция может сбросить этот флаг, сообщив тем самым, что изображение данного конкретного блока обновлять не надо. Когда сработает таймер обновления окна подсистемы, РДС проверит состояние этого флага у каждого блока, изображение которого рисуется программно, и, если у всех блоков этот флаг окажется сброшен, и других изменений в подсистеме не было, обновление окна производиться не будет. Если же модель не сбросит этот флаг (сообщая тем самым о наличии изменений во внешнем виде блока, или просто потому что работа с этим флагом не заложена в модель программистом), все содержимое окна будет перерисовано. Следует отметить, что эффективность этого метода борьбы с задержками не очень высока – для его работы необходимо, чтобы все блоки подсистемы работали с флагом RDS_NEEDSDLLREDRAW. Достаточно хотя бы одного блока, модель которого не сбрасывает этот флаг, чтобы окно подсистемы обновлялось постоянно.

Другой способ уменьшить время, которое РДС будет тратить на обновление окна – это оптимизация самой модели блока. Из функции рисования, входящей в модель, необходимо вынести как можно больше операций, выполнять которые при каждом рисовании нет необходимости. Если посмотреть на функцию рисования из предыдущего примера (стр. 198), можно заметить, что в ней каждый раз заново вычисляются координаты области графика Gr_x1, Gr_x2, Gr_y1 и Gr_y2, которые зависят от размера блока, выбранного в его настройках шрифта и текущего масштаба окна подсистемы. Поскольку эти вычисления связаны с преобразованием вещественных чисел в строки и определением размеров этих строк, выведенных на экран текущим шрифтом, неплохо было бы выполнять их только при изменении масштаба или шрифта. Для этого координаты области графика нужно сделать полями класса TSimplePlotData и ввести в него еще одно дополнительное поле для отслеживания изменений масштаба. Таким образом, изменения, которые необходимо внести в описание класса, выглядят так (выделено жирным шрифтом):

```
//=======================================
// Простой график - личная область данных
//=======================================
class TSimplePlotData
{ private:
    // Запомненные координаты поля графика
    int Gr_x1,Gr_x2,Gr_y1,Gr_y2;
    // Масштаб окна на момент последнего рисования
    double OldZoom;
    // Настроечные параметры графика (цвета, шаг и т.п.)
    double TimeStep;            // Шаг записи отсчетов
    //          ...
    // ... далее без изменений ...
```

В вещественное поле OldZoom при рисовании графика будет записываться текущий масштаб окна подсистемы. Если при очередном вызове функции Draw значение переданного в нее масштабного коэффициента DrawData->DoubleZoom не совпадет со значением OldZoom, значит, с момента последнего рисования масштаб окна подсистемы изменился, и координаты Gr_x1, Gr_x2, Gr_y1 и Gr_y2 нужно вычислить заново. При изменении параметров шрифта, которое тоже должно приводить к пересчету координат поля графика, будем записывать в OldZoom значение –1. Поскольку масштаб окна не может быть отрицательным, это значение никогда не будет совпадать с DrawData->DoubleZoom, поэтому координаты будут вычислены заново при следующем вызове Draw. Шрифт чисел на

осях графика может измениться только в двух случаях: при задании нового шрифта пользователем в окне настроек или при загрузке параметров блока. Таким образом, присваивание `OldZoom=-1` должно выполняться вместе с вызовами функций `Setup` и `LoadText`. Кроме того, его нужно выполнить в конструкторе класса, чтобы при самом первом вызове `Draw` координаты тоже вычислялись, а также при изменении размеров блока, поскольку размеры рабочего поля при этом тоже должны изменяться.

Чтобы оградить себя от возможных ошибок в программировании, мы внесли поле `OldZoom` в закрытую (`private`) область класса. Но нам нужно иметь возможность сбрасывать запомненное значение масштаба из функции модели, присваивая `OldZoom` значение –1. Для этого мы добавим в открытую область класса (`public`) функцию `ResetCoords`, которая будет этим заниматься:

```
//=======================================
// Простой график - личная область данных
//=======================================
class TSimplePlotData
{ private:
    //          ...
  public:
    // Функция сброса запомненного масштаба последнего рисования
    void ResetCoords(void){OldZoom=-1.0;};
    // Функция отведения массивов отсчетов
    void AllocateArrays(void);
    //          ...
    // ... далее без изменений ...
```

В функции рисования мы будем сравнивать текущий масштаб с запомненным, поэтому, вероятно, следует сделать небольшое замечание относительно сравнения вещественных чисел. Обычно, если эти числа являются результатом каких-либо вычислений, их сравнение производится с заданной точностью. Например, если необходимо сравнить результат какой-либо вещественной функции `f(x)` с нулем, обычно пишут следующий оператор: “`if(fabs(f(x))<DELTA) ...`”, где `DELTA` – заданная погрешность. В нашем случае мы сравниваем вещественный масштабный коэффициент с его же прежним значением, которое не является результатом вычислений – это просто запомненное значение той же самой переменной. Таким образом, для отслеживания изменений переменной мы можем использовать оператор точного сравнения “==”.

Внесем изменения в указанные выше функции класса. Начнем с конструктора:

```
// Конструктор класса личной области данных графика
TSimplePlotData::TSimplePlotData(void)
{ // Инициализация OldZoom: значение -1 приведет к принудительному
  // вычислению Gr_x1,Gr_x2,Gr_y1 и Gr_y2 в ближайшем вызове
  // функции Draw
  OldZoom=-1.0;
  // Присвоение начальных значений параметрам
  TimeStep=0.1;            // Шаг записи
  BorderColor=0;           // Цвет рамки вокруг блока
  //       ...
  // ... далее без изменений ...
```

Изменения в функции модели блока выглядят так:

```
        //         ...
        // Функция настройки параметров
        case RDS_BFM_SETUP:
          data->ResetCoords();     // Сброс запомненных значений
          return data->Setup();
```

```
    // Загрузка параметров в текстовом формате
    case RDS_BFM_LOADTXT:
      data->LoadText((char*)ExtParam);
      data->ResetCoords();     // Сброс запомненных значений
      break;

    // Изменение размеров блока
    case RDS_BFM_RESIZE:
      data->ResetCoords();     // Сброс запомненных значений
      break;
    //        ...
```

В реакциях модели на загрузку параметров блока и вызов окна настройки (то есть в реакциях, которые могут привести к изменению параметров шрифта и, следовательно, размеров поля графика) добавлен сброс запомненного масштаба. Технически, при вызове функции настройки пересчитывать координаты имеет смысл только тогда, когда пользователь закрыл окно кнопкой “OK”, подтвердив изменения. Но, на самом деле, можно присваивать `OldZoom` значение –1 при любом вызове функции настройки, поскольку в этом случае, если пользователь закрыл окно кнопкой “Отмена”, вычисление координат выполнится всего один лишний раз – ни РДС, ни, тем более, пользователь этого не заметят.

В модель также добавлена новая реакция на изменение размеров блока `RDS_BFM_RESIZE`, в которой тоже сбрасывается запомненный масштаб.

Теперь нужно сделать вычисление координат `Gr_x1`, `Gr_x2`, `Gr_y1` и `Gr_y2` в функции `Draw` условным – оно должно производиться только при неравенстве `OldZoom` и масштаба окна `DrawData->DoubleZoom`. Кроме того, после вычисления необходимо запоминать новое значение масштаба в `OldZoom`, чтобы заблокировать вычисления координат до его изменения. С этими изменениями функция будет выглядеть следующим образом:

```
// Рисование внешнего вида блока
void TSimplePlotData::Draw(RDS_PDRAWDATA DrawData)
{ // Вспомогательные переменные
  // int Gr_x1,Gr_x2,Gr_y1,Gr_y2;     - эти переменные стали
  //                                    полями класса
  int x1,y1,x2,y2,textheight,w1,w2;
  char buf[80];

  // Рамка графика
  rdsXGSetPenStyle(0,PS_SOLID,1,BorderColor,R2_COPYPEN);
  rdsXGSetBrushStyle(0,RDS_GFS_SOLID,FillColor);
  rdsXGRectangle(DrawData->Left,DrawData->Top,
                 DrawData->Left+DrawData->Width,
                 DrawData->Top+DrawData->Height);

  // Необходимо вычислить координаты поля графика относительно
  // верхнего левого угла блока

  // Установка параметров шрифта с учетом масштаба
  rdsXGSetFontByParStr(&Font,DrawData->DoubleZoom);
  if(DrawData->DoubleZoom!=OldZoom) // Масштаб изменен
    { // Зазор сверху - половина высоты цифры + 1 точка
      rdsXGGetTextSize("0",NULL,&textheight);
      Gr_y1=textheight/2+1;
      // Зазор снизу - полная высота цифры + 1 точка
      Gr_y2=DrawData->Height-textheight-1;
```

```
        // Зазор слева – ширина самого длинного числа вертикальной
        // оси или половина ширины Xmin
        sprintf(buf," %.*lf ",YNumDecimal,Ymin);
        rdsXGGetTextSize(buf,&w1,NULL); // Ширина Ymin
        sprintf(buf," %.*lf ",YNumDecimal,Ymax);
        rdsXGGetTextSize(buf,&w2,NULL); // Ширина Ymax
        if(w2>w1) w1=w2;
        sprintf(buf," %.*lf ",XNumDecimal,Xmin);
        rdsXGGetTextSize(buf,&w2,NULL); // Ширина Xmin
        w2/=2;
        if(w2>w1) w1=w2;
        Gr_x1=w1;
        // Зазор справа – половина ширины Xmax
        sprintf(buf," %.*lf ",XNumDecimal,Xmax);
        rdsXGGetTextSize(buf,&w2,NULL); // Ширина Xmax
        w2/=2;
        Gr_x2=DrawData->Width-w2;

        // Запоминание нового масштаба
        OldZoom=DrawData->DoubleZoom;
      } // if(DrawData->DoubleZoom!=OldZoom)

    // Абсолютные (на рабочем поле) координаты поля графика
    x1=DrawData->Left+Gr_x1;
    x2=DrawData->Left+Gr_x2;
    y1=DrawData->Top+Gr_y1;
    y2=DrawData->Top+Gr_y2;
    //       ...
    // ... далее без изменений ...
```

Фактически, в функцию внесено всего три изменения. Во-первых, координаты `Gr_x1`, `Gr_x2`, `Gr_y1` и `Gr_y2` больше не являются локальными переменными функции, теперь они стали полями класса `TSimplePlotData`. Во-вторых, весь блок вычисления этих координат помещен внутрь условного оператора, который выполняется, только если запомненное значение масштаба отличается от текущего. И, в-третьих, после вычисления координат текущее значение масштаба запоминается в поле `OldZoom`, чтобы не повторять эти вычисления, пока масштаб не изменится.

Если измерить скорости работы старой и новой функций рисования, можно заметить, что прирост скорости получается крайне незначительным – время выполнения старой функции больше времени выполнения новой всего на пару процентов. По сравнению с рисованием чисел на осях (на рис. 61 изображается десять чисел) и построением самого графика, вычисления размеров для четырех чисел и одной дополнительной строки, вынесенные в условный оператор, занимают не так много времени. Однако, для более сложных блоков с большим количеством вычислений, связанных с рисованием, выполнение этих вычислений только при изменениях параметров блока или масштаба окна может дать лучший результат, тем более, что модифицировать подобным образом функцию рисования не так сложно.

Еще один способ повысить скорость работы функции рисования применяется довольно редко: если в окне подсистемы видна только часть блока, можно не рисовать его невидимые фрагменты. Это требует довольно существенного усложнения функции рисования, поэтому такой метод обычно применяют только для очень больших блоков, имеющих размер, существенно больший, чем видимая в окне часть рабочего поля подсистемы, и отображающих много слабо связанных между собой элементов. Например, он хорошо работает для блоков, показывающих внутри своего изображения различные географические карты или схемы. Координаты области рабочего поля, видимой в данный

момент, передаются в модель блока в поле `VisibleRect` структуры `RDS_DRAWDATA`. Это поле имеет тип `RECT*`, то есть указатель на стандартную структуру Windows, описывающую прямоугольник. Если координаты какого-либо элемента изображения блока, который нужно нарисовать, не попадают в этот прямоугольник, элемент можно не рисовать целиком. Мы не будем приводить здесь пример блока с такого рода оптимизациями. Для рассмотренного блока-графика они не дадут прироста скорости (и даже, может быть, замедлят работу за счет большого количества дополнительных проверок), а блоки, в которых это имело бы смысл, слишком сложны сами по себе, чтобы использовать их в качестве примера.

Достаточно часто изображение блока состоит из неподвижной постоянной части (рамки, шкал, элементов оформления) и каких-либо элементов, изменяющихся и двигающихся в процессе работы блока (стрелок, индикаторов, линий и т.п.). Типичный пример такого блока – рассмотренный выше график, в котором в процессе работы схемы постепенно рисуется только сама линия графика, а рамка, числа на осях и сетка остаются неизменными. Было бы логично перерисовывать неподвижные части изображения только тогда, когда нужно обновить все окно, а в процессе расчета (при обновлении окон по таймеру) рисовать только изменившиеся элементы. Разумеется, это возможно только в том случае, если блоки не перемещаются по рабочему полю подсистемы, иначе у них просто не будет неподвижных частей – каждый раз все изображение блока нужно будет рисовать на новом месте. Кроме того, блоки в окне подсистемы не должны перекрываться и накладываться один на другой, иначе перекрытый блок может испортить изображение блока слоем выше, если нарисует что-нибудь на его фиксированной части.

РДС поддерживает специальный режим обновления окна подсистемы (он независимо включается для каждого окна), в котором модели блока сообщается, должна ли она нарисовать изображение блока полностью, или может отобразить только изменения, произошедшие с момента последнего рисования. За этот режим отвечает флаг “в системе только неподвижные не перекрывающиеся блоки” на вкладке “общие” окна параметров подсистемы (рис. 62).

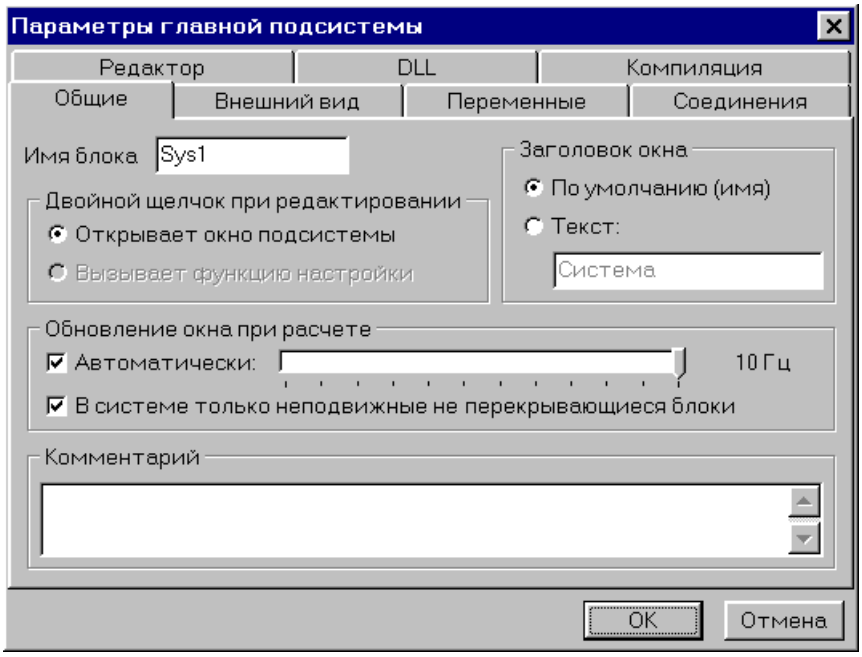


Рис. 62. Вкладка “Общие” окна параметров подсистемы

Если этот флаг включен, РДС в режиме расчета будет по-разному рисовать содержимое окна подсистемы, в зависимости от того, из-за чего возникла необходимость в этом рисовании. Если окно необходимо перерисовать из-за того, что оно было скрыто за

другим окном, изменило размер, пользователь изменил что-либо в редакторе слоев и т.п., все содержимое окна (связи, блоки, сетка, обои окна) будут нарисованы полностью, при этом поле FullDraw структуры RDS_DRAWDATA, указатель на которую передается в модель блока при рисовании его внешнего вида (см. стр. 187), будет иметь значение TRUE. Если же окно должно быть перерисовано из-за срабатывания таймера обновления (при этом изображение, нарисованное в прошлый раз, не было ничем перекрыто), РДС будет рисовать только блоки, причем поле FullDraw вышеупомянутой структуры будет иметь значение FALSE.

Таким образом, чтобы воспользоваться преимуществами, предоставляемыми этим режимом рисования, модель блока должна анализировать поле FullDraw. Если его значение – TRUE, модель должна нарисовать изображение блока полностью. Если же оно – FALSE, модель может "дорисовать" на изображении блока изменения, произошедшие с момента последнего рисования. Выигрыш в скорости при этом будет весьма значительным, ведь чем меньше изменений произошло с момента последнего рисования, тем меньше придется рисовать модели. Если изменений вообще не было, модель завершится немедленно, не потратив процессорного времени зря.

Разумеется, как было указано выше, этот режим не годится для подсистем, блоки которых перемещаются внутри окна или накладываются друг на друга. Его включение в таких подсистемах приведет к появлению "хвостов" за движущимися блоками и другим искажениям изображения.

Ускорим этим методом рисование рассмотренного выше графика. Для этого, прежде всего, необходимо запоминать последнюю построенную точку, чтобы при очередном рисовании продолжать линию графика начиная с нее. Кроме того, нужно ввести в функцию рисования анализ поля FullDraw переданной структуры RDS_DRAWDATA. Чтобы не загромождать уже написанную функцию рисования Draw, которая и так получилась достаточно длинной, мы не будем кардинально переделывать ее – она будет вызываться тогда, когда нужно нарисовать блок полностью. В дополнение к ней мы напишем новую функцию DrawFast, которая будет дорисовывать изменившиеся части блока. Изменения, которые нужно внести в описаний класса блока, выглядят так:

```
//=========================================
// Простой график - личная область данных
//=========================================
class TSimplePlotData
{ private:
    // Запомненные координаты поля графика
    int Gr_x1,Gr_x2,Gr_y1,Gr_y2;
    // Масштаб окна на момент последнего рисования
    double OldZoom;
    // Индекс последнего нарисованного отсчета
    int LastDrawnIndex;
    // Настроечные параметры графика (цвета, шаг и т.п.)
    double TimeStep;            // Шаг записи отсчетов

    //          ...

    void LoadText(char *text);  // Функция загрузки параметров
    void Draw(RDS_PDRAWDATA DrawData); // Функция рисования
    // Функция быстрого рисования
    void DrawFast(RDS_PDRAWDATA DrawData);
    //          ...
    // ... далее без изменений ...
```

В поле LastDrawnIndex будет храниться индекс последнего отсчета в массивах Times и Values, нарисованного функциями Draw или DrawFast. Функция Draw должна, как и

раньше, рисовать линию графика с начала массивов до индекса NextIndex-1, после чего она должна занести значение NextIndex-1 в поле LastDrawnIndex (это будет единственным изменением, которое нужно сделать в Draw). Функция DrawFast должна будет нарисовать отсчеты с LastDrawnIndex+1 по NextIndex-1, и тоже занести NextIndex-1 в LastDrawnIndex. Таким образом, при каждом вызове DrawFast она будет дорисовывать участок графика до последнего запомненного отсчета. В конструкторе класса желательно присвоить полю LastDrawnIndex начальное значение, сигнализирующее об отсутствии уже нарисованной части графика (будем использовать для этого значение –1):

```
// Конструктор класса личной области данных графика
TSimplePlotData::TSimplePlotData(void)
{ // Инициализация OldZoom: значение -1 приведет к принудительному
  // вычислению Gr_x1,Gr_x2,Gr_y1 и Gr_y2 в ближайшем вызове
  // функции Draw
  OldZoom=-1.0;
  // Инициализация LastDrawnIndex
  LastDrawnIndex=-1;
  // Присвоение начальных значений параметрам
  TimeStep=0.1;             // Шаг записи
  BorderColor=0;            // Цвет рамки вокруг блока
  //       ...
  // ... далее без изменений ...
```

Казалось бы, присваивать начальное значение LastDrawnIndex в конструкторе класса не обязательно: при самом первом рисовании РДС потребует от модели нарисовать изображение блока полностью, таким образом, модель должна будет вызвать функцию Draw, которая и установит значение этого поля. Однако, если внимательно посмотреть на текст функции Draw, можно заметить, что в некоторых случаях она завершается, не построив график (например, если массивы отсчетов пусты). Проще инициализировать поле LastDrawnIndex в конструкторе, чем следить за его правильной установкой во всех случаях завершения функции Draw. Кроме того, оставлять важные поля класса без значений по умолчанию в программировании обычно считается дурным тоном.

Добавим в функцию Draw установку поля класса LastDrawnIndex после рисования:

```
// Рисование внешнего вида блока
void TSimplePlotData::Draw(RDS_PDRAWDATA DrawData)
{ // Вспомогательные переменные

  //       ...

      // Строим ломанную линию по отсчетам из массивов
      for(int i=0;i<NextIndex;i++)
        { // Преобразуем вещественные отсчеты в целочисленные
          // координаты на рабочем поле
          int ix=x1+(Times[i]-Xmin)*(x2-x1)/(Xmax-Xmin),
              iy=y2-(Values[i]-Ymin)*(y2-y1)/(Ymax-Ymin);
          if(i) // Не первая точка - строим линию от предыдущей
            rdsXGLineTo(ix,iy);
          else // Первая точка графика - делаем ее текущей
            rdsXGMoveTo(ix,iy);
        }
      // Запоминаем последний нарисованный отсчет
      LastDrawnIndex=NextIndex-1;
      // Отмена отсечения
      rdsXGSetClipRect(NULL);
```

```
    }
  }
  //=========================================
```

Теперь осталось написать функцию DrawFast и обеспечить ее вызов вместо Draw, если модель блока вызвана РДС для рисования изменений. Вынесем эту проверку внутрь функции DrawFast: она сама будет анализировать поле FullDraw в переданной ей структуре и вызывать Draw, если значение поля истинно. В самой функции модели просто заменим вызов Draw на вызов DrawFast:

```
        // Рисование внешнего вида блока
        case RDS_BFM_DRAW:
          // data->Draw((RDS_PDRAWDATA)ExtParam);
          data->DrawFast((RDS_PDRAWDATA)ExtParam);
          break;
```

Наконец, напишем функцию DrawFast:

```
  // Рисование изменений
  void TSimplePlotData::DrawFast(RDS_PDRAWDATA DrawData)
  { int x1,y1,x2,y2;

    // Если РДС требует полного рисования, или полное рисование
    // еще не проводилось, вызывается старая функция Draw
    if(DrawData->FullDraw || DrawData->DoubleZoom!=OldZoom)
      { Draw(DrawData);
        return;
      }

    // Вычисление абсолютных координат поля графика
    x1=DrawData->Left+Gr_x1;
    x2=DrawData->Left+Gr_x2;
    y1=DrawData->Top+Gr_y1;
    y2=DrawData->Top+Gr_y2;

    if(x1>=x2 || y1>=y2) // Негде рисовать
      return;

    // Если массивы не пустые - рисовать график, начиная
    // с последней уже нарисованной точки
    if(Count)
      { RECT r;
        // Установить область отсечения рисования по полю графика
        r.left=x1+1;
        r.top=y1+1;
        r.right=x2-1;
        r.bottom=y2-1;
        rdsXGSetClipRect(&r);

        // Установить сплошной стиль линии, заданный для
        // графика цвет и толщину линии с учетом масштаба
        rdsXGSetPenStyle(0,PS_SOLID,
             LineWidth*DrawData->DoubleZoom,
             LineColor,R2_COPYPEN);

        // Проверка допустимости LastDrawnIndex - на всякий случай
        if(LastDrawnIndex<0) LastDrawnIndex=0;

        // Строим ломанную линию по отсчетам из массивов,
        // начиная с LastDrawIndex
```

```
            for(int i=LastDrawnIndex;i<NextIndex;i++)
              { int ix=x1+(Times[i]-Xmin)*(x2-x1)/(Xmax-Xmin),
                   iy=y2-(Values[i]-Ymin)*(y2-y1)/(Ymax-Ymin);
                if(i!=LastDrawnIndex) rdsXGLineTo(ix,iy);
                else rdsXGMoveTo(ix,iy);
              }

            // Запоминаем последний нарисованный отсчет
            LastDrawnIndex=NextIndex-1;

            // Отмена отсечения
            rdsXGSetClipRect(NULL);
          }
      }
      //=======================================
```

Прежде всего функция выполняет проверку: требуется ли нарисовать изображение блока полностью, или можно ограничиться только отображением изменений. Если `DrawData->FullDraw` имеет значение `TRUE`, то есть требуется полное рисование, вызывается старая функция `Draw`, и работа функции `DrawFast` на этом завершается. То же самое происходит и в случае несоответствия масштаба на в момент предыдущего рисования (`OldZoom`) текущему масштабу (`DrawData->DoubleZoom`). В противном случае, точно так же, как и в `Draw`, вычисляются абсолютные координаты поля графика `x1`, `y1`, `x2` и `y2`, после чего размеры поля проверяются на допустимость. Если с размерами все в порядке и в массивах есть отсчеты (значение `Count` не равно нулю), устанавливается область отсечения и стиль линии для рисования графика. Этот фрагмент функции в точности соответствует аналогичному в `Draw`, а дальше уже начинаются различия.

Если в функции `Draw` рисование линии графика начиналось с нулевого индекса в массиве отсчетов, в этой функции начинать нужно с индекса `LastDrawIndex`, поскольку вплоть до этого индекса линия уже нарисована прошлыми вызовами функций. В случае, если `LastDrawIndex` имеет отрицательное значение (в конструкторе мы присвоили ему –1), ему принудительно присваивается 0, чтобы рисование графика началось с самого первого отсчета. Далее следует цикл рисования, отличающийся от соответствующего цикла в `Draw` только тем, что в качестве начального индекса везде используется не 0, а `LastDrawIndex`. После цикла в `LastDrawIndex` записывается индекс последнего нарисованного отсчета, отключается область отсечения и функция завершается.

Изменения, внесенные в модель графика, позволяют ускорить его рисование в несколько раз, особенно в схемах, работающих синхронно с реальным временем, в которых обновление окон идет постоянно. При этом следует помнить о недопустимости перекрытия изображений блоков в этом режиме, в противном случае можно будет наблюдать эффект, подобный изображенному на рис. 63.

На рисунке видно, что линия графика блока, лежащего ниже, рисуется поверх изображения блока, лежащего выше, хотя она должна быть скрыта под ним. В таких случаях нужно отключать ускоренный режим рисования в окне параметров подсистемы (см. рис. 62). Вносить какие-либо изменения в модели блоков не нужно: РДС в этом случае будет каждый раз вызывать модель с требованием полного рисования изображения блока, что в ней должно быть предусмотрено в любом случае. В рассмотренном примере при каждом рисовании графика просто будет вызываться функция `Draw`, поскольку поле `FullDraw` структуры `RDS_DRAWDATA` никогда не будет иметь значение `FALSE`.

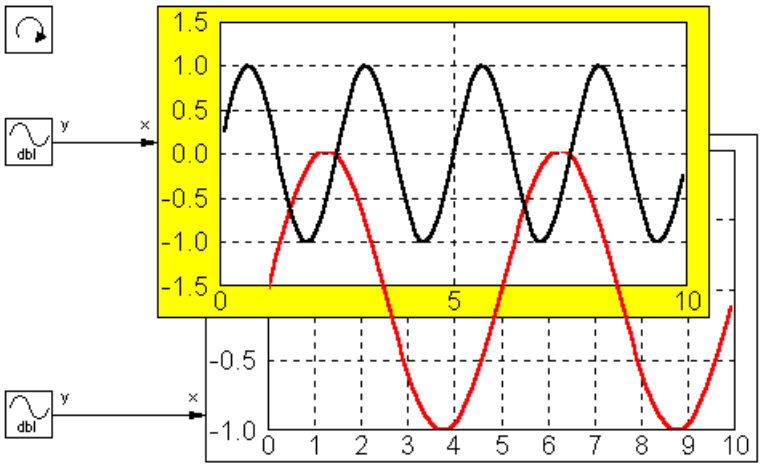


Рис. 63. Искажения изображения у перекрывающихся блоков

§2.10.3. Дополнительное рисование

Рассматривается рисование дополнительных элементов поверх изображения блока. В описанный ранее график добавляется рисование иконки, указывающей на отсутствие доступа к необходимой ему переменной времени.

Достаточно часто возникает необходимость как-либо визуально пометить блок, сигнализируя пользователю о возникшей в блоке ошибке, особом состоянии, недопустимом сочетании настроечных параметров и т.п. Если внешний вид блока рисуется программно, можно отображать такие пометки в функции рисования. В блоке, внешний вид которого задается векторной картинкой, для этого можно предусмотреть соответствующие элементы картинки и управлять их видимостью, но это будет работать только в режимах моделирования и расчета. В режиме редактирования всегда выводятся все элементы векторной картинки, поэтому все предусмотренные пометки будут видны пользователю, и это будет сбивать его с толку. Если же внешний вид блока задается прямоугольником с текстом, модель не сможет изменять его простыми средствами, и не сможет как-либо привлечь внимание пользователя к проблемам этого блока.

Для того, чтобы модель могла нарисовать что-либо поверх изображения блока, независимо от способа, которым задан его внешний вид, в РДС введен специальный режим вызова модели для дополнительного рисования. После того, как изображение блока (будь то прямоугольник с текстом, векторная картинка или результат программного рисования) выведено в окно подсистемы, модель вызывается в режиме `RDS_BFM_DRAWADDITIONAL` и ей передается указатель на структуру `RDS_DRAWDATA`, как и в режиме `RDS_BFM_DRAW`. В этом дополнительном режиме модель может при помощи функций Windows API или сервисных функций РДС рисовать в окне подсистемы что угодно, как и при уже рассмотренном программном рисовании. Главное отличие режима вызова `RDS_BFM_DRAWADDITIONAL` от `RDS_BFM_DRAW` заключается в том, что в режиме `RDS_BFM_DRAW` модель вызывается только при включенном в параметрах блока программном рисовании (см. рис. 58), а в режиме `RDS_BFM_DRAWADDITIONAL` она вызывается всегда при обновлении окна подсистемы. Таким образом, включение в модель блока реакции на вызов в этом режиме позволяет делать на любом блоке произвольные пометки, на которые не будет влиять даже изменение внешнего вида блока пользователем. Модель, которой не нужно ничего рисовать поверх изображения блока, может просто игнорировать этот вызов.

Для иллюстрации дополнительного рисования снова возьмем блок-график, рассмотренный выше. Для построения графика зависимости своего входа от времени этот блок использует стандартную динамическую переменную “DynTime”, из которой он берет текущее значение времени. Создать эту переменную и присваивать ей значения должен другой блок (например, стандартный планировщик динамического расчета из библиотеки блоков РДС), без нее график строиться не будет, и на экран будут выведены только координатные оси с числами. Было бы правильно в этом случае привлечь внимание пользователя к графику, нарисовав на нем иконку, сигнализирующую о неработоспособности блока. Стандартные библиотечные блоки в этом случае рисуют красный восклицательный знак в круге на желтом поле, мы поступим так же.

Прежде всего, введем в класс личной области данных блока новую функцию-член `DrawAdditional`, которая будет выводить указанную иконку при отсутствии доступа к переменной “DynTime”. Эта функция, как и две других функции рисования, уже имеющиеся в классе, будет принимать единственный параметр – указатель на структуру `RDS_DRAWDATA`, полученный моделью блока из РДС при ее вызове в режиме `RDS_BFM_DRAWADDITIONAL`:

```
//=======================================
// Простой график - личная область данных
//=======================================
class TSimplePlotData
{
    //          ...

    void LoadText(char *text);   // Функция загрузки параметров
    void Draw(RDS_PDRAWDATA DrawData); // Функция рисования
    // Функция быстрого рисования
    void DrawFast(RDS_PDRAWDATA DrawData);
    // Рисование иконки при отсутствии доступа к DynTime
    void DrawAdditional(RDS_PDRAWDATA DrawData);
    //          ...
    // ... далее без изменений ...
```

Для вызова `DrawAdditional` из функции модели блока, в оператор `switch` нужно вставить новую метку `case`:

```
// Рисование внешнего вида блока
case RDS_BFM_DRAW:
  // data->Draw((RDS_PDRAWDATA)ExtParam);
  data->DrawFast((RDS_PDRAWDATA)ExtParam);
  break;

// Дополнительное рисование
case RDS_BFM_DRAWADDITIONAL:
  data->DrawAdditional((RDS_PDRAWDATA)ExtParam);
  break;
```

Теперь напишем функцию `DrawAdditional`. Она должна проверять, есть ли у блока доступ к переменной “DynTime” (необходимо проверить поле класса `Time`, в котором должен храниться указатель на структуру подписки), и, при отсутствии доступа, нарисовать иконку. Красный восклицательный знак в желтом круге – одна из стандартных иконок РДС, поэтому для ее рисования мы будем использовать сервисную функцию.

```
// Дополнительное рисование
void TSimplePlotData::DrawAdditional(RDS_PDRAWDATA DrawData)
{
  // Проверка доступа к переменной времени
  if(Time==NULL || Time->Data==NULL)  // Доступа нет
    { int w,h;
```

```
            // Константа, указывающая на стандартную иконку РДС
            DWORD icon=RDS_STDICON_YELCIRCEXCLAM;
            // Определяем размер иконки и выводим ее в центре блока
            if(rdsXGGetStdIconSize(icon,&w,&h))
              rdsXGDrawStdIcon(DrawData->Left+(DrawData->Width-w)/2,
                               DrawData->Top+(DrawData->Height-h)/2,
                               icon);
        }
    }
    //=======================================
```

Прежде всего, в этой функции проверяется наличие доступа к динамической переменной, на которую ссылается указатель на структуру подписки `Time`, и все дальнейшие действия производятся только при отсутствии этого доступа. Чтобы нарисовать иконку в центре изображения блока, где она с большей вероятностью привлечет внимание пользователя, необходимо сначала определить размеры этой иконки. Каждой стандартной иконке РДС соответствует целая константа `RDS_STDICON_*`, используемая в сервисных функциях (полный список констант приведен в приложении А). Красному восклицательному знаку в желтом круге соответствует константа `RDS_STDICON_YELCIRCEXCLAM`. Для улучшения читаемости программы эта константа присваивается вспомогательной переменной `icon`. Затем вызывается сервисная функция `rdsXGGetStdIconSize`, которая определяет ширину и высоту указанной иконки и помещает их в переменные `w` и `h` соответственно. Поскольку мы используем стандартную иконку РДС, можно было бы не определять ее размеры – известно, что ее ширина и высота равны 16 точкам. Однако, при дальнейшей модернизации РДС эти размеры могут быть изменены, поэтому лучше получить их текущее значение при помощи сервисной функции. Зная размеры иконки и координаты прямоугольника блока, которые можно взять из структуры `DrawData`, можно вывести эту иконку в центр прямоугольника вызовом сервисной функции `rdsXGDrawStdIcon`. Первые два параметра функции – горизонтальная и вертикальная координаты верхнего левого угла выводимой иконки, третий – уникальный идентификатор стандартной иконки РДС (в нашем случае – `RDS_STDICON_YELCIRCEXCLAM`).

Теперь, если удалить из схемы планировщик динамических вычислений, в центре графика немедленно появится иконка, сигнализирующая об ошибке (рис. 64). Если вернуть планировщик обратно, иконка немедленно исчезнет. Это позволит пользователю, посмотрев на схему, сразу определить неправильно работающие блоки. На самом деле, кроме иконки, указывающей на наличие ошибки, было бы неплохо информировать пользователя о самой ошибке (в данном случае, указывать на отсутствие в системе переменной времени). Можно было бы вывести на изображении блока какой-либо описывающий ошибку текст, но в мелких масштабах он будет нечитаемым и ничем не поможет пользователю. Такие поясняющие тексты лучше всего выводить во всплывающих подсказках, которые будут рассмотрены в §2.11.

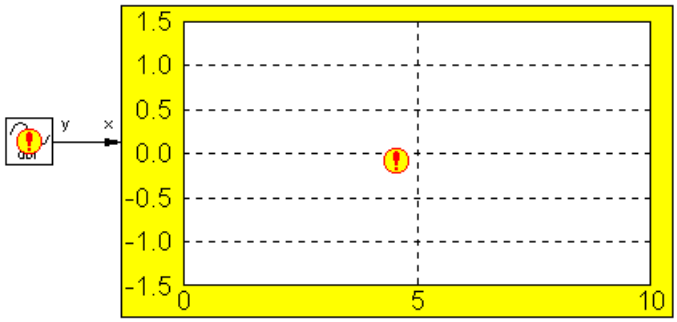


Рис. 64. Внешний вид графика без доступа к динамической переменной

§2.10.4. Панели блоков в окне подсистемы

Рассматривается создание панелей – отдельных окон, принадлежащих блокам и находящихся внутри окна подсистемы. Такие панели нужны для размещения в них полей ввода или привязки к ним вывода внешних библиотек. Приводится пример блока, создающего пустую панель, которую пользователь может открывать, закрывать и перемещать. Затем этот пример модифицируется, и с панелью связывается область вывода трехмерной библиотеки OpenGL, а блок становится индикатором трех угловых координат, рисующим на панели поворачивающийся трехмерный объект.

Программное рисование модели блока с помощью функций API Windows или сервисных функций РДС дает возможность строить сколь угодно сложные изображения в окне подсистемы. Однако, модель не имеет доступа к самому окну подсистемы, и, поэтому, не может создавать внутри него свои собственные объекты (поля ввода, элементы управления и т.п.) Проще говоря, модель блока не может средствами РДС получить дескриптор (типа `HWND`) окна подсистемы и вмешиваться в его работу, добавляя в него свои объекты. Технически, модель может получить этот идентификатор при помощи стандартной функции API Windows `WindowFromDC` и работать с ним, однако, разработчику модели лучше этого не делать из-за возможных проблем с совместимостью – механизм взаимодействия РДС с моделями блоков на это не рассчитан. Вместо этого модель блока, используя сервисные функции РДС, может создавать внутри окна родительской подсистемы специальные объекты – *панели*. Каждая такая панель является оконным объектом Windows и имеет дескриптор `HWND`, который РДС сообщает модели блока при создании панели. РДС также уведомляет модель об уничтожении оконного объекта при закрытии окна подсистемы, о необходимости перерисовать содержимое панели, о перемещении или изменении размера панели пользователем (если это разрешено) и т.д. Модель блока может создавать внутри панели любые другие оконные объекты – главное, чтобы она удалила их при закрытии панели в ответ на соответствующее сообщение от РДС. Следует помнить, что модель может создавать произвольное число панелей, но только в окне родительской подсистемы. Создавать панели в окнах других подсистем нельзя.

Общая процедура работы модели с панелями в окне подсистемы выглядит следующим образом:

- Модель вызывает сервисную функцию `rdsPANCreate` для создания объекта-панели (это один из многих вспомогательных объектов РДС типа `RDS_HOBJECT`). Независимо от того, открыто или закрыто в данный момент окно родительской подсистемы, РДС будет следить за данной подсистемой – как только окно будет открыто, внутри него будет создано дочернее окно-панель.
- Когда окно подсистемы открывается, РДС создает внутри него оконный объект с указанными моделью ранее параметрами и информирует об этом модель блока, передавая ей дескриптор `HWND` этого объекта. В этот момент модель может создать внутри указанного объекта все необходимые ей органы управления, поля ввода и т.п.
- Если при создании панели модель разрешила пользователю перемещать панель или изменять ее размеры, она будет получать уведомления об этих событиях. Также, если это было разрешено, РДС будет информировать ее о получении оконным объектом сообщения `WM_PAINT`, то есть о необходимости перерисовки содержимого.
- При закрытии окна подсистемы РДС сообщит модели о том, что оконный объект сейчас будет уничтожен. В этот момент модель должна уничтожить все органы управления, которые она поместила туда при создании объекта. После закрытия окна РДС снова будет следить за подсистемой, чтобы при очередном открытии окна создать объект и передать модели его идентификатор.
- Если панель больше не нужна модели, она удаляет созданный вспомогательный объект функцией `rdsDeleteObject`, что автоматически приводит к удалению панели. После этого РДС перестает следить за окном подсистемы.

Модель блока также может скрывать и показывать панель, не удаляя вспомогательный объект РДС. При скрытии панели связанный с ней оконный объект уничтожается (о чем, как обычно, РДС сообщает модели блока), а при показе – снова создается. Часто бывает удобно создать объект-панель в момент подключения модели к блоку (например, в конструкторе класса личной области данных, если используется C++), но не показывать эту панель немедленно – функция `rdsPANCreate` предоставляет такую возможность. Панель будет показана пользователю позже, когда возникнет такая необходимость. Затем она может быть снова скрыта, вновь показана и т.д. При этом все параметры панели (положение, размер и т.п.) сохраняются во вспомогательном объекте, поэтому их не нужно устанавливать заново при каждом показе панели. Уничтожение вспомогательного объекта в этом случае целесообразно производить в момент отключения модели от блока – если панель в этот момент видима, она автоматически закроется.

Размещение стандартных органов управления Windows на панелях в окне подсистемы может показаться более удобным, чем программное рисование этих же органов и реализация функций редактирования (если они нужны) силами самой модели блока, однако, у этого метода есть и недостатки – при выводе подсистемы на печать (или сохранении ее графического образа в виде растрового рисунка) панели, являющиеся отдельными оконными объектами, изображаться не будут. Точно так же они не будут отображаться в портах вывода при работе РДС под управлением внешнего приложения (см. §3.6).

В качестве примера рассмотрим блок, выводящий трехмерное изображение средствами библиотеки OpenGL [10, 11]. Для того, чтобы с ее помощью отображать трехмерные сцены в Windows, нужно знать дескриптор окна, в которое будет осуществляться вывод. Конечно, модель блока может открыть отдельное окно средствами Windows API и рисовать в нем трехмерные изображения, но при этом это окно не будет привязано к подсистеме и будет перемещаться независимо от ее окна. Это будет не очень хорошо выглядеть, если подсистема имитирует какой-либо пульт управления, а трехмерные изображения являются частью одного из приборов. При перетаскивании окна такой подсистемы или изменении его масштаба пульт будет “разваливаться” на отдельные окна. Чтобы избежать этого, будем выводить трехмерные сцены на панель блока внутри окна подсистемы.

Для большей ясности примера сначала сделаем модель, которая создает панель в окне подсистемы, но ничего в ней не рисует, а потом добавим в эту модель функции построения трехмерного изображения. Такая “скелетная” модель позволит сосредоточиться на функциях работы с панелями и оставить пока в стороне тонкости OpenGL.

Новую модель имеет смысл разместить в отдельной DLL, а не дописывать к уже имеющимся. В дальнейшем нам придется включить дополнительные описания для работы с OpenGL, и нет смысла перегружать ими модели, не работающие с трехмерными сценами. Главная функция DLL этой модели не будет отличаться от уже рассмотренных ранее:

```
#include <windows.h>
#include <RdsDef.h>
// Подготовка описаний сервисных функций
#define RDS_SERV_FUNC_BODY GetInterfaceFunctions
#include <RdsFunc.h>
// Глобальная переменная для значения ошибки
double DoubleErrorValue;
//========== Главная функция DLL ==========
int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                         unsigned long reason,
                         void* /*lpReserved*/)
{ if(reason==DLL_PROCESS_ATTACH) // Загрузка DLL
    { // Получение доступа к функциям
```

```
      if(!GetInterfaceFunctions())
        MessageBox(NULL,"Нет доступа к функциям","Ошибка",MB_OK);
      else
        rdsGetHugeDouble(&DoubleErrorValue);
    }
  return 1;
}
//========= Конец главной функции =========
```

Поскольку в процессе работы модели блока будет создана панель, идентификатор этой панели, возвращаемый функцией `rdsPANCreate`, нужно где-то хранить. Кроме того, для регулярного обновления панели в процессе расчета потребуется таймер (в отличие от самого окна подсистемы, обновляемого при расчете с заданным интервалом, панели в ней автоматически не перерисовываются – модель блока должна делать это сама при необходимости). Идентификатор панели и идентификатор таймера, который создаст модель, будут храниться в личной области данных блока, которую мы оформим как класс C++:

```
//================================================
// Личная область данных блока
//================================================
class TOpenGLInstr
{ private:
    RDS_TIMERID RefreshTimer;    // Таймер обновления

  public:
    RDS_HOBJECT Panel;           // Объект-панель

    // Сохранение параметров блока
    void SaveText(void);
    // Загрузка параметров блока
    void LoadText(char *text);

    // Создание таймера обновления
    void CreateRefreshTimer(RDS_BHANDLE parent);

    // Конструктор класса
    TOpenGLInstr(void);
    // Деструктор класса
    ~TOpenGLInstr();
};
//================================================
```

В закрытой области класса (`private`) размещаются поля и функции класса, к которым не нужен доступ снаружи этого класса, то есть из функции модели блока. В данном случае это идентификатор таймера обновления `RefreshTimer`. В открытой области (`public`) расположено поле `Panel` для хранения идентификатора созданной панели, а также функции, которые будут вызываться непосредственно из модели блока. Чтобы положение и размер панели, созданной блоком, не терялись при сохранении и последующей загрузке схемы, будем записывать и загружать их функциями `SaveText` и `LoadText` соответственно. Создание таймера обновления панели также вынесем в отдельную функцию-член класса `CreateRefreshTimer` (удаление таймера будет происходить в деструкторе).

Создавать панель мы будем в конструкторе класса, то есть в момент присоединения модели к блоку (при загрузке схемы или добавлении в схему блока с этой моделью из библиотеки блоков):

```
// Конструктор класса личной области данных
TOpenGLInstr::TOpenGLInstr(void)
{ // Создаем панель в окне подсистемы
  Panel=rdsPANCreate(0,0,0,300,300,
```

```
                              RDS_PAN_F_SCALABLE|RDS_PAN_F_BORDER|
                              RDS_PAN_F_SIZEABLE|RDS_PAN_F_MOVEABLE|
                              RDS_PAN_F_CAPTION|RDS_PAN_F_HIDDEN|
                              RDS_PAN_F_PAINTMSG,
                              "Прибор");
        // Инициализируем идентификатор еще не созданного таймера
        RefreshTimer=NULL;
      }
      //================================================
```

Первой же командой конструктора создается объект-панель, и его идентификатор присваивается полю класса `Panel`. Для создания объекта используется функция `rdsPANCreate`:

```
RDS_HOBJECT RDSCALL rdsPANCreate(
     int order,// Близость панели к переднему плану
     int left,int top,     // Верхний левый угол панели
     int width,            // Ширина панели
     int height,           // Высота панели
     int flags,            // Флаги
     LPSTR caption         // Заголовок панели
);
```

Параметр `order` определяет близость панели к переднему плану – панели одного и того же блока с большим значением `order` будут перекрывать панели с меньшим значением. Перекрытие панелей разных блоков определяется взаимным расположением самих блоков, то есть если блок A расположен в окне подсистемы ближе к переднему плану, чем блок B, все панели блока A будут перекрывать панели блока B, независимо от того, с каким значением параметра `order` эти панели были созданы. При этом панели блока A с `order`=10 будут перекрывать панели этого же блока со значением `order`=5. Если блок создает единственную панель, как в нашем случае, значение параметра `order` может быть любым, поскольку от него ничего не зависит.

Параметры `left`, `top`, `width` и `height` задают положение левого верхнего угла панели и ее размеры в координатах рабочей области подсистемы. Значения ширины и высоты указываются для масштаба 100%. Будет ли размер панели изменяться в соответствии с масштабом окна подсистемы, или ее размер в точках экрана будет оставаться неизменным, определяется одним из флагов в параметре `flags`.

Параметр `flags` содержит один или несколько битовых флагов, определяющих внешний вид и поведение панели:

- `RDS_PAN_F_SCALABLE` – если указан этот флаг, размер панели будет меняться синхронно с изменением масштаба в окне подсистемы. Это удобно для блоков, выводящих на панель какую-либо абстрактную графику (как в нашем случае), но, обычно, нежелательно для блоков, размещающих в панели какие-либо поля ввода и органы управления. Как правило, размер поля ввода должен оставаться постоянным, иначе оно может стать слишком мелким для комфортной работы с ним. Если размер панели с полем ввода будет меняться, а размер самого поля – нет, в мелких масштабах поле просто не уместится в панель, а в крупных поле займет только верхний левый угол панели, большая часть которой будет закрывать рабочее поле окна безо всякого смысла.
- `RDS_PAN_F_BORDER` – если указать этот флаг, панель будет иметь вокруг себя рельефную рамку. Панель с рамкой визуально выделяется на рабочем поле системы, что, как правило, и требуется. Однако, если необходимо создать сложное изображение, разместив рядом несколько панелей и блоков, рамку лучше не включать, чтобы панели, расположенные рядом, воспринимались как единое целое. При одинаковых размерах, панель с рамкой имеет несколько меньшую площадь, доступную блоку, чем панель без рамки, поскольку параметры `width` и `height` задают внешние, с учетом рамки, размеры

панели. Следует также помнить, что РДС поддерживает перемещение панели пользователем и изменение ее размеров только для панелей с рамками. Если необходимо дать пользователю возможность перемещать панель без рамки, разработчик модели блока должен заложить такую возможность самостоятельно, например, разместив на панели собственный оконный объект Windows и реагируя на перемещения курсора в его пределах.

- `RDS_PAN_F_CAPTION` – флаг указывает на наличие у панели заголовка с названием в стиле окон Windows. Текст названия панели при этом задается параметром `caption`. Заголовок могут иметь только панели с рамкой. Он также используется для перемещения панели пользователем, если это разрешено (пользователь "перетаскивает" панель за заголовок левой кнопкой мыши, как обычное окно Windows), поэтому, если необходимо дать пользователю возможность перемещать панель, заголовок нужно включить.
- `RDS_PAN_F_MOVEABLE` – панель может перемещаться пользователем (только при наличии у нее рамки и заголовка). При этом модель блока уведомляется о новом положении панели.
- `RDS_PAN_F_SIZEABLE` – размер панели может изменяться пользователем (только при наличии у нее рамки). Изменение размера панели производится точно так же, как и обычного окна Windows – перетаскивается один из углов или одна из сторон рамки. Модель блока при этом уведомляется о новом размере панели.
- `RDS_PAN_F_NOBUTTON` – в заголовке панели нет кнопки закрытия. По умолчанию эта кнопка, как и в обычных окнах Windows, расположена в крайней правой части полосы заголовка, нажатие на нее приводит к закрытию панели. Естественно, у панелей без заголовка этой кнопки нет, и пользователь не может самостоятельно закрыть их – соответствующие функции должна обеспечить модель блока.
- `RDS_PAN_F_HIDDEN` – панель скрыта от пользователя. Если указан этот флаг, функция `rdsPANCreate` создаст внутренний объект РДС для работы с панелью, но сама панель в окне подсистемы не появится до тех пор, пока модель блока не скомандует показать ее, вызвав одну из сервисных функций.
- `RDS_PAN_F_PAINTMSG` – модель блока получает уведомления о необходимости перерисовки содержимого панели. Если на панели размещены какие-либо объекты Windows, в этом нет необходимости – таким объектам не нужно сообщать о перерисовке, они сами получают сообщение Windows `WM_PAINT` и могут отреагировать на него. В нашем же случае мы не будем размещать что-либо на панели, вместо этого, мы свяжем с ней область вывода OpenGL. Поскольку оконный объект, связанный с панелью, создается и обслуживается РДС, модель не может вмешаться в его работу и перехватить поступившее ему от операционной системы сообщение о перерисовке. Вместо этого она может указать при создании панели флаг `RDS_PAN_F_PAINTMSG`, и РДС будет транслировать ей эти сообщения.

В этом примере в вызове `rdsPANCreate` указаны все флаги, кроме `RDS_PAN_F_NOBUTTON`. Модель создает скрытую панель с рамкой, заголовком "Прибор" и кнопкой закрытия, причем размер и положение панели пользователь сможет, при желании, изменять. Модель блока будет получать уведомления о необходимости перерисовки панели.

В конце конструктора полю класса `RefreshTimer`, предназначенному для хранения идентификатора таймера для обновления панели в процессе расчета, присваивается нулевое значение – таймер пока не нужен, он будет создан позднее.

Деструктор класса будет включать всего два вызова: удаление созданного объекта-панели функцией `rdsDeleteObject` и удаление таймера обновления панели (если он был создан в процессе работы блока) функцией `rdsDeleteBlockTimer`:

```
// Деструктор класса личной области данных
TOpenGLInstr::~TOpenGLInstr()
{ // Удаление панели
```

```
  if(Panel)
    rdsDeleteObject(Panel);
  // Удаление таймера
  if(RefreshTimer)
    rdsDeleteBlockTimer(RefreshTimer);
}
//================================================
```

Теперь напишем функцию `CreateRefreshTimer`, которая будет создавать таймер для обновления панели или настраивать его параметры, если таймер уже создан (почему мы не создаем таймер в конструкторе блока, будет объяснено позднее). Эта функция будет вызываться при запуске расчета. Пока расчет не запущен, у нас нет необходимости обновлять панель с заданной частотой – если панель будет перекрыта другим окном, а потом снова появится на переднем плане, Windows автоматически пошлет ей сообщение о необходимости перерисовки, которое будет передано в модель блока, поскольку при создании панели был указан флаг `RDS_PAN_F_PAINTMSG`. Если же не обновлять панель во время расчета, изображение блока никак не будет реагировать на изменение входных переменных – пока панель не перекрыта другими окнами и не изменяет свой размер, никаких сообщений о перерисовке она не получает. Можно, конечно, обновлять панель при каждом срабатывании модели блока, но это было бы неоправданной тратой вычислительных ресурсов. Перерисовка окон – одна из самых длительных процедур, и выполнение ее в каждом такте расчета приведет к существенному замедлению работы всей системы. Тем более, что такое частое обновление просто не нужно – обычно для достижения вполне приемлемого качества мультипликации достаточно перерисовывать изображение не чаще десяти раз в секунду.

В данном случае в момент запуска расчета мы средствами РДС создадим таймер, который с заданной периодичностью будет вызывать модель блока в режиме `RDS_BFM_WINREFRESH`. Этот режим вызова специально предназначен для того, чтобы сообщать блоку о необходимости перерисовки всех окон и панелей, которыми он владеет. В реакции на этот вызов мы, со временем, будем перерисовывать трехмерное изображение на панели. В качестве интервала срабатывания таймера возьмем интервал автоматического обновления окна подсистемы, в которой находится блок. Это логично – панель будет находиться в этом же окне, и будет обновляться с той же частотой, что и содержимое этого окна. Таким образом, функция создания таймера будет иметь вид:

```
// Создание таймера обновления панели
void TOpenGLInstr::CreateRefreshTimer(RDS_BHANDLE parent)
{ // Структура для получения параметров окна подсистемы
  RDS_EDITORPARAMETERS WinParams;
  // Определение интервала обновления окна подсистемы
  WinParams.servSize=sizeof(RDS_EDITORPARAMETERS);
  rdsGetEditorParameters(parent,&WinParams);
  // Интервал - в WinParams.RefreshDelay

  // Создание таймера
  RefreshTimer=rdsSetBlockTimer(RefreshTimer,    // Идентификатор
                WinParams.RefreshDelay,          // Интервал
                RDS_TIMERM_LOOP | RDS_TIMERS_WINREF |
                RDS_TIMERF_FIXFREQ,        // Режим и флаги
                TRUE);                     // Запустить таймер
}
//================================================
```

В функцию передается единственный параметр – идентификатор родительской подсистемы блока `parent`, интервал обновления которой будет использован при создании таймера. Этот идентификатор содержится в структуре `RDS_BLOCKDATA`, указатель на которую передается

в модель блока при каждом вызове, поэтому получение его внутри модели не представляет трудностей. Интервал обновления можно получить вместе с другими параметрами окна подсистемы сервисной функцией rdsGetEditorParameters. В эту функцию передается идентификатор подсистемы (parent) и указатель на структуру типа RDS_EDITORPARAMETERS, в которую функция записывает все параметры окна указанной подсистемы. Как и большинство сервисных функций РДС, работающих со структурами, функция rdsGetEditorParameters проверяет соответствие поля servSize переданной структуры ожидаемому размеру этой структуры, и отказывается работать в случае, если ожидаемый размер окажется больше. Это, в большинстве случаев, позволяет предотвратить фатальные сбои и ошибки общей защиты при ошибке разработчика модели, передавшего в функцию неправильный указатель. Таким образом, чтобы функция rdsGetEditorParameters сработала, перед ее вызовом неободимо присвоить полю servSize передаваемой структуры размер этой структуры, то есть sizeof(RDS_EDITORPARAMETERS). После вызова функции структура будет заполнена различными параметрами подсистемы – размерами и положением окна, числом слоев, цветами и т.п. (подробнее она описана приложении А), но нас будет интересовать только поле RefreshDelay – интервал автоматического обновления окна в миллисекундах. Этот параметр и используется при вызове функции rdsSetBlockTimer для создания таймера. Таймер будет циклическим (константа RDS_TIMERM_LOOP), вызывать модель в режиме RDS_BFM_WINREFRESH (константа RDS_TIMERS_WINREF) и его частота не будет снижаться при слишком больших задержках обновления окна (константа RDS_TIMERF_FIXFREQ), поскольку сейчас для нас плавность анимации важнее экономии вычислительных ресурсов. Таймер сразу создается активным (последний параметр функции – TRUE), но считать он будет только в режиме расчета. Можно заметить, что первый параметр функции – идентификатор таймера RefreshTimer, а не NULL, как в моделях с таймерами, рассмотренных ранее (например, на стр. 174). Это сделано для того, чтобы функцию CreateRefreshTimer можно было вызывать и в том случае, если таймер уже создан. Если таймера еще нет, поле класса RefreshTimer будет иметь значение NULL, и функция создаст новый таймер. Если же таймер уже создан, RefreshTimer будет содержать его идентификатор, и вызов rdsSetBlockTimer изменит параметры существующего таймера. Таким образом, перед вызовом функции CreateRefreshTimer не обязательно удалять таймер – это упрощает программу и предохраняет от возможных ошибок.

Разобравшись с созданием таймера, напишем функции сохранения и загрузки положения и размера панели. Будем хранить данные блока в формате INI-файла Windows (см. §2.8.5), то есть в виде строк “<*ключевое_слово*>=<*значение*>”. Начнем с функции сохранения параметров:

```
// Функция сохранения параметров блока
void TOpenGLInstr::SaveText(void)
{ RDS_HOBJECT ini; // Идентификатор вспомогательного объекта

  // При сохранении блока в отдельный файл на диске параметры
  // панели записывать не нужно
  if(rdsGetSystemInt(RDS_GSISAVELOADACTION)==RDS_LS_SAVETOFILE)
    return;

  // Создание вспомогательного объекта для работы с данными
  ini=rdsINICreateTextHolder(TRUE);

  // Параметры записываются в секцию "[Window]"
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"Window");
```

```
    // Получение параметров панели и запись их в объект
    rdsINIWriteInt(ini,"Left",
                   rdsGetObjectInt(Panel,RDS_PAN_LEFT,0));
    rdsINIWriteInt(ini,"Top",
                   rdsGetObjectInt(Panel,RDS_PAN_TOP,0));
    rdsINIWriteInt(ini,"Width",
                   rdsGetObjectInt(Panel,RDS_PAN_WIDTH,0));
    rdsINIWriteInt(ini,"Height",
                   rdsGetObjectInt(Panel,RDS_PAN_HEIGHT,0));
    rdsINIWriteInt(ini,"Visible",
                   rdsGetObjectInt(Panel,RDS_PAN_VISIBLE,0));

    // Сохранение получившегося текста в файл, в который в данный
    // момент идет запись
    rdsCommandObject(ini,RDS_HINI_SAVEBLOCKTEXT);

    // Удаление вспомогательного объекта
    rdsDeleteObject(ini);
}
//================================================
```

Перед тем, как записывать параметры панели, функция проверяет, из-за чего происходит сохранение блока. Если блок сохраняется в составе системы, или копируется в буфер обмена (что тоже вызывает сохранение его параметров), пользователь ожидает, что после загрузки схемы или вставки блока панель появится на том же самом месте, на котором она находилась раньше. Поэтому в этих случаях положение, размер и видимость панели необходимо сохранять. Если же блок записывается в отдельный файл, например, при создании библиотеки блоков, он, вероятнее всего, будет загружен в совершенно другую систему. В этом случае сохранять параметры панели бессмысленно – возможно, ее текущее положение вообще находится за пределами рабочей области подсистемы, в которую будет вставлен блок.

Для получения информации о том, куда в данный момент сохраняется блок, используется сервисная функция `rdsGetSystemInt` с параметром `RDS_GSISAVELOADACTION`. Эта функция предназначена для получения различных глобальных параметров РДС, в данном случае она возвращает целое число, характеризующее выполняющуюся в данный момент операцию сохранения или загрузки. Нас интересует сохранение блока в файл, поэтому возвращаемое значение сравнивается с константой `RDS_LS_SAVETOFILE`, и, в случае равенства, функция завершается – ничего сохранять не надо. В противном случае функцией `rdsINICreateTextHolder` (см. пример на стр. 168) создается вспомогательный объект РДС для работы с текстом, после чего в нем создается секция “Window”. Далее функциями `rdsINIWriteInt` в эту секцию записываются пять целых параметров: координаты левого верхнего угла панели, ее ширина, высота и видимость (1 – для видимой панели, 0 – для скрытой). Параметры читаются непосредственно из объекта-панели `Panel`, который был создан в конструкторе личной области данных блока, при помощи функций `rdsGetObjectInt` с соответствующими параметрами. Независимо от того, видима ли панель в окне подсистемы и открыто ли вообще окно этой подсистемы, объект будет хранить параметры панели, и их можно считывать и устанавливать стандартными функциями для взаимодействия со вспомогательными объектами РДС (см. приложение А). После того, как текст с параметрами панели сформирован во вспомогательном объекте, он сбрасывается туда, куда в данный момент идет запись (в файл, буфер обмена и т.д.), после чего объект уничтожается и функция завершается.

Функция загрузки параметров `LoadText` будет зеркальным отражением функции сохранения (разумеется, без проверки причины загрузки – если в загружаемом тексте есть параметры, их необходимо считать и установить в любом случае).

```
// Функция загрузки параметров блока
void TOpenGLInstr::LoadText(char *text)
{ RDS_HOBJECT ini; // Идентификатор вспомогательного объекта

  // Создание вспомогательного объекта для работы с данными
  ini=rdsINICreateTextHolder(TRUE);

  // Запись в объект полученного от РДС текста с параметрами блока
  rdsSetObjectStr(ini,RDS_HINI_SETTEXT,0,text);

  // Установка "[Window]" в качестве текущей секции и чтение
  // из нее пяти целых параметров
  if(rdsINIOpenSection(ini,"Window"))
    { rdsSetObjectInt(Panel,RDS_PAN_LEFT,0,
                      rdsINIReadInt(ini,"Left",0));
      rdsSetObjectInt(Panel,RDS_PAN_TOP,0,
                      rdsINIReadInt(ini,"Top",0));
      rdsSetObjectInt(Panel,RDS_PAN_WIDTH,0,
                      rdsINIReadInt(ini,"Width",0));
      rdsSetObjectInt(Panel,RDS_PAN_HEIGHT,0,
                      rdsINIReadInt(ini,"Height",0));
      rdsSetObjectInt(Panel,RDS_PAN_VISIBLE,0,
                      rdsINIReadInt(ini,"Visible",0));
    }
  // Удаление вспомогательного объекта
  rdsDeleteObject(ini);
}
//================================================
```

Внутри функции создается вспомогательный объект `ini` для работы с текстом, в этот объект записывается текст для загрузки, полученный моделью из РДС (функция `LoadText` получает его как параметр), в объекте выбирается секция “Window”, и, если она существует, из нее в объект-панель читаются пять тех же целых параметров окна. Каждый параметр считывается из текста функцией `rdsINIReadInt` и передается в панель функцией `rdsSetObjectInt`. После считывания параметров объект `ini` уничтожается.

Теперь можно написать функцию модели. Структуру переменных блока пока определять не будем – сейчас мы создаем простую модель, умеющую создавать панель в окне подсистемы и обновлять ее по таймеру, но не выводящую на эту панель никаких изображений. В модель будет включена реакция на таймер и на действия пользователя с панелью, но эти реакции пока останутся пустыми.

```
// Модель блока с панелью
extern "C" __declspec(dllexport) int RDSCALL OpenGLInstr(
                int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{ // Указатель на личную область, приведенный к нужному типу
  TOpenGLInstr *data=(TOpenGLInstr*)(BlockData->BlockData);
  // Вспомогательная - указатель на структуру параметров при
  // действиях с панелью (будет использована в реакциях)
  RDS_PPANOPERATION param;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Создание личной области данных
        // (при этом в конструкторе будет создана панель)
        BlockData->BlockData=data=new TOpenGLInstr();
```

```
  break;

// Очистка данных блока
case RDS_BFM_CLEANUP:
  // В деструкторе класса панель будет уничтожена
  delete data;
  break;

// Запуск расчета
case RDS_BFM_STARTCALC:
  // Создание таймера обновления
  data->CreateRefreshTimer(BlockData->Parent);
  break;

// Вызов функции настройки или двойной щелчок
// левой кнопки мыши
case RDS_BFM_SETUP:
case RDS_BFM_MOUSEDBLCLICK:
  // Показать панель
  rdsSetObjectInt(data->Panel,RDS_PAN_VISIBLE,0,1);
  break;

// Сохранение параметров блока
case RDS_BFM_SAVETXT:
  data->SaveText();
  break;

// Загрузка параметров блока
case RDS_BFM_LOADTXT:
  data->LoadText((char*)ExtParam);
  break;

// Действия с панелью
case RDS_BFM_BLOCKPANEL:
  // Приведение указателя на структуру, переданного в
  // ExtParam, к нужному типу
  param=(RDS_PPANOPERATION)ExtParam;
  // Разные действия в зависимости от операции с панелью
  switch(param->Operation)
    { // Создание оконного объекта для панели
      case RDS_PANOP_CREATE:
        // Здесь будет инициализация рисования на панели
        break;
      // Уничтожение оконного объекта панели
      case RDS_PANOP_DESTROY:
        // Здесь будет очистка инициализированного
        break;
      // Размер панели изменен
      case RDS_PANOP_RESIZED:
        // Здесь будет реакция на изменение размера
        // и перерисовка панели
        break;
      // Необходимо перерисовать изображение
      case RDS_PANOP_PAINT:
        // Здесь будет перерисовка панели
        break;
    }
  break;
```

```
        // Необходимо обновить окна блока (вызывается таймером)
        case RDS_BFM_WINREFRESH:
           // Здесь будет перерисовка панели
           break;
      }
    return RDS_BFR_DONE;
  }
  //================================================
```

При инициализации блока (`RDS_BFM_INIT`) и очистке его данных (`RDS_BFM_CLEANUP`) , как обычно, создается и удаляется личная область. В момент ее создания (в конструкторе класса) будет создан объект для панели в окне подсистемы, при этом сама панель в окне не появится, даже если окно подсистемы уже создано, поскольку объект в конструкторе создается с флагом `RDS_PAN_F_HIDDEN`. В деструкторе класса, соответственно, этот объект уничтожается, и, если панель видима, она тоже исчезнет из окна.

При запуске расчета (`RDS_BFM_STARTCALC`) вызывается функция `CreateRefreshTimer`, которая либо создает таймер, если это первый ее вызов, либо задает для таймера новый интервал обновления, если таймер уже существует. В функцию передается идентификатор родительской подсистемы блока `BlockData->Parent`, чтобы она считала из параметров этой подсистемы интервал обновления окна и установила для таймера такой же. В отличие от рассматривавшихся ранее моделей с таймерами, в этой модели таймер создается не при инициализации блока (в конструкторе класса), а позже, при запуске расчета. Дело в том, что в режиме редактирования пользователь может изменить частоту обновления окна подсистемы, и единственный способ отследить это – заново считывать интервал обновления каждый раз при входе в режим моделирования или расчета. В данном случае выбран именно режим расчета, поскольку таймер, создаваемый в функции `CreateRefreshTimer`, работает именно в этом режиме.

В конструкторе блока панель создается скрытой, поэтому необходимо дать пользователю возможность открыть ее. Поскольку у данного блока нет каких-либо настраиваемых пользователем параметров, логично использовать функцию настройки, предусмотренную интерфейсом РДС (`RDS_BFM_SETUP`), для показа панели. В режимах моделирования и расчета для этого будет использоваться реакция на двойной щелчок левой кнопкой мыши (`RDS_BFM_MOUSEDBLCLICK`). Разумеется, чтобы все это работало, необходимо в параметрах блока указать наличие функции настройки (при этом лучше изменить ее название на “Открыть окно” или “Показать панель”) и включить реакцию на мышь (рис. 65). Если еще установить в параметрах блока флаг “Двойной щелчок вызывает функцию настройки” (см. рис. 5), панель будет вызываться на экран по двойному щелчку независимо от режима РДС.

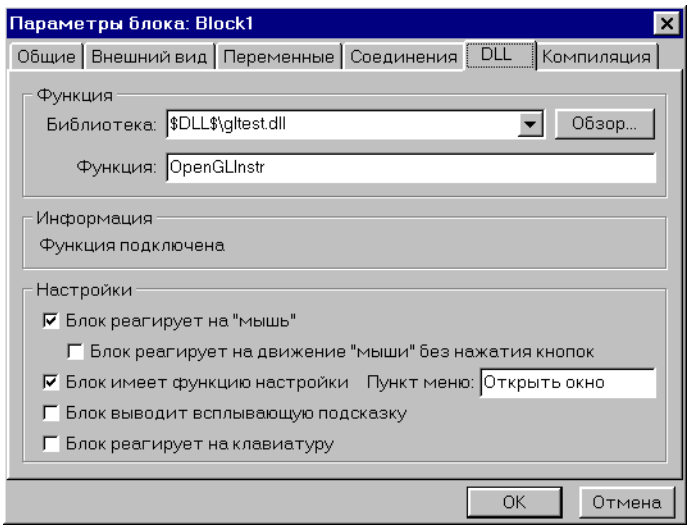


Рис. 65. Параметры блока, создающего панель

Для показа панели в функцию модели введена одинаковая реакция на вызовы `RDS_BFM_SETUP` и `RDS_BFM_MOUSEDBLCLICK`. В ней при помощи функции

rdsSetObjectInt, передающей целое число объекту data->Panel, параметру объекта RDS_PAN_VISIBLE устанавливается значение 1. Это приведет к появлению панели в окне подсистемы и созданию для нее оконного объекта Windows. Если панель уже видима, никаких действий выполнено не будет.

Далее в функции модели следуют реакции на сохранение и загрузку параметров блока, в которых вызываются написанные ранее функции SaveText и LoadText. Функция LoadText устанавливает все параметры панели, включая видимость, согласно ранее записанным значениям, поэтому, если сохранить схему с открытой панелью, при загрузке этой схемы панель будет открыта автоматически.

Для реакции на различные события, связанные с панелями блока, РДС вызывает функцию модели в режиме RDS_BFM_BLOCKPANEL, при этом в параметре ExtParam передается указатель на структуру типа RDS_PANOPERATION (именно для работы с этой структурой в функции модели объявлена вспомогательная переменная param). В поле Operation этой структуры содержится целый идентификатор события, произошедшего с панелью, в поле Panel – указатель на структуру описания самой панели. Структура описания панели, в свою очередь, содержит большой набор параметров панели: идентификатор, положение и размер, флаги, заголовок и т.п. Важнее всего то, что она содержит дескриптор оконного объекта Windows, связанного с панелью – поле Handle типа HWND. Не зная этого дескриптора, нельзя создавать внутри панели дочерние окна, привязывать к панели область вывода трехмерных изображений OpenGL и т.п. Пока мы пишем “скелет” модели, поэтому в данный момент она никак не реагирует на различные события панели. Однако, в модель уже включен оператор switch, анализирующий поле Opeation переданной структуры. Пока внутри него расставлены комментарии, поясняющие, как будет реагировать модель на события, позднее мы заменим их настоящими реакциями.

Всего модель может реагировать на пять действий с панелью, таким образом, поле Operation структуры RDS_PANOPERATION может принимать одно из следующих значений:

- RDS_PANOP_CREATE – для панели создан оконный объект Windows. Модель должна выполнить необходимые действия для создания дочерних оконных объектов, их инициализации и т.п.;
- RDS_PANOP_DESTROY – оконный объект панели сейчас будет уничтожен. Все созданные дочерние объекты, созданные моделью, должны быть уничтожены в этой реакции;
- RDS_PANOP_RESIZED – размер панели изменен. Модель может, при необходимости, перестроить дочерние окна так, чтобы они уложились в панель, изменить размер области вывода и т.п.;
- RDS_PANOP_MOVED – панель перемещена. Как правило, от модели в этом случае не требуется каких-либо действий, но, при необходимости, она может отреагировать и на это;
- RDS_PANOP_PAINT – панель необходимо перерисовать. Этот вызов производится только для панелей, созданных с флагом RDS_PAN_F_PAINTMSG, как в нашем случае.

Наша модель будет реагировать на все действия с панелью, кроме RDS_PANOP_MOVED.

Наконец, последняя в нашей модели – реакция на обновление окон RDS_BFM_WINREFRESH. Она будет вызываться в режиме расчета при срабатывании таймера обновления панели. Сейчас эта реакция пуста, но позже там будет вызов функции, перерисовывающей трехмерное изображение.

Итак, мы получили модель, которая умеет открывать пустую панель в окне подсистемы (рис. 66), дает пользователю перемещать эту панель, закрывать ее и изменять ее размеры, может обновлять панель по таймеру (хотя пока ничего в ней не рисует), а также сохраняет положение и размер панели при сохранении схемы и восстанавливает их при загрузке. Теперь заставим ее строить на панели трехмерное изображение.

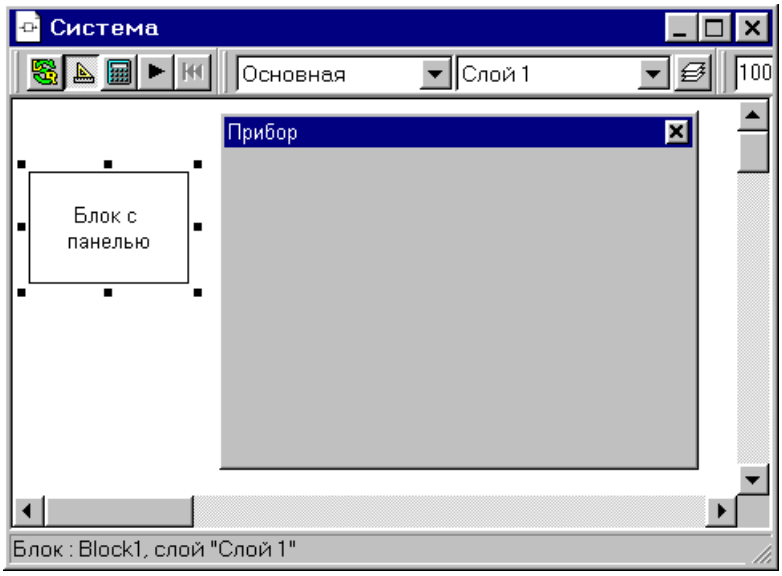


Рис. 66. Блок с пустой панелью

Сделаем из нашего блока индикатор, который будет наглядно отображать три угловых координаты какого-либо объекта. Используя морскую терминологию, назовем эти координаты курсом, дифферентом и креном (рис. 67).

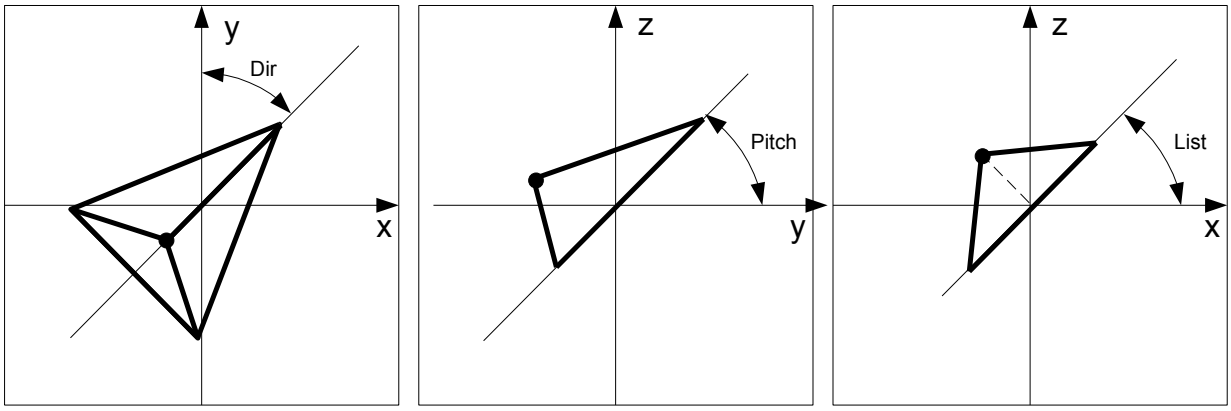


Рис. 67. Угловые координаты объекта

Блок будет иметь три вещественных входа: `Dir` (курс), `List` (крен) и `Pitch` (дифферент). В зависимости от их значений будем поворачивать внутри панели какой-нибудь трехмерный объект на соответствующие углы по трем координатам. Чтобы не усложнять пример, возьмем в качестве объекта-индикатора неправильную треугольную пирамиду, грани которой раскрашены в разные цвета (рис. 68, 69). Нижнюю плоскость пирамиды (треугольник P_1-P_2-P_3) будем считать горизонтальной плоскостью объекта, при этом точка P_1 будет соответствовать носовой части объекта, а отрезок P_2-P_3 – его корме.

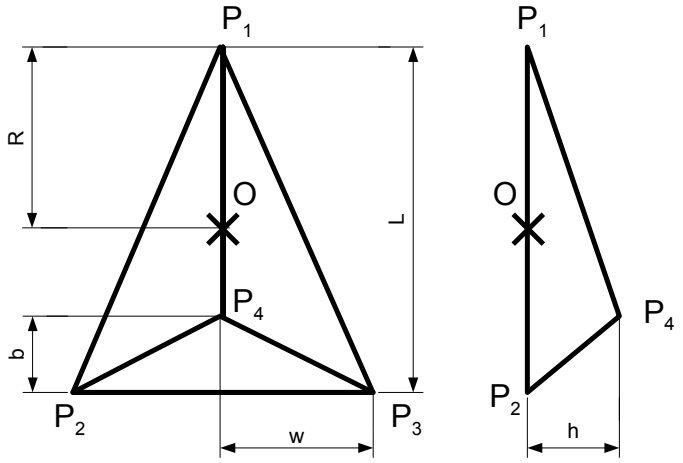


Рис. 68. Центральный объект блока

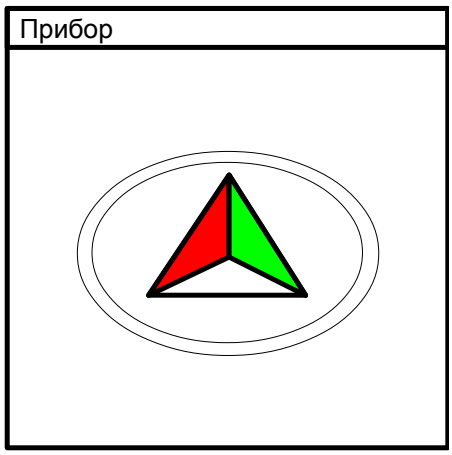


Рис. 69. Внешний вид панели блока

Левая грань пирамиды (P_1-P_2-P_4) будет раскрашена в красный цвет, правая (P_1-P_4-P_3) – в зеленый, задняя (P_2-P_3-P_4) и нижняя (P_1-P_3-P_2) – в белый. Для большей наглядности изображения будем строить его со включенным расчетом освещенности от одного точечного источника (без расчета освещенности трехмерные фигуры с заполненными гранями не выглядят объемными). Точкой “O” на рис. 68 обозначен центр объекта, вокруг этой точки он будет вращаться при изменении угловых координат. Вокруг треугольной пирамиды будут также изображаться две окружности: одна будет лежать в плоскости P_1-P_2-P_3 (и, соответственно, поворачиваться вместе с объектом), а вторая, большего диаметра, в горизонтальной плоскости внешней (“мировой”) системы координат. По взаимному расположению этих окружностей будет легче оценить углы поворота объекта. Точка наблюдения (место расположения условной камеры, через которую пользователь видит трехмерную сцену) будет выбрана так, чтобы объект при нулевом курсе и дифференте был виден сверху с кормы (рис. 69).

Поскольку данная модель будет работать с функциями библиотеки OpenGL, в ее текст необходимо включить файлы заголовков “gl.h” и “glu.h”. Кроме того, нам потребуются некоторые математические функции, поэтому подключим еще и файл “math.h”:

```
#include <windows.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <RdsDef.h>
// Подготовка описаний сервисных функций
#define RDS_SERV_FUNC_BODY GetInterfaceFunctions
// ...
```

В главную функцию DLL этой модели никаких изменений вносить не будем.

Класс личной области данных блока нужно дополнить несколькими полями и функциями. Во-первых, для рисования нам потребуется контекст рисования OpenGL. Во-вторых, действия по настройке OpenGL и рисованию трехмерной фигуры лучше всего вынести в отдельные функции. Класс личной области данных будет иметь следующий вид (изменения выделены жирным):

```
//================================================
// Личная область данных блока
//================================================
class TOpenGLInstr
{ private:
    HGLRC Hrc;  // Контекст OpenGL
```

```
    RDS_TIMERID RefreshTimer;    // Таймер обновления

    // Служебная функция – настройка параметров OpenGL
    BOOL SetupGLParams(HDC *pHdc);
  public:
    RDS_HOBJECT Panel;           // Объект-панель

    // Настройка вывода изображения на панель
    void InitWindow(HWND window);
    // Отключение OpenGL
    void Clear(void);

    // Рисование трехмерной сцены
    void RenderScene(double Dir,double List,double Pitch);

    // Сохранение параметров блока
    void SaveText(void);
    // Загрузка параметров блока
    void LoadText(char *text);

    // Создание таймера обновления
    void CreateRefreshTimer(RDS_BHANDLE parent);

    // Конструктор класса
    TopenGLInstr(void);
    // Деструктор класса
    ~TOpenGLInstr();
};
//=================================================
```

В закрытую область класса (`private`) добавлен контекст OpenGL `Hrc` (он будет использоваться в функциях-членах класса, отвечающих за рисование) и служебная функция `SetupGLParams` для настройки области рисования, освещения и прочих параметров OpenGL (эти настройки выделены в отдельную функцию для улучшения читаемости примера). В открытую область (`public`) добавлено несколько функций, которые будут вызываться непосредственно из модели блока для настройки оконного объекта, его уничтожения, и для рисования изображения.

Кроме внесения изменений в класс личной области, введем еще несколько служебных функций, которые помогут в геометрических расчетах и в построении трехмерного объекта, изображенного на рис. 68 и 69. Можно было бы также сделать их членами класса личной области данных блока, но, поскольку им не нужен доступ к каким-либо данным блока, логичнее сделать их отдельными функциями.

Начнем с функции вычисления вектора, перпендикулярного плоскости треугольника с заданными координатами вершин. Такие векторы единичной длины, называемые “нормалями”, используются в OpenGL для расчета освещения. Поскольку центральный объект блока будет состоять из четырех треугольников, эта функция пригодится при их построении. Для определения нормали будем вычислять векторное произведение векторов, составляющих две стороны треугольника, а потом приводить получившийся вектор к единичной длине.

```
// Вычисление нормали к треугольнику
// p1,p2,p3 – массивы координат вершин {x,y,z}
// norm – массив, в котором возвращаются вычисленные
// координаты нормали
void NormalToTriangle(GLfloat *p1,GLfloat *p2,
```

```
                        GLfloat *p3,GLfloat *norm)
{ double m[3],a[3],b[3],R; // Вспомогательные переменные

  // Символические имена для координат векторов
  #define x 0
  #define y 1
  #define z 2

  // Вычисляем векторное произведение векторов
  // a (p1->p2) и b (p1->p3)
  for(int i=0;i<3;i++)
    { a[i]=p2[i]-p1[i];
      b[i]=p3[i]-p1[i];
    }
  m[x]=a[y]*b[z]-a[z]*b[y];
  m[y]=a[z]*b[x]-a[x]*b[z];
  m[z]=a[x]*b[y]-a[y]*b[x];

  // Длина получившегося вектора
  R=sqrt(m[x]*m[x]+m[y]*m[y]+m[z]*m[z]);

  // Приводим вектор к единичной длине
  for(int i=0;i<3;i++)
    norm[i]=(GLfloat)(m[i]/R);

  // Отмена макроопределений символических имен координат
  #undef x
  #undef y
  #undef z
}
//=================================================
```

Координаты вершин треугольника передаются в функцию в виде трех массивов `p1`, `p2` и `p3`, содержащих координаты вершин. Каждый массив содержит три элемента типа `GLfloat` – вещественного типа, используемого в библиотеке OpenGL. Координата *x* хранится в первом элементе массива, *y* – во втором, *z* – в третьем. Вычисленные координаты вектора нормали записываются в массив `norm` с точно такой же структурой. Для большей ясности текста функции внутри нее введены символические имена `x`, `y` и `z` для индексов массивов координат 0, 1 и 2 соответственно – `p1[x]` выглядит гораздо понятнее, чем `p1[0]`. Для работы с координатами выбраны именно массивы, а не структуры с полями `x`, `y` и `z`, поскольку функции OpenGL могут работать именно с такими массивами. Внутри функции вычисляется векторное произведение сторон треугольника `p1`-`p2` и `p1`-`p3` и длина `R` получившегося вектора, после чего все компоненты вектора делятся на его длину. В результате получается вектор единичной длины, перпендикулярный поверхности треугольника. Следует помнить, что многоугольники в OpenGL – ориентированные, то есть имеют “лицевую” и “изнаночную” стороны. При рисовании объемных фигур чаще всего изображается только лицевая сторона всех многоугольников, изнаночная считается внутренней, и на ее рисование ресурсы не тратятся. Для правильного расчета освещенности вектор нормали каждой грани объемной фигуры должен быть направлен наружу, поэтому при вызове функции `NormalToTriangle` точки треугольника необходимо задавать в правильном порядке, чтобы векторное произведение `p1`-`p2` и `p1`-`p3` также было направлено наружу фигуры. Например, для вычисления нормали к левому переднему (красному) треугольнику P_1-P_2-P_4 необходимо сделать вызов

```
NormalToTriangle(p2, p4, p1, norm);
```

Теперь, имея возможность вычислять нормаль к грани объемной фигуры, напишем функцию, которая будет строить треугольник по трем точкам при помощи команд OpenGL. Будем считать, что библиотека OpenGL уже инициализирована, вывод на панель в окне подсистемы настроен, и все необходимые режимы рисования уже установлены – все это будет рассмотрено позднее. Функция будет достаточно простой:

```
// Построение треугольника p1-p2-p3
void DrawTriangleGL(GLfloat *p1,GLfloat *p2,GLfloat *p3)
{ GLfloat norm[3]; // Массив для вычисления нормали

  // Вычислить нормаль
  NormalToTriangle(p1,p2,p3,norm);
  // Установить нормаль в OpenGL
  glNormal3fv(norm);
  // Задать три точки треугольника
  glVertex3fv(p1);
  glVertex3fv(p2);
  glVertex3fv(p3);
}
//================================================
```

Как и в функцию `NormalToTriangle`, в новую функцию `DrawTriangleGL` координаты вершин треугольника передаются в трех трехэлементных массивах `p1`, `p2` и `p3`. Функция вычисляет вектор нормали к заданному треугольнику и записывает его координаты во вспомогательный массив `norm`, после чего эти координаты передаются библиотеке OpenGL функцией `glNormal3fv` (суффикс "`3fv`" во всех функциях OpenGL указывает на то, что данная функция принимает в качестве входного параметра массив из трех элементов `Glfloat`, именно поэтому во всех написанных нами функциях для хранения координат будут использоваться такие массивы). Затем функцией `glVertex3fv` в библиотеку передаются координаты точек `p1`, `p2` и `p3`. На самом деле, перед установкой координат вершин треугольника и нормалей, необходимо указать библиотеке, что передаваемые точки являются именно вершинами треугольника, а не точками ломаной линии, концами отрезка, и т.п. Для этого служат специальные функции OpenGL `glBegin` и `glEnd`, которые будут вызываться во внешней функции, строящей всю объемную фигуру и вызывающей `DrawTriangleGL`.

В OpenGL в каждой точке многоугольника может быть задана своя нормаль (это позволяет более реалистично отображать кривые поверхности, аппроксимированные многоугольниками), но, в данном случае, для всех точек треугольника будет использован один и тот же вектор нормали, вычисленный в массиве `norm`. Именно поэтому на три вызова `glVertex3fv` в функции приходится всего один вызов `glNormal3fv`. Задав нормаль один раз, можно не повторять задание для каждой очередной точки многоугольника.

Обычно "лицевой" стороной многоугольника считается та, при взгляде на которую вершины следуют друг за другом против часовой стрелки. В связи с этим, при вызове функции `DrawTriangleGL` важно передавать ей точки треугольника в правильном порядке. Функция написана так, что этот порядок совпадает с порядком аргументов функции `NormalToTriangle`. Например, для построения треугольника P_1-P_2-P_4 необходимо сделать следующий вызов:

```
DrawTriangleGL(p2, p4, p1);
```

Еще одна дополнительная функция, которая потребуется для рисования – это функция построения окружности. Изображаемый объект содержит две окружности: подвижную и неподвижную, эти окружности разного диаметра и, для улучшения восприятия, лучше всего их окрасить в разные цвета. Таким образом, функция должна иметь возможность строить окружность произвольного радиуса и произвольного цвета. Для определенности, будем строить окружность в плоскости XY (поворот подвижной окружности будет осуществляться

преобразованиями системы координат перед ее построением, это одно из удобств, предоставляемых библиотекой OpenGL). Окружность будет аппроксимироваться замкнутой ломаной линией с числом звеньев N.

```
// Рисование окружности в OpenGL
void DrawCircleGL(GLfloat R,int N,GLfloat cR,GLfloat cG,GLfloat cB)
{ double aStep=(2.0*M_PI)/N;     // Шаг ломаной по углу, радиан
  // Массивы для задания материала
  GLfloat MaterialAmbDiff[4],
          MaterialSpecular[4]={1.0,1.0,1.0,1.0};
  // Запись цвета материала в массив
  MaterialAmbDiff[0]=cR;
  MaterialAmbDiff[1]=cG;
  MaterialAmbDiff[2]=cB;
  MaterialAmbDiff[3]=1.0; // Непрозрачный
  // Установка отражающих свойств (материала) фигуры
  glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,
               MaterialAmbDiff);
  glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS, 50.0);
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,MaterialSpecular);
  // Нормаль - по оси Z (окружность в плоскости XY)
  glNormal3f(0.0,0.0,1.0);
  // Рисуем замкнутую ломаную линию
  glBegin(GL_LINE_LOOP);
    for(int j=0;j<N;j++)   // Цикл по углу
      { double a=aStep*j;  // Угол
        double x=R*cos(a),y=R*sin(a); // Координаты точки
        glVertex3f((GLfloat)x,(GLfloat)y,0);
      }
  glEnd();
}
//================================================
```

В функцию передаются радиус окружности R, число звеньев ломаной N и три компоненты цвета окружности cR, cG и cB (красный, зеленый и синий соответственно). При задании компонент цвета вещественными числами (например, типа GLfloat, как в данном случае) считается, что интенсивность компоненты изменяется от 0 (минимум) до 1 (максимальная интенсивность). Поскольку мы используем при построении фигур расчет освещенности, для трехмерных объектов недостаточно простого задания цветов их точек и поверхностей. Необходимо задать отражающие свойства этих поверхностей, в терминологии OpenGL – параметры *материала*. Фактически эти свойства аналогичны цвету: например, если материал отражает зеленый цвет на 100%, а красный и синий – на 0%, то при освещении белым светом он будет выглядеть зеленым. Кроме того, может задаваться степень прозрачности материала, также изменяющаяся от 0 (полностью прозрачный) до 1 (непрозрачный). В OpenGL задается несколько параметров материала для разных типов освещения (окружающего, рассеянного и т.п.). Целью данного примера не является подробный разбор особенностей библиотеки OpenGL, поэтому не будем подробно останавливаться на принципах расчета освещенности (все это подробно описано, например, в “OpenGL Programming Guide”).

В начале функции вычисляется угол aStep между условными линиями, соединяющими соседние точки ломаной, изображающей окружность, и ее центр (при числе сегментов ломаной N он будет равен $2\pi/N$) и отводятся два вспомогательных массива, которые будут использованы для задания свойств материала окружности. Далее в первые три элемента массива MaterialAmbDiff записываются переданные компоненты цвета окружности cR, cG и cB. В четвертый элемент записывается число 1, указывающее на непрозрачность данного объекта. После этого занесенные в массивы параметры устанавливаются в качестве текущего материала для разных типов освещения функциями

glMaterialfv и glMaterialf. Теперь все точки многоугольников будут получать эти параметры материала.

Затем функцией glNormal3f устанавливается нормаль к окружности. В отличие от функции glNormal3fv, использовавшейся ранее, функция принимает координаты не в массиве, а в трех отдельных параметрах GLfloat, о чем говорит суффикс "3f" Окружность строится в плоскости XY, поэтому нормаль будем считать направленной вдоль оси Z, то есть координаты единичного вектора будут (0, 0, 1).

Далее начинаем строить ломаную линию. Необходимо указать библиотеке OpenGL, что последовательно передаваемые ей координаты точек должны быть соединены отрезками, причем последняя переданная точка должна соединяться с первой, то есть линия должна быть замкнутой. Построение любой геометрической фигуры в OpenGL начинается с вызова функции glBegin, в параметре которой указывается тип фигуры. В данном случае это константа GL_LINE_LOOP, обозначающая замкнутую ломаную линию. Завершает построение фигуры вызов функции glEnd. Между этими вызовами находится цикл, в котором координаты *x* и *y* точек окружности радиусом R последовательно вычисляются тригонометрическими функциями и передаются в библиотеку вызовом glVertex3f. Для точки окружности с номером i угол a к этой точке будет равен i*aStep, а ее координаты *x* и *y* будут равны R*cos(a) и R*sin(a) соответственно. Поскольку окружность лежит в плоскости XY, координата *z* для всех точек равна нулю. При вызове glBegin был указан параметр GL_LINE_LOOP, поэтому в момент вызова glEnd последняя переданная точка будет соединена отрезком с первой.

Наконец, напишем функцию, которая будет строить центральный объект блока в согласно рис. 68. Все параметры, указанные на рисунке (*R*, *b*, *l*, *w*, *h*) будут параметрами этой функции. Нижний треугольник объекта P_1-P_3-P_2 будет находиться в плоскости XY, причем точка P_1 будет лежать на оси Y, а отрезок P_2-P_3 будет параллелен оси X. Точка O будет совпадать с началом координат. В этой функции точки объекта не будут пересчитываться при изменении углов курса, дифферента и крена – этим будет заниматься преобразование координат OpenGL. Функция будет иметь следующий вид:

```
// Рисование центрального объекта блока
void DrawArrowGL(double R,double l,double w,double h,double b)
{ // Массивы для координат точек фигуры
  GLfloat p1[3],p2[3],p3[3],p4[3];
  // Материал – дно и корма (белый цвет)
  GLfloat MaterialBack[]  = {1.0, 1.0, 1.0, 1.0};
  // Материал – левый борт (красный цвет)
  GLfloat MaterialLeft[]  = {1.0, 0.5, 0.5, 1.0};
  // Материал – правый борт (зеленый цвет)
  GLfloat MaterialRight[] = {0.5, 1.0, 0.5, 1.0};

  // Символические имена для координат точек
  #define x 0
  #define y 1
  #define z 2

  // Вычисление координат точек фигуры
  p1[x]=0;  p1[y]=R;       p1[z]=0;
  p2[x]=-w; p2[y]=R-l;     p2[z]=0;
  p3[x]=w;  p3[y]=p2[y];   p3[z]=0;
  p4[x]=0;  p4[y]=p2[y]+b; p4[z]=h;

  // Построение граней
  glBegin(GL_TRIANGLES);
```

```
    // Свойства материала – общие для всех
    glMaterialf(GL_FRONT,GL_SHININESS,20.0);
    // Свойства материала – корма и дно
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,MaterialBack);
    glMaterialfv(GL_FRONT,GL_SPECULAR,MaterialBack);
    // Нижний треугольник (дно)
    DrawTriangleGL(p1,p3,p2);
    // Задний треугольник (корма)
    DrawTriangleGL(p2,p3,p4);
    // Свойства материала – левый борт
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,MaterialLeft);
    glMaterialfv(GL_FRONT,GL_SPECULAR,MaterialLeft);
    // Левый передний треугольник
    DrawTriangleGL(p2,p4,p1);
    // Свойства материала – правый борт
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,MaterialRight);
    glMaterialfv(GL_FRONT,GL_SPECULAR,MaterialRight);
    // Правый передний треугольник
    DrawTriangleGL(p4,p3,p1);
  glEnd();

  // Отмена макроопределений символических имен координат
  #undef x
  #undef y
  #undef z
}
//=================================================
```

Для всех четырех точек фигуры в начале функции отводятся трехэлементные массивы вещественных чисел `p1`, `p2`, `p3` и `p4` – в этих массивах будут храниться вычисленные координаты точек, они же буду передаваться в функцию `DrawTriangleGL` для построения граней фигуры. Кроме того, описываются три четырехэлементных массива параметров материала: `MaterialBack` для дна и кормы объекта, `MaterialLeft` для левого борта и `MaterialRight` для правого. Эти массивы сразу заполняются значениями, соответствующими белому, красному и зеленому цветам. Далее, как и в уже описанной функции `NormalToTriangle`, вводятся макроопределения для индексов массивов координат, и, согласно значениям `R`, `l`, `w`, `h` и `b`, в массивах `p1`, `p2`, `p3` и `p4` вычисляются координаты всех четырех точек фигуры.

Теперь, когда координаты вычислены, можно приступать к построению граней. Сначала вызывается функция `glBegin` с параметром `GL_TRIANGLES`, указывающим на то, что следующие за этим вызовом тройки точек будут описывать отдельные треугольники. Далее поочередно устанавливаются параметры материала каждой грани и вызывается функция `DrawTriangleGL`, описанная выше. Каждый ее вызов передает библиотеке тройку точек, которые интерпретируются как вершины треугольника соответствующей грани. После того, как все четыре грани построены, вызывается функция `glEnd`.

Все описанные выше функции были вспомогательными, они не относились к классу личной области данных блока. Теперь займемся функциями-членами класса, и начнем с функции настройки вывода OpenGL на панель:

```
// Настройка вывода изображения на панель
void TOpenGLInstr::InitWindow(HWND window)
{ HDC Hdc;      // Контекст устройства Windows
  // Структура для установки формата изображения
  PIXELFORMATDESCRIPTOR pfd = {
     sizeof(PIXELFORMATDESCRIPTOR),   // Размер структуры
     1,                               // Номер версии
     PFD_DRAW_TO_WINDOW |             // Вывод в окно
```

```
            PFD_SUPPORT_OPENGL |        // Поддержка OpenGL
            PFD_DOUBLEBUFFER,           // Двойная буферизация
        PFD_TYPE_RGBA,                  // Формат цвета - RGBA
        24,                             // Глубина цвета - 24 бита
        0,0,0,0,0,0,                    // (здесь не используется)
        0,0,0,0,0,0,0,                  // (здесь не используется)
        32,                             // Z-буфер - 32 бита
        0,0,                            // (здесь не используется)
        PFD_MAIN_PLANE,                 // Главный слой
        0,                              // Зарезервировано
        0,0,                            // Маски слоев (не исп.)
        };
    int PixelFormat;

    // Получение контекста оконного объекта
    Hdc=GetDC(window);

    // Установка формата изображения по структуре pfd
    PixelFormat=ChoosePixelFormat(Hdc,&pfd);
    SetPixelFormat(Hdc,PixelFormat,&pfd);

    // Создание контекста OpenGL
    Hrc=wglCreateContext(Hdc);

    // Установка этого контекста в качестве текущего
    wglMakeCurrent(Hdc,Hrc);
}
//================================================
```

Эта функция получает в качестве параметра дескриптор оконного объекта, созданного РДС для панели. Прежде чем выводить что-либо в это окно при помощи OpenGL, необходимо настроить формат изображения окна, а для этого необходимо предварительно получить контекст рисования этого окна при помощи функции `GetDC`. Контекст записывается в локальную переменную `Hdc` (хотя этот контекст нам и потребуется позже в других функциях, лучше каждый раз получать его вызовом `GetDC` для оконного объекта панели). Теперь для полученного контекста можно установить формат изображения при помощи функций Windows API `ChoosePixelFormat` и `SetPixelFormat`, при этом сами параметры изображения задаются в структуре `pfd` типа `PIXELFORMATDESCRIPTOR` (подробные описания функций и структуры см. в Windows API). В данном случае важно, что мы запрашиваем поддержку OpenGL (флаг `PFD_SUPPORT_OPENGL`), двойную буферизацию (флаг `PFD_DOUBLEBUFFER`), двадцатичетырехбитный цвет формата RGBA (то есть цвет каждого цветового канала задается отдельно одни байтом) и тридцатидвухбитный Z-буфер. Двойная буферизация позволяет иметь два буфера изображения: в одном, невидимом, производится рисование, а другой в это время изображается на экране. Когда изображение нарисовано полностью, буферы меняются местами – новое изображение появляется на экране, а буфер со старым изображением становится невидимым и в нем можно готовить к показу следующий кадр. Такой режим работы существенно улучшает анимацию, убирая мигания изображения при перерисовке, т.к. она производится в невидимом буфере.

В Z-буфере для каждой точки изображения хранится “глубина” – третья координата, перпендикулярная плоскости проекции. Этот буфер используется для отсечения частей трехмерных объектов, перекрытых другими объектами. Перед записью очередной точки объекта в буфер изображения ее глубина сравнивается с уже записанной в Z-буфере. Если новая точка располагается дальше от переднего плана, чем уже записанная, она отбрасывается, если ближе – записывается в буфер изображения и ее глубина записывается в Z-буфер. Чем выше разрядность Z-буфера, тем точнее будут сравниваться точки по глубине,

и тем правильнее будет происходить рисование перекрывающихся (особенно, близко расположенных или пересекающихся) объектов. Тридцати двух бит хватит для большинства случаев, тем более, для такого простого изображения, как наше.

После того, как формат изображения в контексте рисования установлен, при помощи функции `wglCreateContext` создается контекст OpenGL, который записывается в поле класса `Hrc` – он нам понадобится в функциях рисования. Все функции рисования OpenGL работают с текущим контекстом, поэтому, перед их вызовом, необходимо установить какой-либо контекст рисования в качестве текущего. В конце этой функции мы делаем текущим только что созданный контекст.

Перед завершением работы с трехмерной графикой необходимо уничтожить созданный контекст OpenGL. Для этого напишем функцию `Clear` и будем вызывать ее из модели перед уничтожением оконного объекта:

```
// Отключение OpenGL
void TOpenGLInstr::Clear(void)
{ // Отключение текущего контекста OpenGL
  wglMakeCurrent(NULL,NULL);
  // Уничтожение созданного контекста
  wglDeleteContext(Hrc);
}
//================================================
```

В этой функции всего два вызова: сначала при помощи `wglMakeCurrent` с параметром `NULL` отключается текущий контекст (чтобы не удалить выбранный), а затем функцией `wglDeleteContext` созданный ранее в `InitWindow` контекст удаляется.

Теперь запишем служебную функцию, которая будет выполнять настройки OpenGL перед рисованием:

```
// Служебная функция – настройка параметров OpenGL
BOOL TOpenGLInstr::SetupGLParams(HDC *pHdc)
{ // Параметры перспективной проекции
  const GLfloat zNear=0.1;      // Ближняя плоскость отсечения
  const GLfloat zFar=1000.0;    // Дальняя плоскость отсечения
  const GLfloat vAngle=30.0;    // Угол зрения

  HDC Hdc; // Контекст устройства (окна) Windows

  // Расположение источника освещения
  const double lightDistance=500.0;   // Расстояние до объекта
  const double lightRotation=-55.0;   // Азимут в градусах
  const double lightPitch=45.0;       // Угол места в градусах

  // Вспомогательные переменные
  RDS_PANDESCRIPTION descr;
  int width,height;
  GLfloat array[4];
  double l_r,l_p;

  // Получение описания панели и дескриптора ее окна
  descr.servSize=sizeof(RDS_PANDESCRIPTION);
  if(!rdsPANGetDescr(Panel,&descr))   // Не удалось
    return FALSE;
  if(descr.Handle==NULL)   // Нет оконного объекта
    return FALSE;
  // Проверка высоты и ширины панели
  if(descr.Height==0 || descr.Width==0)
    return FALSE; // Негде рисовать
```

```
// Получение контекста окна и передача в вызвавшую функцию
// его через параметр-указатель
Hdc=GetDC(descr.Handle);
if(pHdc) *pHdc=Hdc;

// Установка контекста Hrc, созданного при инициализации,
// в качестве текущего
if(!wglMakeCurrent(Hdc,Hrc))
  return FALSE;     // Не удалось

// Установка различных параметров, влияющих на качество рисования
// и способ закраски многоугольников
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glHint(GL_POLYGON_SMOOTH_HINT,GL_FASTEST);
glEnable(GL_POINT_SMOOTH);
glBlendFunc (GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

glClearColor(0.0f,0.0f,0.0f,1.0f);  // Цвет фона - черный
glClearDepth(1.0);  // Значение для очистки Z-буфера

//----------- Установка освещения ----------
// Общее рассеянное освещение
array[0]=array[1]=array[2]=0.5;     // Белый, интенсивность 0.5
array[3]=1.0;
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,array);
// Вычисление координат источника света
l_r=(lightRotation*M_PI)/180.0;
l_p=(lightPitch*M_PI)/180.0;
array[0]=(GLfloat)(lightDistance*cos(l_p)*sin(l_r));     // X
array[1]=(GLfloat)(lightDistance*sin(l_p));              // Y
array[2]=(GLfloat)(lightDistance*cos(l_p)*cos(l_r));     // Z
array[3]=0.0;
glLightfv(GL_LIGHT0,GL_POSITION,array);
// Рассеянное освещение от источника отсутствует
array[0]=array[1]=array[2]=0.0; // Отсутствует - интенсивность 0
array[3]=1.0;
glLightfv(GL_LIGHT0,GL_AMBIENT,array);
// Другие виды освещения - белый цвет
array[0]=array[1]=array[2]=0.5;     // Белый, интенсивность 0.5
glLightfv(GL_LIGHT0,GL_DIFFUSE,array);
glLightfv(GL_LIGHT0,GL_SPECULAR,array);
// Включение источника света
glEnable(GL_LIGHT0);
// Разрешение расчета освещения
glEnable(GL_LIGHTING);
// Режим расчета "затенения"
glShadeModel(GL_SMOOTH);
// Способ расчета освещения
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
//-----------------------------------------

// Установка области окна для рисования
glViewport(0,0,descr.Width,descr.Height);

// Настройка перспективной проекции на эту область
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
  gluPerspective(vAngle,
           (GLfloat)descr.Width/(GLfloat)descr.Height,
           zNear,zFar);

  // Переключение на матрицу моделей (для рисования объектов)
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  return TRUE; // Установка параметров успешно завершена
}
//================================================
```

Функция принимает единственный параметр – указатель на контекст устройства Windows. Через него она вернет в вызвавшую функцию контекст окна панели, на которой будет рисоваться трехмерное изображение. В вызвавшей функции, конечно, можно было бы получить этот контекст при помощи вызова `GetDC`, но, поскольку `SetupGLParams` все равно его получает, можно этим воспользоваться. Возвращает эта функция логическое значение, указывающее на успешность установки параметров. Если параметры установить не удалось (например, при невидимой панели, не имеющей связанного оконного объекта), пытаться что-либо рисовать бессмысленно.

Большая часть функции состоит из вызовов OpenGL для установки режимов, настройки освещения и перспективной проекции и т.п. Короткие пояснения к этим вызовам вставлены непосредственно в текст функции, мы не будем разбирать их более подробно. Исчерпывающая информация по этим функциям находится в описании OpenGL. Детально рассмотрим только привязку области вывода OpenGL к оконному объекту панели.

Сразу после описаний вспомогательных переменных вызывается сервисная функция РДС `rdsPANGetDescr`, которая записывает описание панели `Panel` в структуру `descr` типа `RDS_PANDESCRIPTION`. В этой структуре много полей, соответствующих различным параметрам панели (см. приложение А), но нас будут интересовать только три: дескриптор окна `Handle`, ширина окна `Width` и его высота `Height`. Прежде всего, мы сравниваем полученный дескриптор с `NULL` – если дескриптор нулевой, окно для панели еще не создано, и настраивать OpenGL и рисовать что-либо бессмысленно (функция возвращает `FALSE`). В противном случае высота и ширина оконного объекта также сравниваются с нулем – если хотя бы один из размеров окна нулевой, рисовать будет негде. Следует помнить, что для панелей с рамкой создаваемый внутри них оконный объект меньше самой панели на размер рамки, при этом `Width` и `Height` в структуре `RDS_PANDESCRIPTION` всегда указывают именно размер оконного объекта, а не самой панели. Вызов `glViewport`, устанавливающий область рисования в окне, задает в качестве верхнего левого угла прямоугольной области точку (0, 0), а в качестве правого нижнего – (`descr.Width`, `descr.Height`). Это полный размер внутреннего оконного объекта панели.

Последняя функция-член класса личной области данных блока, которую мы опишем – функция построения трехмерного изображения (часто называемого “сценой”) `RenderScene`:

```
// Рисование трехмерной сцены
void TOpenGLInstr::RenderScene(double Dir,double List,double Pitch)
{ HDC Hdc; // Контекст устройства Windows
  // Угол места камеры в градусах
  const GLfloat Camera_Pitch=-50.0;
  // Расстояние до камеры
  const GLfloat Camera_Distance=300.0;

  // Настройка параметров OpenGL и получение контекста окна
  if(!SetupGLParams(&Hdc))
    return;
```

```
    // Очистка буфера изображения и Z-буфера
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Записать матрицу трансформаций в стек
    glPushMatrix();
    // Трансформации камеры
    glTranslatef(0.0,0.0,-Camera_Distance);  // Отодвигаем от камеры
    glRotatef(Camera_Pitch,1,0,0);           // Поворачиваем
    // Рисование неподвижной зеленой (0,1,0) окружности
    DrawCircleGL(60,100,0.0,1.0,0.0);

    if(Dir!=DoubleErrorValue &&
       List!=DoubleErrorValue &&
       Pitch!=DoubleErrorValue) // Можно рисовать
      { // Поворот на угол курса
        if(Dir!=0.0)
          glRotatef(Dir,0,0,1);
        // Поворот на угол дифферента
        if(Pitch!=0.0)
          glRotatef(Pitch,1,0,0);
        // Поворот на угол крена
        if(List!=0.0)
          glRotatef(-List,0,1,0);

        // Рисование подвижной белой (1,1,1) окружности
        DrawCircleGL(55,100,1.0,1.0,1.0);
        // Рисование центрального объекта
        DrawArrowGL(50,80,30,10,10);
      }
    // Восстановить матрицу трансформаций из стека
    glPopMatrix();
    // Завершить незавершенное рисование, если нужно
    glFlush();
    // Поменять местами видимый и рабочий буферы
    SwapBuffers(Hdc);
  }
  //================================================
```

Функция принимает три вещественных параметра: `Dir` (курс), `List` (крен) и `Pitch` (дифферент), они используются для поворота центрального объекта при рисовании. Внутри функции прежде всего вызывается `SetupGLParams`, настраивающая параметры OpenGL и устанавливающая текущий контекст рисования. Если параметры настроить не удалось, функция вернет `FALSE`, и `RenderScene` немедленно завершится. Конечно, можно было бы не настраивать *все* параметры OpenGL при каждом рисовании, но целью этого примера является демонстрация принципиальной возможности работы с OpenGL на панелях в окне подсистемы, а не оптимизация рисования. К тому же, без этой оптимизации модель блока будет значительно проще, что сделает пример более понятным.

Сразу после настройки функция очищает рабочий буфер изображения и Z-буфер, заполняя их значениями по умолчанию. Эти значения задаются в `SetupGLParams`: для буфера изображения – черный цвет, для Z-буфера – максимально возможная дальность (1.0). После этого рабочая матрица OpenGL (в данный момент – матрица, определяющая трансформации системы координат) помещается в стек функцией `glPushMatrix`. Стек матриц – удобная возможность OpenGL, позволяющая быстро возвращаться к исходной системе координат. Например, если нужно нарисовать какой-то повернутый объект в стороне от начала текущей системы координат, можно поместить матрицу в стек, затем передвинуть

систему координат, повернуть ее, нарисовать объект в начале координат (он будет смещенным и повернутым), а затем, вместо того, чтобы поворачивать и перемещать систему координат обратно, можно просто извлечь матрицу из стека, что приведет к немедленному возврату к старой системе координат. Помещая исходную систему координат в стек перед различными трансформациями, мы получаем возможность быстро вернуться к исходной.

Первыми выполняются трансформации "камеры", то есть точки наблюдения. Функция `DrawArrowGL`, которую мы написали ранее, строит объект в начале координат, поэтому, если систему координат не переместить, точка наблюдения (начало координат) будет находиться внутри объекта. Чтобы наблюдать объект со стороны, мы перемещаем систему координат на расстояние `Camera_Distance` вдоль оси Z в отрицательном направлении функцией OpenGL `glTranslatef`, а затем поворачиваем ее на угол `Camera_Pitch` вокруг оси X функцией `glRotatef` (`Camera_Distance` и `Camera_Pitch` – константы, описанные в начале функции). Теперь "камера" смотрит на начало координат, где будет построен объект, сзади и немного сверху.

Далее вызовом `DrawCircleGL` строится неподвижная окружность зеленого цвета (красный и синий компоненты цвета – нулевые, зеленый – единица) радиусом 60. Эта окружность лежит в горизонтальной плоскости и не поворачивается вместе с объектом, поэтому для нее никаких трансформаций не требуется.

Теперь нужно нарисовать объект и подвижную окружность, повернутые согласно трем заданным угловым координатам. Прежде всего, нужно сравнить полученные функцией значения углов со специальным значением, сигнализирующим об ошибке вычислений. Это значение получено из РДС в главной функции DLL и помещено в глобальную переменную `DoubleErrorValue`. Такие проверки, как уже не раз отмечалось, нужно выполнять для всех значений, которые передаются из других блоков. Если все три значения углов – допустимые, можно приступать к рисованию.

Для поворота системы координат, в которой будут строиться объект и окружность, используются функции OpenGL `glRotatef`. Каждый из трех поворотов выполняется вокруг определенной оси координат – координаты вектора оси поворота задают три последних параметра функции. Поворот на угол курса выполняется вокруг оси Z, на угол дифферента – вокруг оси X, на угол крена – вокруг оси Y, это в точности соответствует рис. 67. После поворотов вызывается функция `DrawCircleGL` для построения белой окружности радиусом 55, и `DrawArrowGL` для построения центрального объекта (размеры объекта, как и радиусы окружностей и расстояние до точки наблюдения, подобраны вручную).

После того, как изображение построено, мы возвращаемся к исходной системе координат, извлекая ее из стека функцией `glPopMatrix`, завершаем все незавершенные операции рисования функцией `glFlush` и меняем местами рабочий буфер с отображаемым буфером функцией `SwapBuffers`. После этого то, что мы только что нарисовали, будет изображаться на панели. Старое изображение, показывавшееся до этого, оказывается в рабочем буфере – оно будет стерто и заменено новым кадром при следующем вызове.

Теперь все функции класса написаны – можно добавить в модель блока их вызовы. Но прежде нужно добавить в блок входы, через которые он будет получать значения курса, дифферента и крена:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	Dir	double	8	Вход
10	List	double	8	Вход
18	Pitch	double	8	Вход

Теперь добавим в функцию модели новые вызовы и переменные. Изменения, как всегда, выделены жирным:

```
// Модель блока с панелью
extern "C" __declspec(dllexport) int RDSCALL OpenGLInstr(
                int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Dir (*((double *)(pStart+2)))
#define List (*((double *)(pStart+10)))
#define Pitch (*((double *)(pStart+18)))
  // Указатель на личную область, приведенный к нужному типу
  TOpenGLInstr *data=(TOpenGLInstr*)(BlockData->BlockData);
  // Вспомогательная - указатель на структуру параметров при
  // действиях с панелью (будет использована в реакциях)
  RDS_PPANOPERATION param;

  switch(CallMode)
    { // Инициализация блока
      case RDS_BFM_INIT:
        // Создание личной области данных
        // (при этом в конструкторе будет создана панель)
        BlockData->BlockData=data=new TOpenGLInstr();
        break;

      // Очистка данных блока
      case RDS_BFM_CLEANUP:
        // В деструкторе класса панель будет уничтожена
        delete data;
        break;

      // Проверка типа переменных
      case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDDD}"))
          return RDS_BFR_BADVARSMSG;
        return RDS_BFR_DONE;

      // Запуск расчета
      case RDS_BFM_STARTCALC:
        // Создание таймера обновления
        data->CreateRefreshTimer(BlockData->Parent);
        break;

      // Вызов функции настройки или двойной щелчок
      // левой кнопки мыши
      case RDS_BFM_SETUP:
      case RDS_BFM_MOUSEDBLCLICK:
        // Показать панель
        rdsSetObjectInt(data->Panel,RDS_PAN_VISIBLE,0,1);
        break;

      // Сохранение параметров блока
      case RDS_BFM_SAVETXT:
```

```
            data->SaveText();
            break;

          // Загрузка параметров блока
          case RDS_BFM_LOADTXT:
            data->LoadText((char*)ExtParam);
            break;

          // Действия с панелью
          case RDS_BFM_BLOCKPANEL:
            // Приведение указателя на структуру, переданного в
            // ExtParam, к нужному типу
            param=(RDS_PPANOPERATION)ExtParam;
            // Разные действия в зависимости от операции с панелью
            switch(param->Operation)
              { // Создание оконного объекта для панели
                case RDS_PANOP_CREATE:
                  // Настройка вывода изображения на панель
                  data->InitWindow(param->Panel->Handle);
                  break;
                // Уничтожение оконного объекта панели
                case RDS_PANOP_DESTROY:
                  // Отключение OpenGL
                  data->Clear();
                  break;
                // Размер панели изменен
                case RDS_PANOP_RESIZED:
                  // При изменении размера - просто перерисовка
                  data->RenderScene(Dir,List,Pitch);
                  break;
                // Необходимо перерисовать изображение
                case RDS_PANOP_PAINT:
                  data->RenderScene(Dir,List,Pitch);
                  break;
              }
            break;

          // Необходимо обновить окна блока (вызывается таймером)
          case RDS_BFM_WINREFRESH:
            // Перерисовка изображения
            data->RenderScene(Dir,List,Pitch);
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef Pitch
    #undef List
    #undef Dir
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=================================================
```

Теперь, когда у блока появились статические переменные, в его модель добавлена обычная реакция на вызов в режиме RDS_BFM_VARCHECK для проверки допустимости их типа.

Изменения также произошли в реакциях на вызовы RDS_BFM_BLOCKPANEL и RDS_BFM_WINREFRESH.

В режиме RDS_BFM_BLOCKPANEL (при получении от РДС сообщения об операции с панелью) модель теперь выполняет следующие действия:

- при создании окна внутри панели (RDS_PANOP_CREATE) вызывается функция InitWindow, настраивающая формат изображения в окне и создающая для окна контекст OpenGL;
- перед уничтожением окна (RDS_PANOP_DESTROY) вызывается функция Clear, уничтожающая ранее созданный контекст OpenGL;
- при изменении размеров панели (RDS_PANOP_RESIZED) и получении сообщения о необходимости перерисовки (RDS_PANOP_PAINT) вызывается функция рисования изображения RenderScene, в параметрах которой передаются входы блока.

В режиме RDS_BFM_WINREFRESH (при срабатывании таймера в режиме расчета) снова вызывается функция RenderScene с входами блока в качестве параметров.

Таким образом, в режиме расчета получившийся блок постоянно отображает три угловые координаты, поступающие ему на входы, поворачивая треугольную пирамиду и окружность относительно неподвижной “камеры” (рис. 70).

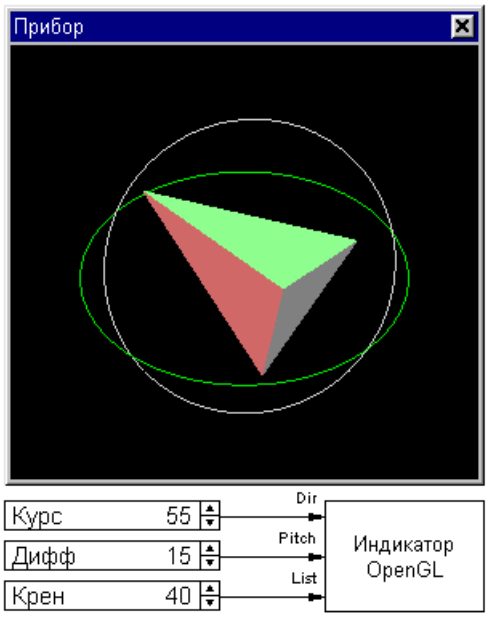


Рис. 70. Работающий блок-индикатор

§2.11. Отображение всплывающих подсказок к блокам

Рассматривается вывод различных текстов во всплывающих подсказках к блокам. Приводятся примеры отображения в подсказке настроечных параметров блока-генератора и координат ближайшей к курсору точки графика.

Всплывающие подсказки – это окна с текстом, появляющиеся через некоторое время после наведения курсора на какой-либо элемент окна: кнопку, поле ввода и т.п. Они широко используются в Windows для вывода пояснений к различным объектам, поскольку текст, показываемый таким образом, появляется только при необходимости и не занимает места на экране в остальное время. РДС позволяет моделям выводить всплывающие подсказки к обслуживаемым этими моделями блокам, причем текст подсказки, время ее нахождения на экране и некоторые другие параметры могут задаваться программно. Если в параметрах блока разрешен вывод всплывающей подсказки (см. соответствующий флаг в окне на рис. 65), при задержке курсора над изображением этого блока РДС вызовет его модель в режиме RDS_BFM_POPUPHINT, передав ей указатель на структуру RDS_POPUPHINTDATA, содержащую различные параметры, связанные с выводом подсказки: координаты курсора мыши, текущий масштаб подсистемы, временной интервал гашения подсказки и т.п. (см. стр. 259). Некоторые параметры в этой структуре модель может изменить, например, увеличить или уменьшить задержку гашения подсказки, или указать на необходимость погасить ее немедленно при перемещении курсора. Установив текст подсказки при помощи сервисной функции rdsSetHintText и, при необходимости, изменив параметры в переданной структуре, модель возвращает управление РДС, после чего подсказка появляется на экране. Если модель не установит никакого текста функцией rdsSetHintText, подсказка выведена не будет.

Следует помнить, что вызов rdsSetHintText устанавливает текст подсказки только внутри реакции модели на сообщение RDS_BFM_POPUPHINT. Во всех остальных реакциях модели вызов rdsSetHintText будет проигнорирован. Таким образом, нельзя установить текст подсказки один раз, например, при создании блока. Этот текст должен передаваться в РДС каждый раз перед выводом подсказки. Поскольку всплывающая подсказка, как правило, должна зависеть от текущего состояния и режима работы блока, такая схема оправдана, так как текст подсказки может меняться от показа к показу.

Ранее (стр. 124) мы рассматривали блок, который, в зависимости от настроек, может выводить на выход синусоидальный и косинусоидальный сигнал или прямоугольные импульсы заданной скважности. Сделаем для этого блока всплывающую подсказку, которая будет отображать параметры блока, а также, при необходимости, сообщать об отсутствии доступа к переменной времени “DynTime”, необходимой блоку для работы.

Добавим в класс личной области данных блока TTestGenData новую функцию PopupHint, которая будет вызываться из модели и формировать всплывающую подсказку.

```
//====== Класс личной области данных ======
class TTestGenData
{ public:
    int Type;               // Тип (0-sin,1-cos,2-прямоугольные)
    double Period;          // Период
    double Impulse;         // Длительность импульса

    RDS_PDYNVARLINK Time;   // Связь с динамической
                            // переменной времени

    int Setup(void);        // Функция настройки
    void SaveText(void);    // Сохранение параметров
    void LoadText(char *text);// Загрузка параметров
    void PopupHint(void);   // Всплывающая подсказка
    TTestGenData(void)      // Конструктор класса
      { Type=0; Period=1.0; Impulse=0.5;
        // Подписка на динамическую переменную времени
        Time=rdsSubscribeToDynamicVar(RDS_DVPARENT,
                                      "DynTime",
                                      "D",
                                      TRUE);
      };
    ~TTestGenData(void)     // Деструктор класса
      { // Прекращение подписки
        rdsUnsubscribeFromDynamicVar(Time);
      };
};
//================================================
```

В функцию модели блока необходимо добавить реакцию на вызов в режиме RDS_BFM_POPUPHINT:

```
        //      ...
        // Очистка
        case RDS_BFM_CLEANUP:
          data=(TTestGenData*)(BlockData->BlockData);
          delete data;
          break;
        // Всплывающая подсказка
        case RDS_BFM_POPUPHINT:
          data=(TTestGenData*)(BlockData->BlockData);
          data->PopupHint();
          break;
```

```
        // Проверка типа переменных
        case RDS_BFM_VARCHECK:
          if(strcmp((char*)ExtParam,"{SSD}")==0)
            return RDS_BFR_DONE;
          return RDS_BFR_BADVARSMSG;
        //      ...
```

В данном случае нам не нужны параметры из структуры RDS_POPUPHINTDATA, поэтому в реакции на RDS_BFM_POPUPHINT мы игнорируем указатель на структуру, переданный в ExtParam, и не передаем ничего в функцию PopupHint.

Функция вывода всплывающей подсказки будет выглядеть следующим образом:

```
// Вывод всплывающей подсказки
void TTestGenData::PopupHint(void)
{ // Есть ли доступ к переменной времени?
  if(Time==NULL || Time->Data==NULL)  // Доступа нет
    rdsSetHintText("ОШИБКА: в схеме нет переменной DynTime");
  else     // Доступ есть
    { char buf[200];  // В этом буфере будем формировать текст
      switch(Type)
        { case 0:     // Синус
            sprintf(buf,"Генератор - синус\nПериод: %lf",Period);
            break;
          case 1:     // Косинус
            sprintf(buf,"Генератор – косинус\n"
                        "Период: %lf",Period);
            break;
          case 2:     // Импульсы
            sprintf(buf,"Генератор – прямоугольные импульсы\n"
                        "Период: %lf\n"
                        "Длительность импульса: %lf",
                        Period,Impulse);
            break;
          default:    // Недопустимый тип
            return;
        }
      // Установка текста подсказки
      rdsSetHintText(buf);
    }
}
//================================================
```

Прежде всего функция проверяет доступ к динамической переменной времени. Если его нет, она устанавливает в качестве текста всплывающей подсказки сообщение об ошибке при помощи функции rdsSetHintText, и на этом ее работа завершается.

Если же доступ к динамической переменной есть, функция формирует текст подсказки с названием типа генерируемого сигнала и его параметрами. Текст записывается во вспомогательный массив buf стандартной библиотечной функцией форматированного вывода sprintf (для ее использования необходимо включить в исходный текст модели заголовочный файл “stdio.h”). Текст, кроме названия сигнала, содержит значение его периода и, если поле Type равно двум, еще и длительность прямоугольного импульса. Период и длительность импульса – вещественные числа двойной точности (тип double), поэтому для их вывода используется формат “%lf”. Далее сформированный в buf текст передается в РДС для использования в качестве подсказки. Символы перевода строки “\n” в тексте служат для разбиения подсказки на строки.

Теперь, если установить в параметрах блока на вкладке “DLL” флаг “блок выводит всплывающую подсказку” (см. рис. 65), при наведении курсора мыши на блок через некоторое время будет появляться окно подсказки со сформированным текстом (рис. 71).

Использование формата “%lf” для вывода периода и длительности импульса в данном случае выглядит не очень удачным, поскольку в обоих числах выведено слишком много незначащих нулей. Явно указать число знаков после десятичной точки затруднительно, поскольку заранее неизвестен масштаб чисел, с которыми будет работать пользователь. Здесь было бы желательно автоматически подбирать формат вывода под введенные пользователем числа. К счастью, в РДС есть подобная сервисная функция, которой мы и воспользуемся для улучшения внешнего вида подсказки. Переделаем функцию `PopupHint` следующим образом:

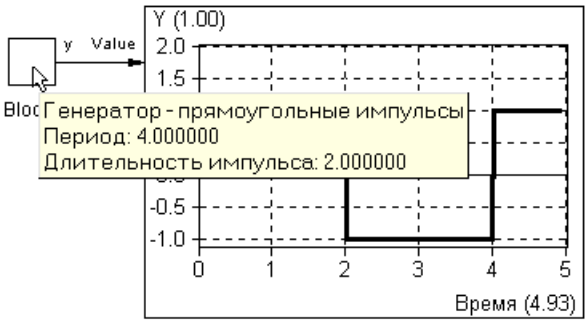


Рис. 71. Всплывающая подсказка к генератору

```
// Вывод всплывающей подсказки
void TTestGenData::PopupHint(void)
{ // Есть ли доступ к переменной времени?
  if(Time==NULL || Time->Data==NULL)  // Доступа нет
    rdsSetHintText("ОШИБКА: в схеме нет переменной DynTime");
  else     // Доступ есть
    { char *str=NULL,
           *period=rdsDtoA(Period,-1,NULL),
           *impulse=rdsDtoA(Impulse,-1,NULL);
      switch(Type)
        { case 0:     // Синус
            str=rdsDynStrCat("Генератор - синус\nПериод: ",
                             period,FALSE);
            break;
          case 1:     // Косинус
            str=rdsDynStrCat("Генератор – косинус\nПериод: ",
                             period,FALSE);
            break;
          case 2:     // Импульсы
            str=rdsDynStrCat("Генератор – прямоугольные импульсы\n"
                        "Период: ",period,FALSE);
            rdsAddToDynStr(&str,"\nДлительность импульса: ",FALSE);
            rdsAddToDynStr(&str,impulse,FALSE);
            break;
        }
      // Установка текста подсказки
      rdsSetHintText(str);
      // Освобождение памяти, занятой динамическими строками
      rdsFree(str);
      rdsFree(period);
      rdsFree(impulse);
    }
}
//=================================================
```

Начало новой функции не отличается от прежнего варианта: при отсутствии доступа к переменной времени в качестве текста подсказки устанавливается сообщение об ошибке. Далее начинаются различия: если раньше мы формировали текст подсказки во вспомогательном массиве `buf`, теперь для этого будут использоваться сервисные функции РДС, которые сами отводят память под строки. Для хранения указателя на динамически отведенную строку с текстом подсказки будем использовать вспомогательную переменную

str. Двум другим вспомогательным переменным, period и impulse, сразу же присваиваются указатели на динамические строки, в которые преобразуются вещественные значения Period и Impulse соответственно. Для преобразования используется сервисная функция РДС rdsDtoA, принимающая три параметра: преобразуемое вещественное число, желаемое число знаков после десятичной точки и указатель на целую переменную, в которую нужно записать длину получившейся строки. Вместо числа знаков после десятичной точки мы в обоих вызовах передаем число –1, указывающее функции на необходимость самостоятельно подобрать точность для преобразования числа и отбросить все незначащие нули в дробной части. Длина получившейся строки нас не интересует, поэтому в третьем параметре функции в обоих случаях передается NULL. Теперь у нас есть две динамические строки, содержащие символьное представление чисел Period и Impulse, которые будут использоваться при формировании текста подсказки. Важно не забыть освободить отведенную под них память, вызвав в конце функции rdsFree.

Далее, в зависимости от значения Type, мы формируем динамическую строку с подсказкой и записываем указатель на нее в переменную str. Эту строку тоже нужно будет освободить в конце функции при помощи rdsFree. Для значений Type 0 и 1 процедура проста: мы используем уже знакомую нам функцию rdsDynStrCat, которая складывает две переданные ей строки. В данном случае первая строка – заголовок подсказки с названием формы сигнала и словом “период”, вторая – значение периода. В третьем параметре функции rdsDynStrCat передается FALSE, что запрещает возврат NULL вместо строки нулевой длины. На этом формирование подсказки для синусоидального и косинусоидального сигналов заканчивается.

Для прямоугольных импульсов (Type==2) подсказка формируется немного сложнее: кроме значения периода сигнала в подсказку нужно добавить значение ширины импульса. Сначала, как и для двух других форм сигнала, при помощи rdsDynStrCat формируется строка, состоящая из заголовка подсказки и значения периода сигнала. После этого к уже сформированной динамической строке необходимо добавить слова “длительность импульса” и значение этой длительности. Для этого мы будем пользоваться не функцией rdsDynStrCat, а очень похожей на нее rdsAddToDynStr. Отличие этой функции заключается в том, что она добавляет к уже имеющейся динамической строке, переданной в первом параметре, строку, переданную во втором, увеличивая отведенную для первой строки память. Фактически, вызову

```
rdsAddToDynStr(&str,text,FALSE);
```

соответствует конструкция следующего вида:

```
char *oldstr=str;
str=rdsDynStrCat(oldstr,text,FALSE);
rdsFree(oldstr);
```

При использовании этой функции следует всегда помнить, что в качестве первого параметра в нее передается указатель на переменную, содержащую указатель на динамическую строку. Эта строка не может быть статической, поскольку rdsAddToDynStr после формирования новой объединенной строки пытается освободить старую строку при помощи rdsFree. Вызов rdsFree не для динамической строки может привести к непредсказуемым последствиям. Например, конструкция

```
char teststr[]="Некоторая статическая строка";
char *str=&teststr;
rdsAddToDynStr(&str,"добавляемая строка",FALSE);
```

является ошибочной, поскольку переменная str в данном случае указывает не на динамическую строку, а на массив teststr, и при попытке rdsAddToDynStr освободить занятую teststr память произойдет ошибка.

Вернемся к переделанной функции PopupHint при Type==2. Первый вызов rdsAddToDynStr добавит к уже сформированному в str тексту подсказки слова

“длительность импульса” с двоеточием, второй – строку, в которую ранее было преобразовано значение этой длительности. Затем сформированный текст передается в РДС при помощи rdsSetHintText, после чего все три использованные в функции динамические строки освобождаются вызовами rdsFree. Внешне подсказка, выведенная этой функцией, будет выглядеть так же, как и на рис. 71, только вместо “4.000000” и “2.000000” будут выведены числа “4” и “2” соответственно.

В описанном примере мы не использовали параметры всплывающей подсказки, передаваемые в модель блока при ее вызове в режиме RDS_BFM_POPUPHINT, и не пытались изменить поведение этой подсказки на экране. Рассмотрим теперь более сложный случай: сделаем всплывающую подсказку к графику, рассмотренному в §2.10.1 (стр. 190). Эта подсказка должна отображать координаты точки графика возле курсора мыши. Таким образом, при перемещении курсора над изображением графика текст всплывающей подсказки должен изменяться, отражая изменение координат точки под курсором. Без изменения некоторых параметров подсказки этого добиться не получится: по умолчанию она просто исчезнет через некоторое время после появления, не реагируя на перемещения курсора в пределах изображения блока.

В описание класса блока нам нужно добавить две новых функции: одна из них будет заниматься выводом всплывающей подсказки, другая, вспомогательная, будет искать в массиве индекс отсчета, ближайшего к переданному в параметре значению времени. Функция, выводящая всплывающую подсказку, будет принимать один параметр: указатель на структуру параметров подсказки RDS_POPUPHINTDATA:

```
//=========================================
// Простой график - личная область данных
//=========================================
class TSimplePlotData
{
    //          ...

    // Рисование иконки при отсутствии доступа к DynTime
    void DrawAdditional(RDS_PDRAWDATA DrawData);
    // Поиск индекса отсчета, соответствующего времени t
    int FindTimeIndex(double t);
    // Вывод всплывающей полсказки
    void PopupHint(RDS_PPOPUPHINTDATA hintdata);
    //          ...
    // ... далее без изменений ...
```

Напишем сначала функцию поиска отсчета в массиве Times, учитывая тот факт, что из-за логики работы блока он упорядочен по возрастанию. Чтобы не усложнять пример, будем искать отсчет линейным поиском (позже мы перепишем эту функцию с использованием более быстрого алгоритма поиска).

```
// Поиск отсчета, соответствующего времени t
int TSimplePlotData::FindTimeIndex(double t)
{
  if(Count==0)        // Массивы пусты
    return -1;

  if(t<=Times[0])     // t раньше начала массива
    return 0;         // Ближайший индекс - 0

  // Поиск первого индекса, большего t
  for(int i=1;i<NextIndex;i++)
    if(Times[i]>t) // t между i-1 и i
      { double d1=fabs(t-Times[i-1]),
               d2=fabs(t-Times[i]);
```

```
        return (d1<d2)?(i-1):i; // Возвращаем ближайший
      }
    // Ничего не нашли - значит, t>Times[NextIndex-1]
    return NextIndex-1;
  }
  //=======================================
```

Эта функция устроена достаточно просто. Сначала мы проверяем, есть ли вообще отсчеты в массивах, и, если их нет, возвращаем вместо индекса значение –1. Затем сравниваем переданное значение времени `t` с началом массива. Если `t` меньше `Times[0]`, значит, все отсчеты массива находятся позднее `t`, и ближайший к `t` индекс – это начало массива, то есть 0. Далее мы в цикле проходим по массиву и ищем в нем самый первый отсчет, больший `t`. Если такой найдется, значит, `t` лежит между ним и предыдущим. Остается только определить, к какому из этих двух отсчетов `t` ближе, и возвратить соответствующий индекс. Если же во всем массиве не нашлось ни одного отсчета, большего `t`, значит, ближайшим к `t` будет конец массива, то есть `NextIndex-1`. При большом количестве отсчетов функция будет работать медленно, но при выводе всплывающей подсказки особенной скорости обычно не требуется. Тем более, что позже мы переделаем эту функцию.

Теперь, когда мы можем определить индекс отсчета по значению времени, можно написать функцию, выводящую подсказку.

```
// Всплывающая подсказка
void TSimplePlotData::PopupHint(RDS_PPOPUPHINTDATA hintdata)
{ int x1,x2,y1,y2,index;
  double t;
  char *text,*str_t,*str_v;

  // Проверка доступа к переменной времени
  if(Time==NULL || Time->Data==NULL)
    { rdsSetHintText("ОШИБКА: в схеме нет переменной DynTime");
      return;
    }

  // Определение абсолютных координат поля графика
  x1=hintdata->Left+Gr_x1;
  x2=hintdata->Left+Gr_x2;
  y1=hintdata->Top+Gr_y1;
  y2=hintdata->Top+Gr_y2;

  // Если курсор мыши не попадает в поле графика,
  // подсказку выводить не нужно
  if(hintdata->x<x1 || hintdata->x>x2 ||
     hintdata->y<y1 || hintdata->y>y2)
    return;

  // Преобразование экранной горизонтальной координаты
  // в значение времени согласно масштабу графика
  t=(hintdata->x-x1)*(Xmax-Xmin)/(x2-x1)+Xmin;

  // Поиск отсчета, соответствующего этому моменту времени
  index=FindTimeIndex(t);
  if(index<0) return; // Ошибка - отчет не найден

  // Преобразование времени и значения отсчета в строки
  str_t=rdsDtoA(Times[index],-1,NULL);
  str_v=rdsDtoA(Values[index],-1,NULL);
  // Формирование текста подсказки
  text=rdsDynStrCat("Время: ",str_t,FALSE);
```

```
    rdsAddToDynStr(&text,"\nЗначение: ",FALSE);
    rdsAddToDynStr(&text,str_v,FALSE);
    // Установка текста подсказки
    rdsSetHintText(text);
    // Освобождение динамических строк
    rdsFree(text);
    rdsFree(str_t);
    rdsFree(str_v);

    // Изменение параметров подсказки таким образом, чтобы при
    // смещении курсора на одну точку она вывелась снова
    hintdata->HZLeft=hintdata->x;
    hintdata->HZTop=hintdata->y;
    hintdata->HZWidth=hintdata->HZHeight=1;
    // Задержка гашения подсказки - одна минута
    hintdata->HideTimeout=60000;
  }
  //=======================================
```

Эта функция принимает единственный параметр – указатель на структуру параметров подсказки RDS_POPUPHINTDATA. Этот указатель РДС передает в модель блока, а она, в свою очередь, должна будет передать ее нашей новой функции. Структура RDS_POPUPHINTDATA описана следующим образом:

```
typedef struct
{ int x,y;            // Координаты курсора
  int BlockX,BlockY;  // Точка привязки блока
  // Положение и размер блока на момент последнего рисования
  int Left,Top;       // Левый верхний угол
  int Width,Height;   // Ширина и высота
  //------- Параметры, которые можно изменить -------
  int HZLeft,HZTop,     // Размер зоны действия подсказки (по
      HZWidth,HZHeight; // умолчанию - все изображение блока)
  int ReshowTimeout;  // Интервал повторного вывода подсказки
                      // (по умолчанию - 0, то есть не выводить)
  int HideTimeout;    // Интервал гашения подсказки (по
                      // умолчанию - стандартное для Windows)
  //---------- Конец изменяемых параметров ----------
  int IntZoom;          // Масштаб окна в %
  double DoubleZoom;    // Масштабный к-т (в долях единицы)
} RDS_POPUPHINTDATA;
typedef RDS_POPUPHINTDATA *RDS_PPOPUPHINTDATA;
```

В этой структуре нас, прежде всего, интересуют координаты курсора мыши (x, y) и верхнего левого угла блока (Left, Top). Зная их, мы можем вычислить, какой точке массива соответствует положение курсора. Кроме того, мы будем менять зону действия подсказки (HZLeft, HZTop, HZWidth и HZHeight). Пока курсор не покинет эту зону, подсказка меняться не будет. Нам нужно, чтобы текст подсказки отражал координаты точки графика, около которой находится курсор, поэтому мы каждый раз будем устанавливать зону действия размером в одну точку точно под курсором. Таким образом, любое перемещение курсора будет приводить к его выходу из зоны действия подсказки и выводу новой, с новым текстом. Мы также увеличим время гашения подсказки (HideTimeout) до одной минуты, чтобы пользователь гарантированно успел прочесть ее. При выходе курсора за пределы блока подсказка будет погашена независимо от значения HideTimeout, поэтому мы можем не беспокоиться о том, что этот интервал слишком велик.

Как и в предыдущем примере с подсказкой, сначала мы проверим доступность динамической переменной времени и, в случае ее отсутствия, выведем вместо подсказки сообщение об ошибке. В совокупности с изображаемой блоком иконкой с восклицательным

знаком (см. §2.10.3), это сообщение позволит пользователю понять причину неработоспособности блока. Если же с переменной времени все в порядке, то, как и в функции рисования, мы вычисляем абсолютные координаты поля графика в окне подсистемы. В функции рисования мы пользовались для этого полями `Left` и `Top` структуры `RDS_DRAWDATA`, здесь пользуемся одноименными полями `RDS_POPUPHINTDATA` – их значения будут одинаковыми. Если курсор мыши, координаты которого также передаются в структуре `RDS_POPUPHINTDATA`, не попадает в поле графика, функция немедленно завершается и подсказка при этом не выводится. Если же курсор находится внутри поля, мы, согласно масштабу горизонтальной оси, вычисляем значение времени, которому соответствует горизонтальная координата курсора, и записываем его в переменную `t`. Используемая при этом формула обратна формуле преобразования времени в координату, использованной в функции рисования (см. стр. 198).

Далее, при помощи написанной ранее функции `FindTimeIndex`, мы вычисляем индекс в массиве отсчетов, которому соответствует значение времени `t` и записываем его в целую переменную `index`. Возврат функцией отрицательного значения свидетельствует об ошибке (массивы пусты), в этом случае функция завершается без вывода подсказки. Теперь, зная индекс в массивах, можно определить значение времени `Times[index]` и вертикальной координаты `Values[index]`, соответствующие текущей точке, и сформировать из них текст подсказки уже описывавшимся в предыдущем примере способом.

Теперь нужно вставить вызов `PopupHint` в функцию модели блока – для этого в оператор `switch(CallMode)` добавляется новая метка `case`:

```
// ...
// Всплывающая подсказка
case RDS_BFM_POPUPHINT:
  data->PopupHint((RDS_PPOPUPHINTDATA)ExtParam);
  break;
// ...
```

Здесь перед передачей в функцию `PopupHint` указатель общего вида (`void*`) `ExtParam`, полученный функцией модели блока, приводится к типу `RDS_PPOPUPHINTDATA`, то есть “указатель на `RDS_POPUPHINTDATA`”.

Осталось только настроить зону действия подсказки и время ее гашения. В координаты левого верхнего угла зоны копируются координаты курсора мыши, а размер зоны устанавливается в одну точку – теперь любое перемещение курсора выведет его за пределы зоны, и подсказка будет показана повторно, уже с новыми значениями координат. Время гашения подсказки устанавливается в 60000 миллисекунд, то есть в одну минуту. Теперь, если разрешить вывод всплывающей подсказки для блока, и навести курсор на поле графика, через некоторое время появится окно подсказки (рис. 72), которое будет перемещаться вслед за курсором мыши, и числа в нем будут изменяться).

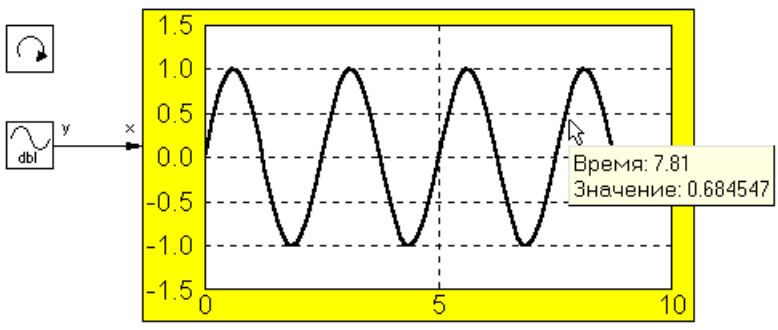


Рис. 72. Всплывающая подсказка к графику

Написанная нами функция `FindTimeIndex` определяет соответствующий заданному времени отсчет в массиве при помощи простого линейного поиска. Перепишем ее так, чтобы она использовала более быстрый поиск делением пополам, пользуясь тем, что массив `Times` упорядочен по возрастанию. Хотя используемый метод поиска и не имеет прямого отношения к рассматриваемому в примерах выводу всплывающих подсказок, следует помнить, что

линейный поиск – достаточно медленный алгоритм, и при больших массивах отсчетов он может вызывать замедление работы при перемещениях курсора по графику. Поэтому лучше ускорить указанную функцию, тем более, что метод деления пополам не особенно сложен. Новая функция `FindTimeIndex` будет иметь следующий вид:

```
// Поиск отсчета, соответствующего времени t
// (метод деления пополам)
int TSimplePlotData::FindTimeIndex(double t)
{ int L,R;
  double dl,dr;

  if(Count==0 || NextIndex==0)  // Нет данных в массивах
    return -1;
  if(NextIndex==1)    // В массиве единственный отсчет
    return 0;

  // В массиве по крайней мере два значения - проверяем границы
  if(t<=Times[0])               // t меньше первого отсчета
    return 0;
  if(t>=Times[NextIndex-1])     // t больше последнего отсчета
    return NextIndex-1;

  // t - внутри диапазона массива
  L=0; R=NextIndex-1;
  while(L<R-1)
    { int m=(L+R)/2;
      if(Times[m]<t) L=m; else R=m;
    }
  // t лежит между L и R
  dl=fabs(t-Times[L]),
  dr=fabs(t-Times[R]);
  return (dl<dr)?L:R; // Возвращаем ближайший
}
//=========================================
```

Проверив, что в массиве есть данные, и исключив случай, когда в нем записан единственный отсчет (при этом ничего искать, естественно, не нужно), мы сравниваем значение `t` с первым и последним отсчетами массива. Если `t` лежит за пределами диапазона отсчетов массива, искать его в массиве не нужно: ближайшим к `t` отсчетом будет первый (если `t<=Times[0]`) или последний (если `t>=Times[NextIndex-1]`). Если `t` – внутри диапазона, мы присваиваем переменным `L` и `R` граничные индексы массива и начинаем в цикле сравнивать отсчет в середине диапазона `L...R` с `t`. В зависимости от того, будет этот отсчет меньше или больше `t`, мы перемещаем одну из границ и повторяем деление до тех пор, пока `L` и `R` не окажутся соседними отсчетами, при этом значение `t` будет лежать между ними. Останется только выбрать из этих отсчетов ближайший к `t`.

§2.12. Реакция блоков на действия пользователя

Рассматриваются различные способы организации взаимодействия блоков схемы с пользователем: реакция на мышь и клавиатуру, добавление новых пунктов в контекстное меню блока и системное меню РДС. Отдельно описываются реакции на действия пользователя при редактировании схемы: добавление блоков, их удаление, изменение их размеров и т.п.

§2.12.1. Реакция на мышь

Рассматриваются возможные реакции модели блока на действия пользователя мышью. Приводится пример блока, позволяющего щелчками по верхней и нижней части его изображения увеличивать и уменьшать значение его выхода. Затем в модель блока добавляется реакция на щелчки по активным элементам векторной картинки.

Основной способ организовать взаимодействия схемы с пользователем – это разрешить некоторым блокам схемы (рукояткам, кнопкам, полям ввода и т.п.) реагировать на нажатие и отпускание кнопок мыши и перемещение ее курсора. В режиме редактирования мышь используется для перетаскивания блоков, рисования связей и прочих действий по изменению схемы, поэтому РДС позволяет моделям блоков реагировать на мышь только в режимах моделирования и расчета. Для того, чтобы модель блока вызывалась в этих случаях, должны одновременно выполниться три условия:

- курсор мыши должен находиться в пределах изображения блока (точнее, его описывающего прямоугольника);
- блок должен находиться на слое, для которого разрешено редактирование;
- в параметрах блока должна быть разрешена реакция на мышь.

Из первого условия есть исключение: при необходимости, установив специальный флаг в структуре данных блока `RDS_BLOCKDATA`, модель может захватить мышь, чтобы получать информацию о перемещениях курсора даже после того, как он покинет пределы изображения блока (подобный пример будет рассмотрен ниже). Отключение редактирования слоя не только запрещает пользователю перемещать блоки и менять их параметры в режиме редактирования, но и блокирует реакции их моделей на мышь в режимах моделирования и расчета. Это позволяет, например, сделать какую-либо группу полей ввода отладочной, и защитить их от случайного изменения. Разрешение или запрещение реакции на мышь в параметрах блока позволяет, например, использовать блоки с одной и той же моделью и как индикаторы, и как поля ввода.

Ранее, в примере блока, создающего панель в окне подсистемы (стр. 225), уже встречалась реакция модели на двойной щелчок мышью по изображению блока, но там она играла только вспомогательную роль, поэтому ей было уделено мало внимания. Теперь более подробно рассмотрим обработку нажатия кнопки мыши на простом примере: создадим блок, который будет увеличивать значение своего целого выхода `v` на единицу при каждом щелчке левой кнопкой мыши по верхней части его изображения, и уменьшать его на единицу при каждом щелчке по нижней части.

Блоку нужен единственный целый выход, поэтому структура переменных блока будет выглядеть следующим образом:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	v	int	4	Выход

Модель блока будет достаточно простой:

```
// Увеличение/уменьшение значения по щелчку
extern "C" __declspec(dllexport)
  int RDSCALL PlusMinus(int CallMode,
```

```
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define v (*((int *)(pStart+2)))
// Вспомогательная — указатель на структуру события мыши
RDS_PMOUSEDATA mouse;

  switch(CallMode)
    { // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSI}")?
                    RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Реакция на нажатие кнопки мыши
      case RDS_BFM_MOUSEDOWN:
        // Приведение ExtParam к нужному типу
        mouse=(RDS_PMOUSEDATA)ExtParam;
        if(mouse->Button==RDS_MLEFTBUTTON)
          { // Нажата левая кнопка
            if(mouse->y<mouse->Top+mouse->Height/2)
              v++;    // В верхней половине блока - увеличиваем
            else
              v--;    // В нижней половине блока — уменьшаем
            // Взводим сигнал готовности
            Ready=1;
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef v
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Кроме стандартной проверки типа переменных,блок реагирует еще только на одно событие: нажатие какой-либо кнопки мыши RDS_BFM_MOUSEDOWN. При вызове модели в этом режиме в параметре ExtParam передается указатель на структуру RDS_MOUSEDATA, описывающую произошедшее событие:

```
typedef struct
{ int x,y;            // Координаты курсора мыши на рабочем поле
  int BlockX,BlockY;  // Координаты точки привязки блока с учетом
                      // связи с переменными
  int Left,Top;       // Верхний левый угол зоны блока
  int Width,Height;   // Ширина и высота зоны блока
  int IntZoom;        // Масштаб окна подсистемы в процентах
  DWORD Button;       // Кнопка мыши (константа RDS_M*)
  DWORD Shift;        // Флаги клавиш (RDS_M*, RDS_K*)
  double DoubleZoom;  // Масштаб в долях единицы
  int MouseEvent;     // Причина вызова (константа RDS_BFM_MOUSE*)
} RDS_MOUSEDATA;
typedef RDS_MOUSEDATA *RDS_PMOUSEDATA;
```

Для упрощения доступа к полям этой структуры указатель на нее, приведенный к нужному типу, записывается во вспомогательную переменную `mouse`.

Сейчас нас в этой структуре будут интересовать координаты курсора (`x`, `y`), координаты и размеры описывающего прямоугольника блока в текущем масштабе (`Left`, `Top`, `Width` и `Height`), а также нажатая кнопка (`Button`). Увеличивать и уменьшать выход блока нужно только при щелчках левой кнопки, поэтому, прежде всего, поле `mouse->Button` сравнивается с константой `RDS_MLEFTBUTTON`, обозначающей левую кнопку мыши. Далее модель сравнивает вертикальную координату мыши `mouse->y` с вертикальной координатой центра описывающего прямоугольника блока `mouse->Top+mouse->Height/2` (координата верхней границы плюс половина высоты). Если координата курсора меньше этого значения, значит, щелчок пришелся на верхнюю часть блока, и значение выхода нужно увеличить. В противном случае значение уменьшается, – щелчок пришелся на нижнюю часть. После этого сигналу готовности блока `Ready` присваивается единица, чтобы в ближайшем такте расчета сработали связи, присоединенные к выходу блока `v`.

Для проверки работы созданной модели необходимо разрешить реакцию на мышь в параметрах блока, к которому она подключена (рис. 73), и присоединить к его выходу числовой индикатор.

В качестве изображения блока можно выбрать прямоугольник с двумя текстовыми строчками “+1” и “–1”, расположенными друг под другом. В режиме расчета при щелчке по верхней части блока значение индикатора, соединенного с выходом блока, будет увеличиваться, а при щелчке по верхней – уменьшаться.

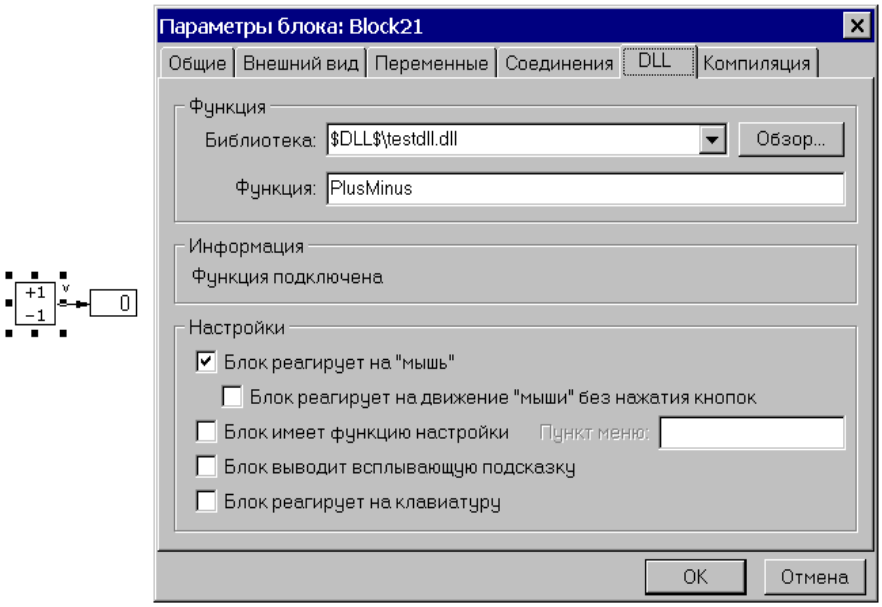


Рис. 73. Включение реакции на мышь в параметрах блока

Модель этого блока получилась достаточно простой – мы обходимся без класса личной области данных, и все действия выполняем непосредственно в функции модели, внутри макроопределений для статических переменных. Здесь, однако, следует внимательно следить за возможными конфликтами имен макросов переменных и используемых в функции имен переменных и полей структур. Представим себе, что выход данного блока мы назвали бы не `v`, а `y`. Это простое, на первый взгляд, переименование приведет к тому, что компиляция функции станет невозможной. Дело в том, что у используемой в модели структуры `RDS_MOUSEDATA` есть поле с точно таким же именем – “`y`”. При обработке текста программы препроцессором языка С имя “`y`” будет воспринято как макроопределение и

раскрыто, в результате чего выражение “mouse->y” будет преобразовано в “mouse->(*((int*)(pStart+2)))”, которое, естественно, не будет понято компилятором. Ошибки, выданные компилятором при разборе такого выражения, могут показаться программисту, особенно начинающему, довольно странными, тем более, что в тексте компилируемой программы написано, на первый взгляд, правильное выражение “mouse->y”, а результат работы препроцессора обычно в явном виде нигде не отображается. Для того, чтобы избежать подобных проблем, в более-менее сложных моделях блоков имеет смысл выносить все действия за пределы главной функции модели с ее макроопределениями, оформляя эти действия как обычные функции или функции-члены класса личной области данных.

Созданный нами блок можно использовать в качестве простейшего органа управления для увеличения и уменьшения какого-либо параметра схемы. При этом его работа не будет зависеть от его внешнего вида: как бы он ни выглядел, щелчок по его верхней части будет увеличивать параметр, щелчок по нижней – уменьшать. Можно, например, написать в прямоугольнике с текстом не “+1” и “–1”, а “увеличить” и “уменьшить”, или задать блоку векторную картинку со стрелками вверх и вниз. Однако, модель блока написана так, что эти стрелки или надписи обязательно должны располагаться друг под другом, одна – в верхней половине прямоугольника блока, другая – в нижней. Чтобы разместить их, например, слева и справа, придется переделывать модель.

Чтобы сделать блок более гибким, можно проверять, в какой конкретный элемент векторной картинки попал курсор мыши (если, конечно у блока есть картинка). РДС позволяет присваивать каждому элементу картинки какое-либо целое число, и модель блока, вызвав одну из специальных сервисных функций, может получить число, соответствующее элементу по заданным координатам или под курсором мыши. Если присвоить элементам картинки, символизирующим увеличение параметра (например, стрелке вверх) одно число, а символизирующим уменьшение (стрелке вниз) – другое, модель сможет правильно реагировать на щелчки по этим элементам, независимо от того, как они расположены.

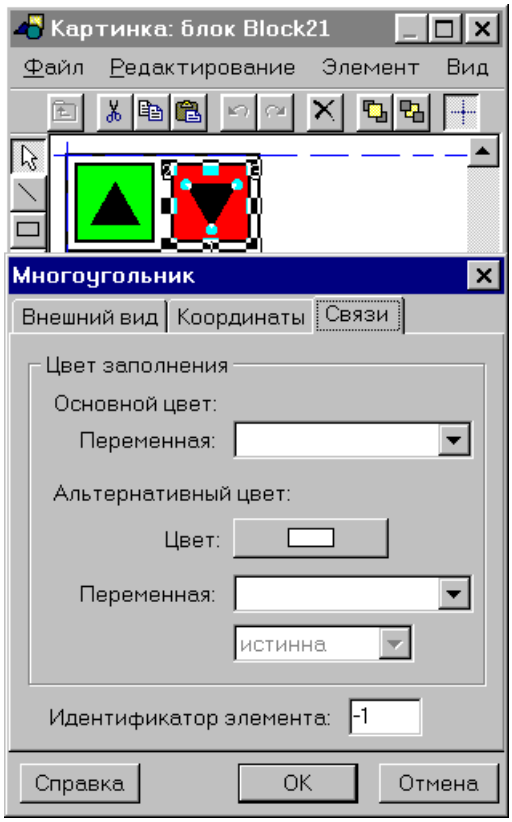


Рис. 74. Ввод идентификатора элемента картинки

Будем присваивать частям картинки, щелчок по которым должен привести к увеличению и уменьшению переменной v, числа-идентификаторы 1 и –1 соответственно. Все остальные элементы картинки будут иметь идентификатор 0 (он присваивается по умолчанию). При отсутствии у блока векторной картинки модель будет вести себя, как и раньше: щелчок по верхней части изображения будет увеличивать v, щелчок по нижней – уменьшать. Поместим на картинку блока (рис. 74) вытянутый по горизонтали белый прямоугольник (идентификатор 0), в левой части которого будет находиться зеленый квадрат (идентификатор 1) с наложенным на него треугольником, направленным вверх (идентификатор 1), а в правой – красный квадрат (идентификатор –1) с направленным вниз

треугольником (идентификатор –1). Идентификаторы квадрата и наложенного на него треугольника должны совпадать, чтобы оба элемента воспринимались моделью блока как единое целое.

Теперь внесем изменения в модель блока. В реакцию на событие RDS_BFM_MOUSEDOWN нужно добавить проверку наличия у блока векторной картинки, и, если она есть, считывание идентификатора ее элемента, на который пришелся щелчок левой кнопкой:

```
// Реакция на нажатие кнопки мыши
case RDS_BFM_MOUSEDOWN:
  // Приведение ExtParam к нужному типу
  mouse=(RDS_PMOUSEDATA)ExtParam;
  if(mouse->Button==RDS_MLEFTBUTTON)
    { // Нажата левая кнопка
      // Проверяем, есть ли у блока картинка (получаем
      // описание блока)
      RDS_BLOCKDESCRIPTION descr;
      descr.servSize=sizeof(descr);
      rdsGetBlockDescription(BlockData->Block,&descr);
      if(descr.Flags & RDS_BDF_HASPICTURE)
        { // Картинка есть – определяем идентификатор
          // элемента под курсором
          int id=rdsGetMouseObjectId(mouse);
          switch(id)
            { case 1:  v++; break;
              case -1: v--; break;
            }
        }
      else if(mouse->y<mouse->Top+mouse->Height/2)
        v++;    // В верхней половине блока - увеличиваем
      else
        v--;    // В нижней половине блока — уменьшаем
      // Взводим сигнал готовности
      Ready=1;
    }
  break;
```

Убедившись, что нажата левая кнопка мыши, модель запрашивает у РДС описание блока, который она обслуживает. Для этого используется уже знакомая нам по примерам структура RDS_BLOCKDESCRIPTION – ранее мы использовали ее для получения текста комментария блока (см. §2.6.3). В служебное поле servSize этой структуры записывается ее размер (так РДС контролирует правильность переданного параметра), после чего вызывается сервисная функция rdsGetBlockDescription, которая записывает описание блока, идентификатор которого передан в первом параметре функции, в структуру, указатель на которую передается во втором. Нас интересует битовый флаг RDS_BDF_HASPICTURE поля Flags: если он взведен, у блока есть векторная картинка. Целая константа RDS_BDF_HASPICTURE, описанная в “RdsDef.h”, имеет единственный единичный бит в позиции интересующего нас флага. Выполняя операцию побитового “ИЛИ” над полем Flags и этой константой, мы получим нулевой результат, если флаг сброшен, и не нулевой в противном случае. Таким образом, оператор

```
if(descr.Flags & RDS_BDF_HASPICTURE)
```

выполнится только в том случае, если для блока задана векторная картинка. В противном случае будет выполняться старая часть модели (после else), определяющая попадание курсора в верхнюю/нижнюю половину изображения.

Если картинка у блока есть, модель вызывает сервисную функцию `rdsGetMouseObjectId`, передавая ей указатель на структуру описания события мыши `RDS_MOUSEDATA`. Функция берет из этой структуры необходимые ей параметры и возвращает идентификатор элемента картинки блока, находящегося под курсором мыши, который присваивается вспомогательной переменной `id`. В данном случае `id` может принимать всего три значения: 1 (курсор мыши попал в зеленый квадрат или лежащий на нем треугольник), –1 (курсор попал в красный квадрат или его треугольник) и 0 (курсор не попал в указанные элементы). Если `id` равен 1 или –1, выход блока соответственно увеличивается или уменьшается на единицу.

Теперь щелчок по зеленому квадрату будет увеличивать выход блока, по красному – уменьшать. Причем, если в редакторе картинки поменять эти квадраты местами, расположить их друг под другом, или вообще в произвольных местах картинки, работа блока не нарушится – модель опознает не какие-то жестко заданные области картинки, а идентификаторы ее элементов, где бы они не находились.

Можно пойти еще дальше, и переделать модель следующим образом:

```
if(descr.Flags & RDS_BDF_HASPICTURE)
  { // Картинка есть - определяем идентификатор
    // элемента под курсором
    int id=rdsGetMouseObjectId(mouse);
    v+=id;
  }
```

Теперь полученное значение идентификатора элемента не анализируется оператором `switch`, а сразу прибавляется к выходу блока `v`. При щелчках на элементах с идентификаторами 1 и –1 поведение блока не изменится: выход будет увеличиваться или уменьшаться на единицу. При этом такая модель позволяет добавить в картинку новые элементы для увеличения или уменьшения выхода на произвольное число. Например, если добавить в картинку еще один квадрат с идентификатором 2, щелчок по нему приведет к увеличению `v` на 2. Блок с новой моделью может иметь любую картинку с любым расположением и количеством активных областей, нажатие на которые будет изменять его выход на число, равное идентификатору области.

Таким образом, из векторной картинки любого блока можно сделать “пульт” с кнопками, нажатие на которые будет отслеживаться моделью, причем внешний вид этого “пульта” ограничен только фантазией разработчика.

§2.12.2. Захват мыши, реакция на перемещение курсора

Рассматривается реакция модели блока на перемещение курсора мыши и захват мыши – режим, в котором блок реагирует на мышь даже при выходе курсора за его изображение. Приводится пример блока, изображающего рукоятку, которую можно двигать мышью по двум координатам, изменяя таким образом значения выходов блока `x` и `y`.

Для создания полноценного пользовательского интерфейса обычно мало реагировать только на щелчки мыши на изображении блока. Достаточно часто нужно отслеживать и перемещения курсора – например, для виртуальных рукояток, которые пользователь может двигать, меняя какие-либо значения, или для выделения области графика, которую нужно рассмотреть подробнее. Чаще всего в таких случаях перемещения курсора производятся при нажатой кнопке мыши: нажатие кнопки отмечает начало операции (перемещения рукоятки, выделения области и т.п.), а отпускание – ее конец. Необходимость закладывать в модель блока реакцию на перемещение курсора мыши без нажатия кнопок встречается довольно редко. Такие реакции могут замедлять систему в режиме расчета, поскольку каждый проход курсора мыши над изображением блока будет порождать большое количество вызовов модели этого блока. При этом, поскольку все реакции на действия пользователя производятся в главном потоке РДС, поток расчета будет каждый раз останавливаться и

ждать завершения реакции модели. Кроме того, если перемещение мыши над блоком должно как-либо отражаться на его внешнем виде, каждое перемещение курсора над блоком будет приводить к необходимости обновления окна подсистемы, что также может приводить к существенному замедлению. По этой причине при включении в параметрах блока реакции на мышь, модель будет реагировать на перемещения курсора по изображению блока только при нажатой кнопке мыши. Для того, чтобы модель всегда реагировала на перемещения курсора, в параметрах блока нужно включить дополнительный флаг (см. рис. 73).

Как уже было написано выше, при покидании курсором мыши описывающего прямоугольника блока, вызовы его модели по умолчанию прекращаются. Это не всегда удобно. Например, если пользователь двигает какую-либо вертикальную рукоятку, и, изменяя ее значение вертикальными движениями курсора, случайно сдвинет курсор по горизонтали, он может выйти за пределы блока и попасть в соседнюю рукоятку. Поскольку взгляд пользователя в этот момент, вероятнее всего, будет прикован к индикаторам, по которым он следит за поведением системы, он не сразу поймет, почему перестал изменяться нужный ему параметр, и при этом начал изменяться какой-то другой.

Чтобы избежать таких проблем, имеет смысл при начале отслеживания перемещения курсора включать захват мыши. При этом модель блока будет получать сообщения о перемещениях курсора, а также нажатии и отпускании кнопок мыши, даже при выходе курсора за пределы изображения блока, до тех пор, пока захват не будет снят. Чаще всего захват включают при нажатии кнопки мыши, а выключают – при отпускании, но возможны и другие варианты.

В качестве примера рассмотрим блок, имитирующий двухкоординатную рукоятку. Внутри прямоугольника, разделенного на четыре части вертикальными линиями, будет изображаться круг синего цвета (рис. 75). Пользователь может “перетаскивать” этот круг мышью, меняя значения выходов блока `x` и `y`. При перемещении круга по горизонтали от левой границы прямоугольника до правой, выход `x` будет изменяться от –1 до 1. При перемещении круга по вертикали от нижней границы до верхней, выход `y` точно так же будет изменяться от –1 до 1. Таким образом, положение круга в центре блока, на пересечении линий, соответствует нулевым значениям обоих выходов. Для лучшей визуальной обратной связи сделаем так, чтобы в процессе перетаскивания круг менял цвет на красный. Чтобы не делать для блока сложную векторную картинку, будем рисовать его внешний вид программно (см. §2.10).

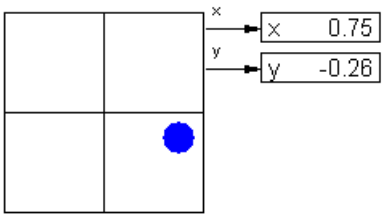


Рис. 75. Двухкоординатная рукоятка

Для упрощения примера мы не будем включать в модель функцию настройки параметров блока (цветов прямоугольника и круга, размера круга), но, тем не менее, сделаем их полями класса личной области данных блока. При необходимости, функции настройки, загрузки и сохранения этих параметров можно будет написать позже. Большую часть реакций модели на мышь мы вынесем в функции-члены класса, чтобы макроопределение для выхода блока `y`, которое будет использоваться в функции модели, не конфликтовало с одноименным полем структуры `RDS_MOUSEDATA` (эта проблема была подробно описана на стр. 264).

Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	x	double	8	Выход
10	y	double	8	Выход

В такте расчета этот блок не будет выполнять никаких действий, поэтому для него следует установить флаг запуска по сигналу, чтобы процессорное время не тратилось зря на пустой вызов модели. Запишем класс личной области данных блока, с указанными выше полями-параметрами и функциями-членами:

```
//====== Класс личной области данных ======
class TSimpleJoystick
{ private:
    // Центр круга (рукоятки) до начала перетаскивания
    int OldHandleX,OldHandleY;
    // Координаты курсора на момент начала перетаскивания
    int OldMouseX,OldMouseY;
  public:
    // Настроечные параметры блока
    COLORREF BorderColor;        // Цвет рамки блока
    COLORREF FieldColor;         // Цвет прямоугольника
    COLORREF HandleColor;        // Цвет круга в покое
    COLORREF MovingHandleColor;  // Цвет круга при таскании
    int HandleSize;              // Диаметр круга

    // Реакция на нажатие кнопки мыши
    int MouseDown(RDS_PMOUSEDATA mouse,double x,double y,
                  DWORD *pFlags);
    // Реакция на перемещение курсора мыши
    void MouseMove(RDS_PMOUSEDATA mouse,double *px,double *py);
    // Рисование изображения блока
    void Draw(RDS_PDRAWDATA draw,double x,double y,BOOL moving);

    // Конструктор класса
    TSimpleJoystick(void)
      { BorderColor=0;           // Черная рамка
        FieldColor=0xffffff;     // Белое поле
        HandleColor=0xff0000;    // Синий круг
        MovingHandleColor=0xff;  // Красный при таскании
        HandleSize=20;           // Диаметр круга
      };
};
//=========================================
```

В закрытой секции класса описаны четыре целых поля, они понадобятся нам при реализации перетаскивания круга-рукоятки по прямоугольнику блока. В момент нажатия левой кнопки мыши в пределах круга мы запомним координаты центра круга в переменных `OldHandleX` и `OldHandleY`, а координаты курсора – в `OldMouseX` и `OldMouseY`. После этого при перемещении курсора относительно (`OldMouseX`, `OldMouseY`) мы будем перемещать центр круга на то же расстояние относительно (`OldHandleX`, `OldHandleY`), таким образом, круг будет двигаться за курсором, не смещаясь относительно него.

В открытой секции класса описаны настроечные параметры блока, функции реакции на мышь и рисования изображения блока и, естественно, конструктор класса. В конструкторе

всем настроечным параметрам присваиваются оговоренные ранее значения. В функции реакции значения выходов блока (или указатели на них, если функция будет в них что-то записывать) передаются в параметрах, поскольку положение круга в блоке зависит от текущего значения его выходов `x` и `y` и мы не собираемся использовать в функциях-членах класса макроопределения для статических переменных.

Теперь рассмотрим функции-члены класса, и начнем с функции рисования `Draw`, которая определяет внешний вид нашего блока. Кроме указателя на структуру параметров рисования `RDS_DRAWDATA` в эту функцию передаются текущие значения переменных блока `x` и `y` (по ним вычисляются координаты центра рисуемого круга) и дополнительный логический параметр `moving`, указывающий на то, идет ли в данный момент перетаскивание круга (в этом случае он должен быть нарисован красным вместо синего). Значение этого параметра будет вычисляться в основной функции модели, которую мы рассмотрим позже. Функция `Draw` имеет следующий вид:

```
// Рисование изображения блока
void TSimpleJoystick::Draw(RDS_PDRAWDATA draw,
                           double x,double y,BOOL moving)
{ int hx,hy,cx,cy;
  RECT r;
  int hR=HandleSize*draw->DoubleZoom/2; // Радиус круга-рукоятки

  // Если размер блока - нулевой, рисовать негде
  if(draw->Height==0 || draw->Width==0)
    return;

  // Рисование поля блока
  rdsXGSetPenStyle(0,PS_SOLID,1,BorderColor,R2_COPYPEN);
  rdsXGSetBrushStyle(0,RDS_GFS_SOLID,FieldColor);
  rdsXGRectangle(draw->Left,draw->Top,
                 draw->Left+draw->Width,draw->Top+draw->Height);

  // Вычисление центра прямоугольника блока
  cx=draw->Left+draw->Width/2;
  cy=draw->Top+draw->Height/2;

  // Вычисление координат центра круга-рукоятки
  hx=cx+x*draw->Width/2;
  hy=cy-y*draw->Height/2;

  // Установка области отсечения
  r.left=draw->Left+1;
  r.top=draw->Top+1;
  r.right=draw->Left+draw->Width-1;
  r.bottom=draw->Top+draw->Height-1;
  rdsXGSetClipRect(&r);

  // Линии перекрестия
  rdsXGMoveTo(cx,draw->Top);
  rdsXGLineTo(cx,draw->Top+draw->Height);
  rdsXGMoveTo(draw->Left,cy);
  rdsXGLineTo(draw->Left+draw->Width,cy);

  // Рисование круга (цвет зависит от параметра moving)
  rdsXGSetPenStyle(RDS_GFSTYLE,PS_NULL,0,0,0);
  rdsXGSetBrushStyle(0,RDS_GFS_SOLID,
     moving?MovingHandleColor:HandleColor);
  rdsXGEllipse(hx-hR,hy-hR,hx+hR+1,hy+hR+1);
```

```
  // Отмена отсечения
  rdsXGSetClipRect(NULL);
}
//=======================================
```

Если длина или ширина блока – нулевые, функция немедленно завершается, поскольку ей просто негде рисовать изображение. В противном случае рисуется прямоугольник размером с весь блок, по которому будет перемещаться перетаскиваемый мышью круг. Координаты центра прямоугольника блока записываются во вспомогательные переменные `cx` и `cy` – этим координатам будет соответствовать положение центра круга при нулевых значениях `x` и `y`. Затем, по размерам прямоугольника и вещественным значениям выходов блока `x` и `y`, вычисляются координаты центра круга (`hx`, `hy`) так, чтобы при нулевом значении выхода круг оказывался в центре блока, а при значении ±1 – на границе.

Перед тем, как нарисовать круг внутри прямоугольника блока, необходимо отсечь возможность рисования за пределами этого прямоугольника. Дело в том, что при `x` или `y`, равных ±1, центр круга будет находиться точно на границе прямоугольника блока, и, если не принять меры, половина круга окажется за пределами изображения блока. Чтобы этого не случилось, вызывается уже знакомая нам по блоку-графику (см. стр. 190) функция отсечения по прямоугольнику `rdsXGSetClipRect`. Ей передаются координаты с отступом на одну точку внутрь прямоугольника блока, чтобы круг, который будет нарисован, не задел рамку прямоугольника.

Далее рисуются линии перекрестия в центре прямоугольника (`cx`, `cy`) – их нужно нарисовать до круга-рукоятки, чтобы они его не перекрывали. Затем, в зависимости от значения параметра `moving`, устанавливается нужный цвет заливки (при `moving==TRUE` – `MovingHandleColor`, то есть красный, при `FALSE` – `HandleColor`, то есть синий), и функцией `rdsXGEllipse` рисуется круг заданного в параметрах блока радиуса с центром в (`hx`, `hy`). После этого отсечение отключается и функция на этом завершается.

Теперь напишем функцию реакции на нажатие кнопки мыши `MouseDown`. Эта функция проверит, левая ли кнопка мыши нажата, попал ли курсор мыши в круг рукоятки (для этого в параметрах функции передаются вещественные значения обоих выходов блока, от которых зависит положение круга), и, если оба условия выполнены, подготовит вспомогательные переменные к перетаскиванию круга и установит в флагах блока, указатель на которые передан в параметре `pFlags`, флаг захвата мыши. Функция `MouseDown` будет возвращать целое значение, которое функция модели без изменений использует в качестве своего результата – таким образом мы сообщим РДС, обработано ли нажатие кнопки мыши.

```
// Реакция на нажатие кнопки мыши
int TSimpleJoystick::MouseDown(RDS_PMOUSEDATA mouse,
                               double x,double y,
                               DWORD *pFlags)
{ int hx,hy,cx,cy,
      hR=HandleSize*mouse->DoubleZoom/2; // Радиус круга

  // Если размер - нулевой, реакция не имеет смысла
  if(mouse->Height==0 || mouse->Width==0)
    return RDS_BFR_DONE;
  // Если нажата не левая кнопка, перетаскивать не надо
  // Разрешаем в этом случае вызов контекстного меню блока
  if(mouse->Button!=RDS_MLEFTBUTTON)
    return RDS_BFR_SHOWMENU;

  // Координаты цента блока
  cx=mouse->Left+mouse->Width/2;
  cy=mouse->Top+mouse->Height/2;
```

```
    // Координаты центра круга-рукоятки
    hx=cx+x*mouse->Width/2;
    hy=cy-y*mouse->Height/2;

    // Проверка попадания курсора в круг
    if(abs(mouse->x-hx)<=hR && abs(mouse->y-hy)<=hR)
      { // Курсор попал в круг
        // Запоминаем координаты центра круга на момент
        // начала перетаскивания
        OldHandleX=hx;
        OldHandleY=hy;
        // Координаты курсора на начало перетаскивания
        OldMouseX=mouse->x;
        OldMouseY=mouse->y;
        // Взводим флаг захвата мыши
        *pFlags|=RDS_MOUSECAPTURE;
      }
    // Курсор не попал в рукоятку - захватывать мышь
    // и подготавливать перетаскивание не нужно
    return RDS_BFR_DONE;
  }
  //=======================================
```

Если ширина или высота блока – нулевые, функция немедленно возвращает `RDS_BFR_DONE` – нормальная работа блока в этом случае невозможна. В противном случае функция проверяет, левая ли кнопка нажата, и, если это не так, возвращает константу `RDS_BFR_SHOWMENU`. Возврат этой константы при реакции на нажатие правой кнопки мыши сигнализирует РДС о том, что, несмотря на то, что нажатие было обработано моделью, необходимо открыть контекстное меню блока (по умолчанию контекстные меню блоков, среагировавших на мышь, не вызываются). На самом деле, нам пока не важно, вызовется ли контекстное меню нашего блока при нажатии на нем правой кнопкой мыши в режимах моделирования и расчета. Но в дальнейшем мы добавим в этот пример новые возможности, которые будут включаться и отключаться именно через контекстное меню, поэтому о его вызове лучше позаботиться сразу.

Если же была нажата левая кнопка мыши, точно так же, как и в функции `Draw`, вычисляются координаты центра блока (`cx`, `cy`), и, через них, координаты центра рукоятки (`hx`, `hy`). Попадание курсора мыши в круг рукоятки проверяется по близости его координат к центру круга – расстояние по обеим координатам не должно быть больше радиуса круга. Если это условие выполняется, можно начинать перетаскивание: текущие координаты центра круга сохраняются в полях класса (`OldHandleX`, `OldHandleY`), а координаты курсора – в (`OldMouseX`, `OldMouseY`). Эти значения будут использоваться в процессе перетаскивания (в реакции на перемещение курсора) как начальные условия. После сохранения начальных значений координат функция захватывает мышь, взводя битовый флаг `RDS_MOUSECAPTURE` в поле `Flags` структуры данных блока `BlockData` (указатель на это поле должен быть передан в функцию `MouseDown` из вызвавшей ее функции модели в параметре `pFlags`). Для взведения флага используется побитовая операция “ИЛИ” (в данном случае использован оператор присваивания `*pFlags|=RDS_MOUSECAPTURE`, который эквивалентен записи `*pFlags=*pFlags | RDS_MOUSECAPTURE`).

Последняя функция класса – `MouseMove` – реагирует на перемещение курсора. Внутри нее по изменившемуся положению курсора вычисляются новые координаты центра перетаскиваемого круга, а по ним – новые значения выходов блока `x` и `y`. Поскольку функция не имеет непосредственного доступа к статическим выходам блока, в ее параметрах передаются указатели на эти выходы `px` и `py`, а она записывает по этим указателям результат своей работы. Мы не будем делать в функции проверку, производится в данный момент

перетаскивание рукоятки или нет – возложим эту обязанность на вызывающую функцию. Пока будем считать, что, раз MouseMove вызвана, значит, в данный момент выполняется перетаскивание.

```
// Реакция на перемещение курсора мыши
void TSimpleJoystick::MouseMove(RDS_PMOUSEDATA mouse,
                                double *px,double *py)
{ int hx,hy,cx,cy;

  // Если размер - нулевой, реакция не имеет смысла
  if(mouse->Height==0 || mouse->Width==0)
    { *px=*py=0.0;
      return;
    }

  // Новые координаты центра рукоятки
  hx=OldHandleX+(mouse->x-OldMouseX);
  hy=OldHandleY+(mouse->y-OldMouseY);

  // Координаты центра блока
  cx=mouse->Left+mouse->Width/2;
  cy=mouse->Top+mouse->Height/2;

  // По новым координатам центра рукоятки вычисляем соответствующие
  // им вещественные значения выходов, ограничивая их
  // диапазоном [-1...1]
  *px=2.0*(hx-cx)/mouse->Width;
  if(*px>1.0) *px=1.0;
  else if(*px<-1.0) *px=-1.0;
  *py=-2.0*(hy-cy)/mouse->Height;
  if(*py>1.0) *py=1.0;
  else if(*py<-1.0) *py=-1.0;
}
//=======================================
```

Как и в двух других функциях, в этой сначала проверяется, не нулевая ли длина или ширина блока (в этом случае выходам блока принудительно присваиваются нули и функция завершается). Если с размерами блока все в порядке, вычисляются новые, с учетом перемещения курсора, координаты центра круга-рукоятки. Зная координаты рукоятки и курсора мыши до начала перетаскивания, вычислить новые координаты рукоятки не сложно. По горизонтали с момента начала перетаскивания курсор переместился на (mouse->x-OldMouseX) точек, рукоятка перемещается вместе с курсором, значит, для вычисления ее новой горизонтальной координаты нужно к старому ее значению OldHandleX добавить перемещение курсора мыши. Вертикальное перемещение вычисляется аналогично.

По новым координатам рукоятки можно вычислить новые значения выходов блока, разделив для каждой из двух координат расстояние от центра круга рукоятки до центра блока на половину размера блока (половину ширины или высоты, в зависимости от вычисляемой координаты). При этом вычисленные значения необходимо ограничить диапазоном [–1...1]. Поскольку блок у нас будет захватывать мышь, его модель будет получать от РДС сообщения о перемещении курсора и, реагируя на них, вызывать функцию MouseMove, даже при выходе курсора за пределы прямоугольника блока. Вместе с координатами курсора координаты центра рукоятки тоже выйдут за прямоугольник блока, что, без ограничений на диапазон выходов, может привести к весьма неприятной ситуации: если перетащить круг рукоятки за пределы блока и отпустить там, в него уже невозможно будет попасть мышью, чтобы перетащить обратно. Действительно, по окончании

перетаскивания модель снимет захват мыши, поэтому реакция на нажатие кнопки будет вызываться только при нахождении курсора в пределах прямоугольника блока. А круг рукоятки находится за его пределами (к тому же еще и не изображается, поскольку функция `Draw` устанавливает отсечение рисования вне прямоугольника). Значит, в этом случае проверка попадания курсора в рукоятку в функции `MouseDown` никогда не выполнится, и новое перетаскивание никогда не начнется. Если же ограничивать выходы блока диапазоном [–1...1], центр рукоятки не сможет покинуть прямоугольник блока, как бы далеко от него ни переместил курсор пользователь.

Теперь, когда все функции класса написаны, можно написать функцию модели блока, которая будет их вызывать и работать с захватом мыши:

```
// Двухкоординатная рукоятка
extern "C" __declspec(dllexport)
  int RDSCALL SimpleJoystick(int CallMode,
                             RDS_PBLOCKDATA BlockData,
                             LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define x (*((double *)(pStart+2)))
#define y (*((double *)(pStart+10)))
  // Вспомогательная переменная - указатель на личную область,
  //приведенный к правильному типу
  TSimpleJoystick *data=(TSimpleJoystick*)(BlockData->BlockData);
  switch(CallMode)
    { // Инициализация
     case RDS_BFM_INIT:
        BlockData->BlockData=new TSimpleJoystick();
        break;
      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;
      // Проверка допустимости типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSDD}")?
                     RDS_BFR_BADVARSMSG:RDS_BFR_DONE;
      // Нажатие кнопки мыши
      case RDS_BFM_MOUSEDOWN:
        return data->MouseDown((RDS_PMOUSEDATA)ExtParam,x,y,
                               &(BlockData->Flags));
      // Отпускание кнопки мыши
      case RDS_BFM_MOUSEUP:
        // Снятие захвата мыши
        RDS_SETFLAG(BlockData->Flags,RDS_MOUSECAPTURE,FALSE);
        break;
      // Перемещение курсора мыши
      case RDS_BFM_MOUSEMOVE:
        // Проверка: включен ли захват мыши
        if(BlockData->Flags & RDS_MOUSECAPTURE)  // Включен
          { // Вызываем функцию реакции
            data->MouseMove((RDS_PMOUSEDATA)ExtParam,&x,&y);
            Ready=1;  // Взводим сигнал готовности
          }
        break;
```

```
        // Рисование
        case RDS_BFM_DRAW:
          data->Draw((RDS_PDRAWDATA)ExtParam,x,y,
                       BlockData->Flags & RDS_MOUSECAPTURE);
          break;
      }
    return RDS_BFR_DONE;
  // Отмена макроопределений
  #undef y
  #undef x
  #undef Ready
  #undef Start
  #undef pStart
  }
  //=======================================
```

Поскольку в этой модели предусмотрена личная область данных блока, при вызовах в режимах `RDS_BFM_INIT` и `RDS_BFM_CLEANUP` эта область, как обычно, создается и уничтожается. Вызов проверки типов статических переменных `RDS_BFM_VARCHECK` тоже не отличается от рассмотренных уже много раз. Мы не будем в очередной раз подробно описывать эти реакции, вместо этого сосредоточимся на реакциях модели на нажатие кнопок и перемещение курсора мыши.

При нажатии какой-либо кнопки мыши модель вызывается в режиме `RDS_BFM_MOUSEDOWN`. При этом она немедленно вызывает функцию-член личной области данных блока `MouseDown`, передавая ей приведенный к правильному типу указатель на структуру описания произошедшего события, текущие значения выходов блока `x` и `y`, а также указатель на поле флагов структуры `BlockData` для взведения в этом поле, при необходимости, флага захвата мыши.

При отпускании кнопки (реакция `RDS_BFM_MOUSEUP`) модель снимает захват мыши, сбрасывая флаг `RDS_MOUSECAPTURE`. Для сброса флага используется макрос для сброса и установки битовых флагов `RDS_SETFLAG`, описанный в “RdsDef.h” следующим образом:

```
#define RDS_SETFLAG(storage,mask,value) \
      ((storage) = (value)? \
      ((storage) | (mask)): \
      ((DWORD)((storage) & (~(mask)))))
```

Параметр `storage` соответствует переменной типа `DWORD`, в которой устанавливается или сбрасывается флаг, параметр `mask` – битовой маске флага, а вместо `value` подставляется `TRUE` для установки флага или `FALSE` для сброса. На самом деле, отключение захвата мыши можно было бы записать и так:

```
BlockData->Flags=BlockData->Flags & (~(RDS_MOUSECAPTURE));
```

однако, запись с макросом читается несколько лучше. Больше при отпускании кнопки мыши никаких действий не выполняется. Модель даже не проверяет, была ли захвачена мышь, прежде чем сбросить флаг захвата – если он и так сброшен, его повторный сброс ничему не помешает.

При перемещении курсора мыши модель вызывается в режиме `RDS_BFM_MOUSEMOVE`. Если флаг `RDS_MOUSECAPTURE` взведен (побитовое “И” поля `Flags` структуры `BlockData` с `RDS_MOUSECAPTURE` дает ненулевой результат), значит, мышь в данный момент захвачена (то есть идет перетаскивание рукоятки), и вызывается функция `MouseMove`. В нее, как и в `MouseDown`, передается указатель на структуру `RDS_MOUSEDATA`, содержащую координаты курсора мыши, текущий размер блока и т.п. Кроме того, ей передаются указатели на выходы блока `x` и `y`, чтобы функция могла изменить их значения. После вызова `MouseMove` выходу готовности блока присваивается единица, чтобы измененные значения выходов блока передались по связям.

Последняя реакция в модели блока – рисование его изображения (`RDS_BFM_DRAW`). В ней вызывается уже описанная функция `Draw`, в которую передается указатель на структуру параметров рисования, текущие значения выходов блока (чтобы функция нарисовала рукоятку в нужном месте) и признак захвата мыши, от которого зависит цвет рукоятки (при перетаскивании она меняет цвет на красный).

Для проверки работы получившейся модели нужно задать в параметрах блока программное рисование, разрешить блоку реагировать на мышь, и подключить к его выходам пару числовых индикаторов (см. рис. 75). В режиме расчета синий круг-рукоятку внутри блока можно будет перетаскивать левой кнопкой мыши, при этом значения на индикаторах будут меняться в соответствии с перемещением рукоятки. При выведении курсора за пределы прямоугольника блока рукоятка останется на его границе, а при возврате курсора обратно в прямоугольник снова будет следовать за ним. На самом деле, рукоятку можно перетаскивать и в режиме моделирования, но значения индикаторов при этом меняться не будут, поскольку данные по связям передаются только в режиме расчета.

§2.12.3. Реакция на мышь в блоках сложной формы

Рассматриваются особенности реакции на мышь в перекрывающихся блоках сложной формы, в которых часть изображения одного блока просматривается сквозь изображение другого. Приводится пример блока с “отверстием”, которое может открываться и закрываться, разрешая или запрещая тем самым щелчки мышью по полю ввода, видимому в отверстии.

Модель блока, если в его параметрах разрешена реакция на мышь, получает информацию о нажатии и отпускании кнопок и перемещении курсора, если курсор находится в пределах описывающего прямоугольника изображения блока (то есть прямоугольника с минимальной шириной и высотой, который можно описать вокруг изображения блока). Однако, довольно часто изображения блоков имеют сложную форму, и, если блоки в подсистеме перекрываются, в описывающий прямоугольник такого изображения могут попасть части изображений других блоков. При этом пользователь, щелкая мышью по ясно видимому в окне изображению, может не подозревать, что на самом деле это изображение перекрыто описывающим прямоугольником какого-либо другого блока, который “перехватит” этот щелчок.

Например, если блок 2 со сложным изображением на рис. 76 расположен на переднем плане, его описывающий прямоугольник (изображен пунктиром) будет перекрывать прямоугольник блока 1. Пользователь, щелкая мышью по изображению блока 1, будет ожидать реакции на этот щелчок именно от этого блока. Однако, поскольку блок 2 с его прямоугольником располагается поверх блока 1, информацию о нажатии кнопки мыши получит блок 2. Если в параметрах блока 2 не разрешена реакция на мышь, РДС не будет вызывать его модель и вызовет модель следующего по близости к переднему плану блока, то есть блока 1, что соответствует замыслу пользователя. Однако, если блок 2, как и блок 1, реагирует на мышь, будет вызвана именно его модель, что будет для пользователя полной неожиданностью, ведь он щелкал по ясно видимому изображению блока 1. Чтобы этого не происходило, модели сложных по форме блоков должны анализировать, не пришелся ли щелчок мыши на прозрачную (не занятую изображением) область описывающего прямоугольника и отказываться реагировать на мышь в таких случаях. Для этого функция модели, вызванная для реакции на мышь, должна

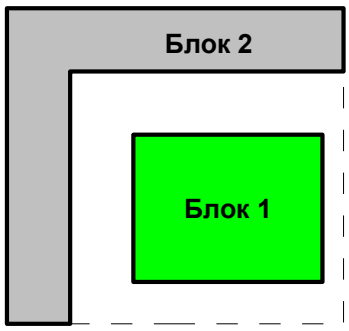


Рис. 76. Перекрывающиеся блоки

вернуть константу RDS_BFR_NOTPROCESSED вместо обычной RDS_BFR_DONE – это укажет РДС на то, что для реакции на мышь нужно вызвать модель другого блока.

В качестве примера создадим блок, который можно использовать для управления полем ввода. Блок будет иметь два вещественных входа: x_ext и x_int, вещественный выход out и внутреннюю логическую переменную bypass. У блока будет два состояния: открытое и закрытое. В открытом состоянии (bypass==0) блок будет выглядеть как рамка зеленого цвета с прозрачным окном внутри, при этом он будет передавать на выход out значение входа x_int. В закрытом состоянии (bypass==1) блок будет сплошным красным прямоугольником, внутри которого отображается значение выхода out, при этом на выход должно передаваться значение входа x_ext. Состояния блока будут переключаться по щелчку мыши. Если разместить этот блок поверх поля ввода, подключенного к его входу x_int, блок сможет управлять этим полем ввода. В открытом состоянии поле ввода будет видно в окне в центре блока, и на выход блока будет передаваться значение, введенное в поле (чтобы пользователь смог работать с полем ввода, необходимо написать модель блока так, чтобы она не реагировала на щелчки мыши внутри прозрачного окна). В закрытом состоянии блок будет перекрывать поле ввода, и, вместо его значения, будет передавать на выход значение с входа x_ext. На рис. 77 изображен пример подключения такого блока к полям ввода: в открытом состоянии на числовой индикатор будет выдаваться значение поля A2, в закрытом – A1. Выход блока out на рисунке подключен еще и ко входу v поля ввода A2, чтобы в закрытом состоянии, когда значение out совпадает с x_ext, это поле автоматически получало значения с поля A1.

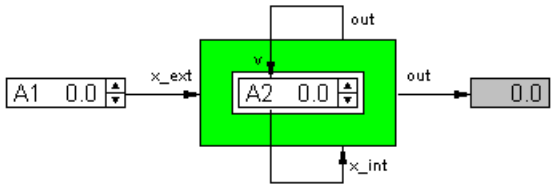


Рис. 77. Подключение блока управления полем ввода

Блок управления будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	1
1	Ready	Сигнал	1	Выход		–
2	x_ext	double	8	Вход	✓	–
10	x_int	double	8	Вход	✓	–
18	out	double	8	Выход		–
26	bypass	Логический	1	Внутренняя		0

Модель блока должна запускаться только при поступлении на его входы новых значений, поэтому в параметрах блока нужно включить запуск по сигналу и установить флаг “пуск” для всех входов блока. Сама модель будет иметь следующий вид:

```
extern "C" __declspec(dllexport)
  int RDSCALL EditControlFrame(int CallMode,
                               RDS_PBLOCKDATA BlockData,
                               LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define x_ext (*((double *)(pStart+2)))
#define x_int (*((double *)(pStart+10)))
```

```
#define out (*((double *)(pStart+18)))
#define bypass (*((char *)(pStart+26)))
  const int fr=20;    // Толщина рамки
  // Вспомогательные переменные
  RDS_PMOUSEDATA mouse;
  RDS_PDRAWDATA draw;
  int frz,x1,y1,x2,y2,xi1,yi1,xi2,yi2;

  switch(CallMode)
    { // Проверка типов статических переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSDDDL}")?
                          RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Реакция на нажатие кнопки мыши
      case RDS_BFM_MOUSEDOWN:
        mouse=(RDS_PMOUSEDATA)ExtParam;
        // Толщина рамки с учетом масштаба
        frz=fr*mouse->DoubleZoom;
        // В открытом состоянии при попадании курсора внутрь
        // прозрачного окна на щелчок реагировать не нужно
        if(bypass==0 &&
           mouse->x>mouse->Left+frz &&
           mouse->y>mouse->Top+frz &&
           mouse->x<mouse->Left+mouse->Width-frz &&
           mouse->y<mouse->Top+mouse->Height-frz)
          return RDS_BFR_NOTPROCESSED;
        // Если не левая кнопка - не обрабатываем щелчок
        // и разрешаем вывести контекстное меню, если нужно
        if(mouse->Button!=RDS_MLEFTBUTTON)
          return RDS_BFR_SHOWMENU;
        // Нажата левая кнопка мыши, причем курсор попал
        // в рамку или блок в закрытом состоянии
        bypass=!bypass;    // Переключаем состояние
        Ready=1;           // Взводим сигнал готовности
        // Здесь намеренно не поставлен оператор break: необходимо
        // выполнить действия в следующем case (такт расчета)

      // Один такт расчета
      case RDS_BFM_MODEL:
        // В зависимости от состояния, подаем на выход
        // один из входов
        out=bypass?x_ext:x_int;
        break;

      // Рисование внешнего вида блока
      case RDS_BFM_DRAW:
        draw=(RDS_PDRAWDATA)ExtParam;
        // Координаты описывающего прямоугольника блока
        x1=draw->Left;
        x2=draw->Left+draw->Width;
        y1=draw->Top;
        y2=draw->Top+draw->Height;
        // Толщина рамки с учетом масштаба
        frz=fr*draw->DoubleZoom;
        // Координаты окна внутри блока
        xi1=x1+frz;
        xi2=x2-frz;
```

```
            yi1=y1+frz;
            yi2=y2-frz;
            if(bypass)
              { // Закрытое состояние
                int w;
                char *text;
                // Рисуем красный прямоугольник с черной рамкой
                rdsXGSetPenStyle(0,PS_SOLID,1,0,R2_COPYPEN);
                rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xff);
                rdsXGRectangle(x1,y1,x2,y2);
                // Устанавливаем шрифт выстой в окно внутри блока
                rdsXGSetFont(0,"Arial",yi2-yi1,0,
                    DEFAULT_CHARSET,0,FALSE,FALSE,FALSE,FALSE);
                // Преобразуем значение выхода в динамическую строку
                text=rdsDtoA(out,-1,NULL);
                // Определяем ширину получившейся строки на экране
                rdsXGGetTextSize(text,&w,NULL);
                // Выводим значение выхода туда, где в открытом
                // состоянии находится прозрачное окно
                rdsXGTextOut(xi2-w,yi1,text);
                // Освобождаем динамическую строку
                rdsFree(text);
              }
            else
              { // Открытое состояние
                rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xff00);
                // Рисуем рамку вокруг прозрачного окна из четырех
                // зеленых прямоугольников
                rdsXGFillRect(x1,y1,x2,yi1);
                rdsXGFillRect(x1,yi2,x2,y2);
                rdsXGFillRect(x1,yi1,xi1,yi2);
                rdsXGFillRect(xi2,yi1,x2,yi2);
                // Обрамляем черными линиями
                rdsXGSetPenStyle(0,PS_SOLID,1,0,R2_COPYPEN);
                rdsXGSetBrushStyle(0,RDS_GFS_EMPTY,0);
                rdsXGRectangle(x1,y1,x2,y2);
                rdsXGRectangle(xi1,yi1,xi2,yi2);
              }
            break;
        }

      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef bypass
    #undef out
    #undef x_int
    #undef x_ext
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

В такте моделирования (RDS_BFM_MODEL) выходу out присваивается значение одного из входов в зависимости от значения переменной состояния bypass – это самая простая реакция модели. Реакция на мышь и процедура рисования немного сложнее.

В реакции на нажатие кнопки мыши (RDS_BFM_MOUSEDOWN) модель сначала проверяет, не попал ли курсор в прозрачное окно внутри блока. Для этого вычисляется

толщина рамки в текущем масштабе `frz` (толщина рамки в единичном масштабе задана как константа `fr` в начале функции). Затем, если блок в открытом состоянии (`bypass==0`), и курсор мыши находится далее чем в `frz` точках от его границ, функция немедленно возвращает константу `RDS_BFR_NOTPROCESSED`, информируя РДС о том, что щелчок пришелся в прозрачное окно и данная модель отказывается от его обработки. В противном случае щелчок пришелся на рамку или на прямоугольник блока в закрытом состоянии. Если была нажата не левая кнопка (нас интересуют только щелчки левой), функция возвращает константу `RDS_BFR_SHOWMENU`, информируя РДС о том, что, хотя модель и обработала нажатие кнопки, контекстное меню блока по правой кнопке вызвать все равно нужно (это стандартная практика при написании моделей блоков – если функция модели, обработав щелчок мыши, не хочет блокировать вызов контекстного меню блока, она должна вернуть `RDS_BFR_SHOWMENU`). Если же была нажата именно левая кнопка мыши, `bypass` инвертируется (состояние блока переключается), взводится сигнал готовности `Ready` для передачи выхода блока по связям в ближайшем такте расчета, и, из-за опущенного оператора `break`, выполняется следующая далее по тексту программы реакция на такт расчета, вычисляющая значение выхода блока по новому значению `bypass`.

При рисовании внешнего вида блока также вычисляется толщина рамки в текущем масштабе `frz` – она потребуется и в открытом состоянии блока (для рисования рамки вокруг прозрачного окна), и в закрытом (для вывода значения выхода блока туда, где в открытом состоянии через окно видно поле ввода). Кроме того, во вспомогательные переменные записываются вычисленные координаты описывающего прямоугольника блока `x1`, `y1`, `x2`, `y2`, и координаты прямоугольника окна внутри блока `xi1`, `yi1`, `xi2`, `yi2` – они смещены внутрь блока на `frz` точек. Далее рисование производится по-разному для открытого и закрытого состояния.

В закрытом состоянии (`bypass!=0`) функция рисует красный прямоугольник размером с описывающий прямоугольник блока. Затем, при помощи функции `rdsXGSetFont`, устанавливаются параметры шрифта, которым будет выведено текущее значение выхода блока, причем в качестве высоты шрифта в функцию передается высота окна внутри блока `yi2-yi1`. Вещественное значение `out` преобразуется в динамически отведенную строку функцией `rdsDtoA` (она уже описывалась ранее, см. стр. 256), после чего эта строка выводится на экран выровненной по правой границе внутреннего окна блока `xi2`. Затем память, отведенная под строку, освобождается функцией `rdsFree`, и рисование на этом завершается.

В открытом состоянии (`bypass==0`) необходимо нарисовать зеленый прямоугольник с прозрачным окном посередине. Поскольку ни в Windows API, ни в графических функциях РДС нет функции для рисования прямоугольника с отверстием, проще всего нарисовать рамку из четырех прямоугольников вокруг прозрачного окна, а затем, отключив заливку, отдельно нарисовать черные линии, ограничивающие блок и окно в его центре. Именно это и делается в модели блока.

Для проверки работы модели следует задать в параметрах блока программное рисование и разрешить ему реакцию на мышь. Затем следует расположить этот блок поверх поля ввода, подобрав его размер так, чтобы это поле было видно сквозь прозрачное окно в блоке, и провести соединения блока с этим полем и еще парой дополнительных блоков, как показано на рис. 77. Теперь, запустив расчет, можно подавать на числовой индикатор значение с “внешнего” или “внутреннего” полей ввода (рис. 78, связи между блоком управления и полем A2 убраны на невидимый слой), переключая состояние блока щелчками мыши. При этом, когда блок находится в открытом состоянии (зеленый цвет), поле ввода, видимое через прозрачное окно, должно реагировать на щелчки мыши, несмотря на то, что формально оно перекрыто блоком управления.

Можно заметить, что у созданного блока есть один недостаток. В режимах моделирования и расчета все работает так, как и было задумано: щелчки в прозрачном окне блока не обрабатываются моделью, попадают в поле ввода и позволяют изменять его значение. Однако, в режиме редактирования модель блока на действия мышью не реагирует, поэтому, хотя поле ввода и видно через окно в прямоугольнике блока, выбрать его мышью, вызвать окно настроек или контекстное меню невозможно: все щелчки будут приходиться на лежащий выше блок. Для того, чтобы модель могла сделать блок “прозрачным” для щелчков и в режиме редактирования, в нее нужно включить реакцию на выбор блока мышью RDS_BFM_MOUSESELECT. Эта реакция очень похожа на реакцию на нажатие кнопки мыши в режимах моделирования и расчета: в параметре ExtParam также передается указатель на структуру RDS_MOUSEDATA, содержащую координаты курсора мыши, размеры описывающего прямоугольника блока, текущий масштаб и т.д., однако, она вызывается только в режиме редактирования. Если в ответ на этот вызов модель вернет RDS_BFR_DONE, точка будет считаться принадлежащей блоку, и РДС выполнит для него все обычные действия режима редактирования (выделит при щелчке левой кнопкой, откроет контекстное меню при щелчке правой, откроет окно параметров или настройки при двойном щелчке). Если же модель вернет RDS_BFR_NOTPROCESSED, РДС пропустит этот блок и будет искать другой, лежащий ниже, по этим же координатам.

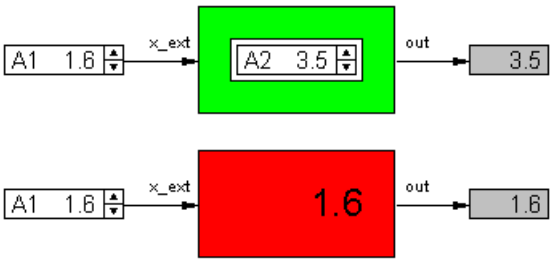


Рис. 78. Блок управления полем ввода в открытом (вверху) и закрытом (внизу) состоянии

Добавим в нашу модель реакцию на это событие. В оператор switch(CallMode) нужно добавить следующий case:

```
// Проверка возможности выбора блока мышью
case RDS_BFM_MOUSESELECT:
  mouse=(RDS_PMOUSEDATA)ExtParam;
  frz=fr*mouse->DoubleZoom; // Толщина рамки
  // Проверка попадания в прозрачное окно
  if(bypass==0 &&
     mouse->x>mouse->Left+frz &&
     mouse->y>mouse->Top+frz &&
     mouse->x<mouse->Left+mouse->Width-frz &&
     mouse->y<mouse->Top+mouse->Height-frz)
    return RDS_BFR_NOTPROCESSED;
  // В окно не попали - функция вернет RDS_BFR_DONE
  break;
```

Новая реакция практически идентична реакции RDS_BFM_MOUSEDOWN, за исключением действий по переключению состояния блока – в режиме редактирования этого делать не нужно. При желании, можно совместить действия этих двух реакций, поставив их операторы case друг за другом.

Из-за похожести реакций на события RDS_BFM_MOUSEDOWN и RDS_BFM_MOUSESELECT у программиста может возникнуть искушение использовать последнее для реакции на щелчок мыши в режиме редактирования. Следует, однако, иметь в виду, что, независимо от того, какую кнопку нажмет пользователь на блоке при его выделении в режиме редактирования, в параметрах события RDS_BFM_MOUSESELECT будет указана левая. Это событие предназначено только для проверки возможности выделения блока щелчком мыши по указанной точке, и использовать его в других целях нежелательно.

В этом примере форма блока (прямоугольник с окном) задавалась программно. Таких же результатов можно достичь и с использованием векторных картинок: нужно каждому

элементу картинки присвоить ненулевой идентификатор. Тогда, если функция `rdsGetMouseObjectId` (см. стр. 265) вернет нулевое значение, это даст модели понять, что курсор не попал ни в один из элементов картинки. Вернув в этом случае константу `RDS_BFR_NOTPROCESSED`, модель позволит отозваться на щелчок мыши какому-либо другому блоку, изображение которого видно под картинкой данного. При этом следует помнить, что проверка попадания курсора мыши в тот или иной векторный элемент картинки блока в функции `rdsGetMouseObjectId` также производится по описывающему прямоугольнику этого элемента. Например, если картинка блока 2 на рис. 76 будет представлять собой один Г-образный многоугольник, любой щелчок мышью в пределах описывающего прямоугольника этого многоугольника будет считаться попаданием в сам многоугольник, и блок 1 снова окажется перекрыт. Чтобы избежать этого, нужно либо составить картинку блока 2 из двух прямоугольников с ненулевыми идентификаторами, либо присвоить многоугольнику нулевой идентификатор (исключив его тем самым из проверки на попадание курсора мыши) и наложить на него два специальных прямоугольных векторных элемента "зона" (которые никак не отображаются на внешнем виде блока) с ненулевыми идентификаторами. Эти элементы специально предназначены для того, чтобы создавать в картинке блока зоны, чувствительные к нажатию кнопок мыши, не затрагивая внешний вид самой картинки. В данном случае, если покрыть такими элементами все непрозрачные детали картинки блока, цель будет достигнута: щелчок по видимым элементам картинки (на самом деле пришедшийся на добавленные "зоны") будет считаться попаданием в блок, щелчок по остальной площади описывающего прямоугольника блока (где зон нет) – не будет.

§2.12.4. Реакция блоков на клавиатуру

Рассматривается реакция модели блока на нажатие и отпускание клавиш. В один из рассмотренных ранее примеров добавляется возможность увеличивать и уменьшать значение выхода блока нажатием клавиш.

При необходимости, модель блока может обрабатывать события нажатия и отпускания клавиш – `RDS_BFM_KEYDOWN` и `RDS_BFM_KEYUP` соответственно. Как и в случае реакции на мышь, реакция блока на клавиатуру возможна только в режимах моделирования и расчета, в режиме редактирования информация о нажатии и отпускании клавиш в модель не передается. Для того, чтобы модель блока могла среагировать на одно из клавиатурных событий, должны одновременно выполниться два условия:

- окно подсистемы, в которой находится блок, должно иметь фокус ввода (быть самым верхним окном, при условии, что РДС – активное приложение);
- в параметрах блока должна быть разрешена реакция на клавиатуру.

В РДС нет понятия "текущего блока" или "блока, имеющего фокус ввода". При нажатии или отпускании какой-либо клавиши поочередно вызываются модели всех блоков в активном окне подсистемы, для которых разрешена реакция на клавиатуру. Порядок вызова моделей определяется внутренней логикой РДС, и ни разработчик, ни пользователь не могут на него повлиять. Единственное, что может сделать модель блока – это, среагировав на интересующее ее сочетание клавиш, сообщить РДС о том, что событие уже обработано, и дальнейший вызов моделей не требуется. При этом РДС не только прекращает перебирать и вызывать блоки, которым разрешена реакция на клавиатуру, но и не выполняет пункт главного меню, которому соответствует нажатая комбинация клавиш (если, конечно, такой имеется). Например, пункту главного меню РДС меню "Расчет | Стоп" соответствует "горячая клавиша" F7, поэтому нажатие этой клавиши обычно останавливает запущенный расчет. Однако, если в активном в данный момент окне подсистемы будет находиться блок, модель которого среагирует на нажатие F7 и сообщит РДС, что событие обработано, расчет остановлен не будет. Таким образом, обработка клавиатуры моделями блоков имеет приоритет над стандартными реакциями РДС. Тем не менее, если в окне подсистемы будет

два блока, обрабатывающих нажатие одной и той же клавиши и сообщающих РДС об успешной обработке события, нельзя заранее сказать, модель которого из блоков будет выполнена.

Реакция на нажатие и отпускание клавиш используется в моделях блоков реже, чем реакция на мышь. Как правило, такие реакции используются для включения и выключения различных кнопок, подачи команд и т.п. На клавиатуру могут реагировать все блоки активной подсистемы, независимо от видимости и активности слоя, на котором они расположены, что позволяет скрыть от пользователя логику обработки клавиатуры в подсистеме. Однако, следует помнить, что при закрытии окна подсистемы или уходе его с переднего плана блоки этой подсистемы перестают реагировать на клавиатуру, поэтому для вызова каких-либо общесистемных функций такие реакции непригодны. Если разработчику необходимо сделать так, чтобы модель блока реагировала на нажатие какой-либо комбинации клавиш независимо от того, открыто ли окно подсистемы с этим блоком и имеет ли это окно фокус, он может добавить в модель регистрацию дополнительного пункта системного меню и связать с этим пунктом "горячую клавишу". При этом нажатие этой клавиши будет вызывать модель блока для реакции на добавленный пункт меню. Механизм регистрации пунктов системного меню и реакции на них будет описан в §2.12.7.

Ранее (стр. 262) мы создали блок, увеличивающий и уменьшающий значение своего выхода при щелчках левой кнопкой мыши на его изображении. Добавим в него похожую реакцию на клавиатуру: нажатие одной клавиши (или сочетания клавиш) будет увеличивать значение выхода на единицу, нажатие другой – уменьшать. Причем клавиши, на которые реагирует блок, мы сделаем настраиваемыми.

Чтобы было где хранить коды клавиш, на которые реагирует блок, создадим для него личную область данных, как всегда оформив ее в виде класса:

```
//====== Класс личной области данных ======
class TPlusMinusData
{ public:
    int KeyPlus;      // Клавиша увеличения
    DWORD ShiftsPlus; // и ее флаги
    int KeyMinus;     // Клавиша уменьшения
    DWORD ShiftsMinus;// и ее флаги

    int Setup(void);       // Функция настройки клавиш
    void SaveBin(void);    // Сохранение параметров
    int LoadBin(void);     // Загрузка параметров

    // Конструктор класса
    TPlusMinusData(void)
      { KeyPlus=ShiftsPlus=KeyMinus=ShiftsMinus=0;};
};
//=========================================
```

В классе содержатся два поля для клавиши увеличения значения (`KeyPlus`, `ShiftsPlus`) и два поля для клавиши уменьшения (`KeyMinus`, `ShiftsMinus`). Каждое сочетание клавиш, на которые будет реагировать блок, задается двумя целыми числами: первое (`KeyPlus`, `KeyMinus`) содержит код клавиши, второе (`ShiftsPlus`, `ShiftsMinus`) – флаги, указывающие на состояние клавиш Ctrl, Alt и Shift. Например, для того, чтобы блок увеличил значение выхода при нажатии сочетания клавиш Ctrl+Alt+F1, в поле `KeyPlus` необходимо записать стандартную константу API Windows `VK_F1`, а в `ShiftsPlus` – объединенные битовым "ИЛИ" флаги `RDS_KCTRL` и `RDS_KALT`, описанные в "RdsDef.h". Нулевое значение поля флагов указывает на то, что блок должен реагировать на нажатие клавиши без Ctrl, Alt и Shift. Нулевое значение кода клавиши не соответствует ни одной

клавише клавиатуры, поэтому может использоваться как признак отсутствия реакции: оно никогда не совпадет с каким-либо кодом.

Для того, чтобы пользователь мог задать сочетания клавиш, на которые должен реагировать блок, в класс включена функция настройки `Setup`. Вводить числовые коды клавиш и флагов неудобно, поэтому в этой функции мы будем пользоваться специальными полями ввода для задания клавиш: при попадании в них курсора пользователь может просто нажать нужное сочетание клавиш, и оно отобразится в поле ввода в понятном ему виде. Например, на рис. 79 для увеличения и уменьшения значения выхода блока заданы сочетания клавиш Ctrl+“8 на цифровом блоке” и Ctrl+“2 на цифровом блоке” соответственно (8 и 2 на цифровом блоке клавиатуры соответствуют курсорным стрелкам вверх и вниз). Нажатие клавиши Backspace в таком поле ввода приведет к обнулению кода клавиши и, таким образом, к отмене реакции (в поле при этом будет написано слово “Нет”).

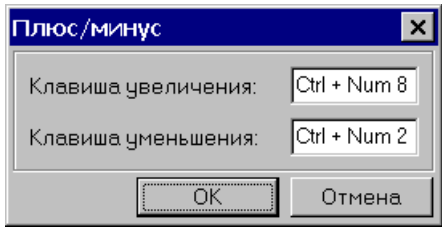


Рис. 79. Задание сочетаний клавиш

С использованием полей для ввода кодов клавиш функция `Setup` будет иметь следующий вид:

```
// Функция задания клавиш
int TPlusMinusData::Setup(void)
{ RDS_HOBJECT window; // Идентификатор вспомогательного объекта
  BOOL ok;            // Пользователь нажал "OK"
  // Создание окна
  window=rdsFORMCreate(FALSE,-1,-1,"Плюс/минус");
  // Добавление полей ввода
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_HOTKEY,
                 "Клавиша увеличения:",80);
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_HOTKEY,
                 "Клавиша уменьшения:",80);
  // Занесение исходных значений кодов клавиш в поля ввода
  rdsSetObjectInt(window,1,RDS_FORMVAL_VALUE,KeyPlus);
  rdsSetObjectInt(window,1,RDS_FORMVAL_HKSHIFTS,ShiftsPlus);
  rdsSetObjectInt(window,2,RDS_FORMVAL_VALUE,KeyMinus);
  rdsSetObjectInt(window,2,RDS_FORMVAL_HKSHIFTS,ShiftsMinus);
  // Открытие окна
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Нажата кнопка OK - запись кодов клавиш в класс
      KeyPlus=rdsGetObjectInt(window,1,RDS_FORMVAL_VALUE);
      ShiftsPlus=rdsGetObjectInt(window,1,RDS_FORMVAL_HKSHIFTS);
      KeyMinus=rdsGetObjectInt(window,2,RDS_FORMVAL_VALUE);
      ShiftsMinus=rdsGetObjectInt(window,2,RDS_FORMVAL_HKSHIFTS);
    }
  // Уничтожение окна
  rdsDeleteObject(window);
  // Возвращаемое значение
  return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
}
//=======================================
```

Для задания кодов клавиш используется новый тип поля ввода `RDS_FORMCTRL_HOTKEY`. Поскольку каждому сочетанию клавиш соответствует два целых числа (код клавиши и флаги), данные в это поле ввода заносятся двумя вызовами функции `rdsSetObjectInt`: с обычной константой `RDS_FORMVAL_VALUE` заносится код клавиши, а с константой `RDS_FORMVAL_HKSHIFTS` – флаги. Чтение данных также производится двумя вызовами

rdsGetObjectInt с теми же константами. В остальном эта функция ничем не отличается от других функций настройки, уже рассматривавшихся ранее.

Для сохранения и загрузки заданных пользователем клавиш (будет не очень хорошо, если ему придется каждый раз после загрузки схемы настраивать клавиши заново) будем использовать функции SaveBin и LoadBin соответственно. Как видно из их названий, на сей раз мы будем сохранять и загружать данные не в текстовом, а в двоичном формате. Недостатки двоичного формата уже были описаны в §2.8.1, но для данного простого блока его вполне можно использовать. Для того чтобы сделать формат более гибким и оставить возможность, при необходимости, легко изменить или дополнить его, будем использовать "теговую" запись: перед каждой группой данных (в данном случае – перед парой целых чисел, описывающих сочетание клавиш) будем записывать байт, значение которого будет указывать на назначение следующих за ним данных. В конце запишем нулевой байт, который будет означать конец данных. Функция сохранения параметров будет такой:

```
// Сохранение параметров
void TPlusMinusData::SaveBin(void)
{ BYTE tag;     // Переменная для байта тега

  tag=1;   // Тег 1 - клавиша увеличения
  rdsWriteBlockData(&tag,sizeof(tag));
  rdsWriteBlockData(&KeyPlus,sizeof(KeyPlus));
  rdsWriteBlockData(&ShiftsPlus,sizeof(ShiftsPlus));

  tag=2;   // Тег 2 - клавиша уменьшения
  rdsWriteBlockData(&tag,sizeof(tag));
  rdsWriteBlockData(&KeyMinus,sizeof(KeyMinus));
  rdsWriteBlockData(&ShiftsMinus,sizeof(ShiftsMinus));

  tag=0;   // Тег 0 - конец данных
  rdsWriteBlockData(&tag,sizeof(tag));
}
//=======================================
```

Если в будущем потребуется добавить в этот блок новые параметры, или изменить формат хранения кодов клавиш, нужно будет записывать новые группы данных с новыми тегами (3, 4, 5 и т.д.). Такая запись позволяет не терять совместимости со старым форматом.

Напишем теперь функцию загрузки для этого формата:

```
// Загрузка параметров
int TPlusMinusData::LoadBin(void)
{ BYTE tag;

  for(;;)  // Цикл до тех пор, пока данные не кончатся
    { // Читаем байт тега
      if(!rdsReadBlockData(&tag,sizeof(tag)))
        break;  // Тег не считан - данные кончились
      // Анализируем считанный тег
      switch(tag)
        { case 0:     // Конец данных блока
            return RDS_BFR_DONE; // Загрузка успешно завершена
          case 1:     // Данные клавиши увеличения
            rdsReadBlockData(&KeyPlus,sizeof(KeyPlus));
            rdsReadBlockData(&ShiftsPlus,sizeof(ShiftsPlus));
            break;
          case 2:     // Данные клавиши уменьшения
            rdsReadBlockData(&KeyMinus,sizeof(KeyMinus));
            rdsReadBlockData(&ShiftsMinus,sizeof(ShiftsMinus));
            break;
```

```
          default:    // Неопознанный тег
            return RDS_BFR_ERROR; // Сообщаем РДС об ошибке
        }
    }
  // Данные кончились до тега 0 - сообщаем об ошибке
  return RDS_BFR_ERROR;
}
//=======================================
```

Внутри этой функции находится цикл `for(;;)` без явного условия завершения, в теле которого первым делом из данных блока читается один байт – это должен быть байт тега. Если байт считать не удалось, значит, данные блока неожиданно закончились, и цикл завершается оператором `break` (при этом функция возвращает `RDS_BFR_ERROR`, сообщая РДС об ошибке). В противном случае считанный байт анализируется оператором `switch`, и, если его значение – 1 или 2, из данных блока читаются два целых поля класса, описывающие клавишу увеличения или уменьшения соответственно. Если считан тег 0, значит, на этом данные блока завершаются – функция возвращает `RDS_BFR_DONE`. Если же считанный байт имеет какое-либо другое значение, функция вернет `RDS_BFR_ERROR` – тег не опознан, в формате записи, реализованном в функции `SaveBin` он не предусмотрен, и загрузка данных невозможна.

Если в будущем мы добавим новые данные и соответствующие им новые теги в функцию `SaveBin`, в функцию `LoadBin` нужно будет просто добавить новые метки `case` и команды загрузки новых данных после них.

Описывая данный формат записи, следует сделать одно, довольно очевидное, замечание. В качестве тега мы здесь используем один байт, поэтому всего может быть 256 различных тегов. Что же делать, если, в процессе модернизации блока, нам потребуется записывать больше различных групп данных? В этом случае одно какое-либо значение тега (например, 255) нужно использовать в качестве префикса, за которым будет следовать новый тег. Таким образом, первые 255 параметров (включая тег конца данных) будут иметь однобайтовые теги 0...254, а следующие за ними – двухбайтовые: (255,0), (255,1), и т.д. Впрочем, в данном блоке нам вряд ли когда-либо понадобятся более 256 параметров.

Теперь, когда класс личной области данных блока и его функции описаны, внесем соответствующие изменения в функцию модели:

```
// Увеличение/уменьшение значения по щелчку и клавишам
extern "C" __declspec(dllexport)
  int RDSCALL PlusMinus(int CallMode,
                        RDS_PBLOCKDATA BlockData,
                        LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define v (*((int *)(pStart+2)))
  // Вспомогательная — указатель на структуру события мыши
  RDS_PMOUSEDATA mouse;
  // Вспомогательная — указатель на структуру события клавиатуры
  RDS_PKEYDATA key;
  // Указатель на личную область данных блока,приведенный к
  // правильному типу
  TPlusMinusData *data=(TPlusMinusData*)(BlockData->BlockData);
  switch(CallMode)
    { // Проверка типа статических переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSI}")?
                      RDS_BFR_BADVARSMSG:RDS_BFR_DONE;
```

```
    // Реакция на нажатие кнопки мыши
    case RDS_BFM_MOUSEDOWN:
      // Приведение ExtParam к нужному типу
      mouse=(RDS_PMOUSEDATA)ExtParam;
      if(mouse->Button==RDS_MLEFTBUTTON)
        { // Нажата левая кнопка
          // Проверяем, есть ли у блока картинка (получаем
          // описание блока)
          RDS_BLOCKDESCRIPTION descr;
          descr.servSize=sizeof(descr);
          rdsGetBlockDescription(BlockData->Block,&descr);
          if(descr.Flags & RDS_BDF_HASPICTURE)
            { // Картинка есть - определяем идентификатор
              // элемента под курсором
              int id=rdsGetMouseObjectId(mouse);
              v+=id;
            }
          else if(mouse->y<mouse->Top+mouse->Height/2)
            v++;    // В верхней половине блока - увеличиваем
          else
            v--;    // В нижней половине блока — уменьшаем
          // Взводим сигнал готовности
          Ready=1;
        }
      break;

    case RDS_BFM_INIT:   // Инициализация
      BlockData->BlockData=new TPlusMinusData();
      break;
    case RDS_BFM_CLEANUP:// Очистка данных
      delete data;
      break;
    case RDS_BFM_SETUP:  // Настройка параметров
      return data->Setup();
    case RDS_BFM_SAVEBIN:// Сохранение параметров
      data->SaveBin();
      break;
    case RDS_BFM_LOADBIN:// Загрузка параметров
      return data->LoadBin();

    // Реакция на нажатие клавиши
    case RDS_BFM_KEYDOWN:
      // Приведение ExtParam к нужному типу
      key=(RDS_PKEYDATA)ExtParam;
      // Сравнение нажатой клавиши с клавишами уменьшения
      // и увеличения
      if(key->KeyCode==data->KeyPlus &&
         key->Shift==data->ShiftsPlus)
        { v++; Ready=1; }
      else if(key->KeyCode==data->KeyMinus &&
              key->Shift==data->ShiftsMinus)
        { v--; Ready=1; }
      break;
  }
return RDS_BFR_DONE;
```

```
// Отмена макроопределений
#undef v
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Добавленные в модель реакции на события инициализации (RDS_BFM_INIT), очистки данных (RDS_BFM_CLEANUP), а также настройки (RDS_BFM_SETUP), загрузки (RDS_BFM_SAVEBIN) и сохранения (RDS_BFM_LOADBIN) параметров стандартны и уже неоднократно описывались ранее. Рассмотрим подробнее реакцию модели на нажатие клавиши: в этом случае она вызывается в режиме RDS_BFM_KEYDOWN.

При каждом вызове модели для реакции на нажатие или отпускание клавиши ей передается указатель на структуру RDS_KEYDATA, описывающую произошедшее событие:

```
typedef struct
{ int KeyCode;       // Код клавиши (Windows API)
  BOOL Repeat;       // При нажатии: нажатие из-за автоповтора
  int  RepeatCount;  // При Repeat==TRUE - число повторенных
                     // с прошлого вызова
  DWORD Shift;       // Флаги клавиатуры (RDS_M*, RDS_K*)
  int KeyEvent;      // Причина вызова - RDS_BFM_KEYDOWN или
                     // RDS_BFM_KEYUP
  BOOL Handled;      // Возвращаемый параметр - событие обработано
} RDS_KEYDATA;
typedef RDS_KEYDATA *RDS_PKEYDATA;
```

Нас будут интересовать поля KeyCode и Shift, содержащие код нажатой клавиши и флаги состояния Ctrl-Shift-Alt соответственно. Они сравниваются с кодами клавиш и флагами в параметрах блока, и, при совпадении с одной из пар чисел, значение выхода блока увеличивается или уменьшается на 1, после чего взводится сигнал готовности Ready, чтобы новое значение передалось по связям.

Если бы мы хотели сообщить РДС, что нажатие клавиши обработано, и дальнейшие реакции не требуются, мы могли бы присвоить полю Handled структуры RDS_KEYDATA значение TRUE, либо вернуть из функции модели не константу RDS_BFR_DONE, как обычно, а RDS_BFR_STOP (или любое другое ненулевое значение). Но нам этого не требуется, наоборот, для данной модели лучше, чтобы обработка клавиатуры продолжалась и после реакции. В этом случае несколько блоков, настроенные на одну и ту же клавишу, одновременно изменят значения своих выходов при ее нажатии.

Для проверки работы измененной модели необходимо разрешить в параметрах всех блоков, к которым она подключена, реакцию на клавиатуру и вызов функции настройки (рис. 80). Затем необходимо в настройках этих блоков задать клавиши для увеличения и уменьшения значений (см. рис. 79). К выходам блоков следует подключить какие-либо индикаторы. Теперь, если расчет запущен, и окно подсистемы, в которой находятся эти блоки, находится на переднем плане (имеет фокус), нажатие на клавиши, указанные в настройках блоков, будет приводить к изменению значений на подключенных к этим блокам индикаторах. Причем если несколько блоков настроены на использование одной и той же клавиши, их значения будут изменяться одновременно. Нужно еще раз подчеркнуть, что все это будет происходить только в том случае, если окно подсистемы будет на переднем плане. Если самым верхним будет другое окно (не обязательно окно другой подсистемы – это может быть главное окно РДС, окно редактора слоев и т.п.), то блоки не будут реагировать на клавиатуру.

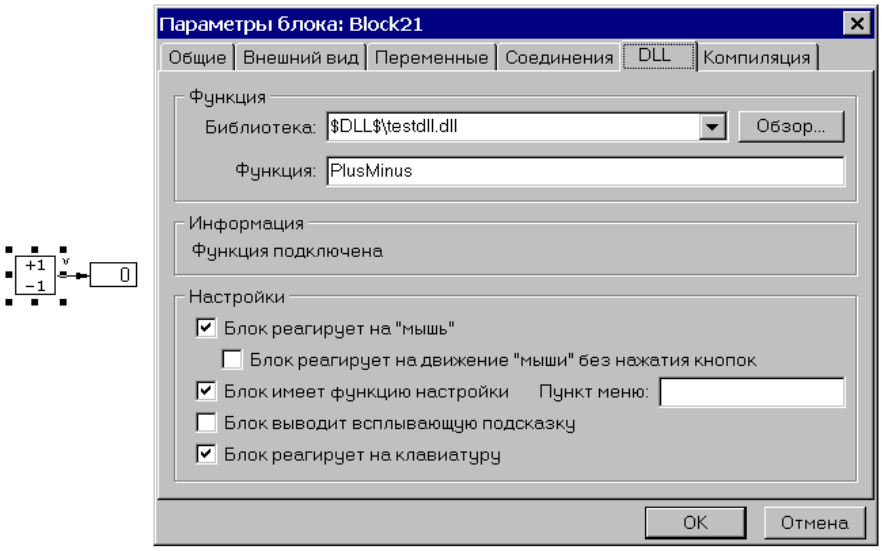


Рис. 80. Включение реакции на клавиатуру в параметрах блока

§2.12.5. Реакция окон подсистем на мышь и клавиатуру

Рассматривается возможность реакции модели подсистемы на действия мышью и нажатие и отпускание клавиш в ее окне, если ни один из внутренних блоков не среагировал на это событие. Приводится пример модели подсистемы, которая перемещает окно по курсорным клавишам и меняет свой масштаб по щелчку правой кнопкой мыши.

Если на нажатие или отпускание кнопок мыши и перемещение ее курсора не среагировал ни один из блоков подсистемы, это событие может обработать модель самой подсистемы (если, конечно, у подсистемы есть модель). Точно так же, если была нажата или отпущена какая-либо клавиша, и ни один из блоков активной подсистемы не сообщил РДС, что это событие обработано, РДС может вызвать соответствующую реакцию модели подсистемы. Разумеется, все это возможно только в режимах моделирования и расчета – в режиме редактирования РДС не передает информацию о мыши и клавиатуре моделям блоков и подсистем. Модели подсистем не участвуют в обработке данных, передающихся по связям от блока к блоку, и не вызываются в цикле в режиме расчета, поэтому в схемах они используются не так часто. Однако, в некоторых случаях, они могут быть полезны.

В качестве примера создадим модель подсистемы, которая при щелчке правой кнопкой мыши на рабочем поле будет переключать масштаб со 100% на 400% и обратно, прокручивая при этом рабочее поле так, чтобы точка щелчка пришлась в центр окна. Кроме того, при нажатии курсорных клавиш одновременно с клавишей Shift, окно подсистемы должно перемещаться в направлении нажатой клавиши на 5 точек экрана.

В целом, модели подсистем выглядят так же, как и модели простых блоков, за исключением того, что модель подсистемы не имеет доступа к статическим переменным. Наша модель будет выглядеть следующим образом:

```
// Модель подсистемы с реакцией на мышь и клавиатуру
extern "C" __declspec(dllexport)
  int RDSCALL SystemReaction(int CallMode,
                     RDS_PBLOCKDATA BlockData,
                     LPVOID ExtParam)
{ // Вспомогательные переменные
  RDS_EDITORPARAMETERS win;
  RDS_PMOUSEDATA mouse;
  RDS_PKEYDATA key;
  int x,y;
```

```
  switch(CallMode)
    { // Нажатие кнопки мыши в окне подсистемы
      case RDS_BFM_WINDOWMOUSEDOWN:
        // Приведение ExtParam к нужному типу
        mouse=(RDS_PMOUSEDATA)ExtParam;
        if(mouse->Button!=RDS_MRIGHTBUTTON)
          break; // Не правая кнопка - выходим
        // Вычисляем координаты щелчка в масштабе 100%
        x=mouse->x/mouse->DoubleZoom;
        y=mouse->y/mouse->DoubleZoom;
        // Установка нового масштаба подсистемы
        rdsSetZoomPercent(BlockData->Block,
                mouse->IntZoom>100?100:400,x,y);
        break;

      // Нажатие клавиши в окне подсистемы
      case RDS_BFM_WINDOWKEYDOWN:
        // Приведение ExtParam к нужному типу
        key=(RDS_PKEYDATA)ExtParam;
        // Проверяем, нажата ли Shift
        if(key->Shift!=RDS_KSHIFT)
          break; // Не нажата - выходим
        // Получаем текущее положение окна подсистемы
        win.servSize=sizeof(win);
        if(!rdsGetEditorParameters(BlockData->Block,&win))
          break;
        // Если окно свернуто или развернуто, перемещать не надо
        if(win.WinMaximized || win.WinMinimized)
          break;
        //В зависимости от нажатой клавиши, вычисляем
        // перемещение окна (x,y)
        x=y=0;
        switch(key->KeyCode)
          { case VK_LEFT:  x=-5;      break;     // Влево
            case VK_RIGHT: x=5;       break;     // Вправо
            case VK_UP:    y=-5;      break;     // Вверх
            case VK_DOWN:  y=5;       break;     // Вниз
          }
        // Если x или y не нулевые - перемещаем окно
        if(x||y)
          rdsSetSystemWindowBounds(BlockData->Block,FALSE,
                win.WinLeft+x,win.WinTop+y,
                win.WinWidth,win.WinHeight);
        break;
    }
  return RDS_BFR_DONE;
}
//=======================================
```

Модель состоит из двух реакций: реакции на нажатие кнопки мыши в окне подсистемы RDS_BFM_WINDOWMOUSEDOWN и реакции на нажатие клавиши в окне RDS_BFM_WINDOWKEYDOWN. Они похожи на соответствующие реакции блока (RDS_BFM_MOUSEDOWN и RDS_BFM_KEYDOWN), в параметре ExtParam при их вызове передаются указатели на те же самые структуры. Нужно помнить, что подсистема тоже является блоком в родительской подсистеме, и там, как и любой блок, она тоже может

реагировать на нажатие кнопок мыши и клавиш, получая обычные сообщения RDS_BFM_MOUSEDOWN и RDS_BFM_KEYDOWN (см. рис. 12).

Рассмотрим сначала реакцию на нажатие кнопки мыши. В параметре ExtParam в этом случае передается указатель на структуру описания события RDS_MOUSEDATA, который, после приведения к соответствующему типу, записывается во вспомогательную переменную mouse. Поскольку нас интересуют только щелчки правой кнопкой, нажатая кнопка сравнивается с константой RDS_MRIGHTBUTTON, и, при несовпадении, реакция завершается. Затем координаты щелчка приводятся к масштабу 100% (для этого их нужно поделить на масштабный коэффициент mouse->DoubleZoom) и вызывается сервисная функция rdsSetZoomPercent, устанавливающая новый масштаб указанной подсистемы. В первом параметре функции передается идентификатор подсистемы (в нашей модели – идентификатор данного блока, то есть BlockData->Block), во втором – устанавливаемый масштаб в процентах, в третьем и четвертом – координаты точки (указанные в масштабе 100%), которую нужно установить в центр окна. Масштаб, который мы устанавливаем, вычисляется на основе текущего масштаба в процентах mouse->IntZoom: если он больше 100, устанавливается 100%, если меньше или равен – 400%. Таким образом, вызов этой функции при масштабе 100% переключит его на 400%, а при 400% – обратно на 100%, что нам и требовалось.

Теперь рассмотрим реакцию на нажатие клавиши. В параметре ExtParam, как и в режиме RDS_BFM_KEYDOWN, передается указатель на структуру описания нажатой клавиши RDS_KEYDATA, который записывается во вспомогательную переменную key. Далее проверяется состояние флага клавиши Shift: если она не нажата, дальнейшая обработка не требуется, поскольку нажатия курсорных клавиш нас интересуют только при нажатой клавише Shift. Затем, вызывая уже знакомую нам функцию rdsGetEditorParameters, мы заполняем структуру win описанием окна подсистемы. В ней нас будут интересовать логические поля WinMaximized и WinMinimized, указывающие на то, что окно полностью развернуто или свернуто (в обоих случаях перемещать его нельзя), а также координаты левого верхнего угла окна подсистемы WinLeft и WinTop, и размеры этого окна WinWidth и WinHeight. Далее, в зависимости от нажатой курсорной клавиши, в переменных x и y формируется необходимое смещение окна на ±5 точек экрана в нужном направлении (если нажатая клавиша – не курсорная, значения x и y останутся нулевыми). Затем, если смещения не нулевые, вызывается функция установки положения и размеров окна подсистемы rdsSetSystemWindowBounds, в которую прежний левый верхний угол окна передается с вычисленными смещениями. В первом параметре этой функции передается идентификатор подсистемы, окно которой двигается, во втором – логическое значение, указывающее на необходимость развернуть окно на весь экран (в данном случае FALSE, разворачивать не нужно). Третий и четвертый параметры – новые координаты левого верхнего угла, пятый и шестой – новые ширина и высота (в нашем случае передаются старые, полученные из структуры описания окна, поскольку размер окна мы не меняем).

Чтобы проверить работу этой модели, необходимо подключить ее к какой-либо подсистеме и разрешить в параметрах этой подсистемы реакцию на необработанные блоками события мыши и клавиатуры (рис. 81).

Теперь в режимах моделирования и расчета нажатие курсорных клавиш одновременно с клавишей Shift должно перемещать окно по экрану, если это окно активно, а щелчок правой кнопкой мыши на рабочем поле окна должен менять масштаб со 100% на 400% и обратно. Разумеется, если щелкнуть правой кнопкой мыши по какому-либо блоку, которому разрешено реагировать на мышь, информацию о щелчке получит модель этого блока, и масштаб окна не изменится, поскольку модель подсистемы не будет вызвана (если только модель среагировавшего блока не вернет константу RDS_BFR_NOTPROCESSED).

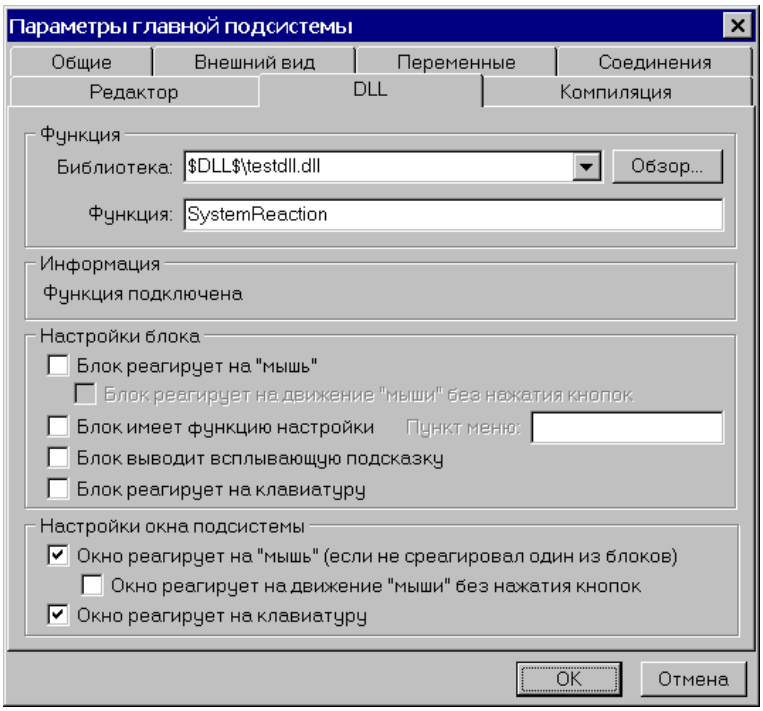


Рис. 81. Включение реакции на клавиатуру и мышь в параметрах подсистемы

Надо отметить, что для того, чтобы управлять окном подсистемы, не обязательно подключать к подсистеме модель. Функции установки масштаба и положения окна, использованные в этом примере, работают с любой подсистемой, идентификатор которой передан им в параметрах. Эти функции можно вызывать и из моделей простых блоков, указывая идентификатор родительской подсистемы. Например, можно разместить в подсистеме на заднем плане (или на самом дальнем слое) блок размером во всю рабочую область этой подсистемы, разрешить ему реакцию на мышь, и в его модели по щелчку правой кнопкой мыши менять масштаб подсистемы так же, как и в описанном примере. Таким же образом один из простых блоков может реагировать на нажатия курсорных клавиш при нажатой клавише Shift и перемещать окно родительской подсистемы. Где реализовывать те или иные функции – в модели подсистемы или в модели одного из ее блоков – разработчик решает сам, исходя из своих представлений об удобстве программирования.

§2.12.6. Добавление пунктов в контекстное меню блока

Рассматривается добавление дополнительных (постоянных и временных) пунктов в контекстное меню блока, вызываемое по правой кнопке мыши. В созданный ранее блок, имитирующий двухкоординатную рукоятку, добавляется возможность фиксировать одну из координат рукоятки, включаемая и выключаемая через контекстное меню. В другой блок, управляющий полем ввода, также добавляется возможность переключения состояния блока пунктом контекстного меню.

Контекстное меню в РДС, как и в большинстве других программ в Windows, вызывается при щелчке правой кнопкой мыши на выделенном объекте или группе объектов. Нас, в данном случае, будет интересовать контекстное меню блока, вызываемое щелчком правой кнопкой мыши на его изображении в окне подсистемы. В этом меню содержатся как пункты общего назначения (переключение режимов РДС, управление масштабом подсистемы), так и пункты, относящиеся к выбранному блоку (вызов окон настройки и параметров блока, копирование блока в буфер обмена, смена слоя и т.п.) Состав пунктов контекстного меню зависит от текущего режима РДС: в режимах моделирования и расчета, например, в меню не будет пунктов, позволяющих изменить какие-либо параметры блока.

На самом деле, в этих режимах в контекстном меню по умолчанию будут только пункты общего назначения, поскольку все стандартные пункты меню, относящиеся к конкретному блоку, предназначены для изменения тех или иных его параметров.

РДС позволяет моделям блоков добавлять собственные пункты в контекстное меню. Таким образом модель может предоставить пользователю быстрый доступ к различным функциям блока, причем возможность выбора этих пунктов меню пользователем может не зависеть от текущего режима РДС. В отличие от пункта контекстного меню, специально предназначенного для вызова окна настройки блока (см. §2.7), дополнительные пункты, добавленные моделью блока, могут вызываться не только из режима редактирования. Модель, при необходимости, может самостоятельно управлять их видимостью, разрешать и запрещать их, оперативно менять их названия и т.п.

Пункты, добавляемые моделью блока в контекстное меню, могут быть как постоянными, так и временными. Постоянные пункты могут добавляться в меню в любой момент и существуют до тех пор, пока не будут удалены из меню вызовом специальной сервисной функции, либо до тех пор, пока существует блок, модель которого их создала. Временные пункты могут добавляться в контекстное меню только в момент его открытия (для этого предусмотрен специальный вызов модели) и автоматически уничтожаются после закрытия меню. У каждого из двух этих типов есть свои достоинства и недостатки. Использование постоянных пунктов меню несколько облегчает работу программиста: в модель не нужно включать реакцию на открытие контекстного меню и размещать создание пунктов именно в ней, кроме того, в любом месте модели можно управлять видимостью и внешним видом пунктов меню. Однако, каждый постоянный пункт меню занимает место в памяти в течение всего времени своей жизни, и, если в системе будет несколько тысяч блоков, каждый из которых создаст один или несколько пунктов меню, общие потери памяти могут стать ощутимыми. При этом тот факт, что тысяча блоков имеет одну и ту же модель, создающую для каждого из этих блоков один и тот же дополнительный пункт меню, не поможет сэкономить память: физически это будут разные пункты меню, иначе блоки не смогли бы управлять своими пунктами независимо.

Временные пункты меню не занимают много памяти, поскольку немедленно уничтожаются при закрытии меню, но создавать их можно только в специально предусмотренной для этого реакции модели на открытие контекстного меню. Управлять внешним видом временных пунктов нельзя – вместо этого их нужно создавать сразу в том виде, в котором их должен увидеть пользователь в данный момент (то есть в текущем режиме, в текущем состоянии блока и т.п.)

Рассмотрим сначала работу с постоянными пунктами меню. Ранее (стр. 268) мы создали блок, имитирующий двухкоординатную рукоятку. Добавим в него возможность фиксировать одну из координат, разрешая пользователю перемещать рукоятку только по другой координате. Зафиксировав координату X, он сможет перемещать рукоятку только по вертикали, зафиксировав Y – только по горизонтали. Для управления этими функциями добавим в контекстное меню блока пункты “Фиксировать X” и “Фиксировать Y”, причем около пункта, соответствующего включенной в данный момент функции, должна ставиться галочка. Первый выбор пункта будет включать фиксацию соответствующей координаты (одновременно отключая фиксацию другой, если она была выбрана: если зафиксировать обе координаты, перемещать рукоятку будет вообще невозможно), второй – выключать. Для того, чтобы по внешнему виду блока пользователь сразу мог определить, что одна из координат зафиксирована, будем закрашивать серым цветом ту часть прямоугольника, в которую пользователь не сможет переместить рукоятку.

Прежде всего, необходимо внести изменения в описание класса блока: нам потребуется несколько новых полей и функция реакции на выбор пункта меню.

```
//====== Класс личной области данных ======
class TSimpleJoystick
{ private:
    // Центр круга (рукоятки) до начала перетаскивания
    int OldHandleX,OldHandleY;
    // Координаты курсора на момент начала перетаскивания
    int OldMouseX,OldMouseY;
    // Флаги фиксации одной из координат
    BOOL LockX,LockY;
    // Идентификаторы добавленных пунктов меню
    RDS_MENUITEM MenuLockX,MenuLockY;
  public:
    // Настроечные параметры блока
    COLORREF BorderColor;       // Цвет рамки блока
    COLORREF FieldColor;        // Цвет прямоугольника
    COLORREF HandleColor;       // Цвет круга в покое
    COLORREF MovingHandleColor; // Цвет круга при таскании
    COLORREF GrayedColor;       // Цвет недоступной области
    int HandleSize;             // Диаметр круга

    // Реакция на нажатие кнопки мыши
    int MouseDown(RDS_PMOUSEDATA mouse,double x,double y,
                  DWORD *pFlags);
    // Реакция на перемещение курсора мыши
    void MouseMove(RDS_PMOUSEDATA mouse,double *px,double *py);
    // Рисование изображения блока
    void Draw(RDS_PDRAWDATA draw,double x,double y,BOOL moving);
    // Реакция на выбор добавленного пункта меню
    void MenuFunction(RDS_PMENUFUNCDATA MenuData);

    // Конструктор класса
    TSimpleJoystick(void)
      { BorderColor=0;          // Черная рамка
        FieldColor=0xffffff;    // Белое поле
        HandleColor=0xff0000;   // Синий круг
        MovingHandleColor=0xff; // Красный при таскании
        GrayedColor=0x7f7f7f;   // Серый
        LockX=LockY=FALSE;      // Фиксация выключена
        HandleSize=20;          // Диаметр круга
        // Создание пунктов меню
        MenuLockX=rdsRegisterContextMenuItem("Фиксировать X",1,0);
        MenuLockY=rdsRegisterContextMenuItem("Фиксировать Y",2,0);
      };

      // Деструктор класса
      ~TSimpleJoystick()
      { // Уничтожение пунктов меню
        rdsUnregisterMenuItem(MenuLockX);
        rdsUnregisterMenuItem(MenuLockY);
      };
};
//=======================================
```

В закрытую область класса добавлены два логических поля `LockX` и `LockY`, которые будут отвечать за фиксацию горизонтальной и вертикальной координаты соответственно. Там же находятся поля `MenuLockX` и `MenuLockY`, в которых будут храниться идентификаторы созданных моделью дополнительных пунктов меню. Эти идентификаторы имеют тип

RDS_MENUITEM (он описан в “RdsDef.h”), мы будем использовать их при вызове сервисных функций РДС для установки и снятия галочек рядом с этими пунктами меню. В настроечные параметры блока добавлено новое поле GrayedColor для хранения цвета, которым мы будем закрашивать недоступную для рукоятки область прямоугольника блока. Также добавлена новая функция-член класса MenuFunction для реакции на выбор пользователем одного из добавленных пунктов меню (позднее мы рассмотрим ее подробно), внесены изменения в конструктор класса и добавлен деструктор.

Самое важное изменение в конструкторе класса – это создание в нем двух постоянных пунктов контекстного меню при помощи сервисной функции РДС rdsRegisterContextMenuItem. Эта функция принимает три параметра: строку названия пункта меню (именно эту строку пользователь увидит в меню) и два произвольных целых числа, которые связываются с создаваемым пунктом. Когда пользователь выберет в меню дополнительный пункт, эти два числа будут переданы в модель блока, и по ним она сможет определить, какой именно пункт выбран. Первое число обычно называется номером (или идентификатором) функции меню, второе – данными меню. В нашем случае пункт “Фиксация X” связывается с парой (1, 0), “Фиксация Y” – с парой (2, 0). На самом деле, практически всегда для точной идентификации выбранного пункта достаточно одного целого числа, второе число связывается с пунктом меню для удобства программиста: например, группа сходных по смыслу пунктов меню может иметь одинаковый номер функции меню, но разные данные. В нашей модели при выборе пункта меню мы будем анализировать только первое число (номер функции), второе будет игнорироваться.

Функция rdsRegisterContextMenuItem возвращает идентификатор созданного пункта меню типа RDS_MENUITEM (на самом деле, этот идентификатор является указателем на внутренний объект РДС, в котором хранятся данные пункта меню, но модель блока не может работать с ним напрямую, не используя сервисные функции). Идентификаторы пунктов “Фиксация X” и “Фиксация Y” присваиваются полям класса MenuLockX и MenuLockY соответственно.

В деструкторе класса созданные пункты меню уничтожаются сервисной функцией rdsUnregisterMenuItem. Можно было бы и не уничтожать их – при отключении модели от блока они уничтожатся сами – но хороший стиль программирования требует явно уничтожать все то, что было создано.

Теперь внесем изменения в функцию рисования блока Draw. Если одна из координат зафиксирована (то есть если одна из переменных LockX или LockY имеет значение TRUE), нужно закрасить серым цветом недоступную для рукоятки область. Если зафиксирована горизонтальная координата, следует нарисовать слева и справа от рукоятки два серых прямоугольника, так что белой останется только узкая вертикальная полоса шириной с рукоятку, по которой круг рукоятки сможет перемещаться. Если зафиксирована вертикальная координата, белая полоса должна быть горизонтальной, то есть нужно рисовать серые прямоугольники выше и ниже рукоятки. Изменения в функции выглядят следующим образом:

```
// Рисование изображения блока
void TSimpleJoystick::Draw(RDS_PDRAWDATA draw,
                           double x,double y,BOOL moving)
{ // ...
  // ...(начало функции без изменений)...
  // ...

  // Установка области отсечения
  r.left=draw->Left+1;
  r.top=draw->Top+1;
  r.right=draw->Left+draw->Width-1;
  r.bottom=draw->Top+draw->Height-1;
```

```
    rdsXGSetClipRect(&r);

    // Рисование ограничений
    if(LockX||LockY) // Фиксируется одна из координат
      { // Установка серого цвета заливки
        rdsXGSetBrushStyle(0,RDS_GFS_SOLID,GrayedColor);
        if(LockX) // Фиксируется X
          { rdsXGFillRect(r.left,r.top,hx-hR,r.bottom); // Слева
            rdsXGFillRect(hx+hR,r.top,r.right,r.bottom);// Справа
          }
        else // Фиксируется Y
          { rdsXGFillRect(r.left,r.top,r.right,hy-hR);  // Сверху
            rdsXGFillRect(r.left,hy+hR,r.right,r.bottom);// Снизу
          }
      }

    // Линии перекрестия
    rdsXGMoveTo(cx,draw->Top);
    rdsXGLineTo(cx,draw->Top+draw->Height);
    rdsXGMoveTo(draw->Left,cy);
    rdsXGLineTo(draw->Left+draw->Width,cy);

    // ...(далее без изменений)...
```

Если LockX или LockY – TRUE, устанавливается серый (GrayedColor из настроек блока) цвет заливки и функцией rdsXGFillRect рисуются два вертикальных или горизонтальных, в зависимости от зафиксированной координаты, прямоугольника. Разрыв между прямоугольниками вычисляется по координатам центра рукоятки (hx, hy) и радиусу рукоятки hR.

В функцию реакции на перемещение курсора мыши тоже необходимо внести изменения: при зафиксированной координате перемещение рукоятки по соответствующей оси должно быть запрещено:

```
// Реакция на перемещение курсора мыши
void TSimpleJoystick::MouseMove(RDS_PMOUSEDATA mouse,
                                double *px,double *py)
{ // ...
  // ...(начало функции без изменений)...
  // ...

  // По новым координатам центра рукоятки вычисляем соответствующие
  // им вещественные значения выходов, ограничивая их
  // диапазоном [-1...1]
  if(!LockX)
    { *px=2.0*(hx-cx)/mouse->Width;
      if(*px>1.0) *px=1.0;
      else if(*px<-1.0) *px=-1.0;
    }
  if(!LockY)
    { *py=-2.0*(hy-cy)/mouse->Height;
      if(*py>1.0) *py=1.0;
      else if(*py<-1.0) *py=-1.0;
    }
}
//=======================================
```

Наконец, добавим в модель реакцию на выбор пользователем пункта меню (RDS_BFM_MENUFUNCTION) и напишем функцию-член класса, которая будет вызываться в

этой реакции. В оператор switch(CallMode) внутри функции модели нужно вставить вызов функции MenuFunction:

```
        // ...

        case RDS_BFM_DRAW:
          data->Draw((RDS_PDRAWDATA)ExtParam,x,y,
                       BlockData->Flags & RDS_MOUSECAPTURE);
          break;

        // Выбор пользователем добавленного пункта меню
        case RDS_BFM_MENUFUNCTION:
          data->MenuFunction((RDS_PMENUFUNCDATA)ExtParam);
          break;
      }
    return RDS_BFR_DONE;
  // Отмена макроопределений
  // ...
```

При вызове функции модели в режиме RDS_BFM_MENUFUNCTION в параметре ExtParam передается указатель на структуру RDS_MENUFUNCDATA, описанную в "RdsDef.h" следующим образом:

```
typedef struct
{ int Function; // Номер функции меню (первое целое число,
                // указанное при создании пункта меню)
  int MenuData; // Данные меню (второе целое число, указанное
                // при создании пункта меню)
} RDS_MENUFUNCDATA;
typedef RDS_MENUFUNCDATA *RDS_PMENUFUNCDATA;
```

Структура содержит всего два поля, в которые записываются целые числа, связанные с выбранным пользователем пунктом меню при его создании. Например, если пользователь выберет пункт "Фиксировать X", в поле Function структуры будет записано значение 1, а в поле MenuData – 0. Указатель на эту структуру, приведенный к правильному типу, передается в функцию MenuFunction, которую мы сейчас напишем:

```
// Функция реакции на выбор одного из пунктов меню
void TSimpleJoystick::MenuFunction(RDS_PMENUFUNCDATA MenuData)
{ switch(MenuData->Function)
    { case 1:         // Выбран пункт "Фиксировать X"
        LockX=!LockX; // Переключаем флаг фиксации X
        LockY=FALSE;  // Отключаем фиксацию Y
        break;
      case 2:         // Выбран пункт "Фиксировать Y"
        LockY=!LockY; // Переключаем флаг фиксации Y
        LockX=FALSE;  // Отключаем фиксацию X
        break;
    }
  // Установка галочек у пунктов меню в зависимости от
  // флагов фиксации координат
  rdsSetMenuItemOptions(MenuLockX,LockX?RDS_MENU_CHECKED:0);
  rdsSetMenuItemOptions(MenuLockY,LockY?RDS_MENU_CHECKED:0);
}
//=======================================
```

В этой функции мы опознаем выбранный пользователем пункт меню по полю Function переданной структуры: оно может иметь значения 1 ("Фиксировать X") или 2 ("Фиксировать Y"). В зависимости от выбранного пункта мы инвертируем флаг фиксации соответствующей координаты и сбрасываем флаг фиксации другой, чтобы не допустить одновременной фиксации обеих координат. Затем, в зависимости от текущего состояния флагов, сервисной

функцией `rdsSetMenuItemOptions` устанавливается галочка у пункта меню, соответствующего зафиксированной координате. В РДС есть и другие сервисные функции, позволяющие управлять параметрами пунктов меню (например, оперативно изменять название пункта или связанные с ним целые числа), но в данном примере нам нужно только ставить и убирать галочку.

Функция `rdsSetMenuItemOptions` позволяет изменить видимость пункта меню, разрешить или запретить его (запрещенные пункты меню отображаются серым цветом), а также установить или сбросить галочку слева от его названия. Она принимает два параметра: первый – идентификатор пункта меню, второй – целое число, представляющее собой набор битовых флагов, управляющих состоянием этого пункта. Для пунктов контекстного меню можно использовать любое сочетание следующих флагов:

- `RDS_MENU_CHECKED` – если флаг установлен, слева от названия пункта меню будет изображаться галочка;
- `RDS_MENU_DISABLED` – если флаг установлен, пункт меню будет запрещен (изображается серым, не может быть выбран пользователем);
- `RDS_MENU_HIDDEN` – если флаг установлен, пункт меню будет невидимым для пользователя;
- `RDS_MENU_DIVIDER` – если флаг установлен, вместо пункта меню будет создан горизонтальный разделитель, при этом переданное название пункта игнорируется, и выбор этого пункта пользователем будет невозможен.

В нашей функции мы используем единственный флаг для управления галочкой. Если координата зафиксирована, мы передаем в функцию значение `RDS_MENU_CHECKED` (флаг галочки взведен, остальные сброшены), это сделает пункт меню видимым, разрешенным и отмеченным галочкой). В противном случае мы передаем значение 0 (все три флага сброшены), что также сделает пункт меню видимым и разрешенным, но уже без галочки.

После внесения изменений в модель блока никакие дополнительные настройки не требуются: во всех режимах в контекстном меню блока должны появиться дополнительные пункты (рис. 82).

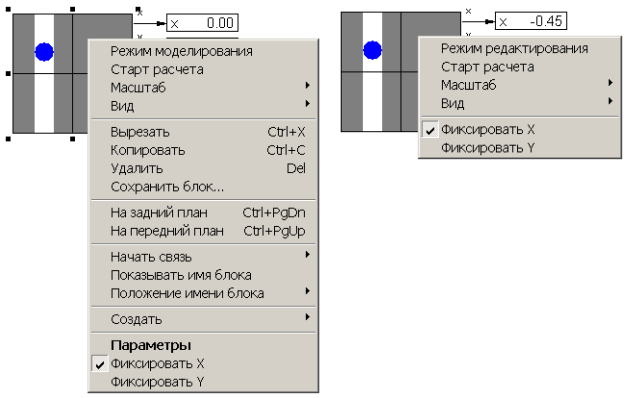


Рис. 82. Дополнительные пункты контекстного меню блока-рукоятки в режимах редактирования (слева) и моделирования (справа)

Приведенный пример демонстрирует работу с постоянными пунктами меню: мы создаем их при инициализации блока в конструкторе класса личной области данных, а затем меняем их параметры (устанавливаем и сбрасываем галочки) в процессе работы блока. Теперь рассмотрим работу с временными пунктами контекстного меню.

Ранее (стр. 277) мы создали блок для управления полем ввода, который может находиться в двух состояниях: открытом (при этом сквозь прозрачное окно в прямоугольнике блока видно лежащее под ним поле ввода) и закрытом. Состояния блока переключались щелчком левой кнопкой мыши. Добавим в этот пример возможность

открывать и закрывать блок пунктом контекстного меню – в этом случае переключать состояние блока можно будет и в режиме редактирования, что ранее было невозможно: в режиме редактирования нажатия кнопок мыши не передаются в модель. И для открытия, и для закрытия блока будем использовать один и тот же пункт меню, меняя его название в зависимости от текущего состояния: при закрытом блоке пункт будет называться “Открыть”, при открытом – “Закрыть”. И, поскольку полей ввода, а, значит, и блоков для их управления, в схеме может быть достаточно много, не будем зря тратить память и сделаем этот пункт контекстного меню временным.

Для реализации задуманного в модель нужно добавить две реакции: во-первых, реакцию на открытие контекстного меню RDS_BFM_CONTEXTPOPUP, в которой мы будем создавать временный пункт, во-вторых, уже знакомую нам реакцию на выбор пункта меню пользователем RDS_BFM_MENUFUNCTION – реакция на временные пункты меню ничем не отличается от реакции на постоянные. Внутрь оператора switch(CallMode) в функции модели добавляются два новых оператора case:

```
// Открытие контекстного меню блока
case RDS_BFM_CONTEXTPOPUP:
  // Добавление временного пункта меню
  rdsAdditionalContextMenuItemEx(bypass?"Открыть":"Закрыть",
                                        0,1,2);
  break;

// Выбор пункта меню пользователем
case RDS_BFM_MENUFUNCTION:
  bypass=!bypass;    // Переключить режим
  out=bypass?x_ext:x_int; // Подать на выход один из входов
  Ready=1;           // Взвести флаг готовности
  break;
```

В реакции на открытие контекстного меню (RDS_BFM_CONTEXTPOPUP) вызывается функция rdsAdditionalContextMenuItemEx, которая и создает нужный нам временный пункт меню. Функция принимает четыре параметра: строку с названием пункта (в нашем случае “Открыть” или “Закрыть”, в зависимости от значения переменной состояния bypass), битовые флаги, указывающие состояние пункта (у нас – 0, то есть видимый, разрешенный пункт меню без галочки) и, как и для постоянного пункта меню, два целых числа, которые будут переданы в модель при выборе этого пункта пользователем. В нашей модели всего один пункт контекстного меню, поэтому в реакции на выбор пункта пользователем мы не будем анализировать эти числа – если бы у нас было несколько пунктов меню, как в предыдущем примере, нам нужно было бы отличать один от другого, а с единственным пунктом такой необходимости нет. В вызове функции с пунктом меню связывается пара чисел (1, 2), но можно было бы указать любые другие числа.

Следует отметить, что функция rdsAdditionalContextMenuItemEx, в отличие от использованной в предыдущем примере rdsRegisterContextMenuItem, не возвращает уникальный идентификатор пункта меню. У временных пунктов меню вообще нет идентификаторов, и после их создания модель никак не может изменить их параметры. Этого и не требуется, поскольку временные пункты контекстного меню существуют только до момента закрытия этого меню, и каждый раз создаются заново в момент его открытия. Именно в момент создания временного пункта модель указывает его название и битовые флаги, определяющие внешний вид пункта на данный момент. В данном примере мы, вместо того, чтобы изменять название постоянного пункта меню в зависимости от текущего состояния блока (значения переменной bypass), в момент открытия меню создаем временный пункт сразу с нужным нам названием.

Во втором параметре функции rdsAdditionalContextMenuItemEx можно использовать уже описанные выше битовые флаги RDS_MENU_CHECKED,

RDS_MENU_DISABLED и RDS_MENU_HIDDEN. На самом деле, использование флага RDS_MENU_HIDDEN при создании временного пункта меню бессмысленно, поскольку этот пункт не будет виден пользователю на протяжении всего своего существования – проще вообще не создавать этот пункт. Может показаться, что создание временного пункта с флагом RDS_MENU_DISABLED тоже не имеет особого смысла – пункт также все время будет запрещенным до момента своего уничтожения, и модель не сможет разрешить его. Однако, этот пункт будет показан в меню и изображен серым цветом, что даст пользователю понять, что в других режимах этот пункт, вероятно, доступен для выбора. Это дает разработчику модели блока выбор: менять ли состав и названия пунктов меню в зависимости от состояния блока, или запрещать недоступные в данный момент пункты, оставляя их видимыми для пользователя. Какой вариант будет выглядеть лучше с точки зрения построения интерфейса – решать программисту.

В реакции на выбор пункта меню (RDS_BFM_MENUFUNCTION), как и было указано выше, не анализируются переданные в модель целые числа, связанные с выбранным пунктом. Наш единственный пункт меню должен инвертировать состояние блока, то есть, выполнять те же самые действия, что и при щелчке левой кнопкой мыши. Именно это и выполняется в реакции на событие RDS_BFM_MENUFUNCTION: значение переменной bypass инвертируется, на выход блока out подается один из входов (в зависимости от значения bypass) и взводится сигнал готовности Ready, чтобы в ближайшем такте расчета новое значение выхода было передано по связям. На самом деле, оператор case реакции на выбор пункта меню можно было бы вставить внутрь реакции на нажатие кнопки мыши перед оператором “bypass=!bypass;”, поскольку выполняемые ими действия совпадают:

```
        // ...

        if(mouse->Button!=RDS_MLEFTBUTTON)
          return RDS_BFR_SHOWMENU;
        // Нажата левая кнопка мыши, причем курсор попал
        // в рамку или блок в закрытом состоянии
      case RDS_BFM_MENUFUNCTION:
        bypass=!bypass;    // Переключаем состояние
        Ready=1;           // Взводим сигнал готовности
        // Здесь намеренно не поставлен оператор break: необходимо
        // выполнить действия в следующем case (такт расчета)

        // ...
```

Этого не было сделано только для большей понятности примера.

Посмотрев на тексты моделей двух рассмотренных примеров, можно заметить, что реакция на выбор пункта меню пользователем не зависит от того, временный это пункт или постоянный. В обоих случаях в модель передаются два целых числа, связанных с выбранным пунктом меню. Технически, создание временных пунктов меню предпочтительнее, поскольку они не занимают лишнего места в памяти, немного проигрывая постоянным только в необходимости включения в модель дополнительной реакции RDS_BFM_CONTEXTPOPUP. Использовать для расширения контекстного меню блока постоянные или временные пункты – решать программисту.

§2.12.7. Добавление пунктов в системное меню РДС

Описывается возможность добавления моделями блоков собственных пунктов в меню “Система”. Рассматривается пример блока, открывающего окно своей подсистемы при выборе пункта системного меню или при нажатии связанной с этим пунктом “горячей клавиши”.

Контекстное меню блока, дополнение которого описано в предыдущем параграфе, может быть вызвано, только если окно подсистемы с этим блоком открыто и сам блок видим

в этом окне. Если модели блока необходимо вынести какие-либо функции на уровень всей системы, чтобы пользователь мог вызвать их независимо от состояния окон подсистем, ей следует добавить пункт в системное меню. Добавляемые моделями блоков пункты размещаются в меню “Система | Дополнительно” главного окна РДС (см. рис. 13). С каждым из этих пунктов может быть связано вызывающее его сочетание клавиш, что позволяет использовать дополнение системного меню РДС для реализации “глобальной” реакции блока на клавиатуру: обычная реакция блока на нажатие клавиш, описанная в §2.12.4, возможна только в режимах моделирования и расчета (при этом окно подсистемы с блоком обязательно должно находиться на переднем плане), а реакция на нажатие сочетания клавиш, связанного с пунктом системного меню, не зависит ни от режима РДС, ни от расположения окна подсистемы, в которой находится добавивший этот пункт блок.

Пункты системного меню, в отличие от пунктов контекстного, не могут быть временными. Их, как правило, немного (слишком большое число пунктов в меню неудобно для пользователя, кроме того, площадь экрана ограничена), поэтому о занимаемом ими объеме памяти можно не беспокоиться. Они создаются сервисной функцией `rdsRegisterMenuItem` и существуют либо до тех пор, пока модель блока не удалит их, либо пока модель не будет отключена от блока (например, при удалении самого блока из схемы). Как и в случае постоянных пунктов контекстного меню, модель может в любой момент изменить название пункта системного меню и его внешний вид, а также связанное с ним сочетание клавиш.

Рассмотрим для примера блок, который добавляет в системное меню РДС пункт, выбор которого открывает окно подсистемы, в которой этот блок находится. Причем сделаем название этого пункта меню и его сочетание клавиш настраиваемыми. Такие блоки, размещенные в важных подсистемах, позволят пользователю быстро открывать их через меню или с клавиатуры, не тратя время на их поиск в схеме, чтобы открыть их двойным щелчком, как обычно.

Начнем написание модели блока с класса личной области данных – он понадобится нам для хранения идентификатора созданного пункта меню, а также настроечных параметров: названия пункта и сочетания клавиш, связанного с ним.

```
// Класс личной области данных блока
class TOpenSysWinData
{ public:
    RDS_MENUITEM MenuItem; // Идентификатор созданного пункта меню
    // Настроечные параметры
    char *Caption;    // Название пункта
    int Key;          // Клавиша (или 0,если ее нет)
    DWORD KeyShifts;  // Состояние Ctrl, Alt и Shift

    // Создание пункта меню с заданными параметрами или
    // изменение параметров уже созданного пункта
    void RegisterMenuItem(void);

    int Setup(void);       // Открыть окно настройки
    void SaveBin(void);    // Сохранить параметры
    int LoadBin(void);     // Загрузить параметры

    // Конструктор класса
    TOpenSysWinData(void)
      { Caption=NULL;
        Key=0; KeyShifts=0;
        MenuItem=NULL;
      };
```

```
    // Деструктор класса
    ~TOpenSysWinData()
      { // Освободить память, занятую строкой Caption
        rdsFree(Caption);
        // Удалить пункт меню
        rdsUnregisterMenuItem(MenuItem);
      };
};
//=======================================
```

Для хранения идентификатора пункта меню, который мы будем создавать, предназначено поле `MenuItem`. Оно имеет уже знакомый нам по работе с контекстным меню тип `RDS_MENUITEM` – в РДС для работы с постоянными пунктами контекстного и системного меню используются одни и те же функции и типы данных. В остальных полях класса хранятся настроечные параметры блока: строка названия пункта меню `Caption` (память под нее мы будем отводить динамически при помощи сервисных функций РДС), код “горячей клавиши” пункта меню `Key` и флаги `KeyShifts`, описывающие состояние клавиш-модификаторов Ctrl, Alt и Shift для этой клавиши.

В классе также описано несколько функций-членов, которые мы рассмотрим позднее, конструктор и деструктор. В конструкторе всем полям класса присваиваются начальные значения, а в деструкторе освобождается динамическая память, занятая строкой `Caption`, и удаляется созданный пункт меню. При удалении мы не проверяем, была ли на самом деле отведена память под строку и был ли создан пункт меню: все сервисные функции РДС, удаляющие какие-либо объекты и освобождающие память, допускают передачу значения `NULL` вместо указателей или идентификаторов, в этом случае никаких действий не производится.

Из всех функций-членов класса для нас интереснее всего функция `RegisterMenuItem`, предназначенная для создания пункта системного меню с заданными параметрами или изменения параметров этого пункта, если он уже создан:

```
// Создание или модификация пункта меню
void TOpenSysWinData::RegisterMenuItem(void)
{ // Вспомогательная переменная для заголовка пункта меню:
  // если Caption==NULL, пункт получит заголовок "Открыть окно"
  char *text=Caption?Caption:"Открыть окно";
  // Флаги пункта меню: если клавиша определена (Key!=0),
  // пункт будет иметь "горячую клавишу"
  DWORD options=Key?RDS_MENU_SHORTCUT:0;

  if(MenuItem)  // Пункт уже есть - изменяем
    rdsChangeMenuItem(MenuItem,text,options,Key,KeyShifts,0,0);
  else          // Пункта еще нет - создаем
    MenuItem=rdsRegisterMenuItem(text,options,Key,KeyShifts,0,0);
}
//=======================================
```

Поскольку `Caption` у нас исходно имеет значение `NULL`, а показывать пользователю пункт меню без названия нехорошо, в функции вводится вспомогательная переменная `text`, которая будет равна `Caption`, если значение последней не `NULL`, или ссылаться на строку “Открыть окно” в противном случае. Это значение будет передано в сервисную функцию в качестве названия пункта, таким образом, пункт не останется без названия в любом случае.

Другая вспомогательная переменная, `options`, содержит битовые флаги пункта меню. Кроме уже знакомых нам по работе с контекстными меню флагов `RDS_MENU_CHECKED`, `RDS_MENU_DISABLED` и `RDS_MENU_HIDDEN`, для пунктов системного меню могут использоваться еще два дополнительных флага:

- RDS_MENU_SHORTCUT – если флаг установлен, пункт меню будет иметь “горячую клавишу”;
- RDS_MENU_UNIQUECAPTION – если флаг установлен, РДС не будет создавать этот пункт меню, если в меню уже есть пункт с точно таким же названием.

Нас интересует только флаг RDS_MENU_SHORTCUT: если значение поля Key не нулевое, значит, клавиша для пункта меню определена, и этот флаг должен быть указан. В противном случае никакие флаги для пункта меню нам не нужны, и переменная options получает значение 0.

Напишем теперь функции записи и загрузки параметров блока. Чтобы не усложнять модель блока, будем, как и в примере на стр. 285, использовать двоичный формат и теговую запись. Функция записи параметров будет иметь следующий вид:

```
// Запись параметров блока
void TOpenSysWinData::SaveBin(void)
{ BYTE tag;      // Переменная для байта тега
  int len=Caption?strlen(Caption):0;  // Длина строки Caption

  tag=1;   // Тег 1 - название пункта меню
  rdsWriteBlockData(&tag,sizeof(tag));
  rdsWriteBlockData(&len,sizeof(len));     // Длина строки
  if(len)  // При ненулевой длине - запись самой строки
    rdsWriteBlockData(Caption,len);

  tag=2;   // Тег 2 - "горячая клавиша"
  rdsWriteBlockData(&tag,sizeof(tag));
  rdsWriteBlockData(&Key,sizeof(Key));
  rdsWriteBlockData(&KeyShifts,sizeof(KeyShifts));

  tag=0;;  // Тег 0 - конец данных
  rdsWriteBlockData(&tag,sizeof(tag));
}
//=========================================
```

Как и в упомянутом выше примере, в этой функции используются однобайтовые теги, указывающие на формат и тип следующих за ними данных. За тегом 1 следует четырехбайтовая целая длина строки (она записывается во вспомогательную переменную len в начале функции) и, если длина не нулевая, len байтов самой строки. За тегом 2 следуют два числа, содержащих код “горячей клавиши” и ее флаги. Тег 0 указывает на конец данных блока.

Функция загрузки считывает байт тега, а затем, в зависимости от его значения, загружает данные в те или иные поля класса:

```
// Загрузка параметров блока
int TOpenSysWinData::LoadBin(void)
{ BYTE tag;      // Переменная для байта тега
  int len;

  // Прежде всего, освобождаем память, которую занимало старое
  // название пункта меню - сейчас загрузится новое
  rdsFree(Caption);
  Caption=NULL;

  for(;;)
    { if(!rdsReadBlockData(&tag,sizeof(tag)))
        break;  // Тег не считан - данные неожиданно кончились
      switch(tag)
        { case 0:     // Конец данных
            RegisterMenuItem(); // Создаем или изменяем пункт меню
```

```
            return RDS_BFR_DONE;
          case 1:     // Название пункта
            rdsReadBlockData(&len,sizeof(len)); // Читаем длину
            if(len)   // Длина ненулевая - читаем len байтов строки
              { // Отводим место с учетом нуля в конце строки
                Caption=(char*)rdsAllocate(len+1);
                rdsReadBlockData(Caption,len);  // Читаем строку
                Caption[len]=0; // Дописываем нулевой байт в конец
              }
            break;
          case 2:     // Данные "горячей клавиши"
            rdsReadBlockData(&Key,sizeof(Key));
            rdsReadBlockData(&KeyShifts,sizeof(KeyShifts));
            break;
          default:    // Тег не опознан - ошибка
            return RDS_BFR_ERROR;
        }
    }
  // Вышли из цикла из-за неожиданного конца данных блока - ошибка
  return RDS_BFR_ERROR;
}
//=======================================
```

В этой функции следует обратить внимание на специфику работы со строкой `Caption`. В самом начале функции, перед тем, как начнут читаться какие-либо данные, текущее содержимое строки уничтожается, и полю класса `Caption` присваивается значение `NULL`. В процессе чтения данных блока должно загрузиться новое название пункта меню, поэтому старое нам больше не нужно. При считывании тега 1 сразу за ним в переменную `len` считывается длина строки названия, и, если она не нулевая, функцией `rdsAllocate` отводится память под `len+1` символов (еще один байт нужен для завершающего строку нуля). После этого символы строки считываются в отведенный буфер и в его конец записывается нулевой байт, которым должна оканчиваться каждая строка.

Второй важный момент в функции – действия при считывании нулевого тега, отмечающего конец данных блока. При этом вызывается функция `RegisterMenuItem`, уже написанная нами раньше, которая, на основе считанного названия и кода “горячей клавиши”, создает пункт системного меню или изменяет его параметры, если он уже был создан ранее. Если эту функцию не вызвать, то параметры будут загружены во внутренние поля класса, но никак не отразятся на внешнем виде пункта в системном меню.

Теперь напишем функцию настройки, которая позволит пользователю задать параметры блока, которые мы уже научились сохранять и загружать:

```
// Настройка параметров блока
int TOpenSysWinData::Setup(void)
{ RDS_HOBJECT window; // Идентификатор объекта-окна
  BOOL ok;            // Пользователь нажал "OK"
  // Создание окна
  window=rdsFORMCreate(FALSE,-1,-1,"Открытие окна");
  // Поле ввода для строки названия
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_EDIT,
                 "Текст пункта меню:",200);
  rdsSetObjectStr(window,1,RDS_FORMVAL_VALUE,Caption);
  // Поле ввода для "горячей клавиши"
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_HOTKEY,
                 "Клавиша:",150);
  rdsSetObjectInt(window,2,RDS_FORMVAL_VALUE,Key);
  rdsSetObjectInt(window,2,RDS_FORMVAL_HKSHIFTS,KeyShifts);
```

```
  // Открытие окна
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Нажата кнопка OK – запись параметров в блок
      rdsFree(Caption);    // Уничтожение старой строки
      // Создание динамической копии введенной строки
      Caption=rdsDynStrCopy(rdsGetObjectStr(window,1,
                            RDS_FORMVAL_VALUE));
      // Чтение параметров клавиши
      Key=rdsGetObjectInt(window,2,RDS_FORMVAL_VALUE);
      KeyShifts=rdsGetObjectInt(window,2,RDS_FORMVAL_HKSHIFTS);
      // Создание пункта меню на основе изменившихся параметров
      RegisterMenuItem();
    }
  // Уничтожение окна
  rdsDeleteObject(window);
  // Возвращаемое значение
  return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
}
//=======================================
```

В этой функции используется уже многократно описанный способ создания окна при помощи вспомогательного объекта РДС. Опять следует обратить внимание на присваивание нового значения заголовку меню: сначала старое содержимое `Caption` уничтожается, затем строка, введенная пользователем в окне, копируется из внутреннего буфера объекта (указатель на который возвращает функция `rdsGetObjectStr`) в динамическую память при помощи функции `rdsDynStrCopy` – эта сервисная функция предназначена для создания в динамической памяти копии переданной в ее параметре строки.

Следует также обратить внимание на то, что, после переписи всех измененных параметров из объекта РДС в поля класса, вызывается функция `RegisterMenuItem`, чтобы параметры передались в пункт меню, и он получил новое название и новую “горячую клавишу”.

Наконец, напишем функцию модели, которая будет создавать и удалять объект описанного нами класса и вызывать его функции:

```
// Блок, открывающий окно родительской подсистемы
extern "C" __declspec(dllexport)
  int RDSCALL OpenSysWin(int CallMode,
                   RDS_PBLOCKDATA BlockData,
                   LPVOID ExtParam)
{ // Указатель на личную область данных блока,приведенный к
  // правильному типу
  TOpenSysWinData *data=(TOpenSysWinData*)(BlockData->BlockData);
  switch(CallMode)
    { case RDS_BFM_INIT:         // Инициализация
        BlockData->BlockData=data=new TOpenSysWinData();
        break;
      case RDS_BFM_CLEANUP:      // Очистка
        delete data;
        break;
      case RDS_BFM_SETUP:        // Настройка
        return data->Setup();
      case RDS_BFM_SAVEBIN:      // Сохранение параметров
        data->SaveBin();
        break;
      case RDS_BFM_LOADBIN:      // Загрузка параметров
        return data->LoadBin();
```

```
        case RDS_BFM_MENUFUNCTION:// Выбор пункта меню
          rdsOpenSystemWindow(BlockData->Parent);
          break;
      }
    return RDS_BFR_DONE;
  }
  //=========================================
```

Поскольку практически все действия мы, как обычно, вынесли в функции-члены класса, сама функция модели получилась довольно простой. Тем более, что блок не работает со статическими переменными, поэтому в тексте функции нет ни макроопределений для них, ни проверки допустимости их типа. Единственное действие, для которого мы не стали писать отдельную функцию – это реакция на выбор пункта меню `RDS_BFM_MENUFUNCTION`. У нашего блока есть единственный пункт меню, поэтому никакой проверки не производится: при выборе пункта сразу вызывается функция `rdsOpenSystemWindow`, открывающая окно подсистемы (в нашем случае – родительской подсистемы этого блока, поскольку в ее параметрах в качестве идентификатора подсистемы передан `BlockData->Parent`).

Нужно отметить, что при выборе пунктов и системного, и контекстного меню вызывается одна и та же реакция `RDS_BFM_MENUFUNCTION`. В связи с этим пары чисел, связанные с пунктами контекстного меню, не должны совпадать с парами чисел пунктов системного, иначе модель не сможет отличить их.

Для проверки работы написанной модели создадим в разных подсистемах блоки с этой моделью и зададим в их настройках разные названия пунктов меню и разные “горячие клавиши”. Например, поместим один такой блок в корневую подсистему, зададим для него название пункта “Открыть главное” и сочетание клавиш Ctrl+F1. В подсистему Sys3 (создадим ее , если такой подсистемы в схеме еще нет) поместим другой блок, и зададим в его настройках название пункта “Открыть Sys3” и сочетание клавиш Ctrl+F2. Теперь, если открыть меню “Система” главного окна РДС (рис. 83), в нем можно будет увидеть пункт “Дополнительно” с двумя подпунктами (их порядок зависит от названий: добавленные пункты системного меню всегда упорядочиваются по алфавиту).

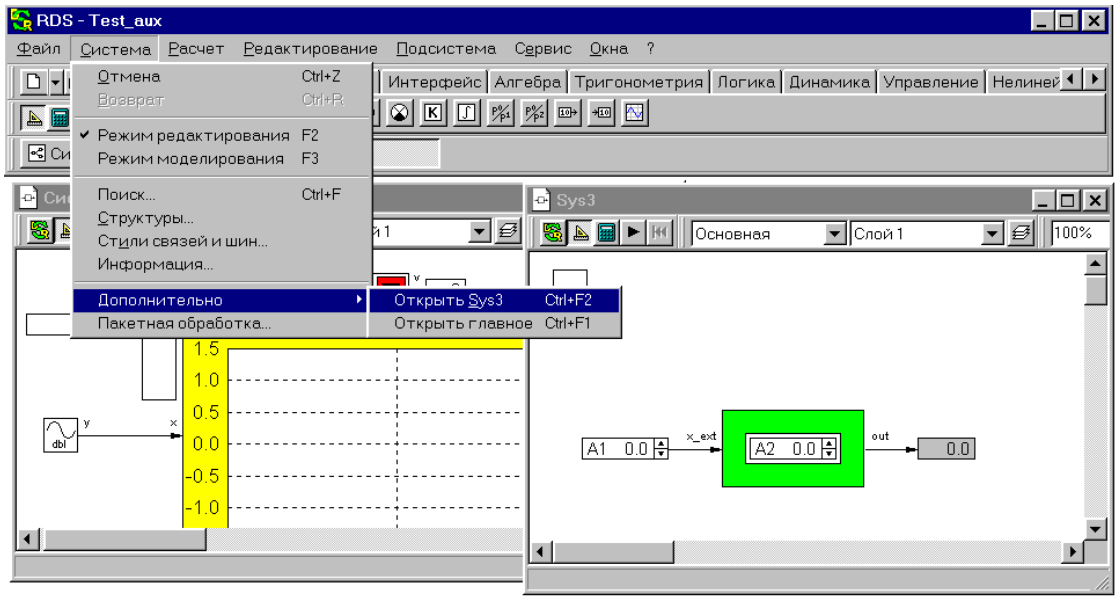


Рис. 83. Дополнительные пункты системного меню

Теперь, если пользователь выберет пункт “Открыть главное” или нажмет Ctrl+F1, окно корневой подсистемы откроется (если оно было закрыто) и переместится на передний план.

При выборе пункта “Открыть Sys3” или нажатии Ctrl+F2 на передний план переместится окно подсистемы Sys3. При этом не важно, в каком режиме находится РДС в данный момент, и какое окно находится на переднем плане.

§2.12.8. Реакция на действия пользователя при редактировании схемы

Рассматриваются различные реакции модели на действия пользователя, связанные с редактированием схемы: добавлением и удалением блоков, изменением их размера и т.п. В рассмотренные ранее примеры добавляются новые функции, влияющие на редактирование схемы пользователем.

Реакция модели блока на действия пользователя в режиме редактирования существенно отличается от реакции в режимах моделирования и расчета. В двух последних режимах пользователь управляет работой схемы – нажимает кнопки, перемещает рукоятки и т.п., и реакции моделей, предусмотренные в РДС, ориентированы именно на эти действия. В режиме редактирования же пользователь собирает схему: добавляет и удаляет блоки, перемещает их, изменяет их параметры и названия. Для того, чтобы облегчить пользователю редактирование схемы и, при необходимости, частично автоматизировать его работу, модель блока может реагировать на эти действия. Такие реакции также иногда применяются в тех случаях, когда РДС работает под управлением внешней программы через библиотеку RdsCtrl.dll – например, для того, чтобы ставить управляющую программу в известность о добавлении и удалении блоков схемы (внешнее управление РДС подробно описано в главе 3).

Мы уже рассматривали некоторые реакции на действия пользователя в режиме редактирования, в том числе и используемый практически в каждом более-менее сложном блоке вызов функции настройки `RDS_BFM_SETUP`. Реакции на вывод всплывающей подсказки `RDS_BFM_POPUPHINT`, открытие контекстного меню блока `RDS_BFM_CONTEXTPOPUP` и выбор пункта меню `RDS_BFM_MENUFUNCTION` вызываются в любом режиме РДС, поэтому их тоже можно отнести к реакциям в режиме редактирования, тем более, что они чаще всего используются именно для облегчения работы пользователя при сборке и настройке схемы. Также нами уже рассмотрена реакция на выбор блока мышью `RDS_BFM_MOUSESELECT` (стр. 281), позволяющая определить, относится ли конкретная точка внутри описывающего прямоугольника блока к изображению этого блока и может ли он быть выбран щелчком по этой точке. В РДС предусмотрено еще несколько реакций, которые мы здесь и рассмотрим.

В предыдущем параграфе мы создали блок, добавляющий в системное меню РДС пункт для открытия окна родительской подсистемы, причем название пункта и его “горячая клавиша” задаются в настройках блока. Если записать этот блок в библиотеку, пользователь сможет легко “вытаскивать” его оттуда и добавлять в важные подсистемы, которые он хотел бы быстро открывать нажатием какого-либо сочетания клавиш. Однако, в библиотеке блок будет записан с какими-то конкретными настройками, поэтому пользователю придется после каждой вставки блока из библиотеки открывать окно его настроек и менять название пункта меню и связанное с ним сочетание клавиш. Было бы логично автоматически открывать окно настроек блока при добавлении его в схему. Воспользуемся для этого специальной реакцией РДС `RDS_BFM_MANUALINSERT`, которая вызывается сразу после того, как пользователь загрузил в схему новый блок из файла, из библиотеки или вставил его из буфера обмена (заметим, что эта реакция не вызывается, если блок появляется в схеме при добавлении в нее подсистемы, внутри которой он содержится).

Добавим в оператор `switch` внутри функции модели `OpenSysWin` (стр. 305) новый оператор `case`:

```
// Добавление блока пользователем
case RDS_BFM_MANUALINSERT:
```

```
            if(data->Setup())  // Параметры изменены
              rdsSetModifiedFlag(TRUE);   // Взвести флаг изменений
            break;
```

Теперь, если блок будет добавлен в схему, модель вызовет функцию Setup класса личной области данных блока, которая открывает окно настройки и возвращает ненулевое значение, если параметры в нем были изменены. При этом вызовется сервисная функция РДС rdsSetModifiedFlag с параметром TRUE, которая взведет флаг наличия изменений в схеме. В реакции RDS_BFM_SETUP нам не приходилось взводить этот флаг вручную: при возврате функцией модели, вызванной для этой реакции, ненулевого значения РДС автоматически взводил этот флаг и предупреждал пользователя, если он хотел завершить работу со схемой, не сохранив изменения. В реакции RDS_BFM_MANUALINSERT это не предусмотрено, поэтому флаг приходится взводить при помощи сервисной функции.

Проверяя работу измененной модели блока, можно заметить в ней один недостаток. Если пользователь выделит несколько таких блоков, скопирует их в буфер обмена, а затем вставит его содержимое в какую-либо подсистему, окно настройки откроется столько раз, сколько блоков было в буфере обмена. Реакция RDS_BFM_MANUALINSERT будет вызвана для каждого вставляемого блока, и каждый раз функция Setup будет открывать окно настройки. Можно не обращать внимания на этот недостаток, поскольку в подсистеме не должно быть более одного такого блока – помещать в подсистему несколько блоков, открывающих ее окно, бессмысленно. Однако, можно и исправить его, проанализировав поля структуры RDS_MANUALINSERTDATA, указатель на которую передается в ExtParam при реакции на событие RDS_BFM_MANUALINSERT. Эта структура описана в “RdsDef.h” следующим образом:

```
typedef struct
{ int Reason;    // Причина добавления: загрузка из файла
                 // (RDS_LS_LOADFROMFILE) или из буфера
                 // обмена (RDS_LS_LOADCLIPBRD)
  BOOL Single;  // Добавляется единственный блок
} RDS_MANUALINSERTDATA;
typedef RDS_MANUALINSERTDATA *RDS_PMANUALINSERTDATA;
```

Нам нужно проверить значение поля Single и открывать окно настроек только в том случае, если оно истинно (TRUE). Если оно ложно, значит, в схему добавляется сразу несколько блоков, и окно открывать не нужно. В приведенный выше оператор case необходимо внести следующие изменения:

```
          // Добавление блока пользователем
          case RDS_BFM_MANUALINSERT:
            if( ((RDS_PMANUALINSERTDATA)ExtParam)->Single )
              { if(data->Setup())    // Параметры изменены
                  rdsSetModifiedFlag(TRUE);// Взвести флаг изменений
              }
            break;
```

Теперь вызов функции настройки заключен внутрь оператора if, который проверяет значение поля Single переданной структуры, предварительно приведя ExtParam к правильному типу.

Кроме добавления блоков в схему иногда имеет смысл реагировать на их удаление, например, чтобы предупредить пользователя об изменениях работы схемы, которые удаление этого блока может повлечь. Для этого используется реакция RDS_BFM_MANUALDELETE – РДС вызывает модель блока в этом режиме непосредственно перед удалением блока из схемы по команде пользователя (при удалении подсистемы в этом режиме также вызываются модели всех ее внутренних блоков). Следует помнить, что модель, реагируя на этот вызов, не может отменить удаление блока – оно будет выполнено,

независимо от того, что будет делать функция модели в этой реакции и какое значение она вернет.

Не следует путать реакции RDS_BFM_MANUALDELETE и RDS_BFM_CLEANUP. Первая вызывается только при удалении блока пользователем вручную, вторая же – при любом отключении модели от блока. В реакции RDS_BFM_CLEANUP невозможно определить, по какой причине отключается модель: из-за удаления блока, из-за подключения к нему другой функции модели, при закрытии РДС и т.д.

Рассмотрим пример, в котором будем реагировать на удаление блока. Ранее (стр. 104) мы создали пару блоков, один из которых перемещается в окне подсистемы, а второй задает скорость и направление его движения через пару динамических переменных. Если удалить из подсистемы блок-задатчик, подвижный блок прекратит работу, поскольку не сможет получать значения скорости и направления. Будем предупреждать об этом пользователя, выводя ему текстовое сообщение.

Прежде, чем предупреждать пользователя о последствиях удаления этого блока, необходимо проверить, подписан ли какой-либо другой блок на созданные им динамические переменные. Если переменными никто не пользуется, блок можно удалять без предупреждения – на другие блоки это не повлияет. Мы будем подсчитывать число подписчиков на каждую динамическую переменных блока. Один подписчик всегда будет обнаружен – это сам создавший переменные блок. Если число подписчиков окажется большим или равным двум, значит, на переменные подписан кто-то еще, и об этом необходимо предупредить пользователя.

В РДС нет сервисной функции, определяющей число подписчиков на динамическую переменную, однако, есть функция rdsEnumDynVarSubscribers, перечисляющая всех подписчиков (то есть вызывающая для каждого блока-подписчика указанную функцию обратного вызова). Ей мы и воспользуемся. Сначала напишем функцию обратного вызова – она должна иметь следующий формат:

```
BOOL RDSCALL имя_функции(
      RDS_BHANDLE block,          // Блок-подписчик
      RDS_PDYNVARLINK link,       // Связь с переменной
      LPVOID ptr);                // Дополнительный параметр
```

В первом параметре функции передается идентификатор обнаруженного блока-подписчика, во втором – указатель на структуру подписки на динамическую переменную, подписчики которой перечисляются, в третьем – дополнительный указатель произвольного типа, который передается в функцию rdsEnumDynVarSubscribers при ее вызове и переносится в функцию обратного вызова без изменений. Мы в этом третьем параметре будем передавать указатель на целое число, которое функция обратного вызова будет увеличивать на единицу для каждого подписчика – таким образом, мы сможем их сосчитать. Возвращаемое функцией обратного вызова значение указывает на необходимость остановить перебор подписчиков (FALSE) или продолжить его (TRUE). Нам не нужно останавливать перебор, поэтому мы всегда будем возвращать TRUE.

Функция обратного вызова будет иметь следующий вид:

```
// Функция обратного вызова для подсчета подписчиков
// динамической переменной
BOOL RDSCALL DynVarUsersCountEnum(RDS_BHANDLE /*block*/,
           RDS_PDYNVARLINK /*link*/,LPVOID ptr)
{ // Приводим ptr к типу "указатель на целое"
  int *pCount=(int*)ptr;
  (*pCount)++;  // Увеличиваем счетчик
  return TRUE;
}
//=======================================
```

Теперь напишем функцию, которая, пользуясь DynVarUsersCountEnum, вернет число подписчиков на заданную переменную. Она будет совсем простой:

```
// Подсчет числа подписчиков на динамическую переменную
int CountDynVarUsers(RDS_PDYNVARLINK link)
{ int count=0;
  // Перебрать всех подписчиков переменной link
  rdsEnumDynVarSubscribers(link,DynVarUsersCountEnum,&count);
  return count;
}
//=======================================
```

Используемая нами сервисная функция rdsEnumDynVarSubscribers принимает три параметра. Первый – указатель на структуру подписки на переменную, подписчиков которой нужно перечислить. Второй – функция обратного вызова (точнее, указатель на нее), которую нужно вызвать для каждого подписчика. Третий – указатель произвольного типа, который без изменений передается в третий параметр функции обратного вызова. В нашем случае это указатель на целую переменную – счетчик подписчиков. После перебора всех подписчиков в переменной count будет находиться их число, которое функция и возвращает.

Теперь внесем изменения в модель. В оператор switch функции модели задатчика TestBlkMoveSetter необходимо добавить новый оператор case:

```
// Удаление блока пользователем
case RDS_BFM_MANUALDELETE:
  // Приведение указателя на личную область данных к
  // правильному типу
  privdata=(TestBlkMoveSetterData*)(BlockData->BlockData);
  // Если блок удаляется в составе подсистемы,
  // предупреждать не нужно
  if( ((RDS_PMANUALDELETEDATA)ExtParam)->WithSys)
    break;
  // Считаем число подписчиков на переменные этого блока
  if(CountDynVarUsers(privdata->VLink)>1 ||
     CountDynVarUsers(privdata->ALink)>1)
    rdsMessageBox("Внимание! Этот блок предоставляет "
                  "данные другим блокам. Его удаление "
                  "приведет к их отключению.",
                  BlockData->BlockName,
                  MB_OK | MB_ICONWARNING);
  break;
```

Прежде всего, мы проверяем, что удаляется: сам блок-задатчик, или вся подсистема, в которой он содержится. В последнем случае вместе с ним будут удалены и все блоки, которые могли пользоваться его динамическими переменными (они создаются задатчиком в родительской подсистеме), поэтому никаких предупреждений выводить не нужно. Для этой проверки анализируется структура RDS_MANUALDELETEDATA, указатель на которую передается в ExtParam в режиме RDS_BFM_MANUALDELETE. Эта структура описана в “RdsDef.h” следующим образом:

```
typedef struct
{ BOOL Single;  // Удаляется один блок (TRUE)
                // или несколько (FALSE)
  BOOL WithSys; // Удаляется внутри подсистемы (TRUE)
                // или самостоятельно, то есть один или в группе
                // выделенных блоков (FALSE)
} RDS_MANUALDELETEDATA;
typedef RDS_MANUALDELETEDATA *RDS_PMANUALDELETEDATA;
```

Нас в этой структуре интересует поле WithSys, значение TRUE которого указывает на удаление блока в составе подсистемы. Если это так, реакция завершается оператором break.

В противном случае подсчитывается число подписчиков на две созданные блоком динамические переменные, и, если хотя бы одно из этих чисел больше единицы (то есть на переменную подписан кто-то кроме создавшего ее блока), выводится сообщение пользователю.

Еще одно событие, реакцию на которое достаточно часто включают в модели блоков, это изменение размеров блока. На самом деле, это не одна, а две реакции. Одна из них – RDS_BFM_RESIZE – вызывается после изменения размеров блока, другая – RDS_BFM_RESIZING – при изменении размеров блока мышью в процессе перетаскивания пользователем одного из восьми прямоугольных маркеров выделения. Реакцию RDS_BFM_RESIZE мы уже использовали при создании блока-графика (стр. 190) для того, чтобы заново вычислить координаты рабочего поля графика при изменении размера всего блока. Эта реакция вызывается один раз после изменения размеров блока, независимо от того, каким способом это изменение размеров было выполнено: перетаскиванием маркеров выделения, вводом точных значений ширины и высоты в окне параметров блока или значения масштабного коэффициента для блоков с векторной картинкой. Реакция RDS_BFM_RESIZING вызывается только при перетаскивании маркеров выделения блока, то есть при изменении размеров блока мышью, причем вызывается она каждый раз при перемещении курсора. В обеих реакциях модель может скорректировать задаваемый пользователем размер, при этом в RDS_BFM_RESIZING пользователь будет сразу же видеть это изменение, поскольку оно отразится на размере перетаскиваемого мышью прямоугольника.

Изменим модель блока, имитирующего двухкоординатную рукоятку (стр. 268) так, чтобы блок всегда оставался квадратным. При этом масштаб блока по горизонтали и вертикали будет одинаковым, то есть горизонтальное и вертикальное перемещение рукоятки на одну и ту же величину будет приводить к одинаковому изменению выходных переменных.

Прежде всего добавим в класс личной области данных блока новую функцию для реакции на изменение размеров блока:

```
void Resizing(RDS_PRESIZEDATA ResData);
```

Эта функция будет принимать один параметр – указатель на структуру RDS_RESIZEDATA, который передается в функцию модели в параметре ExtParam и в режиме RDS_BFM_RESIZE, и в режиме RDS_BFM_RESIZING. Структура описана в “RdsDef.h” следующим образом:

```
typedef struct
{ BOOL HorzResize;    // Изменение размера по горизонтали
  BOOL VertResize;    // Изменение размера по вертикали
  int newWidth,newHeight;  // Новые значения ширины и высоты
  // Данные о сетке редактора
  int GridDx,GridDy;  // Шаг сетки
  BOOL SnapToGrid;    // Включена привязка к сетке
} RDS_RESIZEDATA;
typedef RDS_RESIZEDATA *RDS_PRESIZEDATA;
```

Логические поля структуры HorzResize и VertResize указывают на то, каким именно образом производится изменение размеров блока:

HorzResize	*VertResize*	*Способ изменения*
TRUE	TRUE	Пользователь перетаскивает один из угловых маркеров выделения
TRUE	FALSE	Пользователь перетаскивает левый или правый маркер выделения
FALSE	TRUE	Пользователь перетаскивает верхний или нижний маркер

HorzResize	*VertResize*	*Способ изменения*
		выделения
FALSE	FALSE	Размер изменен пользователем через окно параметров блока

В полях newWidth и newHeight содержатся новые значения ширины и высоты блока. Если необходимо скорректировать размер блока, функция модели может перед возвратом изменить эти значения. В нашем случае, чтобы блок всегда оставался квадратным, его ширина и высота должны быть равны. Нужно только определить, как правильно изменить значения ширины и высоты. Если пользователь перетаскивает левый или правый маркер выделения, он, тем самым, задает ширину блока – в этом случае нужно сделать высоту блока, равной ширине. Если он перетаскивает верхний или нижний маркер, он задает высоту, при этом нужно сделать ширину равной высоте. Если же он перетаскивает один из угловых маркеров, задавая одновременно оба размера блока, или размеры введены в окне параметров, проще всего присвоить ширине и высоте среднее арифметическое заданных пользователем размеров, при этом поведение блока будет выглядеть более-менее естественным.

Реализуем описанный выше алгоритм в функции:

```
// Реакция на изменение размеров блока
void TSimpleJoystick::Resizing(RDS_PRESIZEDATA ResData)
{
  if(ResData->HorzResize && (!ResData->VertResize))
    // Перетаскивается левый или правый маркер
    ResData->newHeight=ResData->newWidth;
  else if((!ResData->HorzResize) && ResData->VertResize)
    // Перетаскивается верхний или нижний маркер
    ResData->newWidth=ResData->newHeight;
  else // Перетаскивается угловой маркер или размер задан точно
    { // Вычисляем среднее арифметическое
      int avg=(ResData->newWidth+ResData->newHeight)/2;
      // Присваиваем его ширине и высоте
      ResData->newWidth=ResData->newHeight=avg;
    }
}
//=======================================
```

Теперь осталось добавить вызов этой функции в функцию модели блока SimpleJoystick. Внутрь оператора switch нужно вставить следующий case:

```
        // Изменение размеров блока
        case RDS_BFM_RESIZE:
        case RDS_BFM_RESIZING:
          data->Resizing((RDS_PRESIZEDATA)ExtParam);
          break;
```

Здесь мы вызываем функцию Resizing в реакциях на оба события, связанных с изменением размера блока. Дело в том, что если вызывать ее только в реакции на RDS_BFM_RESIZE, блок будет всегда оставаться квадратным, но у пользователя при перетаскивании маркеров выделения не будет визуальной обратной связи. Если же вызывать ее только в RDS_BFM_RESIZING, визуальная обратная связь будет, но при вводе размеров блока в окне параметров они не будут скорректированы моделью, поскольку RDS_BFM_RESIZING при этом не вызывается. Таким образом, корректировать заданные пользователем размеры необходимо в обеих реакциях.

Теперь легальными способами пользователь никак не сможет сделать блок-рукоятку не квадратным. На самом деле, эту функцию следовало бы сделать отключаемой, но мы не стали делать этого, чтобы не усложнять пример.

В РДС предусмотрены и другие реакции моделей блоков на действия пользователя, связанные с редактированием схемы. Они применяются значительно реже, и мы не будем

приводить примеры их использования. Просто перечислим их с кратким указанием возможного применения:

- Реакция на перемещение блока `RDS_BFM_MOVED`. Может применяться, например, для синхронного перемещения группы блоков при перемещении одного из них, или для сообщения новых координат блока управляющей программе, если такая есть.
- Реакция на переименование блока `RDS_BFM_RENAME`. Может применяться для синхронного с блоком переименования каких-либо связанных с блоком объектов (например, динамических переменных), имена которых формируются из имени блока.
- Реакция на загрузку схемы `RDS_BFM_AFTERLOAD`, вызываемая у всех блоков схемы сразу после ее загрузки в память. Может применяться для инициализации общесистемных переменных и объектов, которую, по каким-либо причинам, неудобно проводить при подключении модели к блоку.
- Реакция на завершение работы со схемой `RDS_BFM_UNLOADSYSTEM`, то есть на ее выгрузку из памяти. Может применяться, например, для удаления каких-либо временных файлов, созданных для работы с этой схемой.
- Пара реакций на сохранение всей схемы: одна из них (`RDS_BFM_BEFORESAVE`) вызывается до сохранения, другая (`RDS_BFM_AFTERSAVE`) – после. Могут применяться, например, для преобразования файла схемы в какой-либо другой формат.

Структуры данных, используемых в этих реакциях, подробно описаны в приложении А.

§2.13. Вызов функций блоков

Рассматриваются различные способы вызова функций блоков, то есть вызова модели какого-либо блока из модели другого блока. Приводятся примеры использования этого механизма для организации сложного взаимодействия между блоками схемы.

§2.13.1. Общие принципы вызова функций блоков

Рассматриваются общие принципы вызова функций блоков. Описывается регистрация функций блоков в РДС, необходимая для их работы, простейший способ вызова функций, а также включение в модель блока реакции на вызов такой функции.

Рассматривавшиеся ранее способы взаимодействия блоков между собой – передача данных по связям и через динамические переменные – имеют один существенный недостаток: они позволяют передавать от блока к блоку только данные стандартных типов, предусмотренных в РДС. Таким образом нельзя передать, например, дескриптор файла, или указатель на какие-либо данные в памяти. Кроме того, передача данных по связям требует активного участия пользователя (он должен провести эти связи), а обмен через динамические переменные будет слишком сложен при большом количестве блоков. Если блоки обмениваются данными независимо, то для каждой пары связанных таким образом блоков необходима своя, постоянно существующая динамическая переменная, которой нужно дать какое-то уникальное имя, причем такое, чтобы оба связанных блока знали его. При этом нет никакой гарантии, что в обмен данными не вмешается третий блок, тоже подписавшийся на эту переменную.

Чтобы упростить (а в некоторых случаях и ускорить) передачу произвольных данных от блока к блоку, в РДС введен механизм вызова функций блоков, то есть вызова модели одного блока из модели другого. Этот механизм имеет несколько разновидностей: может вызываться модель одного блока или всех блоков подсистемы, модель может вызываться немедленно (и возвращенное ей значение будет передано вызвавшему блоку) или после завершения модели вызвавшего блока, модель может просто поддерживать выполнение какой-либо функции, или объявить свой блок ее исполнителем, чтобы другие заинтересованные блоки схемы знали это – в любом случае, механизм вызова реализуется

через сервисные функции РДС и требует соблюдения некоторых правил. Рассмотрим их подробнее.

Каждая функция блока, доступная для вызова, должна иметь имя. Имя функции блока в данном случае – это не имя функции его модели, это просто некоторая строка, однозначно определяющая для всех моделей вызываемых и вызывающих блоков во всех библиотеках эту функцию и тип данных, который передается при ее вызове. Эту строку придумывает разработчик, который решил добавить в создаваемую им модель поддержку какой-либо новой функции, вызываемой другими блоками. Он обычно включает имя функции и описание передаваемых при ее вызове данных в документацию к блоку, чтобы разработчики других моделей могли ее вызывать или добавить ее поддержку в свои блоки. Нет никаких специальных правил, которым должна соответствовать эта строка – она просто должна быть уникальной. По этой причине имя функции обычно делают достаточно длинным и, для ясности, отражающим смысл выполняемых ей действий. Кроме того, обычно в нем не используют символы национальных алфавитов, чтобы избежать возможных проблем с кодировкой – для того, чтобы разные программисты могли правильно прочесть имя функции и вставить его в свои программы, лучше всего использовать в нем только латинские символы, цифры и знаки препинания. Часто в имя функции включают имя библиотеки, в блоке которой впервые появилась поддержка этой функции, имя разработчика и т.п. – все, что угодно, чтобы точно такое же имя не пришло в голову разработчику какой-либо другой модели, решившему придумать свою функцию. Имена вроде “функция1”, “Установка значения” или “SetValue” использовать не стоит. А вот строка “InstituteOfControlSciences.Lab49.GeneralGraphics.SetValue”, например, достаточно длинная и сложная, чтобы ее можно было относительно спокойно использовать в качестве имени новой функции.

Прежде чем модель сможет вызвать функцию другого блока или сама среагировать на такой вызов, функция должна быть зарегистрирована в РДС. Для этого необходимо вызвать сервисную функцию `rdsRegisterFunction`, передав в ее параметре строку с именем регистрируемой функции. Она вернет уникальный целый идентификатор, присвоенный этой функции. Вся дальнейшая работа с функцией будет вестись не по имени, а по этому целому идентификатору. Функцию `rdsRegisterFunction` для какого-либо конкретного имени функции блока можно вызывать сколько угодно раз, для одного и того же имени она будет возвращать один и тот же идентификатор, до тех пор, пока вся схема не будет выгружена из памяти при завершении работы с РДС или при загрузке другой схемы. Важно помнить, что идентификатор, присвоенный функции, действует только до тех пор, пока схема находится в памяти: при следующей загрузке той же самой схемы и регистрации той же самой функции РДС может присвоить ей другой целый идентификатор.

Регистрация имени функции никак не привязана к какому-либо блоку, поэтому ее можно производить не только в функции модели, но и, например, в главной функции библиотеки (см. §2.2) в момент загрузки DLL. Обычно для каждой функции, используемой в моделях блоков, находящихся в данной DLL, описывают глобальную целую переменную, которой присваивают результат вызова `rdsRegisterFunction`. При этом, поскольку переменная глобальная, с момента регистрации все модели блоков в библиотеке получают доступ к целому идентификатору функции. Чтобы не вызывать `rdsRegisterFunction` лишний раз (это безопасно, но вызывает лишние задержки из-за того, что РДС будет искать переданную строку в таблице уже зарегистрированных функций), можно пользоваться тем фактом, что РДС никогда не присваивает функциям нулевой идентификатор. Таким образом, дав глобальной переменной нулевое начальное значение, можно регистрировать функцию только в том случае, если значение переменной на данный момент осталось нулевым, то есть функция еще не зарегистрирована.

Как регистрировать функции блоков – все сразу в момент загрузки DLL, или по необходимости при инициализации моделей, в которых они нужны – решать программисту. Оба способа не имеют каких-либо явных преимуществ друг перед другом.

Допустим, в какой-либо библиотеке находятся модели `Model1` и `Model2`. В модели `Model1` используется функция “ProgrammersManual.Function1”, а в модели `Model2` – эта же функция, и, кроме нее, еще и “ProgrammersManual.Function2”. Не важно, что это за модели, и что делают указанные функции – они нужны нам просто для примера. Если программист решит регистрировать функции при инициализации моделей блоков, ему нужно будет написать примерно следующее (фрагменты, относящиеся к регистрации функций, выделены жирным):

```
#include <windows.h>
#include <RdsDef.h>
// Подготовка описаний сервисных функций
#define RDS_SERV_FUNC_BODY GetInterfaceFunctions
#include <RdsFunc.h>

// Глобальные переменные для хранения идентификаторов функций
int Function1Id=0,Function2Id=0;

//========== Главная функция DLL ==========
int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                         unsigned long reason,
                         void* /*lpReserved*/)
{ if(reason==DLL_PROCESS_ATTACH) // Загрузка DLL
    { // Получение доступа к функциям
      if(!GetInterfaceFunctions())
        MessageBox(NULL,”Нет доступа к функциям”,”Ошибка”,MB_OK);
    }
  return 1;
}
//========= Конец главной функции =========

// Модель 1
extern "C" __declspec(dllexport)
  int RDSCALL Model1(int CallMode,
                     RDS_PBLOCKDATA BlockData,
                     LPVOID ExtParam)
{ switch(CallMode)
    { case RDS_BFM_INIT:   // Инициализация
        // Регистрация функции
        if(!Function1Id)
          Function1Id=rdsRegisterFunction(
                        "ProgrammersManual.Function1");
  // ...
}
//========================================

// Модель 2
extern "C" __declspec(dllexport)
  int RDSCALL Model2(int CallMode,
                     RDS_PBLOCKDATA BlockData,
                     LPVOID ExtParam)
{ switch(CallMode)
    { case RDS_BFM_INIT:   // Инициализация
        // Регистрация функций
        if(!Function1Id)
```

```
        Function1Id=rdsRegisterFunction(
                              "ProgrammersManual.Function1");
      if(!Function2Id)
        Function2Id=rdsRegisterFunction(
                              "ProgrammersManual.Function2");
  // ...
}
//=======================================
```

В приведенном примере видно, что при таком способе каждая модель регистрирует только те функции, которые нужны именно ей, причем только в том случае, если значение глобальной переменной нулевое, то есть эти функции не были зарегистрированы ранее другой функцией модели или этой же, но присоединенной к другому блоку.

Если программист решит регистрировать функции в момент загрузки DLL, пример будет выглядеть так:

```
#include <windows.h>
#include <RdsDef.h>
// Подготовка описаний сервисных функций
#define RDS_SERV_FUNC_BODY GetInterfaceFunctions
#include <RdsFunc.h>

// Глобальные переменные для хранения идентификаторов функций
int Function1Id,Function2Id;

//========== Главная функция DLL ==========
int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                         unsigned long reason,
                         void* /*lpReserved*/)
{ if(reason==DLL_PROCESS_ATTACH) // Загрузка DLL
    { // Получение доступа к функциям
      if(!GetInterfaceFunctions())
        MessageBox(NULL,"Нет доступа к функциям","Ошибка",MB_OK);
      else
        { // Регистрация функций блоков
          Function1Id=rdsRegisterFunction(
                              "ProgrammersManual.Function1");
          Function2Id=rdsRegisterFunction(
                              "ProgrammersManual.Function2");
        }
    }
  return 1;
}
//========= Конец главной функции =========

// В функциях моделей никакие действия по регистрации функций
// не производятся
```

В этом примере регистрация функций производится один раз за все время существования DLL в памяти процесса, поэтому глобальным переменным не нужны нулевые значения по умолчанию и при регистрации не нужны какие-либо проверки.

После того, как функция зарегистрирована, ее целый идентификатор можно использовать для вызова функций и для реакций на них в моделях. Если необходимо вызвать функцию какого-либо конкретного блока, необходимо знать его уникальный идентификатор (RDS_BHANDLE). Этот идентификатор модель может получить разными способами: перебрав все блоки подсистемы и найдя в ней блок, удовлетворяющий какому-либо критерию; обнаружив блок на другом конце связи, соединенной с данным блоком, при ее анализе, и т.п. В любом случае, модель, вызывающая функцию другого блока, должна знать,

к какому именно блоку схемы она обращается. Для прямого, то есть немедленного, вызова функции конкретного блока (отложенный вызов мы рассмотрим позже) используется сервисная функция rdsCallBlockFunction, принимающая три параметра: идентификатор вызываемого блока, целый идентификатор вызываемой в этом блоке функции и указатель произвольного типа (void*) на область памяти, в которой содержатся параметры этой функции (это может быть массив, структура, или просто набор байтов – РДС передает этот указатель в модель вызываемого блока без какой-либо обработки). Результат возврата rdsCallBlockFunction – целое число, которое вернула функция модели вызванного блока.

Допустим, параметром приведенной в примере выше функции “ProgrammersManual.Function1” является одно вещественное число двойной точности (double). Тогда вызов этой функции (с учетом того, что ее идентификатор уже записан в глобальную переменную Function1Id) будет выглядеть следующим образом:

```
RDS_BHANDLE block=... // Здесь должен быть идентификатор
                      // вызываемого блока
double val=10.0;      // Это число будет параметром функции
int result;           // Сюда запишем возвращенное значение
// Вызов функции
result=rdsCallBlockFunction(block,Function1Id,&val);
```

В момент вызова rdsCallBlockFunction модель блока block вызывается в режиме RDS_BFM_FUNCTIONCALL, при этом в параметре ExtParam передается указатель на структуру RDS_FUNCTIONCALLDATA, в которой содержится идентификатор вызванной функции, указатель на переданные параметры, идентификатор вызвавшего блока, а также другая информация, связанная с вызовом функции:

```
typedef struct
{ int Function;       // Идентификатор функции
  LPVOID Data;        // Указатель на параметры функции
  int Reserved;       // Зарезервировано
  RDS_BHANDLE Caller; // Вызвавший блок
  BOOL Broadcast;     // TRUE - вызов функции сразу у нескольких
                      // блоков, FALSE - только у одного
  int BroadcastCnt;   // При Broadcast==TRUE - порядковый номер
                      // вызванного блока, начиная с 0
  BOOL Stop;          // При Broadcast==TRUE и соответствующих
                      // параметрах вызова - остановить перебор
                      // блоков
  BOOL Delayed;       // TRUE - отложенный вызов,
                      // FALSE - прямой (немедленный)
  DWORD DataBufSize;  // При отложенном вызове - размер параметров
} RDS_FUNCTIONCALLDATA;
typedef RDS_FUNCTIONCALLDATA *RDS_PFUNCTIONCALLDATA;
```

Модель, вызванная в этом режиме, должна сравнить поле Function переданной структуры с идентификаторами поддерживаемых ей функций, и, при совпадении с одним из них, выполнить заданные действия. Если полученный идентификатор не совпал ни с одним из поддерживаемых, модель должна завершиться, чтобы вызов не поддерживаемой блоком функции был безопасен и не приводил к каким-либо неожиданным последствиям.

Реакция на вызов функции “ProgrammersManual.Function1” будет выглядеть примерно следующим образом:

```
// Модель 1
extern "C" __declspec(dllexport)
  int RDSCALL Model1(int CallMode,
                     RDS_PBLOCKDATA BlockData,
                     LPVOID ExtParam)
{ RDS_PFUNCTIONCALLDATA func; // Вспомогательная переменная
```

```
  switch(CallMode)
    {
      // ...
      case RDS_BFM_FUNCTIONCALL:      // Вызов функции
        func=(RDS_PFUNCTIONCALLDATA)ExtParam;
        if(func->Function==Function1Id)
          { // Приведение указателя на параметры функции
            // к указателю на double
            double *pVal=(double*)(func->Data);
            // ... можно выполнять какие-либо действия
            // со значением *pVal ...
          }
        break;
      // ...
    }
  return RDS_BFR_DONE;
}
//=======================================
```

Очевидно, что для сравнения идентификатора вызванной функции с идентификаторами поддерживаемых нельзя использовать оператор `switch`, вместо него приходится использовать конструкции “`if`(...) `else if`(...) ...” Идентификаторы функций – не константы, они каждый раз заново генерируются РДС при регистрации и, в данном случае, хранятся в глобальных переменных `Function1Id` и `Function2Id`, поэтому они не могут использоваться в метках `case`, где допустимы только выражения с константами.

Приведенный пример имеет один существенный недостаток. Мы договорились, что параметром функции “ProgrammersManual.Function1” будет одно число `double`. Но что произойдет, если из-за ошибки программиста при вызове этой функции вместо указателя на вещественное число двойной точности будет передан указатель, например, на тридцатидвухбитное целое? РДС никак не обрабатывает переданный при вызове указатель на параметры, он просто копирует его в поле `Data` структуры `RDS_FUNCTIONCALLDATA`. Если вызванный блок попытается считать 8 байтов (размер числа `double`) по этому адресу, это вызовет ошибку общей защиты, поскольку размер целого – всего 4 байта. Чтобы избежать этого, имеет смысл включить размер параметров функции в сами параметры. Для этого проще всего оформлять параметры функции как структуру, в самом первом поле которой размещать ее собственный размер. Большинство сервисных функций РДС, в параметрах которых передаются указатели на структуры, проверяют правильность параметров именно так.

Если мы хотим передавать одно вещественное число двойной точности, структура параметров функции может выглядеть следующим образом:

```
typedef struct
{ DWORD servSize;     // Сюда будет записан размер структуры
  double Val;         // Собственно параметр функции
} TFunction1Params;
```

Вызывать эту функцию нужно будет так:

```
RDS_BHANDLE block=... // Здесь должен быть идентификатор
                      // вызываемого блока
TFunction1Params params;   // Структура параметров функции
params.servSize=sizeof(params); // Присваиваем размер структуры
params.Val=10.0;      // Это число будет параметром функции
int result;           // Сюда запишем возвращенное значение
// Вызов функции
result=rdsCallBlockFunction(block,Function1Id,&params);
```

Полю servSize введенной нами структуры параметров функции присваивается размер этой структуры, полученный оператором sizeof. Теперь в реакцию на вызов этой функции можно вставить проверку этого размера:

```
case RDS_BFM_FUNCTIONCALL:      // Вызов функции
  func=(RDS_PFUNCTIONCALLDATA)ExtParam;
  if(func->Function==Function1Id)
    { // Приведение указателя на параметры функции
      // к указателю на структуру
      TFunction1Params *par=(TFunction1Params*)(func->Data);
      // Проверка размера параметров
      if(par->servSize>=sizeof(TFunction1Params))
        { // ... можно выполнять какие-либо действия
          // со значением par->Val ...
        }
    }
  break;
```

Можно заметить, что проверка размеров построена так, чтобы размер переданной при вызове структуры параметров par->servSize был не меньше ее ожидаемого размера sizeof(TFunction1Params). Почему бы не проверять их точное совпадение оператором “==”? Представим себе, что разработанная нами функция оказалась весьма полезной, и мы (а, может быть, и другие разработчики тоже) используем ее в большом количестве моделей. Затем мы решили добавить в параметры этой функции новые поля, которые будут использоваться в нескольких новых моделях, но никак не влияют на уже отлаженные старые. Если бы в проверке размеров стоял оператор “==”, все ранее написанные и скомпилированные модели, которые не пользуются этими дополнительными полями, перестали бы работать, поскольку размер структуры изменился, и точное равенство перестало выполняться. Нам пришлось бы компилировать заново все модели, использующие эту функцию, а также заставить сделать это всех остальных разработчиков. Использование оператора “>=” позволяет не делать этого, поскольку он проверяет только *достаточность* размеров переданной структуры: если размер структуры окажется больше ожидаемого, мы можем спокойно продолжать работу, поскольку нужные нам поля в ней присутствуют. Разумеется, дополнительные поля в этом случае можно добавлять только в конец структуры, иначе старые модели, работающие с ее началом, считают неправильные данные. По этой же причине служебное поле, содержащее размер структуры, нужно делать самым первым.

Конечно, такая проверка не дает полной гарантии правильности переданных параметров: у двух структур, совершенно разных по составу полей, может оказаться одинаковый размер. Кроме того, если среди полей структуры есть указатели, типы данных, на которые они ссылаются, нужно проверять отдельно (например, включив в них точно такое же служебное поле размера). Тем не менее, такая проверка существенно повышает надежность работы модели и защищает от многих ошибок программиста.

РДС позволяет вызвать не только функцию конкретного блока, но и, например, функции всех блоков схемы, или всех блоков конкретной подсистемы. Для этого используется сервисная функция rdsBroadcastFunctionCallsEx:

```
int RDSCALL rdsBroadcastFunctionCallsEx(
      RDS_BHANDLE System,   // Подсистема, блоки которой вызываются
      int FuncId,           // Идентификатор вызываемой функции
      LPVOID Parameters,    // Указатель на параметры
      DWORD Flags);         // Флаги вызова
```

В первом параметре указывается подсистема, у всех блоков которой будет вызвана указанная функция. Второй и третий параметры (идентификатор функции и указатель на ее параметры) аналогичны параметрам рассмотренной выше rdsCallBlockFunction. Четвертый

параметр содержит битовые флаги, управляющие перебором блоков подсистемы при вызове их функций:

- `RDS_BCALL_SUBSYSTEMS` – если флаг взведен, будут вызываться не только блоки указанной подсистемы, но и все блоки подсистем, вложенных в нее;
- `RDS_BCALL_ALLOWSTOP` – если флаг взведен, очередной вызванный блок может прекратить дальнейший вызов других блоков, присвоив полю `Stop` структуры `RDS_FUNCTIONCALLDATA` значение `TRUE`;
- `RDS_BCALL_CHECKSUPPORT` – если флаг взведен, перед вызовом функции модель каждого блока будет вызываться в режиме `RDS_BFM_CHECKFUNCSUPPORT`, и, если при этом модель вернет нулевое значение, функция этого блока вызываться не будет. Это устаревший флаг – режим `RDS_BFM_CHECKFUNCSUPPORT` теперь практически не применяется, однако он до сих пор поддерживается и может использоваться по желанию программиста.

Вызов одной функции у всех блоков позволяет, например, уведомить всех “соседей” о наступлении какого-либо события, или подготовить блоки к началу какой-либо операции. Можно также использовать вызов функции всех блоков для поиска блока, удовлетворяющего какому-либо критерию, или для составления списка таких блоков. Реагируя на вызов функции и считав ее параметры, модель блока может передать вызвавшему блоку свой идентификатор для занесения в список (например, вызвав в ответ какую-либо функцию у вызвавшего блока – его идентификатор содержится в поле `Caller` структуры `RDS_FUNCTIONCALLDATA`). Или, если нужно найти единственный блок, она может занести идентификатор своего блока в какое-либо поле структуры параметров и остановить перебор блоков, присвоив `TRUE` полю `Stop` структуры `RDS_FUNCTIONCALLDATA` (при этом вызывать функции блоков нужно с флагом `RDS_BCALL_ALLOWSTOP`).

§2.13.2. Прямой вызов функции одного блока

Рассматривается вызов функции блока, идентификатор которого известен вызывающей модели. Приводятся примеры использования функции “Common.ControlValueChanged”, которая используется блоками пользовательского интерфейса из библиотеки “Common.dll” для общения между собой. В двухкоординатную рукоятку и блок увеличения/уменьшения, созданные ранее, добавляется поддержка этой функции, что позволяет соединять их со стандартными полями ввода и синхронно изменять значения блоков в режиме моделирования.

Перейдем от абстрактных примеров прямого вызова функций блоков к более конкретным, и начнем с самого простого: будем вызывать функцию, которая уже придумана до нас, а также будем реагировать на ее вызов в нашем блоке.

Ранее (стр. 262) мы создали блок, уменьшающий и увеличивающий значение своего целого выхода при щелчках мыши и нажатиях клавиш. Этот блок вполне справляется со своей работой, однако, пользователь обычно хочет не только увеличивать и уменьшать какое-то значение, но и иметь возможность ввести его с клавиатуры. Самое очевидное решение этой проблемы – добавить в переменные блока целый вход, значение которого без изменений передается на выход, и соединить его со стандартным полем ввода. Выход нашего блока также необходимо соединить со входом поля ввода, замкнув их в кольцо (рис. 84). При таком соединении ввод числа в поле ввода будет приводить к попаданию нового значения на вход нашего блока и, следовательно, к изменению его выхода, а изменения значения выхода нашего блока при щелчках мыши или нажатии клавиш будет приводить к изменению значения в поле ввода. Таким образом, значение в поле ввода и выход нашего блока будут изменяться синхронно.

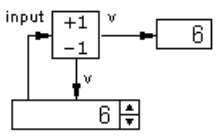


Рис. 84. Кольцевое соединение блока с полем ввода

Однако, все это будет работать только в режиме расчета. Если пользователь захочет остановить расчет, изменить значение, а затем запустить расчет заново, щелчки на нашем блоке будут изменять значение его выхода, но это никак не будет отражаться на значении в поле ввода: в режиме моделирования данные по связям не передаются. К счастью, в полях ввода предусмотрена специальная функция для связи с соседними органами управления – нужно только вызвать ее в соединенном поле ввода, а также добавить в наш блок реакцию на эту же функцию, чтобы поле ввода тоже могло вызывать ее.

Модели всех стандартных органов управления (рукояток, полей ввода и т.п.) из библиотеки “Common.dll” поддерживают функцию “Common.ControlValueChanged”, специально предназначенную для взаимодействия элементов интерфейса пользователя друг с другом. Если пользователь меняет значение в одном из таких интерфейсных блоков, модель этого блока принудительно передает данные выходов по связям, даже если РДС находится на в режиме расчета (для этого используется специальная сервисная функция РДС `rdsActivateOutputConnections`), после чего вызывает во всех блоках, с которыми соединены выходы данного, функцию “Common.ControlValueChanged”. У этой функции нет параметров, блоки должны считать измененное пользователем значение со своих входов. Таким образом, если несколько полей ввода соединены связями в кольцо, когда выход каждого из них соединен со входом другого, как на рис. 85, изменение пользователем значения в поле ввода A вызовет принудительную передачу этого значения по связи на вход соединенного с ним поля B, и вызов в нем функции “Common.ControlValueChanged”. Реагируя на вызов функции, поле ввода B должно изменить свое значение согласно своему входу, снова принудительно передать значение своего выхода по связям, и снова вызвать у соединенных блоков, то есть у поля ввода C, функцию “Common.ControlValueChanged”. Таким образом, введенное пользователем значение будет распространяться по цепочке соединенных полей ввода. Если эта цепочка замкнута в кольцо, как на рисунке, в конце концов значение вернется к полю ввода A, изменение значения в котором и привело к цепочке вызовов функции у соединенных с ним блоков. У этого поля тоже будет вызвана функция “Common.ControlValueChanged”, и здесь очень важно не допустить повторного вызова всей цепочки – если, реагируя на вызов функции, поле ввода A снова передаст значение полю ввода B, вызовы функции по кольцу будут продолжаться бесконечно, точнее, до тех пор, пока не переполнится стек и не произойдет аварийное завершение РДС. Для предотвращения этого проще всего ввести в каждый блок специальный флаг, который взводится перед вызовом “Common.ControlValueChanged” у соседних блоков и сбрасывается после этого вызова. Если, реагируя на вызов функции, модель блока обнаружит этот флаг взведенным, значит, этот блок уже участвовал в цепочке вызовов, и передавать вызов дальше не нужно. В этом случае ввод пользователем числа в поле ввода A приведет к следующей последовательности вызовов:

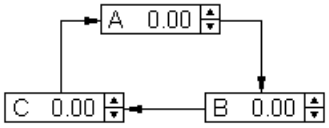


Рис. 85. Кольцо из трех полей ввода

Действие	***Флаг поля A***	***Флаг поля B***	***Флаг поля C***
Исходное состояние блоков.	сброшен	сброшен	сброшен
Пользователь ввел число в поле A, модель поля A, реагируя на его действия, взвела флаг и вызывает модель поля B.	ВЗВЕДЕН	сброшен	сброшен
Модель поля B приняла новое значение, взвела флаг и вызывает модель поля C.	ВЗВЕДЕН	ВЗВЕДЕН	сброшен

Действие	***Флаг поля A***	***Флаг поля B***	***Флаг поля C***
Модель поля C приняла новое значение, взвела флаг и вызывает модель поля A.	ВЗВЕДЕН	ВЗВЕДЕН	ВЗВЕДЕН
Модель поля A обнаружила, что флаг у ее блока взведен, и не стала реагировать на функцию. Управление возвращается вызвавшей модели, то есть модели поля C.	ВЗВЕДЕН	ВЗВЕДЕН	ВЗВЕДЕН
Модель поля C сбрасывает свой флаг и завершает реакцию на функцию. Управление возвращается модели поля B.	ВЗВЕДЕН	ВЗВЕДЕН	сброшен
Модель поля B сбрасывает свой флаг и завершает реакцию на функцию. Управление возвращается модели поля A.	ВЗВЕДЕН	сброшен	сброшен
Модель поля A сбрасывает свой флаг и завершает реакцию на действия пользователя.	сброшен	сброшен	сброшен

Таким образом, использование флага блокировки реакции на функцию позволяет предотвратить бесконечную рекурсию функций моделей при соединении блоков в кольцо. В РДС нет встроенного механизма для поддержки таких флагов, программист должен реализовать его самостоятельно.

Изменим модель и параметры нашего блока таким образом, чтобы его можно было подключать к полю ввода (или другому интерфейсному блоку) как на рис. 84. Прежде всего в переменные блока необходимо добавить целый вход `input`, на который будет подаваться значение с поля ввода. Для этого входа необходимо установить флаг запуска – при срабатывании связи, подключенной к нему, модель блока должна будет передать значение на выход. Новая структура переменных блока будет иметь следующий вид:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	0
1	Ready	Сигнал	1	Выход		–
2	v	int	4	Выход		–
6	input	int	4	Вход	✓	0

Для работы с функцией “Common.ControlValueChanged” введем целую глобальную переменную `ControlValueChangedId`, в нее при регистрации функции будет записываться присвоенный ей уникальный идентификатор. Функцию мы будем регистрировать при инициализации блока, причем только в том случае, если она еще не зарегистрирована, поэтому дадим этой глобальной переменной нулевое значение:

```
// Глобальная переменная для уникального идентификатора функции
// "Common.ControlValueChanged"
int ControlValueChangedId=0;
```

Для того, чтобы реализовать задуманное, нам потребуется функция, принудительно передающая значение выхода блока по связям, после чего вызывающая “Common.ControlValueChanged” у всех соединенных блоков. Мы будем вызывать ее при изменении выхода блока из-за действий пользователя (щелчков мыши или нажатия клавиш), а также при реакции нашего блока на “Common.ControlValueChanged”, чтобы новое значение передавалось по цепочке от блока к блоку. Эту функцию мы не будем делать членом класса

личной области данных нашего блока. Вместо этого сделаем ее глобальной – она может потребоваться нам и в других блоках.

Для принудительной передачи выхода блока по связям мы будем использовать сервисную функцию РДС `rdsActivateOutputConnections`. Она принимает два параметра, первый из которых – идентификатор блока (`RDS_BHANDLE`), данные которого необходимо передать. Если вместо идентификатора блока указать `NULL`, будут переданы данные того блока, модель которого в данный момент выполняется. Второй параметр функции – логическое значение, указывающее на способ передачи данных. Если во втором параметре передается `TRUE`, при передаче, как и в режиме расчета, учитываются логические и сигнальные переменные, привязанные ко входам и выходам блока:

- данные выходов не будут передаваться, если выход готовности блока (вторая сигнальная переменная, “Ready”) будет иметь нулевое значение;
- если выход имеет связанную логическую переменную, его значение будет передано по связи только при ненулевом значении этой переменной;
- если для входа, к которому подсоединена связь, в структуре переменных установлен флаг “Пуск”, первая сигнальная переменная (“Start”) принявшего данные блока автоматически получит значение 1;
- если у входа, к которому присоединена сработавшая связь, есть связанная сигнальная переменная, она тоже автоматически получит значение 1.

Если же передать во втором параметре функции `rdsActivateOutputConnections` значение `FALSE`, связанные логические и сигнальные переменные будут проигнорированы (такая передача данных обычно используется для установки начальных значений при сбросе расчета). Нам нужна обычная передача данных, полностью повторяющая работу связей в режиме расчета, поэтому мы будем передавать значение `TRUE`.

Для того, чтобы после передачи данных по связям вызвать во всех получивших данные блоках “Common.ControlValueChanged”, мы будем использовать сервисную функцию РДС `rdsEnumConnectedBlocks`. Она позволяет вызвать для каждого блока, соединенного связями с заданным, специально написанную функцию обратного вызова, указатель на которую передается в `rdsEnumConnectedBlocks` в одном из параметров. Наша задача – написать эту функцию обратного вызова таким образом, чтобы она вызывала в каждом блоке “Common.ControlValueChanged”.

Функция обратного вызова, используемая в `rdsEnumConnectedBlocks`, должна иметь следующий формат:

```
BOOL RDSCALL имя_функции(
      RDS_PPOINTDESCRIPTION nearpoint,// Точка связи данного блока
      RDS_PPOINTDESCRIPTION farpoint, // Точка связи "соседа"
      LPVOID ptr);                    // Дополнительный параметр
```

В первых двух параметрах функции передаются указатели на структуры описания точки связи `RDS_POINTDESCRIPTION`, соединенной с заданным блоком (первый параметр) и с блоком на другом конце связи (второй параметр). Эти структуры содержат идентификатор блока, имя переменной, к которой подходит связь, идентификатор связи, которой принадлежит точка, и т.п. Фактически, функция обратного вызова вызывается не для каждого блока, а для каждой точки связи, которая соединяет эту связь с каким-либо входом или выходом блока. Например, если связь разветвляется и соединяется с двумя входами одного и того же блока, функция обратного вызова будет вызвана дважды: один раз для первого входа, другой раз – для второго. Нас будет интересовать только идентификатор блока из структуры, переданной во втором параметре функции – именно у этого блока, находящегося на другом конце связи от нашего, мы должны будем вызвать “Common.ControlValueChanged”. Имя переменной, к которой подходит связь, нас, в данном случае, не волнует.

Третий параметр функции обратного вызова – указатель произвольного типа (`void*`, или, в определениях Windows API, `LPVOID`), передаваемый в одном из параметров `rdsEnumConnectedBlocks`. Он никак не обрабатывается РДС и обычно используется для передачи в функцию обратного вызова каких-либо дополнительных параметров. Возвращаемое функцией обратного вызова логическое значение позволяет остановить перебор блоков-соседей, если это необходимо: возврат `FALSE` прекращает перебор, возврат `TRUE` продолжает его. Нем необходимо перебрать все соединенные блоки, поэтому в нашей функции обратного вызова мы будем всегда возвращать `TRUE`.

Функция `rdsEnumConnectedBlocks` позволяет перебрать все блоки, соединенные с выходами заданного, с его входами, или и те, и другие:

```
RDS_BHANDLE RDSCALL rdsEnumConnectedBlocks(
      RDS_BHANDLE Block,          // Заданный блок или NULL
      DWORD Flags,                // Флаги, управляющие перебором
      RDS_BhPdPdpV CallBack,      // Функция обратного вызова
      LPVOID Data);               // Дополнительный параметр
```

В первом параметре функции передается идентификатор блока, соседи которого перебираются, или `NULL`, если необходимо перебрать соседей блока, модель которого выполняется в данный момент. Второй параметр содержит битовые флаги, которые указывают на типы перебираемых блоков:

- `RDS_BEN_INPUTS` – если флаг взведен, будут перебираться блоки, связи от которых идут ко входам заданного блока;
- `RDS_BEN_OUTPUTS` – если флаг взведен, будут перебираться блоки, связи к которым идут от выходов заданного блока;
- `RDS_BEN_TRACELINKS` – если флаг взведен, функция обратного вызова не будет вызываться для обнаруженных подсистем, блоков-входов и блоков-выходов. Вместо этого связи будут прослеживаться внутрь и наружу подсистем до простых блоков.

В третьем параметре передается указатель на функцию обратного вызова уже описанного вида, а в четвертом – дополнительный параметр, который без изменений передается в третьем параметре функции обратного вызова. Возвращает `rdsEnumConnectedBlocks` идентификатор блока, для которого функция обратного вызова вернула `FALSE`, или `NULL`, если перебраны все блоки и функция обратного вызова каждый раз возвращала `TRUE`.

Теперь мы можем написать функцию обратного вызова, которая будет уведомлять блоки, соединенные с нашим, о поступлении на их входы нового значения:

```
// Функция обратного вызова для "Common.ControlValueChanged"
BOOL RDSCALL ControlValChanged_Callback(
      RDS_PPOINTDESCRIPTION /*src*/,
      RDS_PPOINTDESCRIPTION dest,
      LPVOID /*data*/)
{ // Вызов функции "Common.ControlValueChanged" у блока на другом
  // конце связи
  rdsCallBlockFunction(dest->Block,ControlValueChangedId,NULL);
  // Возвращаем TRUE - не останавливаем перебор блоков
  return TRUE;
}
//=======================================
```

Из трех параметров функции обратного вызова нам нужен только второй (`dest`) – указатель на структуру, описывающую точку связи, соединенную с соседним блоком. В параметре `Block` этой структуры содержится идентификатор соединенного блока, который мы и используем в вызове `rdsCallBlockFunction`. Во втором параметре `rdsCallBlockFunction` мы передаем идентификатор вызываемой функции, который должен находиться в глобальной переменной `ControlValueChangedId` (предполагается,

что функция уже зарегистрирована). В третьем параметре должен был бы быть указатель на параметры вызываемой функции, но “Common.ControlValueChanged” не имеет параметров, поэтому мы передаем вместо него NULL.

Теперь напишем функцию, которая будет принудительно передавать по связям данные всех выходов блока, модель которого выполняется в данный момент, и уведомлять об этом все соединенные блоки вызовом “Common.ControlValueChanged”. Мы сможем вызывать эту функцию из любого места модели нашего блока – не важно, из самой ли функции модели, или из какой-либо функции-члена класса его личной области данных (функции-члены вызываются из самой функции модели, и параллельно с ней не сможет выполняться никакая другая из-за блокировки данных). Дополнительно мы включим в эту функцию работу с флагом блокировки, принцип действия которого был описан выше. Функция будет иметь следующий вид:

```
// Функция уведомления соседей об изменении значения
void ControlValueChangedCall(BOOL *pCancelCall)
{ // В параметре pCancelCall передается указатель на флаг
  // блокировки вызова функции
  if(*pCancelCall)    // Вызов заблокирован
    return;

  // Принудительно передаем данные выходов блока, модель которого
  // сейчас выполняется, по связям
  rdsActivateOutputConnections(NULL,TRUE);

  // Взводим флаг блокировки перед вызовом функций
  *pCancelCall=TRUE;
  // Перебираем все простые блоки, соединенные с выходами текущего
  // (для каждого будет вызвана ControlValChanged_Callback)
  rdsEnumConnectedBlocks(NULL,
                         RDS_BEN_OUTPUTS | RDS_BEN_TRACELINKS,
                         ControlValChanged_Callback,
                         NULL);
  // Сбрасываем флаг блокировки после вызова функций
  *pCancelCall=FALSE;
}
//=======================================
```

В параметре pCancelCall в функцию передается указатель на флаг блокировки вызова, который должен быть выполнен в виде логического (BOOL) поля в классе личной области данных блока. Прежде всего функция проверяет, взведен ли этот флаг, то есть истинно ли значение логической переменной, на которую ссылается переданный указатель. Если это так, функция немедленно завершается – блок, для которого она вызвана, уже участвует в цепочке вызовов “Common.ControlValueChanged”, и продолжение работы функции приведет к бесконечной рекурсии. Если флаг сброшен, вызывается функция rdsActivateOutputConnections, которая передает данные выходов блока, из модели которого вызвана функция ControlValueChangedCall (вместо идентификатора блока передано значение NULL), по связям, с учетом логики их работы (второй параметр – TRUE). Теперь нужно уведомить все соединенные блоки о поступлении на их входы новых значений, вызвав в них функцию “Common.ControlValueChanged”. Для этого флаг блокировки вызова взводится, чтобы данный блок, для которого вызвана ControlValueChangedCall, не отреагировал на ее повторный вызов, и вызывается функция перебора соединенных блоков rdsEnumConnectedBlocks. В первом параметре функции перебора передается NULL, чтобы она работала с текущим блоком (то есть с тем блоком, для которого вызвана ControlValueChangedCall). Нам нужно перебрать только блоки, соединенные с выходами данного, причем нам нужно прослеживать связи до

простых блоков внутрь и наружу подсистем (соединенные поля ввода не обязательно находятся в одной и той же подсистеме), поэтому мы используем флаги RDS_BEN_OUTPUTS и RDS_BEN_TRACELINKS. Для каждого блока будет вызываться уже написанная нами функция ControlValChanged_Callback, в которую мы не передаем никаких дополнительных параметров, поэтому последний параметр rdsEnumConnectedBlocks – NULL. После перебора всех соединенных блоков флаг блокировки вызовов снова сбрасывается, чтобы блок был готов к приему новых значений.

В личную область данных блока необходимо добавить логический флаг для блокировки повторных вызовов, и в конструкторе класса присвоить ему значение FALSE. Кроме того, включим в конструктор блока регистрацию функции “Common.ControlValueChanged”:

```
//====== Класс личной области данных ======
class TPlusMinusData
{ public:
    int KeyPlus;      // Клавиша увеличения
    DWORD ShiftsPlus; // и ее флаги
    int KeyMinus;     // Клавиша уменьшения
    DWORD ShiftsMinus;// и ее флаги

    // Флаг блокировки повторных вызовов
    // функции "Common.ControlValueChanged"
    BOOL NoCall;

    int Setup(void);       // Функция настройки клавиш
    void SaveBin(void);    // Сохранение параметров
    int LoadBin(void);     // Загрузка параметров

    // Конструктор класса
    TPlusMinusData(void)
      { KeyPlus=ShiftsPlus=KeyMinus=ShiftsMinus=0;
        NoCall=FALSE; // Исходно флаг блокировки сброшен
        if(ControlValueChangedId==0)  // Регистрация функции
          ControlValueChangedId=
               rdsRegisterFunction("Common.ControlValueChanged");
      };
};
//=========================================
```

Теперь все готово для внесения изменений в функцию модели блока. С учетом того, что структура переменных блока изменилась (мы добавили новый вход), теперь функция будет выглядеть так:

```
// Увеличение/уменьшение значения по щелчку и клавишам
extern "C" __declspec(dllexport)
  int RDSCALL PlusMinus(int CallMode,
                    RDS_PBLOCKDATA BlockData,
                    LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define v (*((int *)(pStart+2)))
#define input (*((int *)(pStart+6)))
  // Вспомогательная – указатель на структуру события мыши
  RDS_PMOUSEDATA mouse;
  // Вспомогательная – указатель на структуру события клавиатуры
  RDS_PKEYDATA key;
```

```
// Указатель на личную область данных блока,приведенный к
// правильному типу
TPlusMinusData *data=(TPlusMinusData*)(BlockData->BlockData);
switch(CallMode)
  { // Проверка типа статических переменных
    case RDS_BFM_VARCHECK:
      return strcmp((char*)ExtParam,"{SSII}")?
                    RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

    // Реакция на нажатие кнопки мыши
    case RDS_BFM_MOUSEDOWN:
      // Приведение ExtParam к нужному типу
      mouse=(RDS_PMOUSEDATA)ExtParam;
      if(mouse->Button==RDS_MLEFTBUTTON)
        { // Нажата левая кнопка
          // Проверяем, есть ли у блока картинка (получаем
          // описание блока)
          RDS_BLOCKDESCRIPTION descr;
          descr.servSize=sizeof(descr);
          rdsGetBlockDescription(BlockData->Block,&descr);
          if(descr.Flags & RDS_BDF_HASPICTURE)
            { // Картинка есть - определяем идентификатор
              // элемента под курсором
              int id=rdsGetMouseObjectId(mouse);
              v+=id;
            }
          else if(mouse->y<mouse->Top+mouse->Height/2)
            v++;    // В верхней половине блока - увеличиваем
          else
            v--;    // В нижней половине блока — уменьшаем
          // Взводим сигнал готовности
          Ready=1;
          // Принудительно передаем в соседние блоки
          ControlValueChangedCall(&(data->NoCall));
        }
      break;

    case RDS_BFM_INIT:   // Инициализация
      BlockData->BlockData=new TPlusMinusData();
      break;
    case RDS_BFM_CLEANUP:// Очистка данных
      delete data;
      break;
    case RDS_BFM_SETUP:  // Настройка параметров
      return data->Setup();
    case RDS_BFM_SAVEBIN:// Сохранение параметров
      data->SaveBin();
      break;
    case RDS_BFM_LOADBIN:// Загрузка параметров
      return data->LoadBin();

    // Реакция на нажатие клавиши
    case RDS_BFM_KEYDOWN:
      // Приведение ExtParam к нужному типу
      key=(RDS_PKEYDATA)ExtParam;
```

```
            // Сравнение нажатой клавиши с клавишами уменьшения
            // и увеличения
            if(key->KeyCode==data->KeyPlus &&
               key->Shift==data->ShiftsPlus)
              { v++; Ready=1; }
            else if(key->KeyCode==data->KeyMinus &&
                    key->Shift==data->ShiftsMinus)
              { v--; Ready=1; }
            // Принудительно передаем в соседние блоки
            if(Ready)
              ControlValueChangedCall(&(data->NoCall));
            break;

          // Такт расчета
          case RDS_BFM_MODEL:
            if(v==input)  // Новое значение равно старому
              { Ready=0;  // Не передаем по связям
                break;
              }
            v=input;      // Передаем значение входа на выход
            // Принудительно передаем в соседние блоки
            ControlValueChangedCall(&(data->NoCall));
            break;

          // Реакция на вызов функции блока
          case RDS_BFM_FUNCTIONCALL:
            if(((RDS_PFUNCTIONCALLDATA)ExtParam)->Function==
                                        ControlValueChangedId)
              { // Вызвана "Common.ControlValueChanged"
                if(data->NoCall)     // Взведен флаг блокировки
                  break;
                if(v==input)   // Новое значение равно старому -
                  break;       // передавать не нужно
                v=input;  // Передаем значение входа на выход
                Ready=1;  // Взводим сигнал готовности
                // Принудительно передаем в соседние блоки
                ControlValueChangedCall(&(data->NoCall));
              }
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef input
    #undef v
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Кроме макроопределения для дополнительной переменной и дополнительного символа “I” для нее в строке типа статических переменных, в функции модели изменены реакции на мышь и клавиатуру, а также добавлены две новые реакции: выполнение такта расчета RDS_BFM_MODEL и реакция на вызов функции блока RDS_BFM_FUNCTIONCALL.

После изменения значения выхода блока при нажатии кнопки мыши или одной из заданных в настройках клавиш, мы теперь вызываем функцию

ControlValueChangedCall, передавая ей указатель на логический флаг блокировки вызова NoCall из класса личной области данных. Следует обратить внимание на то, что этот вызов производится после присваивания переменной блока Ready значения 1: внутри ControlValueChangedCall мы принудительно передаем значения выхода блока на входы соединенных с ним с учетом логики работы связей, поэтому сигнал готовности блока на момент этой передачи должен быть взведен, иначе логика связей не даст им сработать.

В реакции на такт расчета мы просто копируем значение, поступившее на вход блока input, в его выход v, если оно отличается от текущего значения выхода, и вызываем ControlValueChangedCall, чтобы передать данные соседним блокам и вызвать у них "Common.ControlValueChanged". Может возникнуть вопрос: в режиме расчета данные с выхода блока передаются по связям автоматически, так зачем нам предпринимать действия по принудительной передаче этих данных? Дело в том, что, в данном случае, нам лучше передать данные как можно быстрее, а не в конце текущего такта расчета. Представим себе, что, по какой-то причине, на выходе нашего блока и на выходе соединенного с ним поля ввода оказались разные значения, и у обоих блоков взведен сигнал готовности. В этом случае в схеме возникнут колебания: в каждом такте расчета блоки будут обмениваться значениями и взводить сигнал готовности. Принудительная передача данных по связям ликвидирует эти колебания: на момент конца такта, когда происходит обычная передача, значения в блоках уже будут одинаковыми, а сигналы готовности – сброшенными (их сбрасывает rdsActivateOutputConnections после передачи).

В реакции на вызов функции блока мы прежде всего проверяем, какая именно функция вызвана. Для этого мы сравниваем поле Function структуры RDS_FUNCTIONCALLDATA, указатель на которую передан в ExtParam, с глобальной переменной ControlValueChangedId, в которой хранится целый идентификатор, присвоенный функции "Common.ControlValueChanged" при регистрации (мы регистрируем ее в конструкторе класса). Если они совпали, значит, вызвана именно "Common.ControlValueChanged", и нам необходимо принять новое значение со входа и передать его дальше по цепочке соединенных блоков. Сначала проверяется, взведен ли флаг блокировки вызовов data->NoCall: если это так, на вызов функции реагировать нельзя. Если флаг сброшен, мы сравниваем поступившее на вход значение с текущим значением выхода блока: если они совпадают, передавать значение дальше по цепочке не имеет смысла – оно не изменилось. В противном случае мы копируем значение входа в выход, взводим сигнал готовности и вызываем ControlValueChangedCall для принудительной передачи данных дальше по цепочке.

Теперь к нашему блоку можно подключить поле ввода, как на рис. 84, или любой другой блок пользовательского интерфейса. Значения в обоих блоках теперь будут синхронно изменяться и в режиме расчета, и в режиме моделирования.

Если у блока, позволяющего пользователю изменять какие-либо значения, есть несколько независимых выходов, и мы хотим добавить в него поддержку функции "Common.ControlValueChanged", нужно уметь различать, какое именно из значений изменилось. В этом нам могут помочь связанные со входами блока сигналы (см. стр. 75). Хотя это и не имеет прямого отношения к вызову функций блоков, для большей ясности рассмотрим такую ситуацию на примере.

Ранее (стр. 268) мы создали блок, имитирующий рукоятку, позволяющую задавать две независимые координаты, и уже добавили в него несколько дополнительных возможностей. Тем не менее, у блока остался один, достаточно серьезный, недостаток: он не отображает точных значений координат, соответствующих текущему положению рукоятки. Можно подключить к блоку числовые индикаторы, как на рис. 75, но они будут показывать значения выходов блока только в режиме расчета. Если пользователь захочет остановить расчет, изменить значения координат, а затем запустить расчет заново, ему придется двигать

рукоятку наугад. Кроме того, блок не позволяет вводить точные значения координат с клавиатуры. Напрашивается очевидное решение: добавить в блок рукоятки поддержку функции “Common.ControlValueChanged” и подключить к нему пару полей ввода – по одному полю на координату, как на рис. 86. На рисунке в блок добавлены два дополнительных входа: `x_in` для связи с полем ввода `x` и `y_in` для связи с полем ввода `y`.

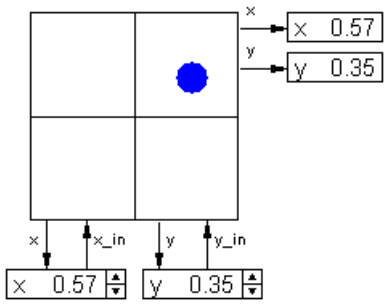


Рис. 86. Рукоятка с полями ввода

Когда мы будем добавлять в блок реакцию на вызов функции, очень важно отличать ее вызов из-за изменения `x_in` от вызова из-за изменения `y_in`. Если, реагируя на вызов, считывать обе переменные одновременно, поведение блока будет неправильным. Допустим, пользователь передвинул рукоятку из положения (0.1, 0.3) в положение (0.5, 0.7). Рассмотрим, что будет происходить в схеме, если мы не различаем изменения переменных в реакции на “Common.ControlValueChanged”, при подключении к блоку полей ввода, как на рисунке:

1. Значения выходов блока-рукоятки изменились: `x`=0.5, `y`=0.7.
2. Изменившиеся значения `x` и `y` передались на входы одноименных полей ввода при помощи вызова `rdsActivateOutputConnections`.
3. Как и в предыдущем примере, мы начали перебирать соединенные с выходами рукоятки блоки при помощи функции `rdsEnumConnectedBlocks`, вызывая у каждого из них “Common.ControlValueChanged”. Допустим, первым обнаруженным блоком оказалось поле ввода “x”.
4. Поле ввода “x”, реагируя на вызов функции, считало значение 0.5 со своего входа, скопировало его на свой выход, передало его по связи на вход рукоятки `x_in`, и вызвало у нее функцию “Common.ControlValueChanged”.
5. Рукоятка, реагируя на вызов функции, считала значения своих входов `x_in` и `y_in`, и переписала их в выходы `x` и `y`. Вход `x_in` имеет правильное значение 0.5, полученное с поля ввода “x”, но вход `y_in` до сих пор имеет старое значение 0.3, поскольку поле ввода “y” еще не вызывалось, и поэтому не успело передать поступившее на его вход значение 0.7 по связи в рукоятку.

Дальнейшие действия можно уже не рассматривать: значение выхода `y`=0.7, заданное пользователем, может быть потеряно из-за того, что блок-рукоятка, реагируя на вызов функции, поступивший от поля ввода “x”, считал не только значение `x_in`, но и `y_in`. Избежать возникновения подобных проблем можно связав входы `x_in` и `y_in` с внутренними переменными-сигналами (назовем их `x_in_ok` и `y_in_ok` соответственно). Поскольку при вызове `rdsActivateOutputConnections` мы не отключаем логику работы связей (второй параметр функции – `TRUE`), эти сигналы автоматически взведутся при срабатывании связи, подключенной к соответствующему входу. Так мы сможем узнать, какая из переменных изменилась.

Таким образом, блок-рукоятка теперь будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Логика*	*Начальное значение*
0	Start	Сигнал	1	Вход		0
1	Ready	Сигнал	1	Выход		–
2	x	double	8	Выход		–
10	y	double	8	Выход		–
18	x_in	double	8	Вход/сигнал	x_in_ok	–

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Логика*	*Начальное значение*
26	y_in	double	8	Вход/сигнал	y_in_ok	–
34	x_in_ok	Сигнал	1	Внутренняя		0
35	y_in_ok	Сигнал	1	Внутренняя		0

Как и в предыдущем примере, добавим в класс личной области данных блока флаг блокировки вызова, а в конструктор этого класса – регистрацию функции “Common.ControlValueChanged”:

```
//====== Класс личной области данных ======
class TSimpleJoystick
{ private:
    // Центр круга (рукоятки) до начала перетаскивания
    int OldHandleX,OldHandleY;
    // Координаты курсора на момент начала перетаскивания
    int OldMouseX,OldMouseY;
    // Флаги фиксации одной из координат
    BOOL LockX,LockY;
    // Идентификаторы добавленных пунктов меню
    RDS_MENUITEM MenuLockX,MenuLockY;
  public:
    // Настроечные параметры блока
    COLORREF BorderColor;        // Цвет рамки блока
    COLORREF FieldColor;         // Цвет прямоугольника
    COLORREF HandleColor;        // Цвет круга в покое
    COLORREF MovingHandleColor;  // Цвет круга при таскании
    COLORREF GrayedColor;        // Цвет недоступной области
    int HandleSize;              // Диаметр круга

    // Флаг блокировки повторных вызовов
    // функции "Common.ControlValueChanged"
    BOOL NoCall;

    // Реакция на нажатие кнопки мыши
    int MouseDown(RDS_PMOUSEDATA mouse,double x,double y,
                  DWORD *pFlags);
    // Реакция на перемещение курсора мыши
    void MouseMove(RDS_PMOUSEDATA mouse,double *px,double *py);
    // Рисование изображения блока
    void Draw(RDS_PDRAWDATA draw,double x,double y,BOOL moving);
    // Реакция на выбор добавленного пункта меню
    void MenuFunction(RDS_PMENUFUNCDATA MenuData);
    // Ограничение входных значений
    void LimitInputValues(double *px_in,double *py_in,
                          double x,double y);
    // Конструктор класса
    TSimpleJoystick(void)
      { BorderColor=0;           // Черная рамка
        FieldColor=0xffffff;     // Белое поле
        HandleColor=0xff0000;    // Синий круг
        MovingHandleColor=0xff;  // Красный при таскании
        GrayedColor=0x7f7f7f;    // Серый
        LockX=LockY=FALSE;       // Фиксация выключена
        HandleSize=20;           // Диаметр круга
```

```
        // Создание пунктов меню
        MenuLockX=rdsRegisterContextMenuItem("Фиксировать X",1,0);
        MenuLockY=rdsRegisterContextMenuItem("Фиксировать Y",2,0);
        NoCall=FALSE; // Исходно флаг блокировки сброшен
        if(ControlValueChangedId==0)  // Регистрация функции
          ControlValueChangedId=
               rdsRegisterFunction("Common.ControlValueChanged");
      };

      // Деструктор класса
      ~TSimpleJoystick()
      { // Уничтожение пунктов меню
        rdsUnregisterMenuItem(MenuLockX);
        rdsUnregisterMenuItem(MenuLockY);
      };
};
//=======================================
```

Глобальную переменную `ControlValueChangedId` для хранения идентификатора функции “Common.ControlValueChanged”, полученного при ее регистрации, мы уже ввели в предыдущем примере. Там же мы написали функцию `ControlValueChangedCall`, которая вызывает “Common.ControlValueChanged” у всех блоков, соединенных с выходами данного. В этом примере она нам тоже понадобится.

Мы также добавили в класс новую функцию `LimitInputValues`, которую будем использовать для ограничения значений, поступивших на вход блока. Значение каждой из координат рукоятки должно лежать в диапазоне [–1...1]. Кроме того, если одна из координат рукоятки зафиксирована, блок должен игнорировать значение, поступившее на соответствующий вход. Поскольку на входы блока могут поступить любые значения, функция `LimitInputValues` будет корректировать значения входов (указатели на них передаются в первом и втором ее параметрах) согласно указанным требованиям. В третьем и четвертом параметрах передаются текущие значения выходов блока: они понадобятся, если координата зафиксирована (в этом случае в соответствующий вход будет записано зафиксированное значение выхода). Функция будет иметь следующий вид:

```
// Ограничение значений входов блока
void TSimpleJoystick::LimitInputValues(double *px_in,double *py_in,
                                       double x,double y)
{ // px_in, py_in - указатели на входы блока
  // x, y - текущие значения выходов
  if(LockX)      // Координата x зафиксирована
    *px_in=x;    // Присваиваем входу x_in зафиксированное значение
  else           // Ограничиваем x_in диапазоном [-1...1]
    { if(*px_in<-1.0) *px_in=-1.0;
      else if(*px_in>1.0) *px_in=1.0;
    }
  if(LockY)      // Координата y зафиксирована
    *py_in=y;    // Присваиваем входу y_in зафиксированное значение
  else           // Ограничиваем y_in диапазоном [-1...1]
    { if(*py_in<-1.0) *py_in=-1.0;
      else if(*py_in>1.0) *py_in=1.0;
    }
}
//=======================================
```

Теперь внесем изменения в функцию модели блока:

```
// Двухкоординатная рукоятка
extern "C" __declspec(dllexport)
  int RDSCALL SimpleJoystick(int CallMode,
                             RDS_PBLOCKDATA BlockData,
                             LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define x (*((double *)(pStart+2)))
#define y (*((double *)(pStart+10)))
#define x_in (*((double *)(pStart+18)))
#define y_in (*((double *)(pStart+26)))
#define x_in_ok (*((char *)(pStart+34)))
#define y_in_ok (*((char *)(pStart+35)))
  // Вспомогательная переменная - указатель на личную область,
  // приведенный к правильному типу
  TSimpleJoystick *data=(TSimpleJoystick*)(BlockData->BlockData);
  switch(CallMode)
    { // ...
      // Реакции RDS_BFM_INIT и RDS_BFM_CLEANUP - без изменений
      // ...

      // Проверка допустимости типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSDDDDSS}")?
                     RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // ... Реакции RDS_BFM_MOUSEDOWN и RDS_BFM_MOUSEUP -
      // без изменений ...

      // Перемещение курсора мыши
      case RDS_BFM_MOUSEMOVE:
        // Проверка: включен ли захват мыши
        if(BlockData->Flags & RDS_MOUSECAPTURE)  // Включен
          { // Запоминаем старые значения
            double oldX=x,oldY=y;
            // Вызываем функцию реакции
            data->MouseMove((RDS_PMOUSEDATA)ExtParam,&x,&y);
            Ready=1;  // Взводим сигнал готовности
            // Если значения изменились, передаем соседям
            if(oldX!=x || oldY!=y)
              ControlValueChangedCall(&(data->NoCall));
          }
        break;

      // ... Реакции RDS_BFM_DRAW, RDS_BFM_MENUFUNCTION,
      // RDS_BFM_RESIZE, RDS_BFM_RESIZING - без изменений ...

      // Один такт расчета
      case RDS_BFM_MODEL:
        if(x_in_ok==0 && y_in_ok==0)
          { // Связанные сигналы не взведены
            Ready=0;  // Сбрасываем сигнал готовности
            break;
          }
```

```
        if(x==x_in && y==y_in)
          { // На входах те же значения, что и на выходах
            Ready=x_in_ok=y_in_ok=0;  // Сбрасываем все сигналы
            break;
          }
        // Ограничиваем входные значения
        data->LimitInputValues(&x_in,&y_in,x,y);
        // В зависимости от того, какой связанный сигнал взведен,
        // копируем соответствующие входы в выходы
        if(x_in_ok) x=x_in;
        if(y_in_ok) y=y_in;
        x_in_ok=y_in_ok=0; // Сбрасываем связанные сигналы
        // Передаем данные соединенным блокам
        ControlValueChangedCall(&(data->NoCall));
        break;

      // Реакция на вызов функции
      case RDS_BFM_FUNCTIONCALL:
        if(((RDS_PFUNCTIONCALLDATA)ExtParam)->Function==
                    ControlValueChangedId)
          { // Вызвана "Common.ControlValueChanged"
            if(data->NoCall)     // Вызов заблокирован
              break;
            // Ограничиваем входные значения
            data->LimitInputValues(&x_in,&y_in,x,y);
            // Если связанные сигналы не взведены - не реагируем
            if(x_in_ok==0 && y_in_ok==0)
              break;
            // Если значения входов те же, что и у выходов -
            // не реагируем
            if(x_in==x && y_in==y)
              break;
            // Копируем входы в выходы согласно связанным сигналам
            if(x_in_ok) x=x_in;
            if(y_in_ok) y=y_in;
            x_in_ok=y_in_ok=0;  // Сбрасываем связанные сигналы
            Ready=1;            // Взводим сигнал готовности
            // Передаем данные соединенным блокам
            ControlValueChangedCall(&(data->NoCall));
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef y_in_ok
#undef x_in_ok
#undef y_in
#undef x_in
#undef y
#undef x
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Поскольку в блоке появилось четыре новых переменных, соответствующим образом изменена строка типов в реакции RDS_BFM_VARCHECK. Изменения внесены также в реакцию на перемещение курсора мыши RDS_BFM_MOUSEMOVE: теперь, если в результате действий пользователя выходы блока изменились (то есть пользователь перетащил круг рукоятки на новое место), вызывается функция ControlValueChangedCall, которую мы написали в предыдущем примере. Она передаст новые значения по связям и вызовет у соединенных блоков функцию “Common.ControlValueChanged”. Это изменит значения в полях ввода, соединенных с рукояткой.

В функцию модели добавлены две новых реакции: выполнение такта расчета RDS_BFM_MODEL и реакция на вызов функции RDS_BFM_FUNCTIONCALL. В такте расчета прежде всего проверяется, взведен ли хотя бы один из флагов x_in_ok и y_in_ok, связанных с входами блока x_in и y_in соответственно. Если оба флага сброшены, блоку не нужно ничего делать, и реакция завершается, предварительно сбросив значение флага готовности Ready. В противном случае значения, поступившие на входы, сравниваются с текущими значениями выходов блока x и y. Если они совпадают, блок тоже может ничего не делать: нет смысла передавать по связям то же самое значение еще раз. В этом случае сбрасывается флаг готовности Ready и оба связанных со входами сигнала, и реакция завершается. Если же значения на входах отличаются от значений на выходах, значит, нужно их обработать и передать на выходы. Поступившие на вход значения обрабатываются функцией LimitInputValues, чтобы не допустить их выход за пределы диапазона [-1...1] и нарушения фиксации одной из координат, если она включена. Затем значения входов, соответствующие взведенным связанным сигналам, копируются в выходы, и связанные сигналы сбрасываются (в РДС сигналы всегда нужно сбрасывать вручную, см. §2.5.2). После этого, как и в предыдущем примере, вызывается функция ControlValueChangedCall для того, чтобы принудительно передать данные соседним блокам, не дожидаясь окончания такта расчета. Это необходимо для устранения возможных колебаний в схеме, если вдруг у нашего блока и соединенного с ним поля ввода окажутся разные значения на выходах при одновременно взведенных сигналах готовности (подробнее см. стр. 329).

В реакции на вызов функции блока RDS_BFM_FUNCTIONCALL мы, прежде всего, проверяем, совпадает ли переданный при вызове идентификатор функции со значением глобальной переменной ControlValueChangedId. Если это так, значит, вызвана функция “Common.ControlValueChanged”. Если при этом взведен флаг блокировки вызовов NoCall, реакция завершается – этот блок уже участвует в цепочке вызовов функции, и реагировать на повторный вызов нельзя. Если же флаг сброшен, выполняются те же действия, что и в такте расчета, с единственным исключением: если в такте расчета флаг готовности блока взводится автоматически, то при вызове функции блока этого не происходит. Его необходимо взвести вручную (выходу Ready присваивается значение 1), причем сделать это необходимо до вызова функции ControlValueChangedCall, поскольку сервисная функция rdsActivateOutputConnections, вызываемая внутри нее, не будет работать при сброшенном сигнале готовности блока.

Теперь, когда пользователь будет перемещать рукоятку, ее координаты будут отображаться в соединенных с ней полях ввода (см. рис. 86) как в режиме расчета, так и в режиме моделирования. В свою очередь, если пользователь будет менять значения в полях ввода, рукоятка будет перемещаться согласно введенным значениям.

В обоих рассмотренных примерах мы использовали одну и ту же стандартную функцию “Common.ControlValueChanged” без параметров. В следующем параграфе мы создадим собственную функцию с параметрами, и будем вызывать ее у всех блоков подсистемы.

§2.13.3. Прямой вызов функции всех блоков подсистемы

Рассматривается вызов функции всех блоков одной подсистемы. В созданную ранее модель блока, управляющего полем ввода, добавляется возможность одновременного открытия, закрытия и переключения состояния всех таких блоков, находящихся в одной и той же подсистеме.

Ранее (стр. 277) мы создали модель блока, управляющего полем ввода, а затем добавили в его контекстное меню пункт, позволяющий переключать его состояние, то есть разрешать или запрещать работу поля ввода, поверх которого расположен наш блок (§2.12.6). Введем в эту модель еще одну модификацию: сделаем так, чтобы через контекстное меню блока можно было управлять всеми такими блоками в подсистеме. Пункт меню “Открыть все” будет разрешать работу всех полей ввода, связанных с управляющими блоками, пункт “Закрыть все” – запрещать работу таких полей, а пункт “Переключить все” – менять режим всех управляющих блоков на противоположный, то есть все открытые блоки должны закрыться, а все закрытые – открыться.

Для того, чтобы открывать, закрывать или переключать все блоки подсистемы, будем использовать вызов функции блока. При выборе пользователем одного из трех перечисленных выше пунктов меню, модель блока, контекстное меню которого открыто, будет вызывать эту функцию у всех блоков своей подсистемы, при этом в параметрах функции будет каким-то образом указываться, что именно должен сделать вызванный блок: открыться, закрыться или переключиться. Название и структуру параметров функции нам предстоит придумать.

Поскольку эта функция нужна нам для примера в этом тексте, реагировать на нее будут только блоки управления полями ввода (мы назвали функцию модели этих блоков `EditControlFrame`), и предназначена она для установки режима работы блока, мы отразим в ее названии три этих момента: назовем ее “ProgrammersGuide.EditCtrlFrame.Set”. Параметры этой функции мы оформим как структуру из двух полей: первое будет содержать размер этой структуры для проверки правильности передачи параметров (см. §2.13.1), во втором будет находиться целое число, указывающее на действие, которое должен выполнить блок: 0 – закрыться, 1 – открыться, 2 – переключиться. Таким образом, для работы с функцией нам потребуются следующие описания:

```
//=======================================
// Функция управления состоянием блока
//=======================================
// Имя функции
#define PROGGUIDEEDITCTRLFUNC_SET \
                        "ProgrammersGuide.EditCtrlFrame.Set"
// Структура параметров функции
typedef struct
{ DWORD servSize;     // Размер этой структуры для проверки
  int Command;        // Команда блоку (0, 1 или 2)
} TProgGuideEditCtrlSetParams;
//=======================================
```

Для строки имени функции мы вводим константу `PROGGUIDEEDITCTRLFUNC_SET`. Использование этой константы позволит избежать ошибок, если мы когда-либо захотим ввести поддержку этой функции в другие модели блоков. Набирая тексте каждой модели имя функции вручную или копируя его через буфер обмена, можно случайно пропустить в нем один символ или как-то еще исказить его, что приведет к трудно выявляемым ошибкам: РДС просто зарегистрирует еще одну функцию, в результате чего в схеме будет одновременно зарегистрировано две независимые функции – одна с правильным именем, а другая – с искаженным, и их целые идентификаторы, естественно, будут различаться. Если же мы будем использовать вместо строки имени функции введенную для нее константу, и случайно исказим имя этой константы, такая ошибка будет обнаружена на этапе компиляции. Вообще,

если создаваемую функцию предполагается в будущем использовать и в других моделях, целесообразно вынести необходимые для нее описания в файл заголовка, чтобы другие программисты могли просто включить его в свои тексты моделей. Например, имена и описания всех функций блоков из стандартной библиотеки “Common.dll ”находятся в файле “CommonBl.h”, поэтому в предыдущих примерах можно было, включив этот файл, вместо строки имени функции “Common.ControlValueChanged” использовать константу COMBL_F_CONTROLCHANGED_NAME.

Кроме константы для имени функции мы также описываем структуру ее параметров TProgGuideEditCtrlSetParams. Полю servSize этой структуры перед вызовом функции необходимо будет присвоить ее размер, полученный оператором sizeof, а в реакции на ее вызов сравнить это поле с этим же размером, и, если значение поля окажется меньше, не выполнять никаких действий, поскольку это говорит о том, что функции переданы какие-то неправильные параметры. В поле Command нужно будет записать команду блоку.

Нам также потребуется глобальная переменная для хранения идентификатора этой функции, полученного при регистрации:

```
// Идентификатор функции PROGGUIDEEDITCTRLFUNC_SET
int EditCtrlFuncSet=0;
```

Теперь можно вносить изменения в функцию модели блока. Нам потребуется добавить в нее регистрацию функции, реакцию на эту функцию, а также три новых пункта контекстного меню, при выборе которых описанная нами функция будет вызываться у всех блоков подсистемы.

Прежде всего, в функции модели нам понадобятся дополнительные переменные для работы с функцией и ее параметрами:

```
extern "C" __declspec(dllexport)
  int RDSCALL EditControlFrame(int CallMode,
                               RDS_PBLOCKDATA BlockData,
                               LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define x_ext (*((double *)(pStart+2)))
#define x_int (*((double *)(pStart+10)))
#define out (*((double *)(pStart+18)))
#define bypass (*((char *)(pStart+26)))
  const int fr=20;    // Толщина рамки
  // Вспомогательные переменные
  RDS_PMOUSEDATA mouse;
  RDS_PDRAWDATA draw;
  int frz,x1,y1,x2,y2,xi1,yi1,xi2,yi2;

  RDS_PFUNCTIONCALLDATA func;    // Описание вызванной функции
  RDS_PMENUFUNCDATA menu;        // Описание выбранного пункта меню
  TProgGuideEditCtrlSetParams callparams; // Параметры функции

  switch(CallMode)
  // ...
```

Теперь нужно добавить внутрь оператора switch(CallMode) новые реакции и внести изменения в уже существующие там, где это необходимо. Регистрацию функции добавим в реакцию на подключение модели к блоку. У этого блока нет личной области данных, и этой реакции в нем раньше не было:

```
        // Инициализация блока
        case RDS_BFM_INIT:
```

```
    if(!EditCtrlFuncSet)
      EditCtrlFuncSet=
            rdsRegisterFunction(PROGGUIDEEDITCTRLFUNC_SET);
    break;
```

Как и в предыдущих примерах, мы регистрируем функцию только в том случае, если глобальная переменная `EditCtrlFuncSet` сохранила свое нулевое значение по умолчанию, то есть если функции еще не присвоен идентификатор.

В контекстное меню мы добавляем три новых пункта:

```
// Открытие контекстного меню блока
case RDS_BFM_CONTEXTPOPUP:
  // Добавление временных пунктов меню
  rdsAdditionalContextMenuItemEx(bypass?"Открыть":"Закрыть",
                                 0,1,2);
  rdsAdditionalContextMenuItemEx(NULL,RDS_MENU_DIVIDER,0,0);
  rdsAdditionalContextMenuItemEx("Открыть все",0,2,1);
  rdsAdditionalContextMenuItemEx("Закрыть все",0,2,0);
  rdsAdditionalContextMenuItemEx("Переключить все",0,2,2);
  break;
```

Первый из четырех добавленных вызовов `rdsAdditionalContextMenuItemEx`, с флагом `RDS_MENU_DIVIDER` и `NULL` вместо названия, добавляет в меню горизонтальный разделитель, чтобы визуально отделить старый пункт меню, меняющий режим этого блока, от новых, управляющих всеми блоками подсистемы. Три следующих вызова добавляют новые пункты меню, задавая для всех них номер функции меню 2. В качестве данных меню (второго целого числа, связанного с пунктом) используется номер команды, которую, как мы решили, мы будем давать блоку: 0 для закрытия, 1 для открытия и 2 для переключения.

Реакцию на выбор пункта меню необходимо изменить следующим образом:

```
// Выбор пункта меню пользователем
case RDS_BFM_MENUFUNCTION:
  menu=(RDS_PMENUFUNCDATA)ExtParam; // Данные пункта меню
  switch(menu->Function)
    { case 1:   // Открыть/закрыть
        bypass=!bypass;    // Переключить режим
        out=bypass?x_ext:x_int; // Подать на выход
        Ready=1;           // Взвести флаг готовности
        break;
      case 2:   // Один из новых пунктов
        // Заполняем структуру параметров функции
        callparams.Command=menu->MenuData;
        callparams.servSize=sizeof(callparams);
        // Вызываем у всех блоков родительской подсистемы
        rdsBroadcastFunctionCallsEx(BlockData->Parent,
                                    EditCtrlFuncSet,
                                    &callparams,
                                    0);
        break;
    }
  break;
```

Раньше выбранный пункт меню не анализировался: он был единственным, и проверять, какой именно пункт выбран пользователем, не имело смысла. Теперь у нас четыре дополнительных пункта меню – старый, как и раньше, имеет номер функции, равный единице, три новых – равный двум. Переданный в параметре `ExtParam` указатель на структуру данных выбранного пункта меню приводится к правильному типу и записывается во вспомогательную переменную `menu`, после чего оператором `switch` проверяется номер функции этого пункта `menu->Function`. Для номера 1 (старый пункт “Открыть/Закрыть”)

выполняются те же действия, что и в старой модели: состояние блока переключается, значение нужного входа подается на выход, взводится сигнал готовности. Для номера 2 (три новых пункта) данные меню `menu->Data` записываются в поле `Command` структуры параметров функции `callparams` – при добавлении пунктов в контекстное меню мы специально сделали так, чтобы данные пункта меню соответствовали команде, которую нужно передать блокам при его выборе. В поле `servSize` структуры параметров записывается ее размер, после чего вызывается сервисная функция РДС `rdsBroadcastFunctionCallsEx`, которая вызовет функцию, идентификатор которой хранится в глобальной переменной `EditCtrlFuncSet`, у всех блоков родительской для данного блока подсистемы (`BlockData->Parent`), передав им в качестве параметра функции указатель на структуру `callparams`. Нам не нужно вызывать функцию у блоков вложенных подсистем или останавливать дальнейшие вызовы по желанию одного из вызванных блоков, поэтому мы не указываем в вызове никаких флагов (последний параметр функции `rdsBroadcastFunctionCallsEx` равен нулю).

Осталось только добавить в `switch` реакцию на вызов функции, и модель готова:

```
// Вызов функции блока
case RDS_BFM_FUNCTIONCALL:
  // Приведение ExtParam к правильному типу
  func=(RDS_PFUNCTIONCALLDATA)ExtParam;
  // Проверяем, какая функция вызвана
  if(func->Function==EditCtrlFuncSet)
    { // Вызвана нужная нам функция
      TProgGuideEditCtrlSetParams *params;
      // Приводим указатель на параметры к нужному типу
      // и проверяем размер переданной структуры
      params=(TProgGuideEditCtrlSetParams*)(func->Data);
      if(params->servSize<
                sizeof(TProgGuideEditCtrlSetParams))
        break; // Размер недостаточен
      // В зависимости от Command меняем режим блока
      switch(params->Command)
        { case 0: bypass=1; break;         // Закрыть
          case 1: bypass=0; break;         // Открыть
          case 2: bypass=!bypass; break;   // Переключить
          default: return RDS_BFR_DONE;    // Ошибка
        }
      // Взводим сигнал готовности и передаем нужный вход
      // на выход
      Ready=1;
      out=bypass?x_ext:x_int;
    }
  break;
```

В этой реакции прежде всего проверяется, совпадает ли идентификатор вызванной функции с `EditCtrlFuncSet`. Если совпадает, переданный указатель на параметры функции приводится к типу “указатель на `TProgGuideEditCtrlSetParams`”, после чего значение поля `servSize` переданной структуры параметров сравнивается с размером типа `TProgGuideEditCtrlSetParams`. Если размер переданной структуры больше ожидаемого значения или равен ему, значит, блок может выполнять функцию – параметры, вероятнее всего, переданы верно. В зависимости от поля `Command` переданной структуры параметров переменной состояния блока `bypass` присваивается нужное значение, взводится сигнал готовности блока и значение выбранного согласно состоянию блока входа передается на его выход.

Рассмотрим работу измененной нами модели подробнее. Допустим, в какой-либо подсистеме есть несколько блоков с моделью `EditControlFrame`, управляющих какими-то полями ввода. Представим себе, что пользователь щелкнул на одном из них правой кнопкой мыши, и выбрал в контекстном меню пункт “Закрыть все”. Модель этого блока будет вызвана в режиме `RDS_BFM_MENUFUNCTION`, при этом в `ExtParam` ей будет передан указатель на структуру с данными выбранного пункта, в которой поле `Function` будет равно 2, а `MenuData` – 0. Реагируя на этот вызов, модель запишет в поле `Command` структуры `callparams` значение 0, взятое из `MenuData`, и запустит вызов функции с идентификатором `EditCtrlFuncSet` у всех блоков подсистемы – все они будут по очереди вызываться в режиме `RDS_BFM_FUNCTIONCALL`. Блоки с моделью `EditControlFrame`, включая и тот блок, контекстное меню которого открыл пользователь, среагируют на вызов функции, присвоив переменной `bypass` значение 1 (поскольку `Command==0`) и “закрыв” тем самым поле ввода. Модели же, в которых не предусмотрена реакция на эту функцию, не опознают переданный им идентификатор, и завершатся, ничего не выполнив. Таким образом выбор пункта меню “Закрыть все” в одном из блоков приведет к закрытию всех таких же блоков подсистемы.

В этом примере мы создали модель, которая сама вызывает функцию, и сама же на нее реагирует. Но удобство использования функций блоков в том, что в будущем мы можем создать другие модели, реагирующие на эту же функцию или вызывающие ее, и блоки с этими моделями смогут открываться, закрываться и переключаться вместе с нашими блоками, или, при необходимости, управлять нашими блоками через эту функцию.

§2.13.4. Пример использования функций блоков для поиска пути в графе

Рассматривается пример использования функций блоков для поиска в ориентированном графе, состоящем из блоков и соединяющих их связей, кратчайшего пути между двумя узлами. Используется вызов функций с параметрами и без них, вызов функции конкретного блока и всех блоков подсистемы.

До сих пор мы добавляли поддержку функций в уже созданные модели для расширения их функциональности. Теперь займемся созданием модели блока, который будет взаимодействовать с другими исключительно через вызов их функций. Мы будем решать задачу поиска кратчайшего пути в ориентированном графе: блоки будут узлами этого графа, а связи – его дугами.

Длиной дуги графа будем считать общую длину связи, к которой, поскольку блок в РДС – не точечный объект, добавлены расстояния между геометрическими центрами блоков и точками присоединения связи к этим блокам (рис. 87). Если блоки 1 и 2 соединены связью общей длины L_C, расстояние между геометрическим центром изображения первого блока и точкой присоединения этой связи к нему – L_1, и расстояние между центром второго блока и точкой присоединения связи – L_2, то длиной дуги, соединяющей узлы графа, представленные блоками 1 и 2, мы будем считать сумму этих трех величин, то есть $L_C + L_1 + L_2$. Конечно, можно было бы пренебречь расстояниями L_1 и L_2, сделав размеры блоков достаточный маленькими (существенно меньшими расстояний между ними), но, для большей универсальности примера, мы все-таки

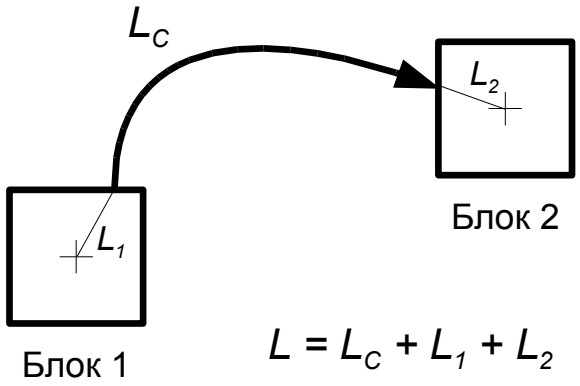


Рис. 87. Длина дуги графа в рассматриваемом примере

будем их учитывать. Вычисление длины дуги мы вынесем в отдельную функцию, чтобы ее, при необходимости, можно было изменить.

Движение по дуге графа будем считать возможным только в направлении стрелки связи. Если связь соединяет выход одного блока со входом другого, можно двигаться только от выхода ко входу: на рис. 87 возможен переход только из первого блока во второй, но не из второго в первый. Если связь соединяет между собой два входа блока, у нее будут изображаться стрелки на обоих концах. В этом случае будем считать, что движение по дуге возможно в обоих направлениях. В режиме расчета такая связь была бы абсолютно бессмысленной: она не присоединена к выходу блока, значение которого она могла бы передать на соединенные входы. Однако, в этом примере мы используем связи только как способ нарисовать дуги графа, и никакие данные по ним передаваться не будут, поэтому в нашем графе будут присутствовать как связи, соединяющие выход со входом, так и связи, соединяющие два входа. Разумеется, связь, изображающая дугу графа, не должна быть разветвленной: дуга может соединять только два узла графа.

Сформулируем задачу, которую мы собираемся решать. В какой-либо подсистеме (не важно – в корневой подсистеме схемы, или в одной из вложенных) собран граф, узлы которого представлены блоками, а дуги – связями между ними. Если пользователь укажет один из блоков как начало маршрута, а другой – как конец, на схеме должен отобразиться кратчайший маршрут между начальным и конечным блоками с учетом направления дуг графа. При этом все блоки и связи, входящие в этот маршрут, должны быть визуально выделены. Пользователь также должен иметь возможность сбросить выделенный маршрут и вернуть граф в исходное состояние.

Для решения этой задачи достаточно написать одну модель: модель блока-узла графа. Все действия по поиску маршрута, его визуальному выделению и взаимодействию с пользователем будут решать блоки-узлы, вызывая друг у друга различные функции. Разобьем задачу на несколько подзадач, и перечислим их в порядке возрастания сложности:

1. Необходимо дать пользователю возможность указывать блок начала маршрута, блок конца маршрута, а также сбрасывать выделенный маршрут. Будем использовать для этого контекстное меню блока.
2. Блоки, принадлежащие и не принадлежащие маршруту, должны выглядеть по-разному. Начальный и конечный блоки маршрута тоже должны иметь специальные отметки. Мы решим эту задачу двумя способами: во-первых, мы сделаем в модели программное рисование, которое будет все это учитывать, и, во-вторых, мы введем в блок статические логические переменные, которые будут отражать принадлежность блока к маршруту, его началу и концу. Если пользователю по какой-то причине не понравится программное рисование, он сможет задать блокам векторную картинку, связав ее элементы с этими переменными.
3. Необходимо изменять внешний вид связи, изображающей дугу графа на выделенном маршруте. При изменении или сбросе маршрута необходимо восстанавливать ее прежний вид. Для этого мы воспользуемся специальным механизмом РДС, позволяющим временно менять параметры внешнего вида связи.
4. Необходимо уметь вычислять длину дуги между двумя блоками. При этом надо учитывать, что связь в РДС может представлять собой набор последовательно соединенных отрезков прямых и кривых Безье. Для вычисления общей длины связи мы напишем специальную функцию.
5. Наконец, самое важное: если в схеме заданы и начальный, и конечный блок, необходимо найти кратчайший маршрут между ними. Мы будем использовать для этого алгоритм, похожий на алгоритм Дейкстры [12], но несколько упрощенный для большей наглядности примера. Упростив алгоритм, мы, вероятно, увеличим время, необходимое на поиск маршрута, но в данном примере это не особенно важно. Способ поиска маршрута, который мы будем использовать, описан ниже.

Для поиска маршрута каждый блок-узел будем помечать вещественным числом – кратчайшим расстоянием до точки начала маршрута (для этой метки мы предусмотрим в блоке специальную переменную). Следует также предусмотреть возможность “бесконечного расстояния”, то есть отсутствия в узле такой метки – например, если добраться до данного узла от начала маршрута невозможно. После того, как такие метки расставлены, легко отследить кратчайший маршрут между начальным и конечным блоком, если начать с конечного: среди блоков, от которых к нему идут дуги, нужно найти блок с наименьшей меткой (то есть блок, расстояние до которого от начала маршрута будет наименьшим), затем найти блок с наименьшей меткой среди соединенных с найденным и т.д., до тех пор, пока мы не доберемся до блока начала маршрута. Цепочка найденных блоков и будет кратчайшим маршрутом от начального блока к конечному, поскольку каждый раз среди соседей очередного блока цепочки мы выбирали ближайший к начальному. Осталось только разобраться, как правильно расставить метки в графе.

Введем для этого специальную операцию: “пометить блок числом *M*” (когда дело дойдет до программирования, мы реализуем ее как функцию блока). В зависимости от наличия и значения уже имеющейся в блоке метки *K*, будут выполняться следующие действия:

- В блоке нет метки. В этом случае блоку присваивается метка *M*, все блоки, в которые можно попасть из данного за один переход по связи, помечаются числом (*M + длина дуги к блоку-соседу*), то есть данная операция выполняется уже для них, но с числом, увеличившимся на расстояние до нового помечаемого блока.
- В блоке есть метка *K*, $K \le M$ (метка, уже имеющаяся в блоке, меньше или равна той, которой мы пытаемся его пометить). Это значит, что этот блок раньше уже помечался, причем тогда расстояние до начального блока было меньше. Такая ситуация возможна при наличии нескольких параллельных путей разной длины между начальным блоком и данным, причем данный блок уже получил метку по более короткому пути. В этом случае метка блока не изменяется, никаких дополнительных пометок блоков не производится.
- В блоке есть метка *K*, $K > M$ (метка, уже имеющаяся в блоке, больше той, которой мы пытаемся его пометить). Это значит, что мы только что нашли более короткий путь от начального блока к данному. В этом случае метка блока заменяется на *M* и все блоки, в которые можно попасть из данного за один переход по связи, помечаются числом (*M + длина дуги к блоку-соседу*).

Можно заметить, что в первом и третьем случае выполняются одни и те же действия (фактически, отсутствие метки в блоке можно считать наличием метки со значением “бесконечность”, то есть бесконечным расстоянием до начального блока маршрута). Поэтому случай отсутствия метки в блоке и случай наличия в блоке метки, большей, чем новое значение, можно обрабатывать одинаково.

Видно, что если сбросить все метки в графе, а затем пометить начальный блок маршрута числом 0, это вызовет цепную реакцию: соседи начального блока будут помечены длинами дуг к нему, что приведет к тому, что их соседи тоже будут помечены и т.д. Это будет продолжаться до тех пор, пока все блоки графа, достижимые из начального, не будут помечены. При этом помечены они будут именно кратчайшим расстоянием до начального, поскольку при обнаружении в блоке метки, большей чем та, которую мы пытаемся ему присвоить, мы заменяем ее на меньшую и помечаем его соседей заново. В результате мы получим желаемую разметку графа, по которой сможем отследить кратчайший маршрут от начального блока к конечному.

Теперь можно приступить к написанию вспомогательных функций, которые потребуются нам для создания модели. Прежде всего, поскольку связь между блоками может содержать не только отрезки прямых, длина которых вычисляется очевидным образом, но и отрезки кривых Безье, нам потребуется функция, вычисляющая длину кривой Безье по координатам четырех задающих ее точек (рис. 88). К сожалению, аналитической формулы

для длины кривой Безье не существует, поэтому мы будем вычислять ее приближенно, при помощи численного интегрирования. Для тех, кого интересует математика, стоящая за вычислением длины кривой Безье, приведем несколько формул, все остальные могут пропустить следующие несколько абзацев.

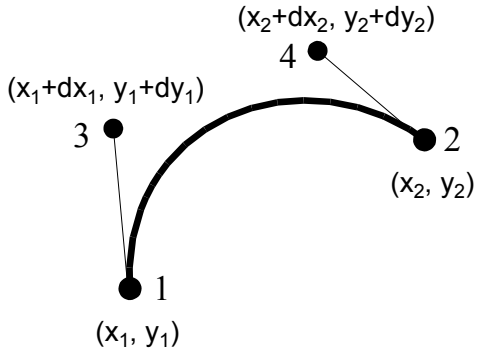


Рис. 88. Координаты кривой Безье

Кривая Безье параметрически задается следующими формулами:

$$\begin{cases} x(t)=a_x t^3+b_x t^2+c_x t+x_1 \\ y(t)=a_y t^3+b_y t^2+c_y t+y_1 \end{cases} \quad t\in[0...1]$$

где t – параметр кривой.

Коэффициенты a, b и c вычисляются по координатам точек рис. 88 следующим образом:

$$a_x=2(x_1-x_2)+3(dx_1-dx_2)$$
$$b_x=3(x_2-x_1+dx_2-2\,dx_1)$$
$$c_x=3\,dx_1$$
$$a_y=2(y_1-y_2)+3(dy_1-dy_2)$$
$$b_y=3(y_2-y_1+dy_2-2\,dy_1)$$
$$c_y=3\,dy_1$$

Длина любой плоской кривой, заданной функциями $x(t)$ и $y(t)$ в диапазоне $0 \le t \le 1$ вычисляется так:

$$L=\int_0^1 \sqrt{\left(\frac{dx}{dt}\right)^2+\left(\frac{dy}{dt}\right)^2}\,dt$$

Для кривой Безье:

$$\begin{cases} \frac{dx}{dt}=3a_x t^2+2b_x t+c_x \\ \frac{dy}{dt}=3a_y t^2+2b_y t+c_y \end{cases} \quad \rightarrow \quad L=\int_0^1 \sqrt{(3a_x t^2+2b_x t+c_x)^2+(3a_y t^2+2b_y t+c_y)^2}\,dt$$

Этот интеграл нам и предстоит вычислить, и вычислять его мы будем методом Гаусса [13, 14]. Реализуя этот алгоритм, получим функцию для вычисления длины кривой Безье с заданной точностью:

```
// Вычисление длины кривой Безье численным интегрированием
// (метод Гаусса)
double BezierLengthGauss(double x1,double y1,double dx1,double dy1,
           double x2,double y2,double dx2,double dy2,double delta)
{ double ax,bx,cx,ay,by,cy;
  double a,b,c,d,e,q,n,k,T,l,h,I,w;
  int m;
  double x[3],s[3];

  // Вычисление коэффициентов параметрического вида
  ax=2*(x1-x2)+3*(dx1-dx2);
  bx=3*(x2-x1+dx2-2*dx1);
  cx=3*dx1;
  ay=2*(y1-y2)+3*(dy1-dy2);
  by=3*(y2-y1+dy2-2*dy1);
  cy=3*dy1;
```

```
  // Интегрирование
  b=1.0;
  n=delta*60.0; m=1; k=0.0;
  do
    { m*=2; a=0.0; T=sqrt(0.6); I=0.0;
      h=(b-a)/m;
      for(int j=1;j<=m;j++)
        { w=a+h;
          c=(w+a)/2; d=(w-a)/2; e=d*5.0/9.0;
          l=d*8.0/9.0; d*=T;
          x[0]=c-d; x[1]=c; x[2]=c+d;
          s[0]=e;   s[1]=l; s[2]=e;
          for(int i=0;i<3;i++)
            { // Вычисление подынтегральной функции (3 раза)
              double vx=3*ax*x[i]*x[i]+2*bx*x[i]+cx,
                     vy=3*ay*x[i]*x[i]+2*by*x[i]+cy;
              double f=sqrt(vx*vx+vy*vy);
              I+=s[i]*f;
            }
          a=w;
        }
      l=k; k=I;
    }
  while(fabs(I-l)>n);
  return I;
}
//=======================================
```

Первые восемь параметров этой функции – координаты точек кривой (см. рис. 88), девятый – требуемая точность вычисления.

Для вычисления длины дуги графа нам также потребуется функция, вычисляющая расстояние между геометрическим центром блока и точкой присоединения связи к нему (L_1 и L_2 на рис. 87). Эта функция будет принимать один параметр: указатель на структуру описания точки связи `RDS_POINTDESCRIPTION`. Мы уже встречались с этой структурой, когда проверяли существование связи, соединенной с конкретной переменной блока (см. стр. 135), она используется во всех сервисных функциях, работающих со связями и их точками. Полное описание этой структуры приведено в приложении А. В ней находятся все данные, которые необходимы нам для вычисления расстояния: идентификатор блока, к которому присоединена связь (поле `Block`) и координаты самой точки (поля `x` и `y`). Следует помнить, что координаты точки присоединения связи к блоку всегда даются относительно точки привязки изображения этого блока (для программно рисуемых блоков и блоков без векторной картинки – относительно верхнего левого угла, для блоков с картинкой – относительно начала координат векторного изображения), и это нужно учитывать при вычислении расстояния от этой точки до центра блока. Функция, вычисляющая это расстояние, будет выглядеть так:

```
// Определение расстояния между точкой связи и геометрическим
// центром блока по структуре RDS_POINTDESCRIPTION
double DistanceFromBlockCenterToPoint(RDS_PPOINTDESCRIPTION point)
{ RDS_BLOCKDIMENSIONS dim; // Структура описания размеров блока
  double dx,dy,xc,yc,xp,yp;

  if(point==NULL)     // Указатель не передан - ошибка
    return -1.0;
  // Проверка - точка ли соединения с блоком передана?
  if(point->PointType!=RDS_PTBLOCK)
    return -1.0;
```

```
  // Определение размеров блока point->Block
  dim.servSize=sizeof(dim);      // Размер структуры
  if(!rdsGetBlockDimensionsEx(point->Block,&dim,RDS_GBD_NONE))
    return -1.0;      // Не удалось получить размеры блока

  // Геометрический центр изображения блока
  xc=dim.Left+dim.Width/2.0;
  yc=dim.Top+dim.Height/2.0;
  // Абсолютные координаты точки связи
  xp=dim.BlockX+point->x;
  yp=dim.BlockY+point->y;
  // Вычисление расстояния между этими точками
  dx=xp-xc; dy=yp-yc;
  return sqrt(dx*dx+dy*dy);
}
//=======================================
```

Прежде всего, в этой функции проверяется правильность переданного параметра: указатель на структуру описания точки связи не должен быть равен NULL, и точка связи, описываемая структурой, должна быть именно точкой соединения связи с блоком (поле PointType должно иметь значение RDS_PTBLOCK). Если одно из этих условий не выполнено, функция возвращает отрицательное значение, сигнализирующее об ошибке.

Далее структура dim типа RDS_BLOCKDIMENSIONS заполняется данными о размерах и положении изображения блока, с которым соединена точка связи. Для этого вызывается сервисная функция РДС rdsGetBlockDimensionsEx, в которую передается идентификатор интересующего нас блока point->Block, указатель на заполняемую структуру &dim и константа RDS_GBD_NONE, указывающая, что при определении размеров и положения блока не нужно учитывать ни масштаб подсистемы, ни возможную связь этих параметров с переменными блока. Таким образом, в структуру dim будут записаны размеры блока в режиме редактирования в масштабе 100%.

Структура RDS_BLOCKDIMENSIONS описана в “RdsDef.h” следующим образом:

```
typedef struct
{ DWORD servSize;     // Размер этой структуры
  int BlockX,BlockY;  // Точка привязки блока
  int Left,Top;       // Верхний левый угол блока
  int Width,Height;   // Ширина и высота блока
} RDS_BLOCKDIMENSIONS;
typedef RDS_BLOCKDIMENSIONS *RDS_PBLOCKDIMENSIONS;
```

Перед вызовом функции в поле servSize (первое поле структуры, как и в большинстве структур, используемых сервисными функциями РДС) необходимо занести размер этой структуры для проверки правильности переданных параметров. После заполнения этой структуры функцией rdsGetBlockDimensionsEx мы вычисляем координаты центра изображения блока (xc, yc) и абсолютные координаты точки связи (xp, yp). Центр блока – это верхний левый угол изображения, смещенный на половину его ширины и высоты, а абсолютные координаты точки связи вычисляются добавлением к точке привязки блока (dim.BlockX, dim.BlockY) относительных координат точки связи (point->x, point->y). Далее вычисляется расстояние между этими точками, и функция возвращает полученное значение.

Теперь, когда мы можем вычислять длину кривой Безье и расстояние от центра блока до точки связи, мы можем написать функцию, которая будет вычислять длину дуги графа, представленную какой-либо связью между блоками. Эта функция будет одновременно решать две задачи: во-первых, она будет вычислять длину дуги, и, во-вторых, она будет проверять, соединяет ли эта связь ровно два блока. Если связь оборвана, она будет соединена

только с одним блоком. Если связь разветвлена, она будет соединять три и более блоков. В обоих случаях, такая связь не может быть дугой графа, и наша функция вместо длины дуги будет возвращать отрицательное значение, сигнализирующее об ошибке. Связь в РДС состоит из одного или нескольких участков, каждый из которых может быть либо отрезком прямой, либо кривой Безье. Если связь не разветвлена, для вычисления длины дуги графа нам необходимо сложить длины всех участков связи и добавить к ним расстояния от центра блока до точки связи для обоих соединенных блоков. Функция будет выглядеть следующим образом:

```
// Проверка связи (должно быть два блока на концах) и вычисление
// длины дуги графа, соответствующей этой связи
double CalcArcLength(RDS_CHANDLE Conn)
{ RDS_CONNDESCRIPTION ConnDescr;      // Структура описания связи
  RDS_POINTDESCRIPTION PointDescr;    // Структура описания точки
  RDS_LINEDESCRIPTION LineDescr;      // Структура описания линии
  int BlockCnt;
  double len=0.0;     // Общая длина дуги
  double x1,y1,x2,y2;

  // Заполнение служебных полей размеров структур
  ConnDescr.servSize=sizeof(ConnDescr);
  PointDescr.servSize=sizeof(PointDescr);
  LineDescr.servSize=sizeof(LineDescr);

  // Получаем описание связи - нам нужно число точек и линий в ней
  if(!rdsGetConnDescription(Conn,&ConnDescr))
    return -1.0;

  if(ConnDescr.ConnType!=RDS_CTCONNECTION)
    return -1.0; // Это не связь, а шина - шины нам не годятся

  // Проверяем число блоков на концах связи (должно быть ровно 2)
  BlockCnt=0;
  for(int i=0;i<ConnDescr.NumPoints;i++)
    { // Получаем описание точки связи i
      rdsGetPointDescription(Conn,i,&PointDescr);
      // Проверяем тип точки
      switch(PointDescr.PointType)
        { case RDS_PTBUS: // Соединение с шиной - связь не годится
            return -1.0;
          case RDS_PTBLOCK: // Соединение с блоком
            BlockCnt++;
            if(BlockCnt>2) // Связь разветвлена
              return -1.0;
            // Найдена точка соединения с блоком. Добавляем к len
            // расстояние между точкой и центром блока
            len+=DistanceFromBlockCenterToPoint(&PointDescr);
            break;
        } // switch(PointDescr.PointType)
    } // for(int i=0;...)
  if(BlockCnt!=2) // Связь соединяет менее двух блоков
   return -1.0;

  // Связь соединена ровно с двумя блоками - суммируем длину всех
  // ее линий
  for(int i=0;i<ConnDescr.NumLines;i++)
```

```
    { // Получаем описание линии связи i
      rdsGetLineDescription(Conn,i,&LineDescr,NULL,NULL);
      // Переводим целые координаты концов линии в double
      // для большей точности вычисления
      x1=LineDescr.x1; y1=LineDescr.y1;
      x2=LineDescr.x2; y2=LineDescr.y2;
      // В зависимости от типа линии, вычисляем ее длину и
      // добавляем к len
      switch(LineDescr.LineType)
        { case RDS_LNLINE: // Отрезок прямой
            len+=sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
            break;
          case RDS_LNBEZIER: // Кривая Безье
            len+=BezierLengthGauss(x1,y1,
                     LineDescr.dx1,LineDescr.dy1,
                     x2,y2,
                     LineDescr.dx2,LineDescr.dy2,
                     2);
            break;
        } // switch(LineDescr.LineType)
    } // for(int i=0;...)
  return len;
}
//=======================================
```

В этой функции мы перебираем все точки связи, идентификатор которой передан в параметре `Conn`, чтобы найти среди них точки соединения с блоками и подсчитать их. Кроме того, мы перебираем все отрезки линий связи и суммируем их длину. Для того, чтобы организовать циклы по всем точкам и всем линиям, нам нужно узнать общее число точек и линий в этой связи. Для этого мы вызываем сервисную функцию `rdsGetConnDescription`, которая заполняет структуру описания связи `ConnDescr` типа `RDS_CONNDESCRIPTION`. В этой структуре много полей, но нас будет интересовать поле типа связи `ConnType`, число точек в ней `NumPoints` и число отрезков `NumLines`. Поле `ConnType` нам нужно только для того, чтобы проверить, не передан ли случайно в нашу функцию идентификатор шины вместо идентификатора связи (в РДС шина является разновидностью связи). Шина, очевидно, не может изображать дугу графа, поэтому если тип связи не равен `RDS_CTCONNECTION`, функция возвращает отрицательное значение, сигнализирующее об ошибке.

Далее мы в цикле перебираем все точки исследуемой связи, подсчитывая в процессе перебора количество соединяемых связью блоков. Внутри цикла мы заполняем структуру `PointDescr` описанием очередной (`i`-й) точки связи `Conn` при помощи функции `rdsGetPointDescription`, после чего при помощи оператора `switch(PointDescr.PointType)` анализируем тип этой точки. Если это точка соединения связи с шиной (поле `PointDescr.PointType` имеет значение `RDS_PTBUS`), функция немедленно возвращает отрицательное значение: связь, соединенная с шиной, не может изображать дугу графа. Если это точка соединения с блоком (тип точки – `RDS_PTBLOCK`), мы увеличиваем на единицу предварительно обнуленную целую переменную `BlockCnt`, в которой подсчитывается число соединенных блоков. Если значение этой переменной превысило 2, значит, связь соединяет три или более блоков, и она не может быть дугой графа – дальнейший перебор точек не имеет смысла, и функция немедленно возвращает отрицательное значение. В противном случае значение расстояния от данной точки связи до центра соединенного с ней блока добавляется к переменной `len`, для этого вызывается ранее написанная функция `DistanceFromBlockCenterToPoint`, в которую передается указатель на заполненную структуру описания точки.

Промежуточные точки связи, имеющие тип RDS_PTINTERNAL, нас не интересуют, поэтому в операторе switch для этого типа нет соответствующего case – при обнаружении таких точек мы их просто пропускаем.

После перебора всех точек связи в переменной BlockCnt будет находиться число блоков, с которыми соединена данная связь, а в переменной len – сумма расстояний от центра блока до точки связи для всех соединенных блоков (L_1+L_2 на рис. 87). Случай BlockCnt>2 мы уже обработали внутри цикла, но связь может оказаться оборванной, то есть соединенной только с одним блоком, или не соединенной ни с одним. В этом случае она тоже не может служить дугой графа, поэтому, если BlockCnt не равно двум, функция возвращает отрицательное значение.

Теперь, когда мы убедились, что связь соединяет ровно два блока, можно добавить к len сумму длин всех ее отрезков, что даст нам длину дуги графа. Для этого мы в цикле перебираем все линии связи. Внутри цикла мы заполняем структуру LineDescr типа RDS_LINEDESCRIPTION описанием i-й линии связи Conn при помощи сервисной функции rdsGetLineDescription. В двух последних параметрах этой функции можно было бы передать указатели на структуры описания точек связи, которые эта линия соединяет, чтобы rdsGetLineDescription их тоже заполнила, но нам эти описания не требуются, поэтому вместо обоих указателей передается значение NULL. Нам нужно только описание самой линии, чтобы вычислить ее длину и добавить к переменной len. Структура RDS_LINEDESCRIPTION описана в “RdsDef.h” следующим образом:

```
typedef struct
{ DWORD servSize;     // Размер этой структуры
  int LineType;       // Тип линии (RDS_LN*)
  int nPoint1,nPoint2;// Номера соединенных ей точек
  int x1,y1;          // Абсолютные координаты точки nPoint1
  int x2,y2;          // Абсолютные координаты точки nPoint2
  int dx1,dy1;// Смещения управляющей точки 1 для кривой Безье
  int dx2,dy2;// Смещения управляющей точки 2 для кривой Безье
  RDS_CHANDLE Owner;  // Связь-владелец линии
} RDS_LINEDESCRIPTION;
typedef RDS_LINEDESCRIPTION *RDS_PLINEDESCRIPTION;
```

Нас в этой структуре интересует поле LineType, определяющее тип отрезка (RDS_LNLINE – отрезок прямой, RDS_LNBEZIER – кривая Безье), координаты его конечных точек (x1, y1) и (x2, y2), а также, в случае кривой Безье, смещения управляющих точек (dx1, dy1) и (dx2, dy2). В зависимости от типа линии мы добавляем к переменной len либо длину отрезка прямой, либо длину кривой Безье, для вычисления которой мы вызываем ранее написанную функцию BezierLengthGauss, задав точность вычисления (последний параметр функции) в две точки экрана.

После завершения цикла по всем линиям связи в переменной len окажется длина дуги графа, которую функция и возвращает.

Кроме вычисления длины дуги нам нужно иметь возможность визуально выделять связи, показывая пользователю найденный кратчайший путь между узлами графа. Мы будем выделять связь, увеличивая ее толщину. В РДС есть сервисная функция rdsSetConnAppearance, позволяющая изменить параметры внешнего вида любой связи, но для наших целей более удобен механизм создания одного или нескольких альтернативных наборов параметров внешнего вида связи, позволяющий оперативно переключаться между ними. У этого механизма есть два преимущества, которые будут нам особенно полезны. Во-первых, он запоминает исходный внешний вид связи и позволяет вернуться к нему одной командой – это будет необходимо нам при снятии визуального выделения связи. Если бы мы пользовались функцией rdsSetConnAppearance, нам бы пришлось где-нибудь запоминать исходную толщину каждой связи перед ее выделением, а

затем, при снятии выделения, восстанавливать запомненное значение. Во-вторых, при сохранении схемы альтернативные наборы параметров связи не запоминаются, поэтому нам можно не опасаться того, что пользователь сохранит схему в тот момент, когда в графе будет выделен маршрут, и измененные толщины линий связей запомнятся как исходные.

Сначала напишем функцию, которая будет выделять связь, идентификатор которой передан в ее параметре:

```
// Визуально выделить связь
void MarkConnection(RDS_CHANDLE Conn)
{ // Определяем число альтернативных внешних видов
  int StylesCount=rdsAltConnAppearanceOp(Conn,RDS_CAOCOUNT,0,NULL);

  if(StylesCount<1)
    { // Для связи еще не определено ни одного внешнего вида
      // Создаем внешний вид 0 и делаем его толще текущего
      RDS_CONNAPPEARANCE style; // Структура описания стиля связи
      // Получаем описание текущего внешнего вида связи
      style.servSize=sizeof(style);
      rdsGetConnAppearance(Conn,&style);
      // Увеличиваем толщину и размер стрелки
      style.LineWidth*=3;
      style.ArrowLength*=2;
      style.ArrowWidth*=3;
      // Запоминаем эти параметры как альтернативный вид 0
      rdsAltConnAppearanceOp(Conn,RDS_CAOSET,0,&style);
    }
  // Устанавливаем альтернативный вид 0
  rdsAltConnAppearanceOp(Conn,RDS_CAOSETCURRENT,0,NULL);
}
//=======================================
```

Все управление альтернативными параметрами внешнего вида связи производится при помощи функции `rdsAltConnAppearanceOp`. Функция принимает четыре параметра: идентификатор связи, одна из констант `RDS_CAO*`, обозначающая выполняемую операцию, номер набора параметров, а также указатель на структуру описания внешнего вида связи типа `RDS_CONNAPPEARANCE` (два последних параметра не используются в некоторых операциях). Обычно используется шесть основных операций:

- `rdsAltConnAppearanceOp(Conn, RDS_CAOSET, n, &style)` – установить параметры из структуры `style` в качестве альтернативного внешнего вида номер `n` для связи `Conn`;
- `rdsAltConnAppearanceOp(Conn, RDS_CAOCOUNT, 0, NULL)` – вернуть число альтернативных внешних видов, созданных для связи `Conn`;
- `rdsAltConnAppearanceOp(Conn, RDS_CAOGET, n, &style)` – считать параметры альтернативного вида номер `n` связи `Conn` в структуру `style`;
- `rdsAltConnAppearanceOp(Conn, RDS_CAOSETCURRENT, n, NULL)` – активировать альтернативный внешний вид номер `n` в связи `Conn` (связь будет выглядеть согласно параметрам, запомненным для этого внешнего вида);
- `rdsAltConnAppearanceOp(Conn, RDS_CAODELETE, n, NULL)` – удалить альтернативный внешний вид номер `n` в связи `Conn` (все наборы с номерами, большими `n`, сдвинутся вниз на 1);
- `rdsAltConnAppearanceOp(Conn, RDS_CAORESTORE, 0, NULL)` – восстановить исходный внешний вид связи `Conn`.

В функции `MarkConnection` мы, прежде всего, определяем число уже существующих в связи `Conn` альтернативных наборов параметров внешнего вида и записываем его в переменную `StylesCount`. Если это значение больше или равно 1, значит, мы уже создали

для этой связи альтернативный внешний вид, нужно только активировать его. В противном случае параметры текущего, то есть заданного пользователем при создании связи, внешнего вида считываются функцией rdsGetConnAppearance в структуру style типа RDS_CONNAPPEARANCE. Эта структура описана в "RdsDef.h" следующим образом:

```
typedef struct
{ DWORD servSize;     // Размер этой структуры
  COLORREF LineColor; // Цвет линии
  int LineWidth;      // Толщна линии
  int LineStyle;      // Стиль линии (константа WinAPI)
  int ArrowLength;    // Длина стрелки (0...255)
  int ArrowWidth;     // Ширина выступа стрелки (0...255)
  int DotSize;        // Размер точки в месте ветвления
} RDS_CONNAPPEARANCE;
typedef RDS_CONNAPPEARANCE *RDS_PCONNAPPEARANCE;
```

Для визуального выделения связи мы увеличиваем толщину линии и ширину стрелки в 3 раза, а длину стрелки – в 2 раза. Затем, при помощи вызова rdsAltConnAppearanceOp с константой RDS_CAOSET, мы создаем в связи альтернативный внешний вид номер 0 с параметрами из структуры style.

В самом конце функции вызовом rdsAltConnAppearanceOp с константой RDS_CAOSETCURRENT в связи Conn включается альтернативный внешний вид номер 0, созданный при самом первом вызове функции MarkConnection для данной связи.

Для снятия визуального выделения связи достаточно восстановить ее исходный внешний вид, который она имела до включения альтернативного. Для лучшей читаемости примера мы оформим это действие в виде отдельной функции:

```
// Снять выделение связи
void UnmarkConnection(RDS_CHANDLE Conn)
{ // Восстанавливаем исходный внешний вид связи
  rdsAltConnAppearanceOp(Conn,RDS_CAORESTORE,0,NULL);
}
//========================================
```

Теперь, когда вспомогательные функции написаны, можно приступать к проектированию функций блоков, с помощью которых узлы графа будут обмениваться друг с другом информацией. Мы уже решили, что в каждом блоке-узле может быть флаг начала или конца маршрута (по одному флагу каждого типа на весь граф), вещественное число-метка, показывающая кратчайшее расстояние от данного узла до начала маршрута, а также признак наличия этой метки, поскольку путь к некоторым узлам графа может и не существовать. Прежде всего нам потребуется функция, с помощью которой можно было бы сбросить флаг начала, флаг конца или вещественную метку блока, чтобы пользователь мог сбрасывать маршрут в графе. Назовем эту функцию "ProgrammersGuide.GraphPath.Reset" и введем для нее следующие описания:

```
// Функция сброса параметров в узле графа
#define PROGGUIDEGRAPHPATHFUNC_RESET \
                      "ProgrammersGuide.GraphPath.Reset"
// Структура парметров функции
typedef struct
{ DWORD servSize;     // Размер этой структуры
  BOOL ResetMark;     // TRUE - сбросить метку узла
  BOOL ResetBegin;    // TRUE - сбросить флаг начала маршрута
  BOOL ResetEnd;      // TRUE - сбросить флаг конца маршрута
} TProgGuideFuncResetParams;
```

Параметром этой функции будет структура TProgGuideFuncResetParams, содержащая, помимо поля размера для проверки правильности передачи параметров, три логических поля, которые указывают, какие именно параметры блока нужно сбросить.

Нам нужно уметь не только сбрасывать параметры узла, но и считывать их. Создадим для этого функцию “ProgrammersGuide.GraphPath.GetParams” со следующими описаниями:

```
// Функция получения параметров узла графа
#define PROGGUIDEGRAPHPATHFUNC_GETPARAMS \
                      "ProgrammersGuide.GraphPath.GetParams"
// Среагировавший на функцию блок должен вернуть значение 1
// Структура парметров функции
typedef struct
{ DWORD servSize;     // Размер этой структуры
  BOOL Marked;        // В узле есть метка
  double Mark;        // Значение метки
  BOOL Begin;         // Этот узел – начало маршрута
  BOOL End;           // Этот узел – конец маршрута
} TProgGuideFuncGetParams;
```

Структура параметров этой функции содержит поля, в которые реагирующий на вызов блок запишет свои параметры. Кроме того, эту же функцию мы будем использовать для того, чтобы выяснить, является ли данный блок узлом графа. Для этого потребуем, чтобы среагировавший на функцию блок-узел графа возвращал значение 1. Функции моделей блоков, в которых не предусмотрена реакция на эту функцию, вернут стандартное значение `RDS_BFR_DONE`, то есть 0 – так мы сможем отличить узлы графа от посторонних блоков. При этом будем считать допустимым и вызов этой функции без параметров, то есть с указателем на параметры, равным `NULL` – если параметры узла графа нам не нужны, а нужно просто выяснить, является ли блок узлом графа, будем вызывать функцию без параметров и анализировать возвращенное значение.

Для того, чтобы можно было легко найти идентификаторы блоков, являющихся началом и концом маршрута, введем специальную функцию поиска начала и конца “ProgrammersGuide.GraphPath.Find”:

```
// Функция поиска начала и конца маршрута
// (вызывается у всех блоков подсистемы)
#define PROGGUIDEGRAPHPATHFUNC_FIND \
                      "ProgrammersGuide.GraphPath.Find"
// Структура параметров функции
typedef struct
{ DWORD servSize;          // Размер этой структуры
  RDS_BHANDLE BeginBlock;  // Найденный идентификатор начала
  RDS_BHANDLE EndBlock;    // Найденный идентификатор конца
} TProgGuideFuncFindParams;
```

Работать эта функция будет следующим образом. Чтобы найти идентификаторы крайних блоков маршрута, необходимо, как обычно, присвоить полю `servSize` структуры `TProgGuideFuncFindParams` размер этой структуры, а в поля `BeginBlock` и `EndBlock` записать значение `NULL`. После этого следует вызвать данную функцию для всех блоков подсистемы, в которой собран граф, при помощи `rdsBroadcastFunctionCallsEx`. Реагируя на вызов этой функции, блок-узел графа должен записать свой идентификатор в поле `BeginBlock`, если он является началом маршрута, и в поле `EndBlock`, если он является концом. Таким образом, после завершения `rdsBroadcastFunctionCallsEx` в поле `BeginBlock` окажется идентификатор начала маршрута, а в поле `EndBlock` – идентификатор конца. Если начало или конец маршрута в графе не заданы, соответствующее поле останется равным `NULL`.

Конечно, можно было бы найти начало и конец маршрута и без этой функции – например, можно просто перебрать все блоки подсистемы, вызывая у каждого “ProgrammersGuide.GraphPath.GetParams” и запоминая идентификаторы блоков, вернувших `TRUE` в полях `Begin` и `End` структуры `TProgGuideFuncGetParams`. Однако, такой способ был бы более громоздким, чем использование специальной функции.

Для пометки узла графа вещественным числом (см. стр. 342) нам тоже потребуется функция:

```
// Пометить узел графа указанным вещественным числом и вызвать
// эту же функцию у его соседей
#define PROGGUIDEGRAPHPATHFUNC_MARK \
                        "ProgrammersGuide.GraphPath.Mark"
// Структура параметров функции
typedef struct
{ DWORD servSize;     // Размер этой структуры
  double Mark;        // Устанавливаемое значение метки
  RDS_BHANDLE Previous;// Блок, от которого пришла метка
} TProgGuideFuncMarkParams;
```

Блок, у которого вызвана эта функция, должен, согласно алгоритму разметки, либо принять новое значение метки `Mark` и пометить всех своих соседей суммой значения метки и длины дуги к соседу, вызвав у каждого из них эту же функцию, либо оставить старое значение метки и не предпринимать никаких дополнительных действий. В поле `Previous` структуры параметров передается идентификатор соседнего блока, пометка которого привела к вызову этой функции у данного блока. Этот идентификатор нужен только для того, чтобы исключить этот блок из перебираемых соседей для ускорения работы функции. Если его не передавать, алгоритм разметки все равно будет работать, поскольку у предыдущего по разметке блока значение метки заведомо меньше, чем у данного, и он в любом случае не будет помечен новым, большим значением.

Наконец, после разметки графа нам потребуется функция, которая выделит кратчайший маршрут между началом и концом. Поскольку она будет отслеживать маршрут от конечного блока к начальному, назовем ее “ProgrammersGuide.GraphPath.BackTrace”:

```
// Выделить маршрут от данного блока к началу
#define PROGGUIDEGRAPHPATHFUNC_BACKTRACE \
                        "ProgrammersGuide.GraphPath.BackTrace"
```

Параметров у этой функции не будет. Работать она будет так: при ее вызове у какого-либо блока этот блок должен изменить свой внешний вид и, если он не является началом маршрута, найти среди соседей, из которых можно в него попасть, блок с наименьшей меткой, выделить связь, ведущую к этому соседу, и вызвать у него эту же самую функцию. Таким образом, вызов этой функции у конечного блока при размеченном графе приведет к выделению всего маршрута.

Если бы мы собирались распространять DLL с моделью блока-узла графа, все описания функций, приведенные выше, имело бы смысл выделить в отдельный заголовочный файл и распространять его вместе с библиотекой. Это дало бы возможность другим разработчикам разобраться в созданных нами функциях и использовать их для взаимодействия с нашей моделью. Но, поскольку модель создается только как пример использования функций блоков, можно разместить эти описания и в основном файле исходного текста.

Для каждой из созданных функций нам потребуется глобальная целая переменная, в которой будет храниться идентификатор, данный этой функции при регистрации:

```
int   GraphFuncFind=0,      //ProgrammersGuide.GraphPath.Find
      GraphFuncGetParams=0, //ProgrammersGuide.GraphPath.GetParams
      GraphFuncReset=0,     //ProgrammersGuide.GraphPath.Reset
      GraphFuncMark=0,      //ProgrammersGuide.GraphPath.Mark
      GraphFuncBackTrace=0; //ProgrammersGuide.GraphPath.BackTrace
```

Регистрировать функции мы будем в момент инициализации модели блока.

Чтобы несколько улучшить читаемость нашей программы, напишем для созданных функций блоков функции-оболочки. Начнем с функции для сброса в заданной подсистеме различных параметров маршрута:

```
// Сбросить у узлов графа в заданной подсистеме заданные маркеры
void GraphPath_Reset(RDS_BHANDLE Sys, // Подсистема
                     BOOL mark,       // Сбросить метки узлов
                     BOOL begin,      // Сбросить начало маршрута
                     BOOL end)        // Сбросить конец маршрута
{ // Структура параметров функции ProgrammersGuide.GraphPath.Reset
  TProgGuideFuncResetParams params;
  // Заполняем поле размера структуры парааметров
  params.servSize=sizeof(params);
  // Заполняем поля структуры параметров
  params.ResetMark=mark;
  params.ResetBegin=begin;
  params.ResetEnd=end;
  // Вызываем ProgrammersGuide.GraphPath.Reset у всех
  // блоков подсистемы Sys
  rdsBroadcastFunctionCallsEx(Sys,GraphFuncReset,&params,0);
}
//=======================================
```

Эта функция принимает четыре параметра: идентификатор подсистемы, в которой находится граф, параметры которого сбрасываются, и три логических значения, указывающих, какие именно параметры графа сбрасываются. Внутри функции заполняется структура `TProgGuideFuncResetParams` и для всех блоков указанной подсистемы вызывается функция “ProgrammersGuide.GraphPath.Reset”, идентификатор которой находится в переменной `GraphFuncReset` (мы считаем, что функция уже зарегистрирована).

Напишем такую же функцию-оболочку для поиска блоков начала и конца маршрута:

```
// Найти блоки начала и конца маршрута в заданной подсистеме
BOOL GraphPath_GetTerminalBlocks(
     RDS_BHANDLE Sys,      // Подсистема с графом
     RDS_BHANDLE *pBegin,  // Возвращаемый идентификатор начала
     RDS_BHANDLE *pEnd)    // Возвращаемый идентификатор конца
{ TProgGuideFuncFindParams params;
  // Заполняем поле размера структуры паараметров
  params.servSize=sizeof(params);
  // Обнуляем поля идентификаторов начала и конца
  params.BeginBlock=params.EndBlock=NULL;
  // Вызываем ProgrammersGuide.GraphPath.Find у всех блоков в
  // подсистеме Sys, разрешая блокам остановить вызовы
  rdsBroadcastFunctionCallsEx(Sys,GraphFuncFind,&params,
                RDS_BCALL_ALLOWSTOP);
  // Копируем найденные идентификаторы в переданные указатели
  if(pBegin) *pBegin=params.BeginBlock;
  if(pEnd) *pEnd=params.EndBlock;
  // Возвращаем TRUE, если установлены и начало, и конец
  return params.BeginBlock!=NULL && params.EndBlock!=NULL;
}
//=======================================
```

В эту функцию передается три параметра – идентификатор подсистемы с графом `Sys` и два указателя `pBegin` и `pEnd`, по которым возвращаются найденные начало и конец маршрута соответственно. Внутри функции обнуляются поля `BeginBlock` и `EndBlock` структуры `TProgGuideFuncFindParams`, после чего она используется как параметр при вызове функции “ProgrammersGuide.GraphPath.Find” у всех блоков подсистемы `Sys`. В сервисную функцию `rdsBroadcastFunctionCallsEx` передается флаг `RDS_BCALL_ALLOWSTOP`, что позволяет любому из вызываемых блоков досрочно прекратить вызовы. Этим можно воспользоваться для ускорения работы функции: если какой-либо блок обнаружит, что и поле `BeginBlock`, и поле `EndBlock` структуры параметров функции уже содержат

идентификаторы, можно прекратить перебор блоков, поскольку и начало, и конец маршрута уже найдены. После завершения работы rdsBroadcastFunctionCallsEx найденные идентификаторы передаются в вызвавшую функцию через указатели pBegin и pEnd, и функция возвращает TRUE, если в графе заданы и начало, и конец маршрута.

Третья функция будет вызывать сразу две функции блоков: функцию разметки графа "ProgrammersGuide.GraphPath.Mark" и, после нее, функцию отслеживания маршрута по разметке "ProgrammersGuide.GraphPath.BackTrace". Мы будем использовать ее для поиска маршрута в графе, если на нем уже отмечены начало и конец, а маркировка графа сброшена:

```
// Поиск маршрута в графе в заданной подсистеме
void GraphPath_FindPath(RDS_BHANDLE System)
{ RDS_BHANDLE StartBlock,EndBlock;
  TProgGuideFuncMarkParams markparams;

  // Считаем, что маркировка всего графа сброшена

  // Ищем начальную и конечную точку маршрута
  if(!GraphPath_GetTerminalBlocks(System,&StartBlock,&EndBlock))
    return; // Начало или конец не найдены
  // Начало маршрута - StartBlock, конец - EndBlock

  // Маркируем граф от начала маршрута
  markparams.servSize=sizeof(markparams);
  markparams.Mark=0.0;     // Начало маркируется значением 0
  markparams.Previous=NULL;// Это значение не пришло от какого-то
                           // соседнего блока
  // Вызываем функцию маркировки для начального блока
  rdsCallBlockFunction(StartBlock,GraphFuncMark,&markparams);

  // Теперь отслеживаем кратчайший путь в обратном направлении
  // (от конечного блока)
  rdsCallBlockFunction(EndBlock,GraphFuncBackTrace,NULL);
}
//=======================================
```

Сначала мы, используя уже написанную функцию GraphPath_GetTerminalBlocks, ищем в графе внутри подсистемы System начало и конец маршрута – их идентификаторы записываются в переменные StartBlock и EndBlock соответственно. Если GraphPath_GetTerminalBlocks вернет FALSE, значит, в графе не отмечены начало или конец маршрута, и поиск невозможен – функция немедленно завершается. В противном случае мы подготавливаем структуру markparams типа TProgGuideFuncMarkParams для пометки начального блока маршрута числом 0: для этого мы записываем 0 в поле Mark и NULL в поле Previous. Поле Prevoius в процессе разметки графа мы используем для того, чтобы, для ускорения работы, при пометке соседей очередного узла не помечать тот узел, из которого мы попали в данный (его метка заведомо меньше метки данного, и, согласно алгоритму со стр. 342, он все равно не будет помечен). При пометке самого первого, начального, узла у нас нет блока, из которого мы попадаем в него, поэтому вместо идентификатора блока мы используем значение NULL.

Далее сервисной функцией rdsCallBlockFunction мы вызываем у блока StartBlock функцию пометки "ProgrammersGuide.GraphPath.Mark" (ее идентификатор должен быть записан в глобальную переменную GraphFuncMark), передавая в качестве параметров функции указатель на структуру markparams. Это приведет к тому, что начальный блок маршрута будет помечен числом 0 и пометит своих соседей расстояниями до них. Они, в свою очередь, пометят своих соседей, те – своих, и т.д., пока все узлы графа не будут помечены расстояниями до начала маршрута.

Когда rdsCallBlockFunction завершится, весь граф окажется размеченным. Теперь мы вызываем функцию “ProgrammersGuide.GraphPath.BackTrace” (ее идентификатор находится в GraphFuncBackTrace) у блока конца маршрута EndBlock: он должен перейти в выделенное состояние, выделить связь, ведущую к нему от блока с наименьшей меткой и вызвать у него ту же функцию. Тот, в свою очередь, тоже перейдет в выделенное состояние, найдет среди блоков, соединенных с ним, блок с наименьшей меткой, выделит связь, ведущую от него к себе и вызовет эту функцию у него, и т.д. Таким образом, весь маршрут от начала до конца окажется выделенным.

Остальные созданные нами функции блоков требуют доступа к переменным конкретного блока, поэтому мы пока не можем написать для них функции-оболочки.

Теперь пришло время подумать о переменных, которые необходимы блоку-узлу графа для работы. Нам потребуются две логических переменных для хранения флагов начала и конца маршрута (назовем их sBegin и sEnd соответственно), логическая переменная, указывающая на принадлежность блока к найденному маршруту (назовем ее sInPath), вещественная переменная для хранения метки узла графа (sPathMark), и еще одна логическая переменная, указывающая на наличие этой метки (sMarked). Эти переменные будут управлять состоянием блока следующим образом:

- sBegin==1 – данный узел графа является началом маршрута;
- sEnd==1 – данный узел графа является концом маршрута;
- sInPath==1 – данный узел графа принадлежит к найденному маршруту и должен визуально выделяться;
- sMarked==0 – у данного узла графа нет метки (расстояние до начала маршрута еще не найдено);
- sMarked==1 – узел графа помечен вещественным числом sPathMark.

Таким образом, блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Значение по умолчанию*
0	Start	Сигнал	1	Вход	0
1	Ready	Сигнал	1	Выход	0
2	sBegin	Логический	1	Внутренняя	0
3	sEnd	Логический	1	Внутренняя	0
4	sInPath	Логический	1	Внутренняя	0
5	sMarked	Логический	1	Внутренняя	0
6	sPathMark	double	8	Внутренняя	0

Блок-узел графа не работает в режиме расчета и не передает никаких данных по связям, поэтому все переменные, кроме двух обязательных сигналов Start и Ready, описаны как внутренние. Блоку все равно нужен по крайней мере один вход и один выход, чтобы можно было соединять такие блоки связями, символизирующими дуги графа – для этой цели мы будем использовать Start и Ready.

Может возникнуть вопрос: почему все имена переменных мы начинаем с буквы “s”? Почему бы не назвать переменную для флага начала маршрута просто Begin, а не sBegin? Дело в том, что в структуре TProgGuideFuncGetParams, описанной нами, есть поля Begin, End и Marked. Если мы хотим использовать для доступа к переменным блока макросы с теми же названиями, эти названия не должны совпадать с именами других переменных и полей структур и классов, которые используются в тех же функциях (эта проблема уже упоминалась на стр. 264). Например, если бы мы назвали переменную блока Begin, а не sBegin, макрос для нее выглядел бы так:

```
#define pStart ((char *)(BlockData->VarData))
// ...
#define Begin (*((char *)(pStart+2)))
```

В результате, встретив в тексте программы конструкцию вида

```
TProgGuideFuncGetParams data;
data.Begin=TRUE;
```

препроцессор посчитал бы слово `Begin` именем макроса и развернул бы его:

```
TProgGuideFuncGetParams data;
data.(*((char *)(((char *)(BlockData->VarData))+2)))=TRUE;
```

Чтобы исключить такую возможность, мы, на всякий случай, сделали так, чтобы имена всех переменных блока не совпадали с полями используемых нами структур.

Мы не будем записывать всю логику работы блока внутри одной-единственной функции модели – это сделало бы ее слишком громоздкой. Часть операций мы вынесем в другие функции, что несколько улучшит читаемость текста программы. Поскольку в этих функциях нам потребуется доступ к переменным блока, в каждую из них мы будем передавать указатель на структуру данных блока `RDS_PBLOCKDATA BlockData` – он используется в макросах переменных, и без него они не будут работать (см. стр. 43). Все эти функции, как и сама функция модели, будут находиться после макроопределений для переменных блока:

```
// Макроопределения для переменных блока
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define sBegin (*((char *)(pStart+2)))
#define sEnd (*((char *)(pStart+3)))
#define sInPath (*((char *)(pStart+4)))
#define sMarked (*((char *)(pStart+5)))
#define sPathMark (*((double *)(pStart+6)))
```

Теперь запишем прототипы функций, которые мы будем вызывать из функции модели (сами функции мы напишем позже):

```
// Сделать этот блок началом маршрута
void GraphNode_SetBlockAsBegin(RDS_PBLOCKDATA BlockData);
// Сделать этот блок концом маршрута
void GraphNode_SetBlockAsEnd(RDS_PBLOCKDATA BlockData);
// Реакция блока на функцию ProgrammersGuide.GraphPath.Reset
void GraphNode_OnReset(RDS_PBLOCKDATA BlockData,
                       TProgGuideFuncResetParams *reset);
// Реакция блока на функцию ProgrammersGuide.GraphPath.Mark
void GraphNode_OnMarkBlock(RDS_PBLOCKDATA BlockData,
                       TProgGuideFuncMarkParams *params);
// Реакция блока на функцию ProgrammersGuide.GraphPath.BackTrace
void GraphNode_OnBackTracePath(RDS_PBLOCKDATA BlockData);
// Программное рисование внешнего вида блока
void GraphNode_Draw(RDS_PBLOCKDATA BlockData,RDS_PDRAWDATA draw);
```

Мы вынесли в отдельные функции установку флагов начала и конца маршрута, программное рисование внешнего вида блока и реакцию на три из пяти созданных нами функций блока (реакция на две оставшихся будет достаточно простой, и мы запишем ее непосредственно в функции модели).

Функция модели блока будет иметь следующий вид:

```
// Модель блока-узла графа
extern "C" __declspec(dllexport)
  int RDSCALL GraphNode(int CallMode,
                        RDS_PBLOCKDATA BlockData,
                        LPVOID ExtParam)
```

```
{ // Вспомогательная - указатель на данные функции блока
  RDS_PFUNCTIONCALLDATA func;

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Регистрация функций
        if(GraphFuncGetParams==0)
          GraphFuncGetParams=rdsRegisterFunction(
                      PROGGUIDEGRAPHPATHFUNC_GETPARAMS);
        if(GraphFuncReset==0)
          GraphFuncReset=rdsRegisterFunction(
                      PROGGUIDEGRAPHPATHFUNC_RESET);
        if(GraphFuncFind==0)
          GraphFuncFind=rdsRegisterFunction(
                      PROGGUIDEGRAPHPATHFUNC_FIND);
        if(GraphFuncMark==0)
          GraphFuncMark=rdsRegisterFunction(
                      PROGGUIDEGRAPHPATHFUNC_MARK);
        if(GraphFuncBackTrace==0)
          GraphFuncBackTrace=rdsRegisterFunction(
                      PROGGUIDEGRAPHPATHFUNC_BACKTRACE);
        break;

      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strncmp((char*)ExtParam,"{SSLLLLD",8)?
                      RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Вызов контекстного меню блока
      case RDS_BFM_CONTEXTPOPUP:
        rdsAdditionalContextMenuItemEx("Начало маршрута",
                          sBegin?RDS_MENU_DISABLED:0,0,0);
        rdsAdditionalContextMenuItemEx("Конец маршрута",
                          sEnd?RDS_MENU_DISABLED:0,1,0);
        rdsAdditionalContextMenuItemEx("Сбросить все",0,2,0);
        break;

      // Быбор пункта меню
      case RDS_BFM_MENUFUNCTION:
        switch(((RDS_PMENUFUNCDATA)ExtParam)->Function)
          { case 0: // Начало маршрута
              GraphNode_SetBlockAsBegin(BlockData);
              rdsRefreshBlockWindows(BlockData->Parent,FALSE);
              break;
            case 1: // Конец маршрута
              GraphNode_SetBlockAsEnd(BlockData);
              rdsRefreshBlockWindows(BlockData->Parent,FALSE);
              break;
            case 2: // Сбросить все
              GraphPath_Reset(BlockData->Parent,TRUE,TRUE,TRUE);
              rdsRefreshBlockWindows(BlockData->Parent,FALSE);
              break;
          }
        break;
```

```
        // Вызов функции блока
        case RDS_BFM_FUNCTIONCALL:
          func=(RDS_PFUNCTIONCALLDATA)ExtParam;
          if(func->Function==GraphFuncReset) // Сброс параметров
            GraphNode_OnReset(BlockData,
                        (TProgGuideFuncResetParams*)(func->Data));
          else if(func->Function==GraphFuncGetParams)
            { // Получение параметров узла
              TProgGuideFuncGetParams *get=
                        (TProgGuideFuncGetParams*)(func->Data);
              if(get!=NULL &&
                  get->servSize>=sizeof(TProgGuideFuncGetParams))
                { // Допустимый размер структуры параметров
                  get->Marked=(sMarked!=0);
                  get->Mark=sPathMark;
                  get->Begin=(sBegin!=0);
                  get->End=(sEnd!=0);
                }
              return 1; // Блок является узлом графа
            }
          else if(func->Function==GraphFuncFind)
            { // Поиск начала и конца маршрута
              TProgGuideFuncFindParams *find=
                        (TProgGuideFuncFindParams*)(func->Data);
              if(find==NULL) break; // Нет параметров
              if(find->servSize<sizeof(TProgGuideFuncFindParams))
                break; // Недопустимый размер структуры параметров
              if(sBegin) // Этот блок - начало маршрута
                find->BeginBlock=BlockData->Block;
              if(sEnd) // Этот блок - конец маршрута
                find->EndBlock=BlockData->Block;
              if(find->BeginBlock!=NULL && find->EndBlock!=NULL)
                func->Stop=TRUE; // Оба конца маршрута найдены
            }
          else if(func->Function==GraphFuncMark) // Пометить граф
            GraphNode_OnMarkBlock(BlockData,
                        (TProgGuideFuncMarkParams*)(func->Data));
          else if(func->Function==GraphFuncBackTrace)
            GraphNode_OnBackTracePath(BlockData); // Выделить маршрут
          break;

        // Рисование
        case RDS_BFM_DRAW:
          GraphNode_Draw(BlockData,(RDS_PDRAWDATA)ExtParam);
          break;
      }
    return RDS_BFR_DONE;
  }
  //=======================================
```

Наш блок не имеет личной области данных, поэтому единственное, что мы должны сделать при вызова нашей модели в режиме `RDS_BFM_INIT` – это зарегистрировать пять функций блока, которые мы собираемся использовать. Как и в предыдущих примерах, функции регистрируются только при инициализации модели самого первого блока – для этого глобальные переменные, в которых хранятся идентификаторы зарегистрированных функций, проверяются на нулевое значение.

Реакция модели на вызов `RDS_BFM_VARCHECK` несколько отличается от аналогичных реакций в других примерах: здесь мы анализируем не всю переданную в

функцию модели строку типов переменных, а только восемь первых ее символов. Таким образом мы позволяем добавлять в конец списка переменных блока любое количество новых переменных любых типов – нас интересуют только первые семь. Хотя мы и договорились использовать для рисования дуг графа связи, соединенные со входами `Start` и `Ready`, пользователю, привыкшему к использованию этих переменных для управления расчетом, это может показаться нелогичным. При желании, он может добавить в блок дополнительные вход и выход, к которым будут подключаться связи, изображающие дуги графа.

В момент открытия контекстного меню блока (при вызове модели в режиме `RDS_BFM_CONTEXTPOPUP`) мы добавляем в меню три временных пункта: “Начало маршрута” (идентификатор пункта 0), “Конец маршрута” (идентификатор 1) и “Сбросить все” (идентификатор 2). Первые два пункта будут помечать блок, для которого вызвано меню, как начало и конец маршрута соответственно, при этом у блока, который уже помечен как начало или конец, соответствующий пункт меню будет запрещенным. Третий пункт меню сбрасывает маршрут во всем графе и приводит граф в исходное состояние.

При выборе пользователем одного из добавленных пунктов меню (режим `RDS_BFM_MENUFUNCTION`) мы вызываем функцию `GraphNode_SetBlockAsBegin` (для пункта “Начало маршрута”), `GraphNode_SetBlockAsEnd` (для пункта “Конец маршрута”) или `GraphPath_Reset` (для “Сбросить все”). Функцию `GraphPath_Reset` мы уже написали ранее – в нее передается идентификатор родительской подсистемы данного блока `BlockData->Parent` и три значения `TRUE`, указывающие, что в графе внутри этой подсистемы необходимо сбросить и флаги начала маршрута, и флаги его конца, и все метки узлов. Функции `GraphNode_SetBlockAsBegin` и `GraphNode_SetBlockAsEnd`, которые должны делать данный блок началом или концом маршрута, мы напишем позже (пока мы написали только их прототипы). После вызова любой из этих трех функций мы перерисовываем окно родительской подсистемы блока сервисной функцией `rdsRefreshBlockWindows`, чтобы отразить изменения, которые, возможно, произошли в состоянии блоков и внешнем виде связей из-за изменения или сброса маршрута. Параметр `FALSE`, переданный в функцию, запрещает обновлять окна подсистем, вложенных в `BlockData->Parent`: изменения в маршруте графа никак не могут отразиться на других подсистемах, и перерисовывать их не нужно.

В реакции на вызов функции блока (`RDS_BFM_FUNCTIONCALL`) мы, прежде всего, выясняем, какая именно функция вызвана, сравнивая переданный идентификатор с глобальными переменными, в которые мы записали идентификаторы зарегистрированных функций. Для функций “ProgrammersGuide.GraphPath.Reset” (идентификатор хранится в глобальной переменной `GraphFuncReset`), “ProgrammersGuide.GraphPath.Mark” (идентификатор в `GraphFuncMark`) и “ProgrammersGuide.GraphPath.BackTrace” (идентификатор в `GraphFuncBackTrace`) мы вызываем одну из внешних функций, для которых пока написаны только прототипы. На две оставшиеся функции мы реагируем прямо внутри модели.

Если идентификатор совпал с переменной `GraphFuncGetParams` (вызвана функция “ProgrammersGuide.GraphPath.GetParams” для получения параметров узла графа), мы приводим переданный указатель на параметры функции к типу `TProgGuideFuncGetParams*` и записываем его во вспомогательную переменную `get`. Мы решили, что эту функцию можно вызывать и без параметров, в этом случае значение `get` окажется равным `NULL`. Если `get` – не `NULL`, и поле `servSize` структуры, на которую указывает `get`, больше или равно размеру типа `TProgGuideFuncGetParams`, значит, функция вызвана с параметрами, и эти параметры переданы правильно: нужно записать в поля структуры, на которую указывает `get`, значения переменных состояния блока. Затем функция возвращает значение 1, указывающее на то, что данный блок является узлом графа.

Если идентификатор совпал с переменной GraphFuncFind, значит, вызвана функция “ProgrammersGuide.GraphPath.Find”. В этом случае мы приводим переданный указатель на параметры функции к типу TProgGuideFuncFindParams* и записываем его в переменную find. Если поле servSize переданной в параметрах структуры меньше размера типа TProgGuideFuncFindParams, значит, функция вызвана с неправильными параметрами, и ее работа немедленно завершается оператором break. В противном случае, если данный блок является началом маршрута, мы записываем его идентификатор в поле BeginBlock переданной структуры, если он является концом – в поле EndBlock (мы договорились, что перед вызовом этой функции для всех блоков графа эти поля должны быть обнулены). Затем, если оба поля имеют ненулевые значения (то есть если в структуре уже отметились и блок начала маршрута, и блок его конца), полю Stop структуры описания вызванной функции (RDS_FUNCTIONCALLDATA) присваивается значение TRUE. Это остановит перебор блоков подсистемы, если функция “ProgrammersGuide.GraphPath.Find” была вызвана через сервисную функцию rdsBroadcastFunctionCallsEx (в уже написанной нами функции GraphPath_GetTerminalBlocks мы делаем именно так).

Наконец, при вызове модели в режиме RDS_BFM_DRAW для программного рисования внешнего вида блока мы вызываем внешнюю функцию GraphNode_Draw, которую нам предстоит написать. Ей мы сейчас и займемся.

Будем изображать блоки выделенного маршрута зелеными прямоугольниками, а все остальные – белыми. На блоке начала и конца маршрута будем выводить буквы “Н” и “К” соответственно. Никаких настроек для программного рисования мы предусматривать не будем: при необходимости, пользователь сможет подключить к блоку векторную картинку любого удобного ему вида. Функция GraphNode_Draw будет выглядеть так:

```
// Рисование узла графа
void GraphNode_Draw(RDS_PBLOCKDATA BlockData,RDS_PDRAWDATA draw)
{ static char beg[]="H",end[]="K"; // Метки начала и конца
  int w,h;

  // Рисуем прямоугольник, цвет которого зависит от переменной
  // блока sInPath
  rdsXGSetPenStyle(0,PS_SOLID,1,0,R2_COPYPEN);
  rdsXGSetBrushStyle(0,RDS_GFS_SOLID,sInPath?0xff00:0xffffff);
  rdsXGRectangle(draw->Left,draw->Top,
                 draw->Left+draw->Width,draw->Top+draw->Height);

  if(sBegin==0 && sEnd==0) // Нет флагов начала и конца маршрута
    return;
  // Устанавливаем шрифт высотой в весь блок
  rdsXGSetBrushStyle(0,RDS_GFS_EMPTY,0);
  rdsXGSetFont(0,"Arial Cyr",
     draw->Height,0,RUSSIAN_CHARSET,0,FALSE,FALSE,FALSE,FALSE);

  if(sBegin) // Рисуем метку начала
    { rdsXGGetTextSize(beg,&w,&h);
      rdsXGTextOut(draw->Left+(draw->Width-w)/2,
                   draw->Top+(draw->Height-h)/2,
                   beg);
    }
  if(sEnd) // Рисуем метку конца
    { rdsXGGetTextSize(end,&w,&h);
      rdsXGTextOut(draw->Left+(draw->Width-w)/2,
                   draw->Top+(draw->Height-h)/2,
                   end);
    }
```

```
}
//=======================================
```

В этой функции мы используем переменные блока, поэтому в нее передается указатель на структуру данных блока `RDS_PBLOCKDATA BlockData`, используемый в макросах для переменных. Кроме того, как и в другие функции рисования, которые мы писали в предыдущих примерах, в нее передается указатель на структуру типа `RDS_DRAWDATA`, в которой находятся параметры, необходимые для рисования изображения блока. Внутри функции мы сначала рисуем прямоугольник размером в весь блок. Цвет этого прямоугольника выбирается согласно значению логической переменной блока `sInPath`: при ее истинном, то есть ненулевом, значении (узел графа находится на выделенном маршруте) прямоугольник будет зеленым, при ложном – белым. Затем, если обе переменных `sBegin` и `sEnd` равны нулю, то есть в блоке нет ни флага начала, ни флага конца маршрута, функция рисования завершается. Если же одна из этих переменных не равна нулю, устанавливается шрифт “Arial” с русским набором символов высотой в весь прямоугольник блока (`draw->Height`) и выводится соответствующая буква – “Н” или “К”. Технически в блоке могут быть установлены оба флага, тогда обе буквы выведутся одна поверх другой, но при поиске маршрута такая ситуация вряд ли встретится (зачем искать маршрут от какого-то узла графа к нему же?) и мы не будем предпринимать по этому поводу каких-либо действий.

Теперь напишем функцию `GraphNode_OnReset`, которая обеспечивает реакцию нашего блока на вызов функции “ProgrammersGuide.GraphPath.Reset”:

```
// Реакция блока на функцию ProgrammersGuide.GraphPath.Reset
void GraphNode_OnReset(RDS_PBLOCKDATA BlockData,
                       TProgGuideFuncResetParams *reset)
{ if(reset==NULL) return; // Нет параметров функции
  if(reset->servSize<sizeof(TProgGuideFuncResetParams))
    return; // Размер переданной структуры меньше ожидаемого

  if(reset->ResetMark)
    { // Сбрасываем выделение всех присоединенных к блоку связей
      RDS_CHANDLE c=NULL;
      for(;;) // Перебираем все связи, подключенные к блоку
        { c=rdsGetBlockLink(BlockData->Block,c,TRUE,TRUE,NULL);
          if(c==NULL) break; // Все связи перебраны
          // Снимаем выделение связи с
          UnmarkConnection(c);
        }
      // Сбрасываем флаги наличия метки и принадлежности
      // к выделенному маршруту
      sMarked=sInPath=0;
    }

  if(reset->ResetBegin) // Сбрасываем флаг начала маршрута
    sBegin=0;
  if(reset->ResetEnd) // Сбрасываем флаг конца маршрута
    sEnd=0;
}
//=======================================
```

В эту функцию передается указатель на структуру данных блока `BlockData` (для обеспечения работы макросов переменных) и указатель на структуру параметров функции “ProgrammersGuide.GraphPath.Reset”. В этой структуре указано, какие именно переменные блока нужно сбросить.

Прежде всего, мы, как обычно, проверяем правильность переданных параметров, сравнивая поле `servSize` переданной структуры `reset` с размером типа `TProgGuideFuncResetParams`. Если значение поля окажется меньше размера типа,

параметры переданы неправильно, и функция завершается. В противном случае анализируются логические поля переданной структуры.

Если поле ResetMark истинно, нужно сбросить метку узла графа и отменить выделение маршрута. Чтобы отменить выделение нужно, кроме изменения переменных данного блока, снять выделение со связей, подключенных к этому блоку, если они изображают дуги, попавшие на маршрут. Мы не будем разбираться, какие связи находятся на маршруте, а какие – нет, мы просто снимем выделение со всех связей, соединенных с этим блоком. Для этого переберем их все функцией rdsGetBlockLink (мы уже встречались с ней в §2.7.4) и для каждой вызовем уже написанную нами функцию UnmarkConnection. Переменным блока sMarked (флаг наличия метки) и sInPath (флаг принадлежности к маршруту) мы присваиваем значение 0.

Если в переданной структуре истинно поле ResetBegin, нужно сбросить у блока флаг начала маршрута: переменной sBegin присваивается значение 0. Если истинно поле ResetEnd, то сбрасывается флаг конца: обнуляется переменная sEnd.

Функция GraphNode_SetBlockAsBegin делает данный блок началом маршрута:

```
// Сделать данный блок началом маршрута
void GraphNode_SetBlockAsBegin(RDS_PBLOCKDATA BlockData)
{ if(sBegin) // Блок уже является началом маршрута
    return;
  // Сбрасываем старый флаг начала маршрута и разметку всего графа
  GraphPath_Reset(BlockData->Parent,TRUE,TRUE,FALSE);
  // Устанавливаем флаг начала маршрута у данного блока
  sBegin=1;
  // Ищем маршрут в графе
  GraphPath_FindPath(BlockData->Parent);
}
//=======================================
```

Прежде всего мы, на всякий случай, проверяем, не является ли уже данный блок (то есть блок, к которому относится переданный в функцию указатель на структуру данных BlockData) началом маршрута. Если это так, никаких действий предпринимать не нужно – функция немедленно завершается. Если данный блок началом маршрута не является, мы вызываем для родительской подсистемы данного блока ранее написанную нами функцию GraphPath_Reset, которая сбросит во всем графе разметку и старый флаг начала маршрута. Флаг конца маршрута в графе не сбрасывается, об этом говорит переданное в четвертом параметре функции значение FALSE. Затем мы устанавливаем флаг начала маршрута у данного блока, присвоив его переменной sBegin значение 1 (это обязательно делать после вызова функции GraphPath_Reset – если сделать это до ее вызова, она, работая со всеми узлами графа, сбросит только что установленный нами флаг и в данном блоке тоже). После этого мы пытаемся найти в графе маршрут от начала к концу, вызывая GraphPath_FindPath. Мы не проверяем, установлен ли флаг конца маршрута: если он не установлен, GraphPath_FindPath просто завершится, не выполнив никаких действий.

Функция, делающая данный блок концом маршрута, устроена аналогично:

```
// Сделать данный блок концом маршрута
void GraphNode_SetBlockAsEnd(RDS_PBLOCKDATA BlockData)
{ if(sEnd) // Блок уже является концом маршрута
    return;
  // Сбрасываем старый флаг конца маршрута и разметку всего графа
  GraphPath_Reset(BlockData->Parent,TRUE,FALSE,TRUE);
  // Устанавливаем флаг конца маршрута у данного блока
  sEnd=1;
  // Ищем маршрут в графе
  GraphPath_FindPath(BlockData->Parent);
```

```
}
//=======================================
```

Точно так же, мы сначала сбрасываем разметку графа и старый флаг конца, устанавливаем новый флаг конца в данном блоке, а затем пытаемся отыскать в графе кратчайший путь от начала маршрута к его концу.

Осталось написать две самых важных функции, которые, собственно, и реализуют описанный выше алгоритм разметки графа и прослеживание в нем кратчайшего маршрута. Начнем с функции `GraphNode_OnMarkBlock`, реализующей реакцию блока на вызов “ProgrammersGuide.GraphPath.Mark”: она должна пометить данный блок указанным в параметрах числом, если оно меньше текущей метки блока (см. описание алгоритма на стр. 342), а также пометить все соседние узлы графа суммой метки данного блока и расстояния до соседнего узла. Для перебора соседей блока мы будем использовать сервисную функцию РДС `rdsEnumConnectedBlocks` (она уже встречалась нам в §2.13.2), поэтому сначала напишем прототип функции, которая будет вызываться для каждого обнаруженного соседа (саму функцию мы напишем позже):

```
// Прототип функции обратного вызова для перечисления соседей блока
BOOL RDSCALL GraphPath_MarkBlock_Callback(
          RDS_PPOINTDESCRIPTION src,
          RDS_PPOINTDESCRIPTION dest,
          LPVOID data);
```

Теперь напишем функцию `GraphNode_OnMarkBlock` (она будет достаточно простой, основная работа будет выполняться в функции обратного вызова):

```
// Пометить данный блок и его соседей
void GraphNode_OnMarkBlock(RDS_PBLOCKDATA BlockData,
                     TProgGuideFuncMarkParams *params)
{ if(params==NULL) return; // Нет параметров функции
  if(params->servSize<sizeof(TProgGuideFuncMarkParams))
    return; // Размер переданной структуры меньше ожидаемого

  if(sMarked!=0 && sPathMark<=params->Mark)
    return; // Блок уже помечен меньшим или таким же числом

  // Блок не помечен вообще или помечен большим числом -
  // помечаем его и его соседей новыми числами
  sMarked=1;                // У блока есть метка
  sPathMark=params->Mark;  // Новая метка блока

  if(sEnd) // Это - конец маршрута, дальше помечать незачем
    return;

  // Помечаем всех соседей блока, кроме params->Previous,
  // суммой метки этого блока и длины дуги к соседу
  rdsEnumConnectedBlocks(BlockData->Block,
          RDS_BEN_INPUTS|RDS_BEN_OUTPUTS,
          GraphPath_MarkBlock_Callback,params);
}
//=======================================
```

Сначала мы, как обычно, проверяем размер переданной структуры параметров. Если он нам подходит, мы сравниваем число, которым предлагается пометить этот блок, с уже имеющейся в нем меткой. Если новая метка больше старой, ничего помечать заново не нужно. Если же новая метка меньше, блок помечается новым, меньшим, значением: оно переносится в переменную блока `sPathMark` (при этом взводится флаг наличия метки `sMarked`). Если данный блок является концом маршрута, помечать его соседей не требуется – достигнув конца, нам не нужно идти дальше. В противном случае при помощи вызова

rdsEnumConnectedBlocks для каждого из блоков, соединенных с данным, вызывается функция обратного вызова GraphPath_MarkBlock_Callback, в которую будут передан указатель params, а также описания двух крайних точек связи, соединяющей эти два блока (см. стр. 323). Эта функция должна проверить допустимость направления связи (мы можем переходить от блока к блоку только по стрелкам), вычислить длину дуги и попытаться пометить соседний блок новым числом. Напишем эту функцию:

```
// Функция обратного вызова для GraphNode_OnMarkBlock
BOOL RDSCALL GraphPath_MarkBlock_Callback(
           RDS_PPOINTDESCRIPTION src,
           RDS_PPOINTDESCRIPTION dest,LPVOID data)
{ TProgGuideFuncMarkParams *src_params=
           (TProgGuideFuncMarkParams*)data;
  TProgGuideFuncMarkParams dest_params;
  double ArcLen;

  // Функция вызвана для пары блоков: src->Block - данный блок,
  // dest->Block - его сосед. Их соединяет связь dest->Owner (или
  // src->Owner - поля Owner у структур src и dest равны, поскольку
  // обе точки принадлежат одной и той же связи)

  // Сравниваем найденного соседа с блоком,
  // который не нужно помечать
  if(src_params->Previous==dest->Block)
    return TRUE;

  // Связь должна подходить ко входу блока dest->Block
  // (движение в графе возможно только по стрелкам)
  if(dest->Source) // Точка dest соединена с выходом блока
    return TRUE;

  // Является ли найденный сосед узлом графа?
  if(!rdsCallBlockFunction(dest->Block,GraphFuncGetParams,NULL))
    return TRUE; // Не является

  // Вычисляем длину дуги между блоками
  ArcLen=CalcArcLength(dest->Owner);
  if(ArcLen<0.0) // Связь разветвленная или оборванная
    return TRUE; // Такая связь не может быть дугой

  // Помечаем найденный соседний блок суммой маркировки данного
  // блока (src_params->Mark) и длины дуги к найденному (ArcLen)
  dest_params.servSize=sizeof(dest_params);
  dest_params.Mark=src_params->Mark+ArcLen;
  dest_params.Previous=src->Block;    // Блок, от которого
                                      // пришла метка
  rdsCallBlockFunction(dest->Block,GraphFuncMark,&dest_params);
  return TRUE;
}
//=========================================
```

В параметре data в эту функцию передается последний параметр функции rdsEnumConnectedBlocks. В функции GraphNode_OnMarkBlock мы подставили туда params – указатель на структуру параметров функции “ProgrammersGuide.GraphPath.Mark”, имеющую тип TProgGuideFuncMarkParams. Поэтому в данной функции мы, прежде всего, приводим параметр data, имеющий тип void*, обратно к типу “указатель на TProgGuideFuncMarkParams” и записываем его во вспомогательную переменную

src_params. Теперь мы имеем доступ к метке блока, соседей которого мы помечаем (src_params->Mark) и к идентификатору блока, от которого пришла эта метка, и который помечать бессмысленно (src_params->Previous). Мы можем сравнить идентификатор найденного соседа (dest->Block) с этим идентификатором и завершить функцию, если они совпали – этого соседа помечать не нужно. Завершая функцию обратного вызова мы всегда будем возвращать TRUE, поскольку возврат FALSE прервал бы работу rdsEnumConnectedBlocks, а нам этого не нужно.

Далее мы вызываем в найденном блоке-соседе функцию блока "ProgrammersGuide.GraphPath.GetParams", которая должна вернуть единицу, если этот блок является узлом графа. Если она вернет ноль, мы не будем работать с этим блоком, поскольку в поиске маршрута участвуют только блоки-узлы графа.

Теперь, когда мы знаем, что найденный сосед dest->Block является узлом графа, который нужно попробовать пометить, мы можем вычислить длину дуги, ведущей к нему. Для этого вызывается написанная ранее функция CalcArcLength, в которую передается идентификатор связи dest->Owner, изображающей эту дугу, и возвращенное ей значение записывается в переменную ArcLen. Функция CalcArcLength не только вычислит длину дуги, но и проверит связь на разветвленность и обрыв – мы работаем только со связями, соединяющими два блока. Если значение ArcLen отрицательно, связь не прошла эту проверку, и мы завершаем работу функции.

Далее нам нужно вызвать у найденного соседа функцию блока "ProgrammersGuide.GraphPath.Mark", передав в ее параметрах новое значение метки, полученное сложением метки данного блока и длины дуги к соседу. Мы не можем использовать для этого структуру, указатель на которую находится в src_params: там хранятся параметры, с которыми был вызван данный блок, и они нам будут нужны при обработке следующего соседа – мы не имеем права их изменять. Поэтому при вызове функции блока мы будем использовать другую структуру – dest_params. Ее поля нам сейчас предстоит заполнить.

В поле servSize мы, как обычно, записываем размер самой структуры для проверки правильности передачи параметров. В поле Mark мы записываем новое значение метки соседа: сумму src_params->Mark и длины дуги ArcLen. В поле Previous мы записываем идентификатор данного блока src->Block: когда вызванный нами сосед начнет помечать своих соседей, данный блок будет из этого процесса исключен. Теперь мы вызываем у найденного соседа функцию "ProgrammersGuide.GraphPath.Mark" (ее идентификатор хранится в глобальной переменной GraphFuncMark), передав ей в качестве параметров указатель на структуру dest_params. Сосед, реагируя на этот вызов, вызовет функцию GraphNode_OnMarkBlock, которая начнет перебирать уже его соседей, вызывая для каждого GraphPath_MarkBlock_Callback, и т.д., пока весь граф не будет размечен.

Последняя функция, которую мы должны написать – это GraphNode_OnBackTracePath, реакция на вызов функции блока "ProgrammersGuide.GraphPath.BackTrace". Эта функция должна пометить данный блок как принадлежащий к выделенному маршруту, найти среди соседей блок с наименьшей меткой, выделить связь, идущую от него, и тоже вызвать для него "ProgrammersGuide.GraphPath.BackTrace". Так, начав с конца маршрута, блок за блоком, будет помечен весь маршрут.

Как и в GraphNode_OnMarkBlock, в этой функции нужно будет перебирать всех соседей блока, чтобы найти среди них блок с наименьшей меткой. Можно, как и там, сделать это с помощью сервисной функции rdsEnumConnectedBlocks, но, для разнообразия, воспользуемся уже знакомой нам функцией rdsGetBlockLink. Текст программы получится более громоздким, зато не потребуется писать функцию обратного вызова.

```
// Проследить и выделить маршрут от данного блока в
// обратном направлении
void GraphNode_OnBackTracePath(RDS_PBLOCKDATA BlockData)
{ RDS_BHANDLE PrevBlock;
  RDS_CHANDLE PrevConn,c;
  double minMark;
  RDS_CONNDESCRIPTION conndescr;
  RDS_POINTDESCRIPTION pointdescr;
  TProgGuideFuncGetParams params;

  if(!sMarked) // У блока нет метки
    return;

  sInPath=1; // Помечаем данный блок как принадлежащий маршруту

  if(sBegin) // Это начало маршрута - мы выделили его весь
    return;

  // Заполняем служебные поля размера у всех структур,
  // которые нам потребуются
  conndescr.servSize=sizeof(conndescr);
  pointdescr.servSize=sizeof(pointdescr);
  params.servSize=sizeof(params);

  // Ищем среди соседей данного блока блок с наименьшей меткой
  PrevBlock=NULL;
  // Перебираем все связи блока
  c=NULL;
  for(;;)
    { RDS_BHANDLE connBlock;
      double connMark;
      int numBlocks;
      // Получаем очередную связь блока, соединенную с его входом
      c=rdsGetBlockLink(BlockData->Block,c,TRUE,FALSE,NULL);
      if(c==NULL) break; // Связи кончились
      // Получаем параметры этой связи
      rdsGetConnDescription(c,&conndescr);
      // Перебираем все точки связи с и ищем среди них соединение
      // с узлом графа на другом ее конце
      connBlock=NULL;
      numBlocks=0;
      for(int i=0;i<conndescr.NumPoints;i++)
        { // Получаем описание точки связи i
          rdsGetPointDescription(c,i,&pointdescr);
          // Проверяем тип точки
          if(pointdescr.PointType==RDS_PTBUS)
            { // Связь, соединенная с шиной, нам не годится
              connBlock=NULL;
              break;
            }
          if(pointdescr.PointType!=RDS_PTBLOCK)
            continue;
          // Найдена точка соединения с блоком
          numBlocks++;
          if(numBlocks>2) // Разветвленная связь - не годится
            { connBlock=NULL;
              break;
            }
```

```
            if(pointdescr.Block==BlockData->Block)
              continue; // Это не сосед, а сам данный блок
            // Получаем параметры узла блока-соседа
            if(!rdsCallBlockFunction(pointdescr.Block,
                        GraphFuncGetParams,&params))
              { // Блок не является узлом графа
                connBlock=NULL;
                break;
              }
            // Нашли соседний блок, являющийся узлом графа
            if(params.Marked) // У соседнего узла есть метка
              { connBlock=pointdescr.Block; // Запоминаем
                connMark=params.Mark;
              }
          } // for(int i=0;...)
        // Цикл перебора точек связи закончен. В connBlock должен
        // находиться идентификатор узла, который эта связь соединяет
        // с данным блоком, а в connMark – его метка.
        if(connBlock) // Есть узел графа на другом конце связи
          { if(PrevBlock==NULL || minMark>connMark)
              { // Найден узел с меньшей маркировкой
                PrevBlock=connBlock;
                minMark=connMark;
                PrevConn=c;
              }
          }
      } // for(;;)

     // Перебор связей данного блока закончен. В PrevBlock должен
     // находиться идентификатор соседа с наименьшей меткой, из
     // которого можно попасть в данный блок, а в PrevConn –
     // идентификатор соединяющей их связи

    if(PrevBlock)
      { // Визуально выделяем связь
        MarkConnection(PrevConn);
        // Вызываем функцию обратного прослеживания маршрута
        // от найденного блока
        rdsCallBlockFunction(PrevBlock,GraphFuncBackTrace,NULL);
      }
  }
  //=======================================
```

Сначала мы проверяем, есть ли у данного блока метка. Если ее нет, он не достижим из начального блока маршрута, и выделять нечего – функция завершается. Если же она есть, мы взводим в данном блоке флаг принадлежности к маршруту `sInPath` и, если наш блок является началом маршрута, прекращаем работу – весь маршрут уже выделен. В противном случае начинаем перебирать всех соседей блока и искать среди них узел графа с наименьшей меткой, то есть ближайший к началу маршрута.

Для поиска соседей мы в цикле перебираем все связи блока при помощи сервисной функции `rdsGetBlockLink`. Мы просматриваем маршрут от конца к началу, поэтому нас интересуют только связи, стрелки которых направлены к данному блоку, то есть присоединенные к его входам, поэтому в третьем параметре `rdsGetBlockLink` передается `TRUE`, а в четвертом – `FALSE`. В пятом параметре вместо указателя на структуру описания точки связи передается `NULL`, поскольку это описание нас сейчас не интересует.

В найденной связи c мы будем перебирать точки в поисках точек соединения с блоками, поэтому нам необходимо узнать общее число точек в связи. Для этого мы считываем параметры связи в структуру conndescr при помощи сервисной функции rdsGetConnDescription. Теперь мы можем организовать цикл по всем точкам связи, чтобы найти блок на ее конце: счетчик цикла i будет изменяться от 0 до conndescr.NumPoints-1. Идентификатор найденного блока будет записан в переменную connBlock.

В цикле мы, прежде всего, вызываем сервисную функцию rdsGetPointDescription, чтобы считать параметры i-й точки связи c в структуру pointdescr. Затем мы проверяем тип этой точки. Если это точка соединения связи с шиной (RDS_PTBUS), эта связь не может быть дугой графа, даже если где-то, на одном из ее концов, будет находиться блок. В этом случае мы обнуляем переменную connBlock и прекращаем перебор точек связи оператором break. Если это внутренняя точка связи, мы пропускаем ее и переходим к следующей. Если же это точка соединения с блоком, мы увеличиваем счетчик найденных блоков numBlocks. Если его значение превысит 2, связь разветвлена, и она не может быть дугой графа – мы прекращаем перебор точек. В противном случае мы сравниваем идентификатор соединенного блока с идентификатором данного, и пропускаем эту точку, если они совпали – нас интересует блок на другом конце этой связи. Если идентификаторы не совпали, мы вызываем у найденного блока функцию “ProgrammersGuide.GraphPath.GetParams” (ее идентификатор хранится в глобальной переменной GraphFuncGetParams), которая заполнит структуру params типа TProgGuideFuncGetParams параметрами узла графа, если, конечно, найденный блок им является. Если блок не является узлом графа (функция вернула 0), мы обнуляем connBlock и прекращаем перебор точек – нам нужны только связи, идущие от узлов графа. В противном случае мы запоминаем его идентификатор в переменной connBlock, а его метку – в connMark.

После завершения цикла по всем точкам связи в переменной connBlock должен находится идентификатор найденного на противоположном конце связи узла графа или NULL, если такого узла нет, а в переменной connMark – значение метки этого узла. Теперь мы можем сравнить найденное значение метки с найденным на данный момент минимальным значением minMark и, если найденное значение окажется меньше, заменить значение в minMark на connMark и запомнить идентификатор найденного блока узла в переменной PrevBlock, а идентификатор идущей от него связи – в PrevConn.

Таким образом, после завершения перебора всех связей в переменной PrevBlock должен оказаться идентификатор блока-соседа с наименьшей меткой, то есть идентификатор предыдущего блока кратчайшего маршрута, а в PrevConn – связь, соединяющая этот блок с данным. Нам остается только визуально выделить эту связь, вызвав для нее ранее написанную функцию MarkConnection, и вызвать для найденного блока функцию “ProgrammersGuide.GraphPath.BackTrace”, реагируя на которую он выделит себя, найдет среди своих соседей блок с минимальной меткой, вызовет для него “ProgrammersGuide.GraphPath.BackTrace” и т.д., пока не будет достигнут начальный блок маршрута.

Функции, начиная с модели блока GraphNode и заканчивая последней написанной GraphNode_OnBackTracePath, обращаются к переменным блока, поэтому в каждую из них передается параметр BlockData, и все они располагаются после макроопределений переменных. Теперь, когда макросы для переменных нам больше не нужны, мы можем, как обычно, отменить их:

```
// Отмена макроопределений переменных блока
#undef sPathMark
#undef sMarked
```

```
#undef sInPath
#undef sEnd
#undef sBegin
#undef Ready
#undef Start
#undef pStart
```

Мы написали все функции, необходимые для работы модели, и теперь можно приступать к ее тестированию. Но сначала разберемся, как же она работает.

Представим себе, что в подсистеме собран граф из блоков с этой моделью, и все его блоки находятся в исходном состоянии, то есть все их флаги сброшены. Допустим, пользователь нажимает на каком-то из блоков правую кнопку мыши и выбирает в контекстном меню пункт “Начало маршрута”. Модель блока, в ответ на это, вызывает функцию `GraphNode_SetBlockAsBegin`, которая сбрасывает во всем графе метки и флаги начала маршрута (ничего не изменится – мы считаем, что все блоки и так находятся в исходном состоянии), взводит в своем блоке флаг начала маршрута `sBegin` и вызывает `GraphPath_FindPath`, чтобы найти в графе кратчайший путь. Это ни к чему не приводит – в графе еще не задан флаг конца маршрута, и `GraphPath_FindPath` завершается, не выполнив никаких действий.

Теперь пользователь нажимает правую кнопку мыши на другом блоке и выбирает в контекстном меню пункт “Конец маршрута”. Модель блока вызывает функцию `GraphNode_SetBlockAsEnd`, которая сбрасывает в графе метки и флаги конца маршрута (опять ничего не изменится – взведен только флаг начала маршрута, а его функция не трогает), взводит в своем блоке флаг конца маршрута `sEnd` и опять вызывает `GraphPath_FindPath`. Теперь в графе уже установлены оба флага, поэтому функция `GraphPath_FindPath` вызовет “ProgrammersGuide.GraphPath.Mark” в блоке начала маршрута, передав ему метку 0. Это приведет к цепной реакции: блок примет метку и пометит своих соседей расстояниями до себя, те, в свою очередь, пометят своих соседей, и так будет продолжаться до тех пор, пока все достижимые из начального блока узла графа не будут помечены кратчайшим расстоянием до начала маршрута. После этого `GraphPath_FindPath` вызовет “ProgrammersGuide.GraphPath.BackTrace” у блока конца маршрута. Это тоже приведет к цепочке вызовов функций: блок конца маршрута выделится, взведя у себя флаг `sInPath`, просмотрев метки своих соседей, найдет среди них ближайшего к началу и передаст управление ему, тот – своему соседу и т.д. пока не будет достигнут блок начала маршрута. В результате все блоки на кратчайшем маршруте (а, заодно, и связи, их соединяющие) окажутся выделенными.

Для тестирования модели можно собрать в какой-либо подсистеме граф, соединяя между собой выход `Ready` одного блока со входом `Start` другого (для создания однонаправленной дуги) или два входа `Start` друг с другом (для двунаправленной дуги) связями произвольного вида. Теперь, установив через контекстное меню один из блоков началом, а другой — концом маршрута, можно увидеть кратчайший путь между этими блоками (рис. 89). Выбор в контекстном меню пункта “Сбросить все” вернет граф в исходное состояние.

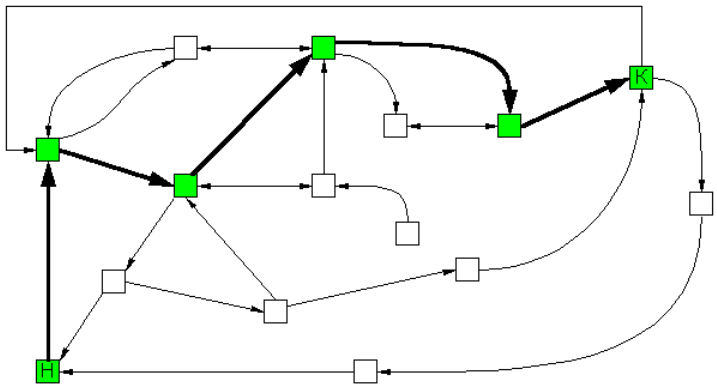


Рис. 89. Кратчайший путь в графе

Конечно, этот пример весьма несовершенен, и его можно существенно улучшить. Можно, например, при разметке графа запоминать в личной области данных каждого блока идентификатор узла графа, от которого пришло значение метки (при вызове функции “ProgrammersGuide.GraphPath.Mark” он передается в поле `Previous` структуры параметров) и связь, соединяющую с ним, тогда при прослеживании маршрута функцией “ProgrammersGuide.GraphPath.BackTrace” не придется перебирать все соседние блоки в поисках узла с наименьшей меткой. Можно вместо используемого упрощенного алгоритма разметки графа реализовать полноценный алгоритм Дейкстры. Но, в любом случае, для взаимодействия узлов между собой нужно будет использовать вызов функций блоков.

§2.13.5. Отложенный вызов функций блоков

Рассматривается отложенный вызов функций блоков, позволяющий избежать переполнения стека при глубокой рекурсии вызовов. В примере, рассмотренном в предыдущем параграфе, прямые вызовы заменяются на отложенные.

Рассмотренный в предыдущем параграфе пример имеет один существенный недостаток, из-за которого созданная модель не будет работать в графах с очень длинными маршрутами. Причина этого в способе, которым мы осуществляем разметку графа.

Когда функция `GraphPath_FindPath` помечает начальный блок маршрута значением 0, функция модели этого блока вызывается в режиме `RDS_BFM_FUNCTIONCALL` для реакции на “ProgrammersGuide.GraphPath.Mark”. Модель блока, реагируя на вызов, вызывает функцию `GraphNode_OnMarkBlock`, которая, начав перебор соседних блоков, помечает первый из них, то есть вызывает его модель для реакции на ту же самую функцию “ProgrammersGuide.GraphPath.Mark”. Эта модель тоже начинает помечать соседние блоки, и опять вызывает модель одного из них. В результате, если в графе существует какой-либо маршрут, состоящий из N блоков, мы получим N вызовов функции модели блока, произведенных один из другого. Поскольку каждый вызов функции занимает в стеке определенное место, тем большее, чем больше у нее параметров и локальных переменных, при размерах графов в несколько тысяч блоков мы можем столкнуться с переполнением стека – его объем не безграничен. Таким образом, попытка найти кратчайший путь между блоками в очень большом графе приведет к аварийному завершению РДС, что будет для пользователя полной неожиданностью, ведь граф, который он нарисовал, умещается в оперативную память, и, следовательно, для его обработки должно хватить ресурсов.

Чтобы избавиться от проблемы переполнения стека из-за большого количества рекурсивных вызовов, можно использовать механизм отложенного вызова функций блоков. При отложенном вызове функция блока не вызывается немедленно, вместо этого ее параметры копируются в специально отведенную область памяти, после чего управление немедленно возвращается вызвавшей функции. До тех пор, пока функция модели не завершится, все отложенные вызовы, сделанные из этой функции, будут ставиться в очередь. Сразу после завершения функции модели отложенные вызовы начнут выполняться один за другим, и, после выполнения каждого из них, память, отведенная под параметры функции, будет освобождаться.

Может показаться, что отложенные вызовы не дают никакого выигрыша: раньше параметры функций при вызове помещались в стек, теперь память под них отводится отдельно, но они все равно продолжают занимать объем тем больший, чем больше функций вызвано. Однако, здесь есть принципиальная разница: при отложенных вызовах память под параметры отводится в динамической области, так называемой “куче” (heap), объем которой, как правило, существенно больше объема стека. Фактически, он ограничен только объемом оперативной памяти, доступной программе. В этой же памяти размещаются данные всех блоков, связей и вспомогательных объектов, созданных РДС, поэтому, если в эту память уместилась созданная пользователем схема, то, вероятнее всего, несколько десятков

дополнительных байтов для каждого из ее блоков (а параметры функций редко занимают больше) не приведут к переполнению памяти. Кроме того, в стек помещаются не только параметры, используемые при вызове функции блока, но и все локальные переменные каждой из выполняемых функций. Избавившись от рекурсивных вызовов модели блока мы, тем самым, перестаем заполнять стек этими локальными переменными. Функции моделей при отложенных вызовах вызываются не друг из друга, а по очереди, поэтому размер стека не увеличивается, сколько бы моделей подряд мы не вызвали: следующая модель вызовется только после завершения предыдущей и освобождения занятой ей памяти в стеке.

Для отложенного вызова функции блока используется сервисная функция `rdsQueueCallBlockFunction`:

```
void RDSCALL rdsQueueCallBlockFunction(
     RDS_BHANDLE Block,    // Блок, у которого вызывается функция
     int FuncId,           // Идентификатор вызываемой функции
     LPVOID ParamBuf,      // Указатель на область параметров
     DWORD ParamBufSize,   // Размер области параметров
     DWORD Flags);         // Флаги
```

В первых трех параметрах этой функции, как и у уже знакомой нам `rdsCallBlockFunction`, передаются идентификатор вызываемого блока, идентификатор вызываемой в нем функции и указатель на начало области параметров этой функции (указатель произвольного типа, то есть `void*`). Дальше начинаются различия: при прямом вызове функций РДС немедленно передает указатель на область параметров в модель вызванного блока и ждет, пока та не вернет управление, поэтому `rdsCallBlockFunction` не требуется знать размер этой области. При отложенном же вызове необходимо сделать копию области параметров функции, а для этого необходимо знать ее размер. По этой причине в четвертом параметре `rdsQueueCallBlockFunction` передается размер области, указатель на которую передан в третьем параметре. В пятом параметре указываются флаги, управляющие постановкой отложенного вызова функции в очередь: флаг `RDS_BCALL_FIRST` поставит вызов в начало очереди, флаг `RDS_BCALL_LAST` – в конец. Один из этих флагов может быть скомбинирован побитовым ИЛИ с уже знакомым нам устаревшим флагом `RDS_BCALL_CHECKSUPPORT`, что позволит перед вызовом функции проверить, поддерживает ли ее данный блок.

При всех достоинствах отложенных вызовов функций блоков у них есть и недостатки, поэтому не во всех случаях они могут заменить прямые вызовы. Прежде всего, параметры, передаваемые при отложенном вызове, не должны содержать внутри себя указателей на какие-либо локальные объекты (строки, структуры, массивы и т.д.), созданные в вызывающей функции. При отложенном вызове вызвавшая функция модели завершится раньше, чем будет произведен вызов, и все ее локальные переменные будут уничтожены, поэтому указатели в скопированной области параметров будут ссылаться на уже освобожденную память. Допустим, например, что мы хотим включить в параметры какой-либо функции блока указатель на строку, и описываем структуру параметров этой функции следующим образом:

```
typedef struct
{ DWORD servSize;     // Сюда будет записан размер структуры
  char *String;       // Указатель на строку
  double Val1,Val2;   // Какие-то другие параметры функции
} TFunction3Params;
```

Будем считать, что эта функция уже зарегистрирована, и ее идентификатор находится в глобальной переменной `Function3Id`. Теперь мы хотим произвести отложенный вызов этой функции у какого-то блока:

```
RDS_BHANDLE block=... // Здесь должен быть идентификатор
                      // вызываемого блока
TFunction3Params params;   // Структура параметров функции
```

```
char buf[100];              // Вспомогательный массив

params.servSize=sizeof(params); // Присваиваем размер структуры
params.Val1=10.0;               // Записываем параметры
params.Val2=15.0;
// Формируем строку во вспомогательном массиве
sprintf(buf,"Val1+Val2=%lf",params.Val1+params.Val2);
// Записываем указатель на строку в структуру параметров
params.String=buf;
// Отложенный вызов функции
rdsQueueCallBlockFunction(block,Function3Id,
          &params,sizeof(params),
          RDS_BCALL_LAST);
```

При вызове `rdsQueueCallBlockFunction` РДС сделает копию структуры `params` (то есть копию области памяти с начальным адресом `&params` и размером `sizeof(params)`) и немедленно вернет управление. Поле `String` скопированной структуры при этом будет ссылаться на массив `buf`, расположенный в стеке функции, фрагмент которой приведен выше. Как только эта функция завершится, массив `buf` будет уничтожен вместе со всеми ее локальными переменными. Теперь поле `String` скопированной структуры ссылается на уничтоженные данные – когда дело дойдет до вызова функции блока, это не приведет ни к чему хорошему.

Может возникнуть соблазн обойти эту проблему следующим образом: отводить строку динамически при помощи сервисной функции `rdsAllocate`, чтобы она не уничтожилась при завершении создавшей ее функции, а обязанность освобождения памяти, занятой строкой, переложить на вызываемую модель блока. То есть изменить приведенный выше текст следующим образом:

```
RDS_BHANDLE block=... // Здесь должен быть идентификатор
                      // вызываемого блока
TFunction3Params params;   // Структура параметров функции
char *buf;                 // Вспомогательный массив
buf=rdsAllocate(100);      // Отводим память под массив
params.servSize=sizeof(params); // Присваиваем размер структуры
params.Val1=10.0;               // Записываем параметры
params.Val2=15.0;
// Формируем строку во вспомогательном массиве
sprintf(buf,"Val1+Val2=%lf",params.Val1+params.Val2);
// Записываем указатель на строку в структуру параметров
params.String=buf;
// Отложенный вызов функции
rdsQueueCallBlockFunction(block,Function3Id,
          &params,sizeof(params),
          RDS_BCALL_LAST);
```

Однако, решив проблему с преждевременным уничтожением строки, мы создадим новую: если окажется, что вызванная модель блока не поддерживает эту функцию, она не освободит память, отведенную под строку, что приведет к утечке памяти. Поэтому лучше всего взять за правило: при отложенном вызове функций – никаких указателей на локальные или динамически создаваемые объекты в параметрах.

Разумеется, в приведенном примере ничто не мешает включить в структуру параметров функции не указатель на строку, а массив символов фиксированного размера:

```
typedef struct
{ DWORD servSize;     // Сюда будет записан размер структуры
  char String[100];   // Строка (массив символов)
  double Val1,Val2;   // Какие-то другие параметры функции
} TFunction3Params;
```

В этом случае структура параметров не содержит указателей на внешние объекты, и никаких проблем при отложенном вызове функции с такими параметрами не будет.

Второй крупный недостаток отложенных вызовов – невозможность получения значения, возвращенного вызванной функцией. rdsQueueCallBlockFunction только готовит данные для отложенного вызова, а сам вызов будет произведен позднее, когда вызвавшая функция уже завершится. Если вызывающая модель хочет получить от вызываемой какой-то ответ, для этого нужно предпринимать специальные действия. Например, вызванная функция модели может считать из поля Caller структуры RDS_FUNCTIONCALLDATA идентификатор вызвавшего ее блока, и вызвать у него в ответ какую-либо специально разработанную для этого функцию, передав в ее параметрах результат выполнения отложенного вызова. Это несколько усложняет создание моделей блоков, поскольку вызов функции и получение ее результата оказываются разнесены во времени.

Как и прямым вызовом, отложенным можно вызывать не только функцию конкретного блока, но и функции всех блоков какой-либо подсистемы. Для этого служит сервисная функция rdsBroadcastFuncCallsDelayed:

```
void RDSCALL rdsBroadcastFuncCallsDelayed(
     RDS_BHANDLE System,   // Подсистема, блоки которогй вызываются
     int FuncId,           // Идентификатор вызываемой функции
     LPVOID ParamBuf,      // Указатель на область параметров
     DWORD ParamBufSize,   // Размер области параметров
     DWORD Flags);         // Флаги
```

Ее параметры аналогичны параметрам функции rdsQueueCallBlockFunction, за исключением того, что в первом параметре передается не идентификатор вызываемого блока, а идентификатор подсистемы, блоки которой вызываются. В флагах этой функции можно указывать не только RDS_BCALL_CHECKSUPPORT, RDS_BCALL_FIRST и RDS_BCALL_LAST, но и уже знакомые нам по rdsBroadcastFunctionCallsEx RDS_BCALL_SUBSYSTEMS и RDS_BCALL_ALLOWSTOP (см. стр. 319).

Вернемся к нашему примеру, в котором мы ищем кратчайший путь между двумя узлами графа. Чтобы перевести функции “ProgrammersGuide.GraphPath.Mark” и “ProgrammersGuide.GraphPath.BackTrace” на отложенный вызов, достаточно изменить всего две из написанных нами вспомогательных функций – GraphPath_FindPath и GraphPath_MarkBlock_Callback. Начнем с первой из них:

```
// Поиск маршрута в графе в заданной подсистеме
void GraphPath_FindPath(RDS_BHANDLE System)
{ RDS_BHANDLE StartBlock,EndBlock;
  TProgGuideFuncMarkParams markparams;

  // Считаем, что маркировка всего графа сброшена

  // Ищем начальную и конечную точку маршрута
  if(!GraphPath_GetTerminalBlocks(System,&StartBlock,&EndBlock))
    return; // Начало или конец не найдены
  // Начало маршрута - StartBlock, конец - EndBlock

  // Маркируем граф от начала маршрута
  markparams.servSize=sizeof(markparams);
  markparams.Mark=0.0;     // Начало маркируется значением 0
  markparams.Previous=NULL;// Это значение не пришло от какого-то
                           // соседнего блока
  // Вызываем функцию маркировки для начального блока
  // rdsCallBlockFunction(StartBlock,GraphFuncMark,&markparams);
```

```
  rdsQueueCallBlockFunction(StartBlock,GraphFuncMark,
                &markparams,sizeof(markparams),RDS_BCALL_FIRST);
  // Теперь отслеживаем кратчайший путь в обратном направлении
  // (от конечного блока)
  // rdsCallBlockFunction(EndBlock,GraphFuncBackTrace,NULL);
  rdsQueueCallBlockFunction(EndBlock,GraphFuncBackTrace,
                NULL,0,RDS_BCALL_LAST);
}
//=======================================
```

В этой функции мы в двух местах заменили `rdsCallBlockFunction` на `rdsQueueCallBlockFunction`, но флаги вызова мы при этом используем разные. Нам нужно сначала разметить весь граф, и только потом вызвать у блока конца маршрута функцию “ProgrammersGuide.GraphPath.BackTrace” для выделения найденного пути. Раньше это получалось само собой: первый вызов `rdsCallBlockFunction` для функции “ProgrammersGuide.GraphPath.Mark” (идентификатор функции находится в переменной `GraphFuncMark`) не возвращал управления до тех пор, пока весь граф не оказывался размеченным, и только потом вызывалась функция выделения пути (идентификатор – в `GraphFuncBackTrace`). Теперь функция `rdsQueueCallBlockFunction` возвращает управление сразу, поэтому нам нужно принять меры, чтобы функция выделения пути “ProgrammersGuide.GraphPath.BackTrace” вызвалась только после того, как прекратятся все отложенные вызовы функции разметки “ProgrammersGuide.GraphPath.Mark”. Для этого достаточно все время ставить функцию разметки в начало очереди, используя флаг `RDS_BCALL_FIRST`, а функцию выделения – в конец, с флагом `RDS_BCALL_LAST`. Таким образом, пока в очереди будет находиться хотя бы один отложенный вызов функции разметки, до вызова функции выделения дело не дойдет.

В функции `GraphPath_MarkBlock_Callback` нам тоже нужно заменить прямой вызов функции разметки на отложенный:

```
// Функция обратного вызова для GraphNode_OnMarkBlock
BOOL RDSCALL GraphPath_MarkBlock_Callback(
           RDS_PPOINTDESCRIPTION src,
           RDS_PPOINTDESCRIPTION dest,LPVOID data)
{ TProgGuideFuncMarkParams *src_params=
           (TProgGuideFuncMarkParams*)data;
  TProgGuideFuncMarkParams dest_params;
  double ArcLen;

  // ...
  // начало функции – без изменений
  // ...

  // Помечаем найденный соседний блок суммой маркировки данного
  //  блока (src_params->Mark) и длины дуги к найденному (ArcLen)
  dest_params.servSize=sizeof(dest_params);
  dest_params.Mark=src_params->Mark+ArcLen;
  dest_params.Previous=src->Block;    // Блок, от которого
                                      // пришла метка
  // rdsCallBlockFunction(dest->Block,GraphFuncMark,&dest_params);
  rdsQueueCallBlockFunction(dest->Block,GraphFuncMark,
                &dest_params,sizeof(dest_params),RDS_BCALL_FIRST);
  return TRUE;
}
//=======================================
```

Здесь мы тоже постоянно ставим вызов функции “ProgrammersGuide.GraphPath.Mark” в начало очереди. Вызов этой функции у какого-либо узла графа добавляет в начало очереди

такие же вызовы для его соседей, при их выполнении в начало очереди добавляются вызовы уже для их соседей и т.д., а единственный вызов функции выделения, сделанный в GraphPath_FindPath, будет оставаться в конце этой очереди.

Вызов функции “ProgrammersGuide.GraphPath.BackTrace” в конце функции GraphNode_OnBackTracePath тоже нужно сделать отложенным:

```
// Проследить и выделить маршрут от данного блока в
// обратном направлении
void GraphNode_OnBackTracePath(RDS_PBLOCKDATA BlockData)
{ RDS_BHANDLE PrevBlock;
  RDS_CHANDLE PrevConn,c;

  // ...
  // начало функции - без изменений
  // ...

  if(PrevBlock)
    { // Визуально выделяем связь
      MarkConnection(PrevConn);
      // Вызываем функцию обратного прослеживания маршрута
      // от найденного блока
      // rdsCallBlockFunction(PrevBlock,GraphFuncBackTrace,NULL);
      rdsQueueCallBlockFunction(PrevBlock,GraphFuncBackTrace,
                NULL,0,RDS_BCALL_LAST);
    }
}
//=======================================
```

Для оставшихся функций блоков, используемых в этом примере, перевод на отложенные вызовы не нужен: они не вызываются рекурсивно и не могут привести к переполнению стека.

Теперь наша модель сможет искать пути даже в очень больших и сильно разветвленных графах. Функции, которые мы перевели на отложенные вызовы, не возвращали никаких значений и их параметры не содержали указателей (функция “ProgrammersGuide.GraphPath.BackTrace” вообще не имеет параметров), поэтому изменения, которые нам пришлось внести, оказались довольно небольшими.

Следует помнить, что основное назначение отложенных вызовов – борьба с переполнением стека из-за глубокой рекурсии. Во всех остальных случаях использование прямых вызовов более целесообразно.

§2.13.6. Регистрация исполнителя функции

Рассматривается механизм регистрации блока в качестве исполнителя какой-либо функции, позволяющий остальным блокам схемы легко находить его идентификатор и вызывать эту функцию. В приводимом примере один блок регистрируется как исполнитель функции вывода сообщения, а другой выводит сообщение с его помощью.

Для того, чтобы вызвать функцию какого-либо блока (не важно, будет ли это прямой вызов или отложенный), необходимо знать его идентификатор. Если в блоке реализована какая-либо функция, к которой должны обращаться блоки в разных местах схемы, то как они узнают идентификатор блока, выполняющего функцию? До сих пор мы вызывали функцию либо у блоков, соединенных связями с вызывающим, либо у всех блоков подсистемы. Можно, конечно, вызвать интересующую нас функцию у всех блоков схемы (для этого нужно вызвать функцию у блоков корневой подсистемы и разрешить вызов вложенных в нее подсистем), тогда, рано или поздно, она будет вызвана у блока, который ее поддерживает, и он выполнит нужные нам действия. Однако, для этого РДС придется перебрать все блоки схемы, что сильно замедлит работу. Кроме того, если в схеме окажется несколько блоков,

выполняющих интересующую нас функцию, придется принимать специальные меры, чтобы только один из этих блоков сработал. Чтобы избежать этих проблем, можно воспользоваться механизмом регистрации блока как исполнителя какой-либо функции, тогда все блоки схемы смогут получить идентификатор этого блока и вызывать функцию непосредственно у него.

Механизм регистрации исполнителя функции и получения доступа к его идентификатору похож на механизм создания динамических переменных и подписки на них (см. §2.6.1). Блок, поддерживающий какую-либо функцию, может зарегистрироваться в РДС как ее исполнитель при помощи вызова `rdsRegisterFuncProvider`. РДС будет помнить факт его регистрации до тех пор, пока блок не отменит ее вызовом `rdsUnregisterFuncProvider` или не будет удален. Блок, которому нужен доступ к этой функции, подписывается на информацию о ее исполнителе вызовом `rdsSubscribeToFuncProvider`. Эта функция создает для вызвавшего ее блока структуру `RDS_FUNCPROVIDERLINK` и возвращает указатель на нее. В поле `Block` этой структуры находится идентификатор ближайшего в иерархии блока, зарегистрировавшегося как исполнитель данной функции (это очень похоже на подписку на динамическую переменную с поиском по иерархии). РДС постоянно отслеживает регистрацию новых блоков и ее отмену, и поддерживает поле `Block` в актуальном состоянии. Если, например, на момент подписки в системе не было зарегистрировано и одного исполнителя данной функции, в этом поле будет находиться значение `NULL`. Если, со временем, зарегистрируется один из блоков в корневой подсистеме, РДС сразу же запишет в поле `Block` его идентификатор. Если потом какой-либо из более близких к подписавшемуся блоков (например, блок в одной подсистеме с ним) тоже зарегистрируется как исполнитель этой функции, в поле `Block` запишется его идентификатор и т.д. Таким образом, подписавшись на информацию об исполнителе функции, блок всегда имеет доступ к идентификатору ближайшего такого исполнителя. Как и в случае динамических переменных, блок может получать информацию только об исполнителях в своей иерархической цепочке: в своей подсистеме, в родительской подсистеме своей подсистемы и т.д. до корневой подсистемы. Об исполнителе в соседней подсистеме блок не узнает.

В качестве примера создадим блок, который будет выводить пользователю сообщение, текст и заголовок которого будут передаваться в параметрах функции. Мы зарегистрируем этот блок в качестве исполнителя функции вывода сообщения, так что любой блок схемы, находящийся в той же иерархической цепочке, сможет найти его и вызвать эту функцию, если ему потребуется сообщить что-то пользователю. Разумеется, гораздо проще было бы вместо вызова функции у какого-то блока выводить сообщение при помощи сервисной функции `rdsMessageBox`, однако, в дальнейшем мы создадим еще один блок, который также будет поддерживать функцию вывода сообщения, но вместо демонстрации этого сообщения пользователю будет записывать его в файл. Таким образом, не переделывая модели блоков, которые будут формировать сообщения, мы сможем легко менять способ вывода этих сообщений, просто заменяя один блок-исполнитель на другой.

Прежде всего, нам нужно придумать имя функции вывода сообщений и описать структуру ее параметров. Функцию мы назовем “ProgrammersGuide.UserMessage”, а в структуру ее параметров включим поле размера для проверки правильности передачи, указатель на строку с текстом сообщения, и целое поле, которое будет определять тип заголовка: 0 – информационное сообщение, 1 – предупреждение, 2 – сообщение об ошибке.

```
// Функция вывода сообщения
#define PROGGUIDEMESSAGEFUNC    "ProgrammersGuide.UserMessage"
// Структура параметров функции
typedef struct
{ DWORD servSize;     // Размер этой структуры
  char *MessageStr;   // Текст сообщения
```

```
  int Level;          // Уровень важности (0, 1 или 2)
} TProgGuideMessageFuncParams;
//=======================================
// Глобальная переменная для идентификатора функции
int MessageFunc=0;
//=======================================
```

Очевидно, функцию с такой структурой параметров нельзя использовать в отложенных вызовах (структура содержит указатель на “постороннюю” строку, см. §2.13.5), но нам это и не потребуется, мы будем использовать только прямой вызов.

Напишем модель блока, который, при вызове этой функции, будет показывать сообщение пользователю:

```
// Блок-исполнитель функции, показывающей сообщение пользователю
extern "C" __declspec(dllexport)
  int RDSCALL MessageFuncBlock_Box(int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{ RDS_PFUNCTIONCALLDATA func;
  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Регистрируем функцию
        if(MessageFunc==0)
          MessageFunc=rdsRegisterFunction(PROGGUIDEMESSAGEFUNC);
        // Объявляем этот блок ее исполнителем
        rdsRegisterFuncProvider(MessageFunc,FALSE);
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        // Отменяем регистрацию данного блока как исполнителя
        rdsUnregisterFuncProvider(MessageFunc);
        break;

      // Вызов функции
      case RDS_BFM_FUNCTIONCALL:
        // Приводим ExtParam к правильному типу
        func=(RDS_PFUNCTIONCALLDATA)ExtParam;
        if(func->Function==MessageFunc)
          { // Вызвана "ProgrammersGuide.UserMessage"
            TProgGuideMessageFuncParams *params=
                      (TProgGuideMessageFuncParams*)(func->Data);
            DWORD icon;
            char *caption;
            // Проверяем наличие параметров и их размер
            if(params==NULL || params->servSize<
                          sizeof(TProgGuideMessageFuncParams))
              break; // Параметров нет или неверный размер
            // В зависимости от уровня важности сообщения
            // устанавливаем иконку и заголовок
            switch(params->Level)
              { case 0:  icon=MB_ICONINFORMATION;
                         caption="Информация";
                         break;
                case 1:  icon=MB_ICONWARNING;
                         caption="Предупреждение";
                         break;
```

```
                    default: icon=MB_ICONERROR;
                             caption="Ошибка";
                  }
                // Показываем сообщение пользователю
                rdsMessageBox(params->MessageStr,caption,icon | MB_OK);
              }
            break;
        }
      return RDS_BFR_DONE;
    }
    //=======================================
```

Модель получилась не особенно сложной. При ее инициализации мы, как обычно, регистрируем функцию “ProgrammersGuide.UserMessage” и записываем полученный идентификатор в глобальную переменную `MessageFunc`. Затем мы вызываем `rdsRegisterFuncProvider`, объявляя тем самым данный блок исполнителем этой функции. В первом параметре `rdsRegisterFuncProvider` передается идентификатор функции (`MessageFunc`), второй параметр важен только при вызове этой сервисной функции из моделей подсистем: значение `TRUE` указывает на то, что подсистема предоставляет доступ к этой функции только своим внутренним блокам, `FALSE` – внутренним блокам и блокам-соседям по родительской подсистеме. Мы вызываем `rdsRegisterFuncProvider` из модели обычного блока, у которого не может быть внутренних блоков, поэтому этот параметр будет проигнорирован РДС, и мы можем передать в нем любое значение. В результате этого вызова все блоки, находящиеся в одной подсистеме с нашим, а также все блоки, для которых эта подсистема находится в цепочке родителей (блоки всех вложенных подсистем, блоки всех подсистем, вложенных во вложенные, и т.д.) смогут получить доступ к этой функции, если подпишутся на идентификатор ее исполнителя.

При очистке (событие `RDS_BFM_CLEANUP`) мы отменяем регистрацию блока-исполнителя функции с идентификатором `MessageFunc` вызовом `rdsUnregisterFuncProvider`. При этом РДС переключит все подписавшиеся блоки на идентификатор другого блока, зарегистрировавшегося как исполнитель этой функции, если, конечно, такой имеется в их цепочке родителей. Если такого блока не окажется, вместо идентификатора исполнителя они будут получать значение `NULL` до тех пор, пока исполнитель снова не появится.

Наконец, мы должны включить в модель блока реакцию на функцию, исполнителем которой мы его регистрируем. В реакции на событие `RDS_BFM_FUNCTIONCALL` мы сравниваем идентификатор вызванной функции с `MessageFunc` и, если они совпали, проверяем наличие (переданный указатель на структуру параметров не должен быть равен `NULL`) и правильность (поле `servSize` структуры параметров должно быть не меньше ожидаемого размера структуры) переданных параметров. Если параметры переданы верно, мы, в зависимости от поля `Level` переданной структуры, записываем во вспомогательную переменную переменную `icon` одну из стандартных констант Windows API, используемых для указания иконок сообщений, а в переменную `caption` – указатель на строку заголовка окна сообщения, соответствующего этой иконке. Затем мы показываем текст, переданный в поле `MessageStr` структуры параметров функции, в стандартном окне сообщения при помощи сервисной функции `rdsMessageBox`. Здесь мы используем сервисную функцию РДС, а не стандартную функцию Windows `MessageBox`, поскольку сервисную функцию можно безопасно вызывать при запущенном расчете: при вызове из потока расчета она, в отличие от ее “тезки” из Windows API, не останавливает работу потока, а возвращает управление немедленно. В качестве заголовка окна мы передаем значение переменной `caption`, в качестве флагов, влияющих на внешний вид окна – значение переменной `icon`,

объединенное битовым ИЛИ с константой MB_OK. Таким образом, на экране появится окно с переданным в параметрах функции текстом, имеющее заголовок и иконку, соответствующие уровню важности сообщения. Окно будет иметь единственную кнопку с надписью “OK”.

Теперь напишем пример модели блока, который будет пользоваться этой функцией. Сделаем блок, который, при поступлении сигнала на вход, будет выводить сообщение, текст и уровень важности которого будет задаваться пользователем в настройках блока. Такой блок можно подключить, например, к сигналу переполнения счетчика, или к блоку сравнения, и он выдаст пользователю заранее определенное сообщение при наступлении соответствующего события. В качестве входного сигнала для выдачи сообщения мы будем использовать стандартный вход запуска модели (это всегда первая переменная блока), поэтому в параметрах блока нам нужно будет включить режим запуска по сигналу (если включить режим запуска каждый такт расчета, блок будет постоянно выдавать сообщения, игнорируя вход запуска). Текст сообщения и уровень его важности мы будем хранить в значениях по умолчанию переменных блока, как мы уже не раз делали (см. стр. 135). Для этого нам потребуется две переменных: строковая для текста и целая для уровня важности. Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Номер переменной*
0	Show	Сигнал	1	Вход	0
1	Ready	Сигнал	1	Выход	1
2	Message	Строка	4	Внутренняя	2
6	Type	int	4	Внутренняя	3

Первую переменную блока, вход запуска модели, мы переименуем в Show, чтобы название переменной лучше отражало назначение этого входа.

Для вывода сообщения нашему блоку необходимо знать идентификатор блока-исполнителя функции “ProgrammersGuide.UserMessage”. Как и при подписке на динамическую переменную, при подписке на идентификатор блока-исполнителя РДС создает во внутренней памяти структуру, содержащую необходимую информацию, и возвращает указатель на нее. Этот указатель необходимо где-то хранить на протяжении всего существования нашего блока, поэтому блоку потребуется личная область данных. Оформим ее в виде класса, в конструкторе и деструкторе которого будем подписываться на идентификатор блока-исполнителя и прекращать эту подписку соответственно:

```
// Личная область данных блока, выводящего сообщение
class TMessageFuncUserData
{ public:
    // Указатель на структуру подписки
    RDS_PFUNCPROVIDERLINK Link;

    // Функция настройки блока
    BOOL Setup(RDS_BHANDLE Block,int NumTypeVar,int NumMessVar);

    // Конструктор класса
    TMessageFuncUserData(void)
      { // Регистрируем функцию
        if(MessageFunc==0)
          MessageFunc=rdsRegisterFunction(PROGGUIDEMESSAGEFUNC);
        // Подписываемся на блок-исполнитель
        Link=rdsSubscribeToFuncProvider(MessageFunc);
      };
```

```
    // Деструктор класса
    ~TMessageFuncUserData()
      { // Прекращаем подписку
        rdsUnsubscribeFromFuncProvider(MessageFunc);
      };
};
//=======================================
```

В поле `Link` описанного нами класса будет храниться указатель на структуру подписки на блок-исполнитель функции. Структура имеет всего два поля:

```
// Структура подписки на блок-исполнитель функции
typedef struct
{ RDS_BHANDLE Block;  // Идентификатор блока-исполнителя
  int FuncId;         // Идентификатор функции
} RDS_FUNCPROVIDERLINK;
typedef RDS_FUNCPROVIDERLINK *RDS_PFUNCPROVIDERLINK;
```

В поле `Block` этой структуры всегда находится либо идентификатор ближайшего по иерархии блока, зарегистрировавшегося как исполнитель данной функции, либо `NULL`, если такого блока нет (РДС всегда поддерживает это поле в актуальном состоянии). В поле `FuncId` хранится идентификатор самой функции. На самом деле, идентификатор функции нам известен и без этой структуры: не зная его, мы не смогли бы подписаться на исполнителя. В структуру он включен только для удобства программирования.

В классе объявлена функция настройки блока, с помощью которой пользователь будет задавать текст и важность сообщения. В функцию передается идентификатор данного блока и порядковые номера переменных, в которых мы храним параметры (эти номера потребуются нам для получения значений переменных по умолчанию). Тело этой функции мы напишем позже.

В конструкторе класса мы регистрируем функцию “ProgrammersGuide.UserMessage”, если она еще не регистрировалась ранее, а затем при помощи вызова `rdsSubscribeToFuncProvider` подписываемся на ее исполнителя. Возвращаемый указатель на структуру `RDS_FUNCPROVIDERLINK` мы записываем в поле класса `Link`. Теперь, если нам потребуется вызвать функцию вывода сообщения, мы сможем использовать конструкцию вида

```
if(Link!=NULL)
  rdsCallBlockFunction(Link->Block,Link->FuncId,...);
```

Сервисная функция `rdsCallBlockFunction` устроена таким образом, что, если вместо идентификатора блока в первом параметре передается значение `NULL`, она немедленно завершится, не выполнив никаких действий, поэтому поле `Link->Block` можно не сравнивать с `NULL` перед ее вызовом. Однако, со значением `NULL` необходимо сравнить само значение поля `Link`, поскольку функция `rdsSubscribeToFuncProvider`, которая вернула нам это значение, возвращает `NULL` в случае ошибки (например, если мы вызовем ее из модели корневой подсистемы, иерархическая цепочка которой не содержит ни одного блока, поэтому и исполнителя функции искать негде).

В деструкторе класса мы отменяем подписку на исполнителя функции при помощи вызова `rdsUnsubscribeFromFuncProvider`. Технически, можно было бы и не делать этого, поскольку при отключении модели от блока все подписки отменяются автоматически, однако явная отмена подписки улучшает читаемость текста программы.

Функция модели блока будет иметь следующий вид:

```
// Блок, выводящий сообщение по сигналу
extern "C" __declspec(dllexport)
  int RDSCALL ShowMessage(int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
```

```
{ // Указатель на личную область данных
  TMessageFuncUserData *data=
                (TMessageFuncUserData*)(BlockData->BlockData);
// Макроопределения для статических переменных блока
#define pShow ((char *)(BlockData->VarData))
#define Show (*((char *)(pShow)))          // 0
#define Ready (*((char *)(pShow+1)))       // 1
#define Message (*((char **)(pShow+2)))    // 2
#define Type (*((int *)(pShow+6)))         // 3

  switch(CallMode)
    { // Инициализация - создание личной области
      case RDS_BFM_INIT:
        BlockData->BlockData=new TMessageFuncUserData();
        break;
      // Очистка - удаление личной области
      case RDS_BFM_CLEANUP:
        delete data;
        break;
      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSAI}")?
                                  RDS_BFR_BADVARSMSG:RDS_BFR_DONE;
      // Настройка блока
      case RDS_BFM_SETUP:
        return data->Setup(BlockData->Block,3,2)?1:0;

      // Такт расчета (реакция на сигнал Show)
      case RDS_BFM_MODEL:
        if(data->Link!=NULL)     // Есть структура подписки
          { TProgGuideMessageFuncParams params;
            // Готовим структуру параметров функции
            params.servSize=sizeof(params);
            params.MessageStr=Message;
            params.Level=Type;
            // Вызываем функцию по данным подписки
            rdsCallBlockFunction(data->Link->Block,
                                    data->Link->FuncId,&params);
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Type
#undef Message
#undef Ready
#undef Show
#undef pShow
}
//=======================================
```

При инициализации и очистке модели мы здесь, как обычно, создаем и уничтожаем личную область данных блока (все действия по регистрации функции и подписке выполняются в конструкторе класса, по прекращению подписки – в деструкторе), а при проверке типов переменных – сравниваем переданную строку со строкой, соответствующей используемому нами набору переменных. Реагируя на вызов в режиме `RDS_BFM_SETUP`, модель вызывает функцию настройки `Setup`, которую нам еще предстоит написать, передавая ей порядковые номера переменных `Type` (3) и `Message` (2). Главные действия, ради которых, собственно,

и создавалась эта модель, производятся в реакции на такт расчета RDS_BFM_MODEL, то есть при поступлении единицы на вход блока Show.

Прежде всего, указатель на структуру подписки, запомненный в поле Link класса личной области данных блока, сравнивается с NULL – равенство укажет на то, что подписка на блок-исполнитель функции принципиально невозможна, и вывести сообщение нельзя. Если поле Link указывает на структуру во внутренней памяти РДС, структура params заполняется параметрами функции (текст сообщения берется из переменной блока Message, важность сообщения – из переменной Type), после чего у блока, идентификатор которого хранится в Link->Params, вызывается функция Link->FuncId. В качестве идентификатора функции можно было бы использовать и значение глобальной переменной MessageFunc, они, очевидно, будут равны.

Нам осталось только написать функцию настройки параметров блока Setup, и модель блока будет готова:

```
// Функция настройки блока
BOOL TMessageFuncUserData::Setup(RDS_BHANDLE Block,
                                 int NumTypeVar,int NumMessVar)
{ RDS_HOBJECT window; // Объект-окно
  BOOL ok;            // Пользователь нажал "OK"
  char *defval;

  // Создание окна
  window=rdsFORMCreate(FALSE,-1,-1,"Сообщение");

  // Важность сообщения
  // Получение значения переменной по умолчанию
  defval=rdsGetBlockVarDefValueStr(Block,NumTypeVar,NULL);
  // Добавление поля - выпадающий список
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_COMBOLIST,"Тип:",150);
  // Установка списка вариантов
  rdsSetObjectStr(window,1,RDS_FORMVAL_LIST,
                           "Информация\nПредупреждение\nОшибка");
  // Установка текущего значения поля
  rdsSetObjectStr(window,1,RDS_FORMVAL_ITEMINDEX,defval);
  // Освобождение defval
  rdsFree(defval);

  // Текст сообщения
  // Получение значения переменной по умолчанию
  defval=rdsGetBlockVarDefValueStr(Block,NumMessVar,NULL);
  // Добавление поля - многострочное
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_MULTILINE,"Текст:",80);
  // Установка текущего значения поля
  rdsSetObjectStr(window,2,RDS_FORMVAL_VALUE,defval);
  // Установка высоты поля в точках экрана (~3 строки)
  rdsSetObjectInt(window,2,RDS_FORMVAL_MLHEIGHT,3*24);
  // Освобождение defval
  rdsFree(defval);

  // Открытие окна
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Пользователь нажал OK - запись измененных параметров
      defval=rdsGetObjectStr(window,1,RDS_FORMVAL_ITEMINDEX);
      rdsSetBlockVarDefValueStr(Block,NumTypeVar,defval);
      defval=rdsGetObjectStr(window,2,RDS_FORMVAL_VALUE);
```

```
        rdsSetBlockVarDefValueStr(Block,NumMessVar,defval);
      }
    // Уничтожение окна
    rdsDeleteObject(window);
    return ok;
  }
  //=======================================
```

Эта функция похожа на другие функции настройки, которые мы уже не раз делали. Единственное новшество в ней – использование многострочного поля ввода `RDS_FORMCTRL_MULTILINE` для текста сообщения и установка высоты этого поля вызовом `rdsSetObjectInt` с константой `RDS_FORMVAL_MLHEIGHT`.

Теперь можно приступать к тестированию модели. Необходимо подключить ее к блоку с указанной выше структурой переменных, включить в параметрах этого блока запуск только по сигналу (см. рис. 5) и разрешить функцию настройки (см. рис. 7). К входу `Show` блока подключим выход `Click` стандартной кнопки (нажатие на эту кнопку будет выводить сообщение), а в настройках нашего блока зададим тип (важность) и текст сообщения (рис. 90). Нам также понадобится блок-исполнитель функции с созданной нами моделью `MessageFuncBlock_Box`: можно разместить его в той же подсистеме, что и блок с кнопкой, либо в одной из подсистем в иерархической цепочке, например, в корневой. Теперь, если запустить расчет и нажать на кнопку, подключенную к блоку, на экране появится сообщение, выведенное блоком-исполнителем (рис. 91).

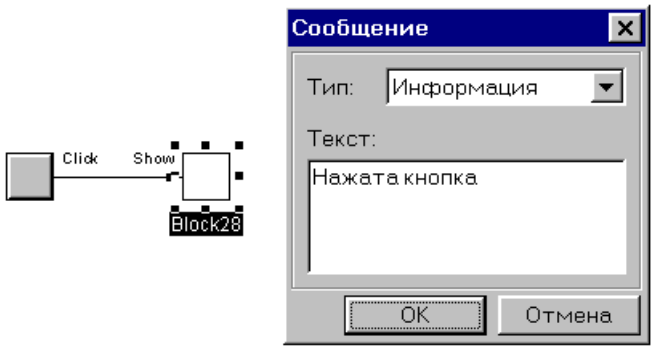


Рис. 90. Блок, выводящий сообщение по сигналу, и окно его настройки

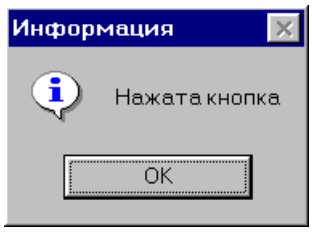


Рис. 91. Сообщение, выводимое блоком

Теперь сделаем другой блок-исполнитель для той же самой функции “ProgrammersGuide.UserMessage”. Вместо немедленной демонстрации сообщения пользователю модель блока будет записывать его в файл, имя которого будет задаваться в настройках блока. Кроме того, чтобы этот файл не разрастался до бесконечности, сделаем в настройках блока возможность включения автоматической очистки этого файла при загрузке схемы. Параметры блока мы будем хранить в значениях статических переменных по умолчанию: имя файла – в строковой переменной `FileName`, флаг необходимости очистки при загрузке схемы – в логической `ClearOnLoad`. Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Номер переменной*
0	Start	Сигнал	1	Вход	0
1	Ready	Сигнал	1	Выход	1
2	FileName	Строка	4	Внутренняя	2
6	ClearOnLoad	Логический	1	Внутренняя	3

Сначала напишем функцию настройки блока. В ее параметрах, как и у остальных функций настройки, работающих со значениями переменных по умолчанию, необходимо передавать идентификатор блока и порядковые номера переменных, с которыми она будет работать:

```
// Функция настройки блока вывода сообщения в файл
BOOL MessageFuncBlockFileSetup(
     RDS_BHANDLE Block,    // Идентификатор блока
     int NumFileVar,       // Номер переменной имени файла
     int NumClearVar)      // Номер переменной флага очистки
{ RDS_HOBJECT window; // Идентификатор объекта-окна
  BOOL ok;            // Пользователь нажал "OK"
  char *defval;

  // Создание окна
  window=rdsFORMCreate(FALSE,-1,-1,"Запись в файл");

  // Поле ввода для имени файла
  // Чтение значения переменной по умолчанию
  defval=rdsGetBlockVarDefValueStr(Block,NumFileVar,NULL);
  // Поле ввода - выбор файла с диалогом сохранения
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_SAVEDIALOG,"Файл:",300);
  // Фильтр типов файлов для диалога сохранения
  rdsSetObjectStr(window,1,RDS_FORMVAL_LIST,
     "Текстовые файлы (*.txt)|*.txt\nВсе файлы|*.*");
  // Запись значения в поле ввода
  rdsSetObjectStr(window,1,RDS_FORMVAL_VALUE,defval);
  // Освобождение defval
  rdsFree(defval);

  // Очистка при загрузке схемы
  // Чтение значения переменной по умолчанию
  defval=rdsGetBlockVarDefValueStr(Block,NumClearVar,NULL);
  // Поле ввода - флаг
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_CHECKBOX,
     "Очищать при загрузке схемы",0);
  // Запись значения в поле ввода
  rdsSetObjectStr(window,2,RDS_FORMVAL_VALUE,defval);
  // Освобождение defval
  rdsFree(defval);

  // Открытие окна
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Пользователь нажал OK - запись измененных значений
      defval=rdsGetObjectStr(window,1,RDS_FORMVAL_VALUE);
      rdsSetBlockVarDefValueStr(Block,NumFileVar,defval);
      defval=rdsGetObjectStr(window,2,RDS_FORMVAL_VALUE);
      rdsSetBlockVarDefValueStr(Block,NumClearVar,defval);
    }
  // Уничтожение окна
  rdsDeleteObject(window);
  return ok;
}
//=======================================
```

В этой функции мы применяем не встречавшийся раньше тип поля ввода: указание имени файла с кнопкой вызова диалога сохранения (тип `RDS_FORMCTRL_SAVEDIALOG`, см. рис. 92). Кроме имени файла, в него необходимо занести список шаблонов имен файлов, доступных пользователю в выпадающем списке в стандартном диалоге сохранения. Этот

список передается в поле в виде строки обычной сервисной функцией `rdsSetObjectStr` с константой `RDS_FORMVAL_LIST`. Каждый элемент выпадающего списка состоит из текста, который видит пользователь, и шаблона имени (с использованием обычных метасимволов “*” и “?”), разделенных символом вертикальной черты “|”. Можно указать несколько шаблонов, разделив их точкой с запятой. Элементы списка отделяются друг от друга символом перевода строки, который в языке С записывается как “\n”. В нашем случае в списке будет два элемента: “Текстовые файлы (*.txt)” (с шаблоном “*.txt”) и “Все файлы” (с шаблоном “*.*”). Если бы мы, например, захотели, чтобы при выборе в диалоге сохранения варианта “Текстовые файлы” отображались файлы не только с расширением “txt”, но и с расширением “log”, вызов установки списка шаблонов выглядел бы так:

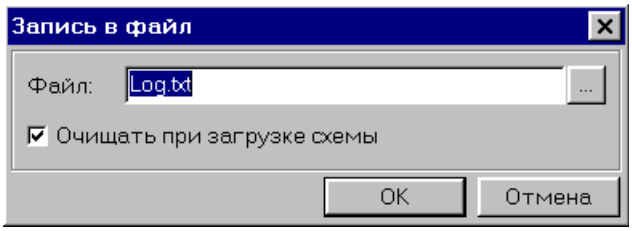


Рис. 92. Окно настройки с полем выбора файла

```
// Фильтр типов файлов для диалога сохранения
rdsSetObjectStr(window,1,RDS_FORMVAL_LIST,
   "Текстовые файлы|*.txt;*.log\nВсе файлы|*.*");
```

В остальном эта функция не отличается от других функций настройки, которые нам уже приходилось делать.

Теперь напишем модель блока:

```
// Блок-исполнитель функции, выводящий сообщение в файл
extern "C" __declspec(dllexport)
  int RDSCALL MessageFuncBlock_File(int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{ RDS_PFUNCTIONCALLDATA func;
  char *fullpath;
// Макроопределения для статических переменных блока
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))              // 0
#define Ready (*((char *)(pStart+1)))            // 1
#define FileName (*((char **)(pStart+2)))        // 2
#define ClearOnLoad (*((char *)(pStart+6)))      // 3
  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Регистрация функции
        if(MessageFunc==0)
          MessageFunc=rdsRegisterFunction(PROGGUIDEMESSAGEFUNC);
        // Объявляем данный блок ее исполнителем
        rdsRegisterFuncProvider(MessageFunc,FALSE);
        break;
      // Очистка
      case RDS_BFM_CLEANUP:
        // Отмена регистрации блока как исполнителя
        rdsUnregisterFuncProvider(MessageFunc);
        break;
      // Проверка типов статических переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSAL}")?
                          RDS_BFR_BADVARSMSG:RDS_BFR_DONE;
```

```
// Вызов функции настройки
case RDS_BFM_SETUP:
  return MessageFuncBlockFileSetup(BlockData->Block,2,3)?1:0;

// Загрузка схемы только что завершилась
case RDS_BFM_AFTERLOAD:
  if(ClearOnLoad)    // Включена очистка файла при загрузке
    { // Формируем в fullpath полный путь к файлу, т.к.
      // введенный пользователем может быть неполным
      fullpath=rdsGetFullFilePath(FileName,NULL,NULL);
      if(fullpath)   // Полный путь существует
        { DeleteFile(fullpath); // Удаляем файл
          rdsFree(fullpath);    // Освобождаем память
        }
    }
  break;

// Вызов функции
case RDS_BFM_FUNCTIONCALL:
  func=(RDS_PFUNCTIONCALLDATA)ExtParam;
  if(func->Function==MessageFunc)
    { // Вызвана наша функция - приводим параметры
      // к правильному типу
      TProgGuideMessageFuncParams *params=
                (TProgGuideMessageFuncParams*)(func->Data);
      char *levelstr,*fullpath;
      if(params==NULL || params->servSize<
                     sizeof(TProgGuideMessageFuncParams))
        break;  // Параметров нет или неверный размер
      // Записываем в переменную levelstr строку,
      // соотвествующую важности сообщения
      switch(params->Level)
        { case 0:  levelstr="Информация";
                   break;
          case 1:  levelstr="Предупреждение";
                   break;
          default: levelstr="Ошибка";
        }
      // Формируем в fullpath полный путь к файлу
      fullpath=rdsGetFullFilePath(FileName,NULL,NULL);
      if(fullpath) // Полный путь сформирован
        { HANDLE h;
          // Открываем файл fullpath на запись
          h=CreateFile(fullpath,GENERIC_WRITE,0,NULL,
               OPEN_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);
          if(h!=INVALID_HANDLE_VALUE)
            { // Файл открыт
              char buf[100]; DWORD temp; SYSTEMTIME time;
              // Перемещаем указатель файла в конец
              SetFilePointer(h,0,NULL,FILE_END);
              // Получаем текущую дату и время
              GetLocalTime(&time);
              // Формируем строку с датой и временем в buf
              sprintf(buf,"%02d-%02d-%04d %02d:%02d:%02d "
                "%s: ",time.wDay,time.wMonth,time.wYear,
                time.wHour,time.wMinute,time.wSecond,
                levelstr);
```

```
                        // Записываем дату, время и важность
                        WriteFile(h,buf,strlen(buf),&temp,NULL);
                        // Записываем текст сообщения
                        if(params->MessageStr)
                          WriteFile(h,params->MessageStr,
                               strlen(params->MessageStr),&temp,NULL);
                        // Записываем перевод строки
                        WriteFile(h,"\r\n",2,&temp,NULL);
                        // Закрываем файл
                        CloseHandle(h);
                      }
                    // Освобождаем память, отведенную под полный путь
                    rdsFree(fullpath);
                  }
              }
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef ClearOnLoad
    #undef FileName
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Реакции этой модели на события `RDS_BFM_INIT` и `RDS_BFM_CLEANUP` в точности совпадают с соответствующими реакциями модели `MessageFuncBlock_Box` – как исполнитель функции “ProgrammersGuide.UserMessage”, этот блок выполняет те же самые действия по регистрации функции и ее отмене. В реакциях на `RDS_BFM_VARCHECK` и `RDS_BFM_SETUP` тоже нет ничего принципиально нового. Две оставшихся реакции мы рассмотрим подробно.

Как только загрузка схемы, в состав которой входит данный блок, завершится, модель блока будет вызвана в режиме `RDS_BFM_AFTERLOAD`. В этот момент модель должна проверить, включена ли в настройках блока очистка файла сообщений при загрузке схемы и, если этот так, стереть этот файл. Таким образом, если значение переменной `ClearOnLoad` не нулевое, мы должны удалить файл с именем `FileName`. Однако, не все так просто: в переменной `FileName` может содержаться только имя файла без пути. Например, если файл находится в одной папке со схемой, поле ввода `RDS_FORMCTRL_SAVEDIALOG`, которое мы использовали в окне настройки, автоматически отбросит этот путь, оставив только имя файла. Для удаления файла необходимо знать полный путь к нему, поэтому сначала нам необходимо воспользоваться сервисной функцией `rdsGetFullFilePath`. Эта функция формирует в динамической памяти строку, содержащую полный путь к файлу, имя которого передано в ее первом параметре. Функция автоматически подставляет вместо стандартных обозначений путей, используемых в РДС (“DLL”, “INI” и т.д.) реальные пути к соответствующим папкам. Если в имени файла, переданном в функцию, отсутствует путь, функция добавит к имени путь по умолчанию, указанный в ее втором параметре, либо, если этот параметр равен `NULL`, путь к файлу схемы. Нам нужен именно последний вариант, поэтому во втором параметре мы передаем `NULL`. В третьем параметре можно передать указатель на целое число, в которое функция запишет длину сформированной строки, но нам это не нужно, поэтому третий параметр в вызове `rdsGetFullFilePath` тоже равен `NULL`.

Указатель на сформированную функцией строку с полным путем к файлу записывается во вспомогательную переменную `fullpath`. Если `fullpath` не равен `NULL`,

то есть если полный путь удалось сформировать, мы удаляем файл функцией Windows API DeleteFile и освобождаем память, занятую строкой, при помощи rdsFree.

Если у блока вызвана функция (режим RDS_BFM_FUNCTIONCALL), и идентификатор этой функции совпадает со значением переменной MessageFunc, в которую мы записали идентификатор нашей функции "ProgrammersGuide.UserMessage" при регистрации, модель проверяет наличие и размер переданной вместе с функцией структуры параметров и завершается, если параметров нет или размер недостаточен. В противном случае во вспомогательную переменную levelstr записывается указатель на строку "Информация", "Предупреждение" или "Ошибка" в зависимости от значения поля Level переданной структуры параметров (точно так же мы выбирали заголовок сообщения в модели MessageFuncBlock_Box на стр. 377, записывая указатель в переменную caption). Затем мы уже описанным способом формируем полный путь к файлу, в конец которого нам предстоит дописать сообщение. Если путь сформирован, мы открываем этот файл на запись функцией Windows API CreateFile и перемещаем указатель в его конец функцией SetFilePointer. Теперь все, что мы запишем в файл, будет добавлено после уже имеющихся в нем данных.

Чтобы пользователь мог понять, когда произошло то или иное событие, сообщение о котором записано в файле, нам следует добавить к записываемому тексту текущую дату и время. Для этого мы используем функцию Windows API GetLocalTime, которая заполняет структуру time типа SYSTEMTIME. Затем мы, используя стандартную функцию sprintf (для ее использования должен быть включен файл заголовков "stdio.h"), формируем во вспомогательном массиве buf строку, содержащую дату и время в привычном для пользователя виде, после которых добавлена строка levelstr, характеризующая важность сообщения. Строка из массива buf записывается в файл функцией Windows API WriteFile, за ней записывается текст сообщения, переданный в поле MessageStr структуры параметров вызванной функции, а за ним – коды перевода строки "\r\n". Затем файл закрывается функцией CloseHandle, а память, отведенная под строку полного пути fullpath, как обычно, освобождается сервисной функцией rdsFree.

Следует отметить, что при ведении какого-либо журнала сообщений, как в нашем случае, лучше всего работать именно в таком режиме: открыть файл, дописать в него новое сообщение, и снова закрыть его. Если бы мы открыли файл при поступлении первого сообщения, и не закрывали бы его до завершения работы со схемой, у пользователя могли бы возникнуть проблемы с доступом к этому файлу, его удалением и т.п. Кроме того, если открывать файл только для записи в него очередного сообщения, несколько разных схем смогут работать с одним и тем же файлом (кончено, при этом лучше отключить очистку файла при загрузке). Постоянные открытия-закрытия будут замедлять работу схемы, но, если сообщения в системе возникают не очень часто, на это можно не обращать внимания.

Если теперь заменить блок с моделью MessageFuncBlock_Box на блок с только что написанной нами моделью, разрешить в его параметрах вызов функции настройки и указать в ней какой-нибудь файл для записи сообщений, то при нажатии на кнопку в схеме, изображенной на рис. 90, в файл будет записана строка следующего вида:

```
04-07-2010 14:30:03 Информация: Нажата кнопка
```

Каждое новое нажатие на кнопку будет добавлять в файл новую строку.

Может возникнуть вопрос: зачем нам понадобилось делать новую модель для вывода сообщений в файл? Не лучше ли было бы добавить эту возможность в модель MessageFuncBlock_Box, и сделать в ее настройках возможность выбора – показывать сообщение пользователю или записывать его в файл? Дело в том, что в нашем случае мы имеем доступ к модели MessageFuncBlock_Box и можем, при желании, переделать ее. Однако, если другому разработчику понадобится реализовать какой-либо другой способ обработки сообщений (например, отправлять их по электронной почте), у него будет

единственный выход – написать свою собственную модель блока-исполнителя функции “ProgrammersGuide.UserMessage”. После этого ему достаточно будет заменить блок-исполнитель, имеющийся в схеме, на свой, и обработка сообщений в схеме изменится. При этом он может разместить свой блок не в корневой подсистеме, а в одной из вложенных, тогда только блоки этой подсистемы и входящих в нее дочерних подсистем будут пользоваться новым блоком-исполнителем, а сообщения всех остальных блоков будут обрабатываться по-старому.

§2.14. Программное управление расчетом

Рассматриваются сервисные функции, позволяющие останавливать, перезапускать и сбрасывать расчет. С их помощью можно выполнять сложные итерационные вычисления.

§2.14.1. Запуск и остановка расчета

Описывается использование сервисных функций для запуска и остановки расчета. Приводится пример блока, запускающего и останавливающего расчет при нажатии кнопки мыши на его изображении, а также останавливающего расчет поступлении сигнала на вход.

Из всех функций управления расчетом чаще всего используется функция остановки расчета `rdsStopCalc`. Обычно пользователь запускает расчет самостоятельно, после чего один из блоков схемы останавливает его, когда результат расчета получен. Например, стандартный блок управления динамическим расчетом (планировщик, см. стр. 87) может останавливать расчет по истечении заданного в его настройках времени, стандартный блок параметрической оптимизации останавливает расчет тогда, когда значения оптимизируемых параметров определены с заданной точностью, и т.д. Необходимость в программном запуске расчета функцией `rdsStartCalc` возникает значительно реже, однако, в некоторых случаях, она позволяет сделать работу со схемой более удобной. Например, если моделируемая система состоит из нескольких схем, работающих на разных машинах и обменивающихся данными по сети, имеет смысл при запуске расчета в одной из таких схем автоматически запускать расчет во всех остальных, иначе пользователю придется постоянно перемещаться от машины к машине и запускать расчет каждой из схем вручную. Возможны и другие случаи, когда программный запуск расчета может быть полезен: блок может запустить расчет сразу после загрузки схемы, при срабатывании какого-либо аппаратного устройства и т.п.

Функции `rdsStartCalc` и `rdsStopCalc` не имеют параметров, поэтому работать с ними несложно: для того, чтобы запустить расчет, нужно вызвать `rdsStartCalc`, чтобы остановить – `rdsStopCalc`. Следует только помнить, что расчет запускается и останавливается не мгновенно, вызов этих функций только начинает процедуру запуска или остановки. Вызвав `rdsStopCalc`, не следует ожидать, что, когда функция вернет управление вызвавшей программе, расчет будет уже остановлен. На самом деле расчет остановится только после завершения текущего такта, когда модели всех блоков сработают и данные передадутся по связям. Функция `rdsStartCalc` тоже не запускает расчет немедленно, он не запустится, по крайней мере, до тех пор, пока не завершится вызвавшая функцию модель блока (возможно, перед запуском расчета будут выполнены еще какие-то действия, например, отложенные вызовы функций блоков, описывавшиеся в §2.13.5).

Напишем модель, которая будет запускать и останавливать расчет по щелчку мыши на изображении блока, а также останавливать расчет по сигналу, поступившему на вход (очевидно, запускать расчет по сигналу не получится, поскольку при остановленном расчете данные по связям не передаются). Поскольку никаких других входов, кроме сигнального входа остановки расчета, у нашего блока не будет, в качестве этого входа можно использовать фиксированный сигнал запуска модели блока. Обычно он называется `Start`,

но мы переименуем его в Stop, поскольку подача на него сигнала будет приводить к остановке расчета. Таким образом, блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Начальное значение*
0	Stop	Сигнал	1	Вход	0
1	Ready	Сигнал	1	Выход	0

Выходной сигнал Ready нам, в данном случае, не нужен, но этот выход должен обязательно присутствовать в любом простом блоке РДС, поэтому удалить его мы не сможем. При поступлении сигнала на вход Stop модель автоматически запустится, поэтому в реакции на такт расчета RDS_BFM_MODEL нужно просто вызвать функцию rdsStopCalc без каких-либо проверок. Разумеется, в параметрах блока нужно отключить флаг “Запуск каждый такт”, иначе он будет останавливать расчет постоянно, независимо от значения входа Stop.

Сама модель будет достаточно простой:

```
extern "C" __declspec(dllexport)
  int RDSCALL StopCalc(int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{ switch(CallMode)
    { case RDS_BFM_MODEL:       // Один такт расчета
        rdsStopCalc();
        break;
      case RDS_BFM_MOUSEDOWN:   // Нажатие кнопки мыши
      case RDS_BFM_SETUP:       // Вызов функции настройки
        if(rdsCalcProcessIsRunning()) // Расчет сейчас запущен
          rdsStopCalc();
        else    // Расчет сейчас остановлен
          rdsStartCalc();
        break;
    }
  return RDS_BFR_DONE;
}
//=======================================
```

В такте расчета, как и было написано выше, мы просто останавливаем расчет: если модель запустилась в этом режиме, значит, на первый сигнальный вход пришел сигнал, а этот вход мы используем для остановки расчета. Мы не обнуляем сигнал Stop после срабатывания, как мы это делали с другими сигнальными входами, поскольку первый сигнальный вход блока обнуляется автоматически при запуске модели. Если пользователь нажал на изображении блока кнопку мыши (режим RDS_BFM_MOUSEDOWN) или в режиме редактирования вызвал функцию настройки (RDS_BFM_SETUP), вызывается сервисная функция rdsCalcProcessIsRunning, которая возвращает TRUE, если расчет в данный момент работает, и FALSE, если он остановлен. Если расчет работает, мы его останавливаем, если нет – запускаем. Таким образом, нажатие кнопки мыши будет то запускать, то останавливать расчет.

Реакция на вызов функции настройки добавлена в модель для того, чтобы блок мог запустить расчет в режиме редактирования. Если разрешить в параметрах блока с этой моделью вызов функции настройки и назвать ее “Запустить расчет”, выбор пункта контекстного меню с этим названием приведет к вызову модели в режиме RDS_BFM_SETUP. В этот момент, поскольку РДС находится в режиме редактирования, расчет работать не будет, rdsCalcProcessIsRunning вернет FALSE, и расчет будет запущен вызовом rdsStartCalc.

Для тестирования этого блока можно подключить к его входу Stop выход Click стандартной кнопки из библиотеки РДС (см. рис. 93) и разрешить в параметрах блока

реакцию на мышь. Теперь нажатие на кнопку при работающем расчете приведет к его остановке. Нажатие на изображение блока будет запускать расчет, если РДС находится в режиме моделирования, или останавливать его, если РДС находится в режиме расчета.

Рис. 93. Тестирование блока запуска и остановки расчета

§2.14.2. Сброс подсистемы в начальное состояние

Описывается программный сброс расчета, то есть способ возврата какой-либо подсистемы или всей схемы в начальное состояние. Рассматриваются два примера, использующих программный сброс расчета: поиск угла возвышения метательной машины, при котором снаряд летит на заданную дальность, и построение графика зависимости дальности полета снаряда от угла возвышения этой машины.

Сброс расчета – совершенно необходимая операция при работе с некоторыми видами схем. Например, если схема моделирует какой-либо протяженный во времени процесс, перед его расчетом с новыми параметрами следует сбросить все блоки в начальное состояние. Часто возникают задачи, в которых необходимо многократно повторять расчет для получения нужного результата. К таким задачам относятся, например, задачи оптимизации, в которых требуется подобрать такие параметры схемы, при которых она ведет себя наилучшим образом. Что такое “вести себя наилучшим образом” обычно формально задается при помощи вычисления некоторого критерия, позволяющего оценить качество поведения системы. Многократно повторяя расчет с разными значениями параметров, находят такой их набор, при котором значение этого критерия будет минимальным (или максимальным – все зависит от того, как задать критерий). Другой пример задачи, требующей многократного моделирования – построение графика зависимости установившегося значения на выходе схемы от одного из ее параметров. Поскольку значение на выходе схемы устанавливается не сразу, для каждого значения параметра нужно запускать расчет на некоторое время, затем фиксировать очередную точку графика, сбрасывать расчет, менять параметр, снова запускать расчет на некоторое время и т.д. Естественно, проводить все эти сбросы и перезапуски вручную было бы слишком трудоемко, такие процессы необходимо автоматизировать.

РДС позволяет программно вернуть всю схему или отдельную ее подсистему в начальное состояние вызовом сервисной функции `rdsResetSystemState`. При этом все статические переменные блоков вернутся к своим значениям по умолчанию и модели всех блоков подсистемы вызовутся в режиме `RDS_BFM_RESETCALC`. Динамические переменные сброшены не будут – если сбрасывать их необходимо, это должна сделать она из моделей блоков. Как и функции запуска и остановки расчета, функция сброса обычно срабатывает не мгновенно – при работающем расчете она будет ждать завершения очередного такта.

Для иллюстрации применения этой функции сделаем модель метательной машины, которая выпускает цилиндрический снаряд известной массы и диаметра под углом к горизонту с некоторой начальной скоростью. Необходимо найти такой угол возвышения (то есть угол между горизонтальной плоскостью и осью канала ствола или направляющей нашей метательной машины), при котором снаряд упадет на землю как можно ближе к точке, находящейся на заданном расстоянии от машины. Поскольку уравнения, описывающие полет снаряда с учетом сопротивления воздуха, не решаются аналитически, нам необходимо будет сделать блок, который, многократно запуская расчет, подберет такой угол возвышения, при котором дальность полета снаряда будет как можно ближе к заданной.

Для решения этой задачи нам потребуется создать два блока: модель полета снаряда (расчет внешней баллистики) и блок, который будет подбирать угол возвышения для заданной дальности полета снаряда. Сначала займемся блоком, который будет моделировать полет снаряда, выпущенного метательной машиной. В этом примере мы не будем глубоко вдаваться в тонкости баллистики: будем считать, что угол возвышения и угол бросания

совпадают, и опустим некоторые другие детали, не особенно существенные для данного примера. Чтобы рассчитать траекторию полета снаряда, нам необходимо получить дифференциальные уравнения, описывающие ее, и численно проинтегрировать их.

Пусть наша метательная машина имеет ствол, который может поворачиваться относительно горизонтальной оси, расположенной на высоте h_0 от поверхности земли (рис. 94). Расстояние от оси вращения ствола до его конца (точки вылета снаряда) назовем l_0. Полет снаряда мы будем рассматривать в двумерной системе координат "высота-дальность",

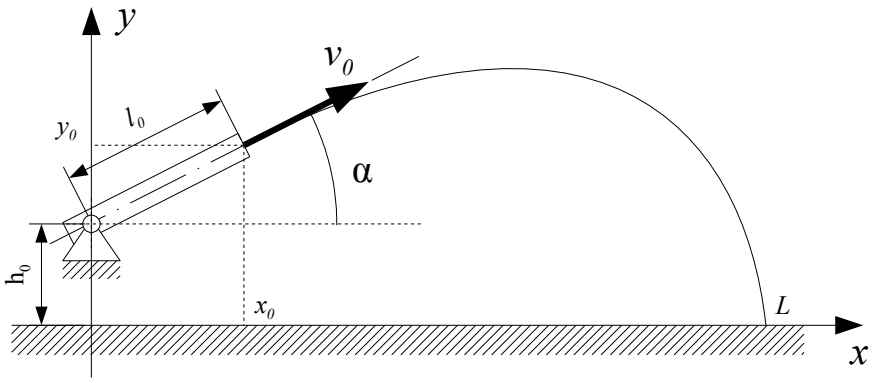


Рис. 94. Метательная машина в системе координат

как на рисунке: ось x будет лежать на поверхности земли, а ось y будет направлена вертикально вверх, проходя через точку поворота ствола. Будем считать, что при любом угле возвышения α наша метательная машина обеспечивает снаряду при вылете из ствола заданную начальную скорость v_0, которая не зависит от угла возвышения. Таким образом, в точке вылета (x_0, y_0) снаряд будет иметь скорость v_0, вектор которой направлен под углом α к горизонту. Координаты точки вылета можно вычислить следующим образом:

$$\begin{cases} x_0 = l_0 \cos\alpha \\ y_0 = h_0 + l_0 \sin\alpha \end{cases}$$

С момента вылета из ствола на снаряд будут действовать всего две силы: сила тяжести mg, направленная вертикально вниз (m – масса снаряда), и сила сопротивления воздуха F, направленная против вектора мгновенной скорости снаряда v (рис. 95).

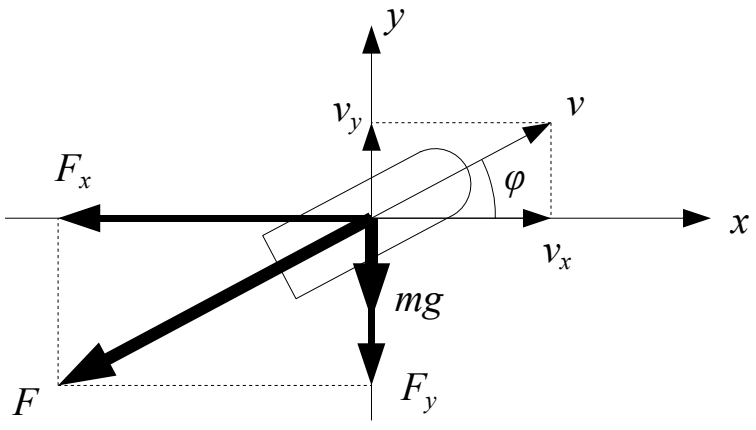


Рис. 95. Силы, действующие на снаряд в полете

Величина силы сопротивления воздуха F зависит от модуля вектора скорости снаряда. Для ее вычисления мы будем пользоваться формулой, полученной Н. А. Забудским [15, 16]:

$$F = A(v)\pi R^2 \frac{\Delta}{\Delta_0} v^{n(v)}$$

где R – радиус поперечного сечения снаряда,
Δ – плотность воздуха в момент выстрела,
Δ_0 – плотность воздуха при температуре 15ºC и давлении 760 мм.рт.ст.,
v – скорость снаряда,
A, n – параметры, зависящие от скорости снаряда следующим образом:

Диапазон v, м/с	*A*	*n*
0...240	0.014	2
240...295	0.0000583	3
295...375	0.00000000067	5
375...419	0.000094	3
419...550	0.0394	2
550...800	0.2616	1.7
800...1000	0.713	1.55

Для простоты мы не будем учитывать разницу плотностей воздуха – будем считать, что мы стреляем именно при температуре 15ºC и давлении 760 мм. Заменив в формуле радиус поперечного сечения снаряда R на его диаметр D (мы договорились задавать именно диаметр), получим:

$$F = 0.25\pi A(v) D^2 v^{n(v)}$$

Разложим силу сопротивления воздуха F на горизонтальную и вертикальную составляющие F_x и F_y. Считая, что вектор мгновенной скорости снаряда v в данный момент направлен под углом φ к горизонту, то есть имеет горизонтальную составляющую $v_x = v\cos\varphi$ и вертикальную $v_y = v\sin\varphi$, получим:

$$\begin{cases} F_x = F\cos\varphi = F\dfrac{v_x}{v} \\ F_y = F\sin\varphi = F\dfrac{v_y}{v} \end{cases}$$

Обозначив через a_x и a_y горизонтальную и вертикальную составляющие ускорения снаряда соответственно, из второго закона Ньютона получим:

$$\begin{cases} ma_x = -F_x \\ ma_y = -mg - F_y \end{cases}$$

Поскольку ускорение – это первая производная скорости по времени, получаем следующие дифференциальные уравнения:

$$\begin{cases} m\dfrac{dv_x}{dt} = -F_x \\ m\dfrac{dv_y}{dt} = -mg - F_y \end{cases}$$

Нулевым моментом времени ($t = 0$) будем считать момент вылета снаряда из ствола. В этом случае начальными условиями для приведенных выше уравнений будут горизонтальная и вертикальная составляющие вектора начальной скорости v_0 (см. рис. 94):

$$\begin{cases} v_x(0) = v_0\cos\alpha \\ v_y(0) = v_0\sin\alpha \end{cases}$$

Нам нужно рассчитать траекторию полета снаряда, то есть функции *x(t)* и *y(t)*. Поскольку скорость – первая производная координаты по времени, получаем следующие дифференциальные уравнения (начальными условиями для них будут координаты точки вылета):

$$\begin{cases} \dfrac{dx}{dt} = v_x & x(0) = x_0 = l_0 \cos\alpha \\ \dfrac{dy}{dt} = v_y & y(0) = y_0 = h_0 + l_0 \sin\alpha \end{cases}$$

Теперь, объединив все приведенные выше уравнения и добавив к ним вычисление модуля вектора скорости по его компонентам v_x и v_y, получим систему дифференциальных и алгебраических уравнений, описывающих полет нашего снаряда:

$$\begin{aligned}
& v = \sqrt{v_x^2 + v_y^2} \\
& F = 0.25\pi\, A(v)\, D^2\, v^{n(v)} \\
& F_x = F\,\frac{v_x}{v} \\
& F_y = F\,\frac{v_y}{v} \\
& \frac{dv_x}{dt} = -\frac{F_x}{m} && v_x(0) = v_0 \cos\alpha \\
& \frac{dv_y}{dt} = -g - \frac{F_y}{m} && v_y(0) = v_0 \sin\alpha \\
& \frac{dx}{dt} = v_x && x(0) = l_0 \cos\alpha \\
& \frac{dy}{dt} = v_y && y(0) = h_0 + l_0 \sin\alpha
\end{aligned}$$

Из-за нелинейной зависимости силы сопротивления воздуха от скорости полета снаряда эта система не имеет аналитического решения. Мы будем решать ее численно, используя простейший метод численного интегрирования – метод Эйлера [13]. От непрерывного времени *t* мы перейдем к дискретным отсчетам t_k, интервал между которыми постоянен и равен *Δt* (рис. 96). Для некоторой непрерывной функции *z(t)* моменту времени t_k будет соответствовать значение $z_k = z(t_k)$. Если функция *z(t)* задана дифференциальным уравнением вида

$$\frac{dz}{dt} = f(t) \quad ,$$

то при достаточно малом *Δt* можно приближенно заменить *dz/dt* на *Δz/Δt*, то есть заменить дифференциалы конечными разностями, и получить следующее соотношение для момента времени t_k:

$$\left.\frac{dz}{dt}\right|_{t=t_k} \approx \frac{\Delta z}{\Delta t} = \frac{z_{k+1} - z_k}{\Delta t}$$

Поскольку

$$\left.\frac{dz}{dt}\right|_{t=t_k} = f(t_k) \quad ,$$

рекуррентное выражение, позволяющее вычислить значение z_{k+1} по известному z_k и правой части дифференциального уравнения *f(t)*, будет выглядеть следующим образом:

$$z_{k+1}=z_k+\Delta t\, f(t_k)$$

Фактически, мы считаем, что на интервале времени *Δt* производная функции *z(t)*, то есть скорость изменения этой функции, не меняется и равна $f(t_k)$. Чем меньше будет интервал времени *Δt*, тем точнее набор отсчетов $\{z_0, z_1, ..., z_k, ...\}$ будет совпадать с функцией *z(t)*.

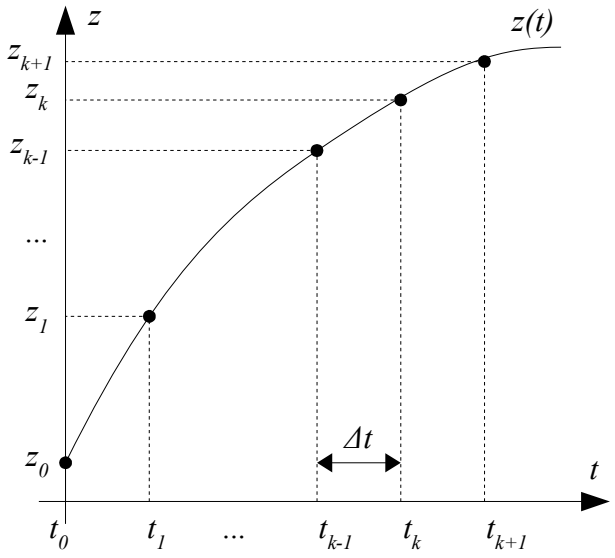


Рис. 96. Переход к дискретному времени

Применив те же рассуждения к дифференциальным уравнениям, описывающим полет снаряда, получим следующую систему разностных уравнений:

$$\begin{cases} v(t_k)=\sqrt{v_x^2(t_k)+v_y^2(t_k)} & \\ F(t_k)=0.25\pi\, A[v(t_k)]D^2 v^{n[v(t_k)]}(t_k) & \\ F_x(t_k)=F(t_k)v_x(t_k)/v(t_k) & \\ F_y(t_k)=F(t_k)v_y(t_k)/v(t_k) & \\ v_x(t_{k+1})=v_x(t_k)-\Delta t\, F_x(t_k)/m & v_x(0)=v_0\cos\alpha \\ v_y(t_{k+1})=v_y(t_k)-\Delta t\,(g+F_y(t_k)/m) & v_y(0)=v_0\sin\alpha \\ x(t_{k+1})=x(t_k)+\Delta t\, v_x(t_k) & x(0)=l_0\cos\alpha \\ y(t_{k+1})=y(t_k)+\Delta t\, v_y(t_k) & y(0)=h_0+l_0\sin\alpha \end{cases}$$

Эти уравнения позволяют по текущим значениям *x*, *y*, v_x и v_y вычислить значения, которые примут эти переменные через интервал времени *Δt*. Такие уравнения уже можно закладывать в программу модели блока.

По условиям задачи нам требуется определить дальность полета снаряда при выстреле с заданным углом возвышения *α*. Дальность полета *L* – это значение горизонтальной координаты снаряда в тот момент, когда его вертикальная координата стала равна нулю (см. рис. 94). Поскольку при моделировании полета снаряда мы численно интегрируем описывающую его движение систему дифференциальных уравнений, траекторию (*x(t)*, *y(t)*) мы получаем в виде набора отсчетов (x_k, y_k), отстоящих друг от друга на интервал времени *Δt*. Мы считаем, что на этом временном интервале производные координат и компонентов скорости снаряда не изменяются, то есть движение снаряда между вычисляемыми точками траектории считается прямолинейным с постоянной скоростью. Таким образом, траектория снаряда в сделанном нами приближении представляет собой кусочно-линейную функцию. Для того, чтобы определить дальность полета, нам нужно дождаться, когда вертикальная координата снаряда станет отрицательной и определить точку пересечения отрезка, соединяющего предпоследний и последний отсчеты траектории, с горизонтальной осью (рис. 97). Поскольку прямая, проходящая через точки (x_{k-1}, y_{k-1}) и (x_k, y_k) описывается уравнением

$$\frac{y-y_{k-1}}{y_k-y_{k-1}}=\frac{x-x_{k-1}}{x_k-x_{k-1}},$$

координата L пересечения этой прямой с горизонтальной осью $(y=0)$ вычисляется следующим образом:

$$L=x_{k-1}-y_{k-1}\frac{x_k-x_{k-1}}{y_k-y_{k-1}}$$

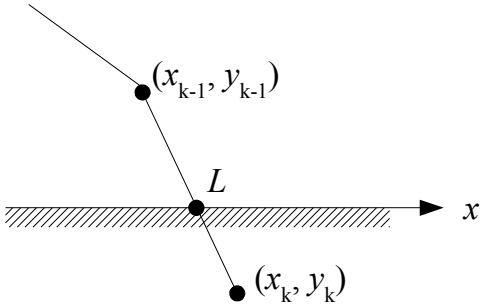


Рис. 97. Определение точки встречи снаряда с поверхностью

Теперь у нас есть все формулы, необходимые для расчета траектории и дальности полета снаряда. Однако, прежде чем приступать к созданию функции модели блока, который будет производить этот расчет, напишем одну вспомогательную функцию, которая существенно облегчит нам настройку этого и некоторых других блоков. В нашем блоке довольно много вещественных параметров: диаметр снаряда, его масса, начальная скорость и т.п. Для удобства пользователя следует предусмотреть в блоке функцию настройки этих параметров, причем хранить эти параметры имеет смысл в статических переменных блока, чтобы у пользователя был выбор: вводить значения параметров в окне настройки или получать их значения по связям от других блоков. Ранее, в §2.7.4, мы создали универсальную функцию `AddWinEditOrDisplayDouble`, которая добавляет в окно настроек поле ввода для редактирования значения по умолчанию вещественной статической переменной блока, если к ней в данный момент не подключена связь. Сейчас, на ее основе, мы напишем универсальную функцию настройки, в которую передаются номера переменных блока и заголовки для соответствующих им полей ввода, а функция самостоятельно создает эти поля, открывает окно, ждет ввода пользователя и, если он нажал в окне кнопку “OK”, записывает введенные им значения обратно в переменные. Эта функция пригодится нам не только для блока моделирования полета снаряда, но и для других блоков, все параметры которых – вещественные числа.

Для того, чтобы в универсальную функцию настройки можно было передать произвольное число пар “номер переменной – заголовок поля”, сделаем одним из ее параметров массив строк. Каждая строка будет начинаться с символьного представления номера переменной (нужно помнить, что нумерация переменных в РДС начинается с нуля), за которым без пробела будет следовать заголовок поля ввода. Например, если в пятнадцатой переменной блока хранится масса снаряда в килограммах, в массив нужно будет включить строку “15Масса снаряда, кг”. Чтобы не передавать в функцию вместе с указателем на начало массива еще и его размер, будем считать, что он всегда завершается значением `NULL`. В этом случае функция, встретив в массиве `NULL` вместо очередной строки, будет знать, что элементы массива закончились.

Универсальная функция настройки вещественных параметров будет выглядеть следующим образом:

```
// Функция настройки вещественных параметров, хранящихся в
// статических переменных блока.
// Возвращает: 1 - OK, 0 - отмена.
int SetupDoubleVars(
      RDS_BHANDLE Block,    // Настраиваемый блок
      char *wintitle,       // Заголовок окна настройки
      char **vars)          // Массив строк-описаний полей ввода
{ RDS_HOBJECT window; // Идентификатор вспомогательного объекта
  BOOL ok;            // Пользователь нажал "OK"
  int count=0;
  BOOL *Connections;
```

```
  int *VarNums;

  if(vars==NULL)      // Ошибка - массив не передан
    return 0;

  // Подсчитываем число полей ввода в массиве vars
  while(vars[count]!=NULL)
    count++;
  if(!count)     // Ошибка - массив пуст
    return 0;

  // Отводим вспомогательный массив логических значений, в котором
  // будем запоминать наличие связей, присоединенных к переменным,
  // а также массив целых чисел для номеров переменных, считанных
  // из строк массива vars
  Connections=new BOOL[count];
  VarNums=new int[count];

  // Создаем окно с заголовком wintitle
  window=rdsFORMCreate(FALSE,-1,-1,wintitle);

  // Добавляем поля ввода из массива vars, заполняя массивы
  // Connections и VarNums
  for(int i=0;i<count;i++)
    { char *str;
      // Считываем из строки номер переменной. Указатель на первый
      // после номера символ запишется в str.
      VarNums[i]=strtol(vars[i],&str,0);
      // Если после номера строка кончается, присваиваем str
      // значение NULL, чтобы в качестве заголовка было
      // использовано имя переменной
      if(*str==0)
        str=NULL;
      // Добавляем поле ввода, запоминаем наличие связи
      Connections[i]=AddWinEditOrDisplayDouble(window,Block,
                                               VarNums[i],i,str);
    }

  // Открываем окно
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Нажата кнопка OK - запись параметров в переменные
      for(int i=0;i<count;i++)
        if(!Connections[i]) // У переменной нет связи
          { char *str=rdsGetObjectStr(window,i,RDS_FORMVAL_VALUE);
            // Устанавливаем новое значение по умолчанию
            rdsSetBlockVarDefValueStr(Block,VarNums[i],str);
          }
    }
  // Удаляем вспомогательные массивы
  delete[] Connections;
  delete[] VarNums;
  // Уничтожаем окно
  rdsDeleteObject(window);
  // Возвращаем 1 или 0
  return ok?1:0;
}
//=======================================
```

Функция принимает три параметра: идентификатор блока, для которого вызывается функция настройки (Block), текст заголовка окна (wintitle) и массив строк описанного выше формата (vars). Эта функция, прежде всего, подсчитывает число строк в массиве vars, перебирая их до тех пор, пока не встретит значение NULL. Число обнаруженных строк записывается в переменную count. Затем отводятся два вспомогательных массива из count элементов: логический массив Connections, в котором будет запоминаться наличие связи у каждой из перечисленных в массиве vars переменных, и целый массив VarNums, в который будут записаны номера переменных блока, полученные при разборе строк из vars. Затем, как обычно, сервисной функцией РДС rdsFORMCreate создается вспомогательный объект-окно, идентификатор которого присваивается переменной window.

Далее в цикле перебираются строки из массива vars, каждая из которых разбивается на номер переменной и заголовок поля при помощи функции strtol из стандартной библиотеки языка С (ее прототип описан в файле “stdlib.h”). Эта функция преобразует строку, переданную в ее первом параметре, в целое число, и возвращает полученное значение. В третьем параметре функции передается основание системы счисления преобразуемого числа (в нашем случае передается 0, при этом функция сама определит систему счисления по форме записи). Во втором параметре функции передается указатель на переменную, в которую записывается указатель на первый символ строки, который функция не смогла распознать, то есть первый не относящийся к числу символ. Нам именно это и нужно: в строке из массива vars сразу за числом, указывающим номер переменной в блоке, следует текст заголовка поля ввода, и мы получим указатель на первый следующий за числом символ, то есть на начало заголовка. Например, если в vars[i] будет записана строка “15Масса снаряда, кг”, то после вызова

```
VarNums[i]=strtol(vars[i],&str,0);
```

VarNums[i] получит значение 15, а str будет указывать на символ “М” в слове “Масса”. Таким образом, при помощи одного вызова мы получаем и номер переменной, и указатель на начало заголовка поля ввода.

Если str указывает на нулевой символ, то есть на маркер конца строки, значит, строка содержала только номер переменной, а заголовка поля ввода в ней не было. В этом случае переменной str принудительно присваивается значение NULL, поскольку вызываемая нами функция AddWinEditOrDisplayDouble написана так, чтобы при передаче ей NULL в качестве заголовка поля ввода использовалось имя переменной блока (иногда имена переменных достаточно информативны сами по себе). В качестве номера переменной в функцию передается только что полученное нами значение VarNums[i], в качестве номера поля ввода – значение индекса цикла i, а результат ее возврата, то есть наличие связи у переменной, записывается в элемент массива Connections[i]. Теперь можно открывать окно функцией rdsFORMShowModalEx, которая вернет управление только тогда, когда пользователь закроет это окно кнопками “OK” или “Отмена”.

Если пользователь нажал “OK” (переменная ok в этом случае будет иметь значение TRUE), мы снова перебираем все переменные из массива vars в цикле от 0 до count-1 и для тех из них, у которых нет присоединенной связи (Connections[i] имеет значение FALSE), записываем строку, введенную пользователем в поле ввода i, в значение по умолчанию переменной с номером VarNums[i].

Перед завершением функции уничтожаются вспомогательные массивы и объект-окно, после чего функция возвращает 1, если пользователь нажал в окне кнопку “OK”, и 0, если кнопку “Отмена” (именно такие значения должна возвращать функция модели, при наличии и отсутствии изменений после вызова функции настройки).

С использованием функции SetupDoubleVars написание моделей блоков, в которых нужно настраивать вещественные параметры, сильно упрощается. Например, если у нас есть три переменных блока типа double с номерами 2, 3 и 8, которые используются для

хранения настроечных параметров, то часть модели, ответственная за вызов функции настройки, может выглядеть так:

```
// Массив строк, описывающих номера переменных и заголовки полей
// ввода (объявлен как static, чтобы не занимал место в стеке)
static char *setup[]={
      "2Параметр 1",
      "3Параметр 2",
      "8Еще один параметр",
      NULL};

      ...

      // Вызов функции настройки
      case RDS_BFM_SETUP:
        return SetupDoubleVars(BlockData->Block,
                  "Заголовок окна",setup);
```

Вернемся к блоку, моделирующему полет снаряда, и определимся с необходимыми для его работы переменными. Будем считать, что выстрел производится при запуске расчета. При выстреле блок должен будет установить начальные значения переменных v_x и v_y, вычислив их по значениям модуля начальной скорости v_0 и угла возвышения α, которые мы сделаем входами блока. Здесь мы встретимся с проблемой, на которой нужно остановиться подробно.

Согласно логике расчета, значения v_0 и α нужны блоку только в момент выстрела, поскольку они используются для определения начальных значений компонентов вектора скорости. Дальше все вычисления ведутся уже с переменными v_x и v_y согласно разностным уравнениям на стр. 395. Если значения на входы блока поступают по связям с других блоков, может случиться так, что блок считает v_0 и α до того, как установятся их правильные значения. Например, если значение на вход блока будет подаваться по цепочке из трех последовательно соединенных блоков, значение этого входа установится только к третьему такту расчета. Мы не можем заранее знать, сколько тактов потребуется для того, чтобы правильные значения добрались до входа блока, но мы можем оценить это число сверху. Если в во всей схеме содержится N простых блоков (именно простых, поскольку в расчете участвуют только простые блоки), длина любой цепочки из соединенных блоков в этой схеме не может быть больше N, а, значит, пропустив N тактов, мы гарантируем установку всех начальных значений на входах всех блоков схемы. Таким образом, мы могли бы отсчитать N тактов после запуска расчета, и только потом устанавливать начальные значения переменных v_x и v_y.

К счастью, возможность пропуска заданного числа тактов уже реализована в стандартном блоке-планировщике динамического расчета. Число пропускаемых тактов задается в настройках этого блока (рис. 98), причем вместо того, чтобы вводить число тактов вручную, можно установить флаг “Авто” – в этом случае число начальных тактов будет равно числу блоков в подсистеме с учетом подсистем, вложенных в нее.

Блок-планировщик изменит значение динамической переменной времени “DynTime” только после пропуска заданного числа начальных тактов, поэтому модель нашего блока должна дождаться изменения “DynTime” и только после этого считать значения v_0 и α и начать расчет разностных уравнений. Для управления моментом считывания введем в блок внутреннюю логическую переменную `ValSet` с начальным значением 0. Пока это значение будет оставаться нулевым, модель не будет вести расчет. Как только “DynTime” изменится при нулевом значении `ValSet`, модель считает v_0 и α и присвоит `ValSet` значение 1, разрешив тем самым расчет траектории.

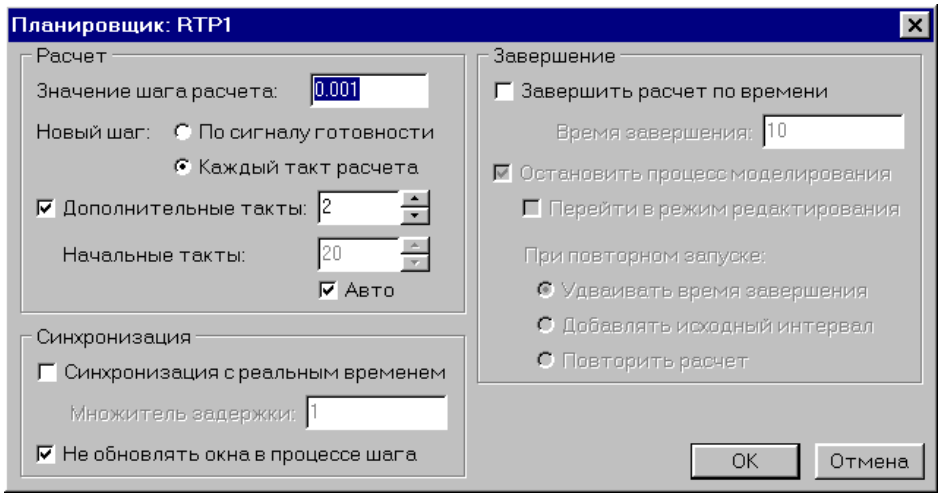


Рис. 98. Настройки планировщика для расчета траектории снаряда

После встречи снаряда с землей блок должен прекратить вычисление его координат и скоростей: в любой момент времени после падения горизонтальная координата снаряда x должна быть равна дальности полета L (см. рис. 94), а вертикальная координата y и обе компоненты вектора скорости v_x и v_y должны быть равны нулю. Кроме того, после падения снаряда на землю блок должен выдать на выход сигнал, сообщающий другим блокам о том, что дальность полета вычислена. Назовем этот сигнальный выход `Impact`, позже мы подключим к нему блок, который будет подбирать угол вылета для достижения заданной дальности. Для прекращения моделирования полета после падения снаряда будем использовать внутреннюю логическую переменную `InFlight` с нулевым начальным значением: блок будет считать траекторию снаряда только тогда, когда ее значение равно единице. В момент начала расчета траектории модель присвоит ей значение 1, а затем, как только снаряд встретится с землей, модель обнулит ее, и расчет траектории прекратится.

Таким образом, наша модель может находиться в одном из следующих трех состояний:

- `ValSet==0`, `InFlight==0` – расчет сброшен, модель ждет, пока планировщик не отсчитает заданное число начальных тактов и не изменит значение времени;
- `ValSet==1`, `InFlight==1` – снаряд в полете, идет расчет траектории;
- `ValSet==1`, `InFlight==0` – снаряд встретился с землей (при переходе в это состояние выдается сигнал `Impact`).

Нужно отметить, что, хотя сигнал `Impact` и будет получать единичное значение синхронно с обнулением переменной `InFlight`, для описания состояния машины он не подходит, поскольку при передаче по связям сигнальные выходы автоматически сбрасываются.

Кроме трех перечисленных переменных, управляющих логикой работы блока, и вещественных переменных для хранения координат, компонентов скорости и параметров снаряда, нам потребуется еще одна дополнительная внутренняя вещественная переменная. Для вычисления интервала времени Δt в разностных уравнениях на стр. 395, нам нужно запоминать значение времени предыдущего шага расчета. Течением времени в нашей схеме будет управлять стандартный блок – планировщик динамического расчета, поэтому значение текущего времени наш блок будет брать из динамической переменной “DynTime”, как мы уже неоднократно делали ранее. Если мы введем в блок статическую переменную `t0` и будем каждый раз запоминать в ней текущее значение времени, то при изменении “DynTime” мы сможем вычислить величину изменения времени, то есть значение Δt, как `DynTime-t0`. Мы уже использовали такой механизм когда создавали модель блока, движущегося внутри окна подсистемы с заданной скоростью (см. стр. 108).

Структура переменных блока моделирования полета снаряда будет следующей:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Назначение*
0	Start	Сигнал	1	Вход	Стандартный сигнал запуска
1	Ready	Сигнал	1	Выход	Стандартный сигнал готовности
2	D	double	8	Вход	Диаметр снаряда, м
10	m	double	8	Вход	Масса снаряда, кг
18	v0	double	8	Вход	Начальная скорость снаряда, м/с
26	Angle	double	8	Вход	Угол возвышения, градусов
34	l0	double	8	Вход	Длина ствола (направляющей), м
42	h0	double	8	Вход	Высота оси ствола (направляющей), м
50	ValSet	Логический	1	Внутр.	Признак того, что v0 и Angle считаны (начальное значение – 0)
51	x	double	8	Выход	Координата *x* снаряда, м
59	y	double	8	Выход	Координата *y* снаряда, м
67	vx	double	8	Выход	Горизонтальная скорость v_x, м/с
75	vy	double	8	Выход	Вертикальная скорость v_y, м/с
83	v	double	8	Выход	Модуль вектора скорости, м/с
91	Impact	Сигнал	1	Выход	Снаряд встретился с землей (начальное значение – 0)
92	t0	double	8	Внутр.	Значение времени предыдущего шага, с (начальное значение – 0)
100	InFlight	Логический	1	Выход	Снаряд в данный момент – в полете (начальное значение – 0)

К выходам блока мы добавили переменную `v` – модуль вектора скорости. Для решения поставленной задачи этот выход нам не нужен, но он даст возможность, при желании, построить график зависимости скорости снаряда от времени.

Теперь можно написать модель блока:

```
// Расчет внешней баллистики
extern "C" __declspec(dllexport)
    int RDSCALL Ballistics(int CallMode,
                           RDS_PBLOCKDATA BlockData,
                           LPVOID ExtParam)
{ RDS_PDYNVARLINK Link;
  double t,dt,A,n,F,Fx,Fy,xp,yp;
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))        // 0
#define Ready (*((char *)(pStart+1)))      // 1
#define D (*((double *)(pStart+2)))        // 2
#define m (*((double *)(pStart+10)))       // 3
#define v0 (*((double *)(pStart+18)))      // 4
#define Angle (*((double *)(pStart+26)))   // 5
#define l0 (*((double *)(pStart+34)))      // 6
#define h0 (*((double *)(pStart+42)))      // 7
```

```
#define ValSet (*((char *)(pStart+50)))    // 8
#define x (*((double *)(pStart+51)))       // 9
#define y (*((double *)(pStart+59)))       // 10
#define vx (*((double *)(pStart+67)))      // 11
#define vy (*((double *)(pStart+75)))      // 12
#define v (*((double *)(pStart+83)))       // 13
#define Impact (*((char *)(pStart+91)))    // 14
#define t0 (*((double *)(pStart+92)))      // 15
#define InFlight (*((char *)(pStart+100))) // 16
  // Массив описания параметров для универсальной функции настройки
  static char *setup[]={
    "6Длина ствола, м",
    "7Высота оси, м",
    "2Диаметр снаряда, м",
    "3Масса снаряда, кг",
    "4Начальная скорость, м/с",
    "5Угол вылета, град.",
    NULL};

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Подписка на динамическую переменную "DynTime"
        Link=rdsSubscribeToDynamicVar(RDS_DVPARENT,"DynTime",
                "D",TRUE);
        BlockData->BlockData=Link;
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        // Прекращение подписки на "DynTime"
        rdsUnsubscribeFromDynamicVar(
                     (RDS_PDYNVARLINK)BlockData->BlockData);
        break;

      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSDDDDDDLDDDDDSDL}")?
                     RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Вызов функции настройки
      case RDS_BFM_SETUP:
        return SetupDoubleVars(BlockData->Block,
                "Внешняя баллистика",setup);

      // Изменение динамической переменной
      case RDS_BFM_DYNVARCHANGE:
        // Получаем доступ к динамической переменной времени
        Link=(RDS_PDYNVARLINK)BlockData->BlockData;
        if(Link==NULL || Link->Data==NULL) // Нет доступа
          break;
        t=*((double*)Link->Data);     // В t - текущее время
        if(t==t0)     // Время не изменилось - ждем
          break;
        dt=t-t0;      // В dt - интервал времени с прошлого шага
        t0=t;    // Запоминаем время в t0 чтобы при следующем
                 // изменени DynTime (t) можно было вычислить dt
```

```
      if(!ValSet)   // Начальные значения еще не считаны
        { // Начинаем расчет траектории снаряда
          double alpha=Angle*M_PI/180.0; // Угол в радианах
          // Занесение в переменные начальных значений
          vx=v0*cos(alpha);
          vy=v0*sin(alpha);
          v=v0;
          x=l0*cos(alpha);
          y=h0+l0*sin(alpha);
          // Сбрасываем сигнал падения
          Impact=0;
          // Разрешаем расчет и взводим ValSet
          InFlight=ValSet=1;
        }
      else if(!InFlight) // Снаряд не в полете - ничего не делаем
        break;
      // Вычисление модуля вектора скорости и силы сопротивления
      // по формуле Забудского
      v=sqrt(vx*vx+vy*vy);
      if(v<240.0)
        { A=0.0140; n=2; }
      else if(v<295.0)
        { A=0.0000583; n=3; }
      else if(v<375.0)
        { A=0.000000000670; n=5; }
      else if(v<419.0)
        { A=0.0000940; n=3; }
      else if(v<550.0)
        { A=0.0394; n=2; }
      else if(v<800.0)
        { A=0.2616; n=1.7; }
      else
        { A=0.713; n=1.55; }
      F=A*M_PI*D*D*pow(v,n)/4.0;    // Сила сопротивления
      // Горизонтальная и вертикальная компоненты F
      Fx=F*vx/v;
      Fy=F*vy/v;
      // Запоминаем текущие значения координат
      xp=x; yp=y;
      // Вычисляем новые значения координат по
      // разностным уравнениям
      x+=dt*vx;
      y+=dt*vy;
      // Вычисляем новые значения компонент вектора скорости
      // по разностным уравнениям
      vx-=dt*Fx/m;
      vy-=dt*(9.807+Fy/m);
      if(y<0.0)     // Снаряд встретился с поверхностью
        { Impact=1; // Взводим выходной сигнал падения снаряда
          InFlight=0;    // Прекращаем расчет траектории
          // Вычисляем координату встречи с поверхностью
          x=xp-yp*(x-xp)/(y-yp);
          y=vx=vy=0;
        }
      Ready=1;      // Для передачи выходов по связям
      break;
  }
return RDS_BFR_DONE;
```

```
// Отмена макроопределений переменных
#undef InFlight
#undef t0
#undef Impact
#undef v
#undef vy
#undef vx
#undef y
#undef x
#undef ValSet
#undef h0
#undef l0
#undef Angle
#undef v0
#undef m
#undef D
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

В самом начале функции модели, после описаний локальных переменных и макроопределений, описывается статический массив строк `setup`, который будет использоваться для вызова написанной нами ранее универсальной функции настройки `SetupDoubleVars` (см. стр. 396). Массив описан как статический, чтобы он не занимал место в стеке функции – его содержимое будет общим для всех блоков с моделью `Ballistics` и не будет изменяться в процессе их работы. Каждая строка массива содержит номер переменной в символьном представлении и заголовок соответствующего ей поля ввода, которое будет добавлено в окно настройки. Всего в этом блоке у нас шесть параметров, которые пользователь сможет ввести в окне настройки, либо, при необходимости, подать с других блоков через связи.

При инициализации блока (вызов с параметром `RDS_BFM_INIT`) модель вызывает функцию `rdsSubscribeToDynamicVar`, подписывая блок на переменную “DynTime” типа `double` (строка “D”) с поиском по иерархии (последний параметр функции равен `TRUE`) в родительской подсистеме (константа `RDS_DVPARENT`). Возвращенный функцией указатель на созданную РДС структуру подписки типа `RDS_DYNVARLINK` присваивается полю `BlockData` структуры данных блока. Обычно в этом поле хранится указатель на личную область данных блока, с которой работает функция модели, но у этого блока нет специально выделенной личной области, поэтому мы используем это поле для хранения указателя на структуру подписки. При вызове для очистки данных (`RDS_BFM_CLEANUP`) модель отменяет подписку, передавая запомненный указатель в функцию `rdsUnsubscribeFromDynamicVar`. Эти функции уже не один раз встречались в описываемых примерах. В реакции модели на проверку типа переменных (`RDS_BFM_VARCHECK`) тоже нет ничего нового: переданная в параметре `ExtParam` строка сравнивается со строкой, соответствующей нужной нам последовательности переменных, и, в зависимости от результата сравнения, возвращается константа `RDS_BFR_BADVARSMSG` или `RDS_BFR_DONE`.

В режиме `RDS_BFM_SETUP` модель вызывает универсальную функцию настройки `SetupDoubleVars`, в которую передается идентификатор настраиваемого блока, заголовок окна настройки “Внешняя баллистика” и массив строк описаний полей ввода `setup`, описанный в начале функции модели. Все необходимые действия по созданию полей ввода, открытию окна и записи измененных пользователем параметров в значения переменных блока выполняются внутри этой функции.

Расчет траектории полета снаряда производится в реакции на изменение динамической переменной RDS_BFM_DYNVARCHANGE, реакция на такт моделирования RDS_BFM_MODEL в этой модели отсутствует: все действия выполняются только при изменении времени. Блок подписан на единственную динамическую переменную – время “DynTime”, поэтому вызов RDS_BFM_DYNVARCHANGE означает, что время в системе изменилось. Прежде всего модель проверяет, есть ли у нее доступ к переменной времени. Если его нет, функция модели немедленно завершается, если есть – значение времени записывается во вспомогательную переменную t. Для расчета координат и составляющих вектора скорости снаряда по разностным уравнениям необходимо знать интервал времени Δt, прошедший с последнего вычисления этих значений. Если значение t будет равно значению t0, то есть значению времени на момент последнего вычисления, значит, время в системе с прошлого раза не изменилось, и ничего вычислять не нужно – модель немедленно завершается. В противном случае из текущего значения времени t вычитается значение времени при прошлом вычислении t0 и полученная разность присваивается переменной dt – это и есть интересующий нас интервал времени. Затем переменной t0 присваивается текущее время, чтобы в следующий раз при изменении времени можно было снова определить интервал Δt.

После того, как интервал dt вычислен, и переменной t0 присвоено новое значение времени, модель проверяет, считаны ли уже значения начальной скорости снаряда и угла возвышения метательной машины. Если значение логической переменной ValSet нулевое, эти значения нужно считать сейчас: мы уже знаем, что системное время изменилось, значит, планировщик уже выполнил все начальные такты, заданные в его настройках. В этом случае угол Angle переводится в радианы и записывается во вспомогательную переменную alpha, вычисляются начальные значения горизонтальной и вертикальной составляющих скорости снаряда vx и vy и начальные координаты снаряда x и y (они соответствуют координатам конца ствола метательной машины при данном угле возвышения). После этого сбрасывается сигнал падения снаряда Impact, и взводятся логические переменные ValSet (мы уже считали v0 и Angle и больше ничего не ждем) и InFlight (снаряд теперь в полете, и мы начинаем расчет его траектории).

Если значение ValSet было ненулевым, мы проверяем InFlight: если ее значение нулевое, значит, снаряд уже закончил свой полет, и модель немедленно завершается.

После того, как модель убедилась, что снаряд находится в полете, вычисляется модуль вектора его скорости v – он нужен для определения силы сопротивления воздуха. В зависимости от скорости по приведенной выше таблице определяются параметры A и n, используемые в формуле Забудского, после чего вычисляется модуль силы сопротивления F, который затем раскладывается на горизонтальную и вертикальную составляющие Fx и Fy. По приведенным на стр. 395 разностным уравнениям вычисляются новые значения координат снаряда x и y и составляющих его скорости vx и vy, причем предыдущие значения координат запоминаются в переменных xp и yp – они понадобятся для вычисления координат точки встречи снаряда с поверхностью, если она произошла.

Если очередное вычисленное значение координаты y стало отрицательным, значит, траектория снаряда пересеклась с поверхностью земли. В этом случае выходному сигналу Impact присваивается единица, а логической переменной InFlight – ноль, что прекратит дальнейший расчет траектории. Затем, согласно рис. 97, вычисляется горизонтальная координата встречи снаряда с поверхностью, которая присваивается переменной x. Переменной y присваивается значение 0 – снаряд лежит на земле. Поскольку, начиная с этого момента, дальнейший расчет траектории вестись не будет, значения x и y больше не изменятся, и значение x на момент выдачи блоком сигнала Impact можно считать дальностью полета снаряда.

В самом конце реакции на вызов `RDS_BFM_DYNVARCHANGE` переменной `Ready` присваивается значение 1, чтобы выходы, вычисленные в этом вызове, передались по связям. На этом работа модели блока завершается.

Теперь можно протестировать созданный нами блок. Для этого, прежде всего, нужно создать новый блок, задать для него указанную выше структуру переменных, подключить к нему созданную модель `Ballistics`, включить в параметрах блока функцию настройки и задать для блока запуск по сигналу. Теперь можно собрать вокруг блока схему, представленную на рис. 99.

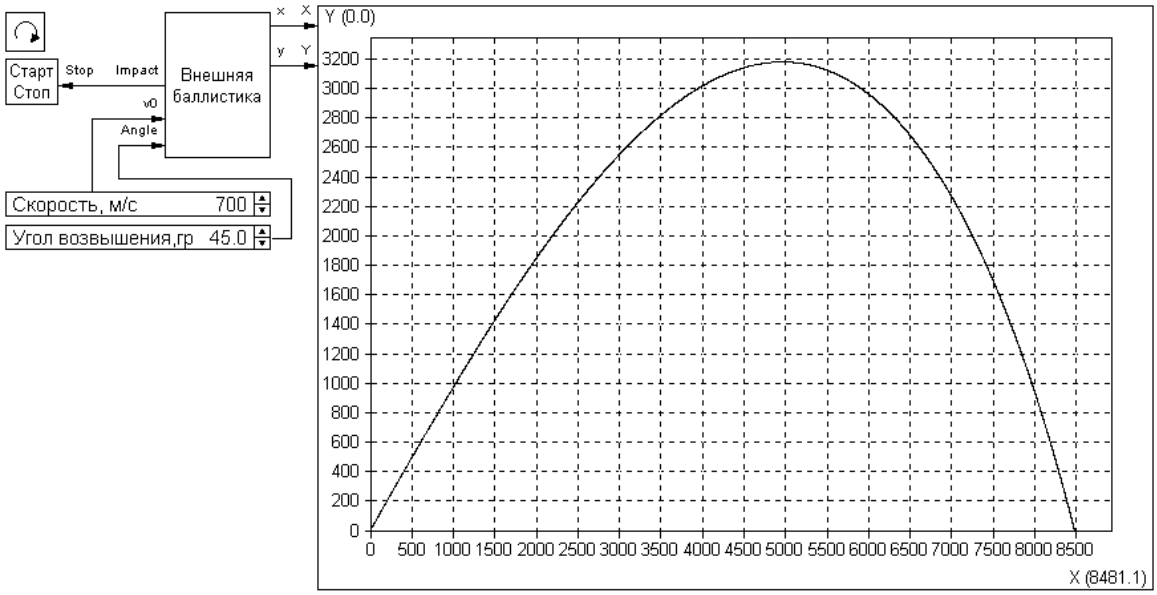


Рис. 99. Схема для проверки блока расчета траектории снаряда

Все параметры блока, кроме начальной скорости снаряда `v0` и угла возвышения `Angle`, заданы через функцию настройки (рис. 100). К выходам `x` и `y` нашего блока присоединен стандартный график для построения фазового портрета, то есть параметрической функции $(x(t), y(t))$ – он будет отображать траекторию полета. Выход `Impact` нашего блока соединен с входом `Stop` блока запуска и остановки расчета, созданного нами в предыдущем параграфе. Как только снаряд достигнет поверхности и наш блок выдаст сигнал `Impact`, этот блок остановит расчет.

Для правильной работы схемы необходимо настроить блок-планировщик согласно рис. 98. Во-первых, необходимо отключить завершение расчета по времени – для остановки расчета после падения снаряда мы подключили специальный блок. Во-вторых, нужно выбрать шаг расчета. Самые быстро изменяющиеся величины в модели полета снаряда – это координаты `x` и `y`, максимально возможная скорость снаряда, для которой мы еще можем считать сопротивление воздуха, равна 1000 м/с, поэтому для определения

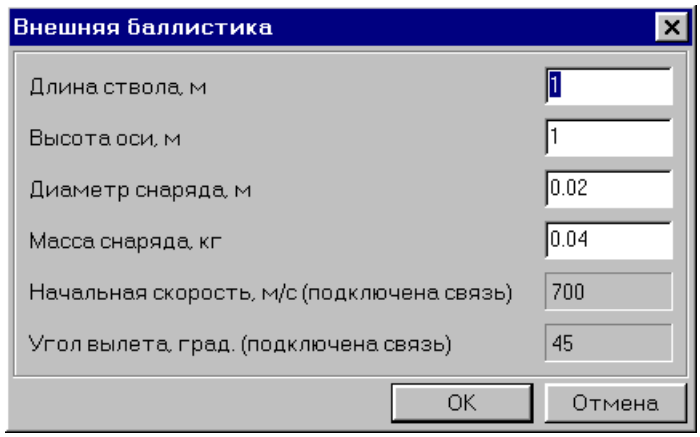


Рис. 100. Окно настройки блока расчета траектории снаряда

дальности с точностью около одного метра можно выбрать шаг порядка одной тысячной секунды. И, в-третьих, как уже было указано выше, следует задать некоторое количество начальных тактов или, лучше всего, установить флаг “Авто”. Так мы будем уверены, что на момент первого изменения времени планировщиком значения `v0` и `Angle` уже успеют установиться.

Результат моделирования полета снаряда с указанными в настройках параметрами при начальной скорости 700 м/с и угле возвышения 45º изображен на рис. 99. Видно, что из-за влияния сопротивления воздуха траектория снаряда отличается от параболы.

Теперь мы перейдем к главной задаче: сделаем блок, который подберет такой угол возвышения метательной машины, чтобы дальность полета снаряда была как можно ближе к заданной. Это достаточно обычная оптимизационная задача, для решения которой существует множество способов. Будем считать, что у метательной машины есть допустимый диапазон углов возвышения (например, от 0º до 90º), в пределах которого мы можем установить любой угол, но только с заданной точностью. Например, если точность установки угла будет равной одной десятой градуса, мы сможем установить углы в 30.1º, 30.2º, 30.3º, но не сможем установить угол в 30.15º – у нас просто не хватит точности исполнительного механизма, или цены деления шкалы, если угол устанавливается вручную. Будем решать нашу задачу следующим образом: установим максимально возможный угол возвышения, выполним расчет и, дождавшись падения снаряда на землю (получив от блока расчета траектории сигнал `Impact`), определим величину промаха, то есть разность между заданной и получившейся дальностью для этого угла. Выберем какой-либо шаг, с которым мы будем изменять угол – например, одну пятую всего диапазона. Сбросим в исходное состояние подсистему, в которой будет находиться наш блок вместе с блоком расчета траектории, и уменьшим угол на выбранную величину шага (мы начали с максимально возможного угла возвышения, поэтому мы можем только уменьшать его). Поскольку расчет мы не останавливали, полет снаряда будет промоделирован снова, но уже с другим углом возвышения. Снова дождемся падения снаряда, и определим новую величину промаха. Если она стала меньше, будем уменьшать угол с этим шагом, каждый раз сбрасывая расчет и повторяя его заново до тех пор, пока промах будет уменьшаться. Как только промах начнет увеличиваться, вернемся к предыдущему значению угла, уменьшим шаг по углу вдвое и снова попробуем уменьшить угол, но уже с новым шагом. Если промах уменьшился, продолжим уменьшать угол, если же он увеличился – попробуем, наоборот, увеличивать его. Если и увеличение, и уменьшение угла с данным шагом приводит к увеличению промаха, еще раз уменьшим шаг по углу вдвое и снова попытаемся уменьшать или увеличивать угол. Так мы будем действовать до тех пор, пока шаг, с которым мы изменяем угол, не станет меньше точности установки угла, определяемой конструкцией механизма наведения нашей машины. Тогда последнее значение угла, при котором промах был наименьшим, и будет искомым углом возвышения для заданной дальности. Описанный метод поиска является упрощением стандартного метода покоординатного спуска для случая единственной координаты – угла возвышения.

После того, как алгоритм поиска завершится и искомый угол возвышения будет найден, нужно будет еще раз провести расчет для этого значения угла, чтобы, если пользователь включил в схему какие-либо графики (например, график траектории, как на рис. 99), на них отобразились бы координаты или скорости снаряда именно для найденного значения. Предыдущий расчет, проведенный в рамках алгоритма, был очередной попыткой изменить угол, поэтому он, очевидно, не будет соответствовать углу возвышения, промах для которого минимален.

Несмотря на простой алгоритм поиска, модель блока, который будет его осуществлять, будет довольно сложной. После каждого вычисления дальности этот блок должен сбросить в исходное состояние подсистему, в которой он будет находиться вместе с блоком расчета траектории снаряда, и изменить значение угла возвышения согласно

алгоритму. Поскольку все статические переменные блока при сбросе вернутся к начальным значениям, в них нельзя хранить такие значения, как текущий проверяемый угол возвышения, наименьший на данный момент промах и соответствующий ему угол и т.д. Эти значения не должны сбрасываться при каждом расчете дальности, иначе алгоритм не будет работать. Мы воспользуемся тем, что сброс расчета не затрагивает личную область данных блока, и будем хранить эти переменные в ней. Оформим личную область данных как класс:

```
//=============================================
// Поиск угла возвышения для заданной дальности
//=============================================
// Личная область данных блока
class TArtSearchData
{ public:
    BOOL SelfReset;   // Блок сам сбросил подсистему
    int Mode;         // Текущее состояние алгоритма:
      #define ASMODE_READY      0 // Готов к поиску
      #define ASMODE_SEARCHING  1 // Идет поиск
      #define ASMODE_FINALRUN   2 // Последний прогон
      #define ASMODE_FINISHED   3 // Поиск завершен
    double AngleStep; // Текущий шаг изменения угла
    double AngleToSet;     // Угол, который нужно установить
                           // после сброса
    double OptAngle;  // Наилучший на данный момент угол
    double OptMiss;   // Наименьший на данный момент промах
    BOOL FirstStep;   // Проведенное моделирование - первое с
                      // новым шагом изменения угла


    // Ограничение диапазона и точности установки угла
    double FixAngle(double a,double amin,double amax,double acc)
      { if(a<amin) return amin;
        if(a>amax) return amax;
        return floor((a-amin)/acc)*acc+amin;
      };

    // Вывод сообщения о результатах поиска
    void ShowResults(void)
      { char *str,
             *angle=rdsDtoA(OptAngle,-1,NULL),
             *miss=rdsDtoA(OptMiss,0,NULL);
        // Формирование динамической строки с сообщением
        str=rdsDynStrCat("Угол возвышения: ",angle,FALSE);
        rdsAddToDynStr(&str," гр.\nПромах: ",FALSE);
        rdsAddToDynStr(&str,miss,FALSE);
        rdsAddToDynStr(&str," м",FALSE);
        // Вывод текста
        rdsMessageBox(str,"Поиск завершен",MB_OK);
        // Освобождение всех динамических строк
        rdsFree(str);
        rdsFree(angle);
        rdsFree(miss);
      };

    // Конструктор класса
    TArtSearchData(void)
      { SelfReset=FALSE;
        Mode=ASMODE_READY;
      };
```

```
};
//=======================================
```

Наш блок должен по-разному реагировать на сброс и на запуск расчета в зависимости от того, чем он в данный момент занят. Он должен отличать сброс расчета, выполненный пользователем, от сброса состояния подсистемы, выполненного самим блоком: в первом случае нужно прекратить поиск и вернуть все параметры в исходное состояние, во втором – продолжить поиск с новым значением угла. Для того, чтобы различать эти два случая, мы ввели в класс личной области данных блока логическое поле `SelfReset` с исходным значением `FALSE` (исходное значение задается в конструкторе класса). Перед программным сбросом модель будет присваивать этому полю значение `TRUE`, а в конце реакции на сброс расчета (`RDS_BFM_RESETCALC`) снова сбрасывать его в `FALSE`. Таким образом, если расчет сброшен пользователем, при вызове модели в режиме `RDS_BFM_RESETCALC` поле `SelfReset` будет иметь значение `FALSE`, а если расчет сброшен моделью блока – `TRUE`.

Нам также необходимо хранить текущее состояние алгоритма, то есть выполняемое им в данный момент действие. Для этого в класс введено целое поле `Mode` и определены символические константы для каждого из состояний блока:

- `ASMODE_READY` – поиск угла еще не проводился, блок в исходном состоянии (это значение присваивается полю `Mode` в конструкторе класса);
- `ASMODE_SEARCHING` – идет поиск угла;
- `ASMODE_FINALRUN` – идет последний расчет с найденным значением угла возвышения;
- `ASMODE_FINISHED` – поиск завершен.

При самом первом запуске расчета или после сброса расчета пользователем блок будет находиться в состоянии `ASMODE_READY`. Как только расчет будет запущен, блок перейдет в состояние `ASMODE_SEARCHING`, и будет находиться в нем до тех пор, пока алгоритм не закончит работу. Затем блок перейдет в состояние `ASMODE_FINALRUN` и будет проведен последний расчет для найденного значения угла. После завершения расчета состояние блока изменится на `ASMODE_FINISHED`, и расчет остановится. В зависимости от значения переменной `Mode` блок будет по-разному реагировать на запуск расчета и на получение значения дальности полета снаряда от блока расчета траектории (то есть на конец очередного расчета дальности).

Для того чтобы в процессе поиска после очередного программного сброса состояния подсистемы установить новое значение угла, в личной области данных нам потребуется поле для хранения этого значения: значение, которое нужно установить, мы сможем вычислить только в конце очередного расчета дальности, а установить его можно только после сброса, иначе установленное значение сбросится вместе со всеми остальными переменными. С этой целью в класс введено вещественное поле `AngleToSet`: в конце расчета мы запишем в него новое значение угла, а в реакции модели на сброс расчета скопируем значение из этого поля в соответствующий выход блока, который будет подключен к входу `Angle` блока расчета траектории. Три оставшихся вещественных поля класса хранят текущие параметры алгоритма: шаг по углу, используемый в данный момент (`AngleStep`), наилучший на данный момент угол возвышения (`OptAngle`) и промах при этом значении угла (`OptMiss`).

Последнее логическое поле класса, `FirstStep`, будет использоваться для переключения направления изменения угла. При каждом уменьшении шага по углу мы будем присваивать этому полю значение `TRUE`, а после любого изменения угла – `FALSE`. Согласно алгоритму поиска, после очередного уменьшения величины шага мы сначала должны попытаться уменьшить угол с этим шагом, а если промах при этом возрастет, попытаться увеличить угол. Если при возрастании промаха поле `FirstStep` имело значение `TRUE`, значит, это была первая попытка изменить угол с новым шагом, и нужно изменить знак шага, то есть перейти от уменьшения угла к увеличению с тем же шагом. Если же поле имело значение `FALSE`, значит, либо это не первый шаг по углу в этом направлении (в этом случае

двигаться обратно с тем же шагом бессмысленно – мы там уже побывали), либо мы только что изменили знак шага, не меняя его величину, то есть начали изменять угол в другом направлении, поскольку старое направление изменения угла приводило к возрастанию промаха. В обоих случаях нужно будет уменьшать шаг и снова пытаться изменить угол, но уже с новым шагом. Таким образом, если при очередном расчете промах увеличится и `FirstStep` будет иметь значение `FALSE`, нужно уменьшить шаг по углу вдвое.

На самом деле, не важно, будем ли мы после уменьшения шага сначала пытаться уменьшать угол, а потом увеличивать, или, наоборот, сначала увеличивать, а потом уменьшать. Поэтому мы не будем следить за знаком шага при его уменьшении – будем оставлять его прежним, а если промах возрастет, менять знак шага на противоположный.

Кроме полей, необходимых для реализации алгоритма поиска, в классе личной области данных блока описаны две вспомогательные функции. Первая из них, `FixAngle`, корректирует переданное ей значение угла `a` таким образом, чтобы оно, во-первых, оставалось в заданном диапазоне `amin...amax`, и, во-вторых, соответствовало заданной точности `acc`. Изменяя угол в процессе поиска, мы можем выйти из допустимого диапазона углов возвышения, а уменьшая шаг по углу – получить значение угла, которое не может быть установлено механизмом нашей метательной машины. Допустим, метательная машина позволяет установить угол возвышения с точностью до 0.1º, текущее значение угла – 45º, а текущий шаг по углу – 0.5º. Если мы, согласно алгоритму, уменьшим шаг вдвое и попытаемся уменьшить угол возвышения на этот шаг, мы получим 45º – 0.25º = 44.75º. Такой угол не может быть установлен с точностью 0.1º, и нам придется выбрать ближайшее допустимое значение, то есть 44.8º. Функция `FixAngle` округляет значение угла до заданной точности, возвращая для переданного ей угла `a` ближайшее значение вида `amin` + N × `acc`, где N – целое число.

Вторая функция класса, `ShowResults`, служит для демонстрации результатов поиска пользователю. В ней значения найденного угла `OptAngle` и соответствующего ему промаха `OptMiss` преобразуются в динамически отводимые строки при помощи уже использовавшейся ранее сервисной функции РДС `rdsDtoA`. Затем при помощи функций `rdsDynStrCat` и `rdsAddToDynStr` из этих строк формируется сообщение пользователю, которое выводится функцией `rdsMessageBox`. В конце функции все динамически созданные строки уничтожаются при помощи `rdsFree`.

Для того, чтобы блок поиска угла можно было присоединить к блоку расчета траектории, и чтобы было где хранить настроечные параметры поиска, потребуется следующая структура переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Назначение*
0	Start	Сигнал	1	Вход	✓	Стандартный сигнал запуска
1	Ready	Сигнал	1	Выход		Стандартный сигнал готовности
2	MinAngle	double	8	Внутр.		Минимально возможный угол возвышения, градусов
10	MaxAngle	double	8	Внутр.		Максимально возможный угол возвышения, градусов
18	Accuracy	double	8	Внутр.		Точность установки угла, градусов
26	Distance	double	8	Внутр.		Дальность полета, для которой ищется угол возвышения, м

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Назначение*
34	x	double	8	Вход		Текущая горизонтальная координата снаряда, м (от блока расчета траектории)
42	Impact	Сигнал	1	Вход	✓	Сигнал о падении снаряда (от блока расчета траектории)
43	Angle	double	8	Выход		Текущий угол возвышения, градусов (к блоку расчета траектории)

Значения переменных `MinAngle`, `MaxAngle`, `Accuracy` и `Distance` будут задаваться пользователем в функции настройки блока – это параметры алгоритма поиска. Входы `x` и `Impact` нужно будет соединить с одноименными выходами блока расчета траектории снаряда: когда блок расчета определит точку падения снаряда и выдаст сигнал `Impact`, с входа `x` можно будет считать дальность полета снаряда. Выход `Angle` будет соединяться с одноименным входом блока расчета траектории, через него блок поиска угла передаст текущий угол возвышения, для которого нужно определить дальность.

Модель блока поиска угла будет такой:

```
// Модель блока поиска угла для заданной дальности
extern "C" __declspec(dllexport)
    int RDSCALL ArtSearch(int CallMode,
                          RDS_PBLOCKDATA BlockData,
                          LPVOID ExtParam)
{ TArtSearchData *data=(TArtSearchData*)(BlockData->BlockData);
  double delta;
// Макроопределения для переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))        // 0
#define Ready (*((char *)(pStart+1)))      // 1
#define MinAngle (*((double *)(pStart+2))) // 2
#define MaxAngle (*((double *)(pStart+10)))// 3
#define Accuracy (*((double *)(pStart+18)))// 4
#define Distance (*((double *)(pStart+26)))// 5
#define x (*((double *)(pStart+34)))       // 6
#define Impact (*((char *)(pStart+42)))    // 7
#define Angle (*((double *)(pStart+43)))   // 8
  // Массив описания параметров для универсальной функции настройки
  static char *setup[]={
    "2Минимальный угол, град.",
    "3Максимальный угол, град.",
    "4Точность по углу, град.",
    "5Дальность цели, м",
    NULL};

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        BlockData->BlockData=new TArtSearchData();
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;
```

```
// Проверка типов переменных
case RDS_BFM_VARCHECK:
  return strcmp((char*)ExtParam,"{SSDDDDDSD}")?
                    RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

// Вызов функции настройки
case RDS_BFM_SETUP:
  return SetupDoubleVars(BlockData->Block,
                         "Поиск угла",setup);

// Запуск расчета
case RDS_BFM_STARTCALC:
  switch(data->Mode)
    { case ASMODE_READY:
        // Начать поиск угла
        Angle=MaxAngle;   // Начальное значение
        data->OptMiss=DoubleErrorValue; // Пока не определено
        data->AngleStep=-(MaxAngle-MinAngle)/5;
        data->FirstStep=TRUE;
        data->Mode=ASMODE_SEARCHING;
        Ready=1; // Для передачи угла по связи
        break;
      case ASMODE_FINISHED:
        // Поиск уже проведен - показать результаты
        rdsStopCalc();
        data->ShowResults();
        break;
    }
  break;

// Сброс расчета
case RDS_BFM_RESETCALC:
  if(data->SelfReset)
    { // Блок сам сбросил подсистему
      data->SelfReset=FALSE; // Очиска признака самосброса
      // Устанавливаем новый угол
      Angle=data->AngleToSet;
      Ready=1; // Для передачи угла по связи
    }
  else // Расчет сброшен пользователем
    data->Mode=ASMODE_READY;
  break;

// Один такт моделирования
case RDS_BFM_MODEL:
  if(!Impact) // Снаряд еще не долетел
    break;

  // Снаряд долетел - дальность полета в x
  Impact=0; // Сбрасываем входной сигнал
  if(data->Mode==ASMODE_FINALRUN)
    { // Это был последний (демонстрационный) прогон
      data->Mode=ASMODE_FINISHED;
      rdsStopCalc();
      data->ShowResults();
      break;
    }
```

```
// Это - очередной прогон в поиске угла
delta=fabs(x-Distance); // Промах при угле Angle
if(data->OptMiss==DoubleErrorValue || // Первый прогон
   delta<data->OptMiss) // При новом угле попали точнее
  { // Запоминаем новый наилучший угол и промах
    data->OptAngle=Angle;
    data->OptMiss=delta;
    // Двигаемся дальше с тем же шагом
    data->AngleToSet=Angle+data->AngleStep;
    // Новый расчет будет уже не первым с этим шагом
    data->FirstStep=FALSE;
  }
else // Промах увеличился или не изменился
  { if(data->FirstStep)
      { // Мы только что провели первый расчет с новым
        // значением шага. Промах увеличился - попробуем
        // двигаться в обратном направлении с тем же
        // шагом
        data->AngleStep=-data->AngleStep;
        // Новый расчет будет уже не первым с этим шагом
        data->FirstStep=FALSE;
      }
    else
      { // Это был не первый расчет с данным значением
        // шага. Мы либо сделали несколько шагов в этом
        // направлении, либо уже пробовали двигаться
        // в обратном. Теперь нужно уменьшить шаг вдвое,
        //  если это возможно.
        if(fabs(data->AngleStep)<=Accuracy)
          { // Достигли минимального шага по углу -
            // выполняем последний расчет
            data->Mode=ASMODE_FINALRUN;
            // Устанавливаем найденный угол
            data->AngleToSet=data->OptAngle;
            data->SelfReset=TRUE; // Флаг самосброса
            // Сбрасываем родительскую подсистему
            rdsResetSystemState(BlockData->Parent);
            break;
          }
        // Минимальный шаг еще не достигнут
        // Дальше будем двигаться от наилучшего на данный
        // момент угла с меньшим шагом
        data->AngleStep=data->AngleStep/2.0;
        // Не даем шагу стать меньше минимального
        if(fabs(data->AngleStep)<Accuracy)
          data->AngleStep=(data->AngleStep<0)?
                                  (-Accuracy):Accuracy;
        // Новый расчет будет первым с этим значением шага
        data->FirstStep=TRUE;
      }
    // Новый угол, который установиться после сброса
    data->AngleToSet=data->OptAngle+data->AngleStep;
  }
// Ограничиваем угол и привязываем его к точности
data->AngleToSet=data->FixAngle(
             data->AngleToSet,MinAngle,MaxAngle,Accuracy);
```

```
            // Сбрасываем родительскую подсистему
            data->SelfReset=TRUE; // Флаг самосброса
            rdsResetSystemState(BlockData->Parent);
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef Angle
    #undef Impact
    #undef x
    #undef Distance
    #undef Accuracy
    #undef MaxAngle
    #undef MinAngle
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Первые четыре реакции модели не содержат ничего специфического: при инициализации блока создается личная область данных, при очистке эта область уничтожается, проверка типов переменных ничем не отличается от такой же реакции во всех остальных моделях, для настройки параметров блока, как и в предыдущей модели, используется универсальная функция `SetupDoubleVars` (массив параметров `setup` описан в начале функции модели, сразу после макроопределений для переменных).

Реакция модели на запуск расчета (`RDS_BFM_STARTCALC`) зависит от того, в каком режиме находится блок. Если поиск угла еще не проводился, то есть поле `Mode` класса личной области данных имеет значение `ASMODE_READY`, необходимо установить начальное значение угла, с которого мы начнем поиск (выбираем в качестве начального максимально возможный угол возвышения, то есть `MaxAngle`) и начальный шаг изменения угла (выбираем одну пятую всего диапазона, при этом мы будем уменьшать угол от максимального, поэтому шаг делается отрицательным). Полю `OptMiss`, в котором должно храниться наименьшее на данный момент значение промаха, мы присваиваем специальное значение – индикатор ошибки из глобальной переменной `DoubleErrorValue` (см. стр. 44), поскольку мы еще не провели ни одного расчета, и у нас пока нет ни наилучшего на данный момент значения угла, ни минимального значения промаха. Полю `FirstStep` присваивается значение `TRUE` (это самый первый расчет с таким значением шага), и блок переводится в режим `ASMODE_SEARCHING`: алгоритм поиска угла начал свою работу. Затем взводится сигнал `Ready`, чтобы установленное значение угла `Angle` передалось по связям, и реакция на этом завершается.

Если поиск угла уже завершен, в поле `Mode` будет находиться константа `ASMODE_FINISHED`. В этом случае запущенный расчет принудительно останавливается, и вызывается функция `ShowResults`, чтобы показать пользователю найденное значение угла и соответствующий ему промах. Во всех остальных случаях модель никак не реагирует на запуск расчета.

Следует отметить, что реакция на запуск расчета будет вызываться чаще, чем может показаться на первый взгляд. Дело в том, что наша модель будет время от времени сбрасывать состояние подсистемы при работающем расчете. РДС технически не может выполнить сброс, пока расчет работает, поэтому при вызове сервисной функции `rdsResetSystemState`, которой мы пользуемся для сброса, расчет сначала останавливается, затем состояние подсистемы сбрасывается, и расчет запускается снова. В результате модель нашего блока, как и модели всех остальных блоков схемы, будет вызываться в режимах `RDS_BFM_STOPCALC` и `RDS_BFM_STARTCALC` не только при

остановке и запуске расчета пользователем, но и при каждом сбросе состояния подсистемы. Тем не менее, на работу нашей модели этот факт никак не повлияет.

При сбросе расчета (режим RDS_BFM_RESETCALC) модель прежде всего проверяет, сам ли блок сбросил состояние подсистемы, или это сделал кто-то другой, то есть пользователь или какой-то другой блок. Для этого проверяется значение поля SelfReset: мы договорились, что перед сбросом состояния подсистемы модель будет присваивать этому полю значение TRUE. Таким образом, если значение SelfReset истинно, то блок сам сбросил расчет в процессе поиска угла, и модель устанавливает на выходе блока Angle новое значение из поля AngleToSet. Кроме этого, SelfReset снова сбрасывается в FALSE до следующего программного сброса, и взводится сигнал Ready для передачи выхода Angle по связям. Если же поле SelfReset имело значение FALSE, блок переводится в состояние готовности к поиску ASMODE_READY. Таким образом, если пользователь сбросит, а затем запустит расчет, поиск начнется заново.

Основная работа алгоритма поиска выполняется в реакции на такт расчета RDS_BFM_MODEL. Прежде всего проверяется значение сигнального входа Impact, на котором должна появиться единица, как только снаряд упадет на землю. Если Impact имеет нулевое значение, модель немедленно завершается: снаряд еще в полете, и дальность, соответствующая текущему углу возвышения, пока неизвестна. Как только сигнал Impact станет равным 1, модель сбрасывает его (любая модель должна самостоятельно сбрасывать все свои входные сигналы), а дальше действует в зависимости от текущего режима работы блока. При запущенном расчете блок может находиться в одном из двух состояний: ASMODE_SEARCHING в процессе поиска угла и ASMODE_FINALRUN при последнем, “демонстрационном”, расчете с найденным углом возвышения, который выполняется только для того, чтобы показать пользователю графики и значения, соответствующие именно этому, наилучшему, значению угла. Если поле Mode имеет значение ASMODE_FINALRUN, значит, только что был проведен последний расчет. При этом блок переводится в состояние ASMODE_FINISHED (поиск окончен), расчет останавливается, и вызывается функция ShowResults для демонстрации результатов пользователю. В противном случае поиск еще не завершен – нужно вычислять новое значение промаха, соответствующее текущему углу возвышения Angle, и сравнивать его с запомненным.

Значение промаха delta вычисляется как модуль разности текущей дальности полета, которая считывается с входа x, и заданной в настройках блока дальности Distance. Этот промах нужно сравнить с наименьшим на данный момент промахом OptMiss. После самого первого расчета, когда у нас еще нет наименьшего запомненного промаха, в OptMiss будет находится значение-индикатор ошибки DoubleErrorValue. В этом случае наилучшим на данный момент нужно считать текущее значение угла, поскольку никакого другого у нас еще нет. Текущее значение угла нужно считать наилучшим и в том случае, если с углом Angle промах получился меньше запомненного, то есть delta < data->OptMiss. Таким образом, в обоих этих случаях полю OptAngle присваивается текущее значение угла Angle, а полю OptMiss – текущий промах delta: текущее значение угла теперь будет наилучшим на данный момент. Раз изменение угла с текущим шагом улучшило ситуацию, мы продолжаем двигаться в том же направлении с тем же шагом: полю AngleToSet, откуда будет взято новое значение угла после программного сброса, присваивается сумма текущего значения Angle и текущего шага AngleStep. Затем сбрасывается логическое поле FirstStep: это будет уже не первый шаг в данном направлении.

Если же значение промаха delta получилось большим OptMiss, то есть сделанный шаг по углу привел к ухудшению ситуации, нам нужно либо изменить знак шага (если мы увеличивали угол – начать уменьшать его, если уменьшали – начать увеличивать), либо, если мы уже попробовали оба знака, уменьшить шаг вдвое. Если значение поля FirstStep

истинно, значит, мы еще не пробовали менять знак шага – в этом случае знак AngleStep меняется на противоположный, а FirstStep сбрасывается, поскольку это будет уже не первая попытка изменить угол с этим абсолютным значением шага. В противном случае нужно уменьшать шаг, но сначала нужно проверить, не достигли ли мы уже заданной точности по углу.

Если текущий шаг AngleStep по модулю меньше или равен точности Accuracy, значит, уменьшать его дальше невозможно. В этом случае блок переводится в режим ASMODE_FINALRUN, устанавливается наилучшее на данный момент значение угла возвышения, и проводится последний расчет, чтобы показать пользователю правильные графики. Для этого полю SelfReset присваивается значение TRUE (сейчас блок будет сам сбрасывать расчет) и вызывается функция rdsResetSystemState для родительской подсистемы данного блока, то есть для подсистемы, в которой он находится вместе с блоком расчета траектории. Затем функция модели завершается.

Если заданная точность еще не достигнута, шаг уменьшается в два раза и ограничивается снизу значением точности по углу Accuracy (мы не можем устанавливать угол точнее). Затем полю FirstStep присваивается значение TRUE, поскольку это будет первый расчет с новым значением шага – если новый шаг ухудшит ситуацию, нам нужно будет менять знак шага, а не его абсолютное значение.

Наконец, когда все ситуации, возникающие при увеличении промаха, рассмотрены, в поле AngleToSet записывается новое значение угла, которое нужно будет установить после сброса, то есть наилучший на данный момент угол OptAngle, к которому прибавлено новое значение шага AngleStep (оно либо поменяло знак, либо уменьшилось вдвое).

В самом конце реакции на такт расчета новое значение угла возвышения AngleToSet “пропускается” через функцию FixAngle, которая ограничит его диапазоном MinAngle...MaxAngle и округлит до точности Accuracy. Затем родительская подсистема сбрасывается в начальное состояние: реагируя на сброс, как было описано выше, модель нашего блока подаст на выход Angle новое значение AngleToSet, и расчет повторится снова.

Для тестирования созданной модели ArtSearch нужно поместить в одну подсистему блок с этой моделью и блок, моделирующий полет снаряда, и собрать схему, подобную изображенной на рис. 101.

В этой схеме выход Angle блока поиска угла подается на одноименный вход блока расчета траектории не напрямую, а через поле ввода – так удобнее наблюдать за изменяющимся в процессе поиска значением угла. Перед запуском расчета нужно настроить параметры блока поиска (рис. 102) и блока расчета траектории (в данном случае начальная скорость снаряда установлена в 200 м/с, а все остальные параметры блока соответствуют рис. 100). В процессе расчета график траектории будет постоянно изменяться, пока, в конце концов, блок поиска не проведет последний расчет в режиме ASMODE_FINALRUN и не выведет сообщение об окончании поиска.

Для каждой начальной скорости снаряда существует максимальная дальность полета, поэтому для дальностей, больших максимальной, невозможно подобрать угол возвышения, при котором промах станет близким к нулю. Наш блок в этом случае найдет такой угол, при котором промах будет наименьшим из всех возможных, то есть угол, соответствующий максимальной дальности полета. Для большинства дальностей, меньших максимально возможной, на самом деле существует два значения угла возвышения, при которых промах близок к нулю: в одну и ту же точку можно попасть как настильной траекторией (при малом угле возвышения), так и навесной (при большом угле). Наш блок найдет только один из этих углов, причем нельзя заранее сказать, какой именно. Это – недостаток данной реализации алгоритма поиска, но мы не будем его исправлять, чтобы не усложнять пример: со своей задачей он, в принципе, справляется.

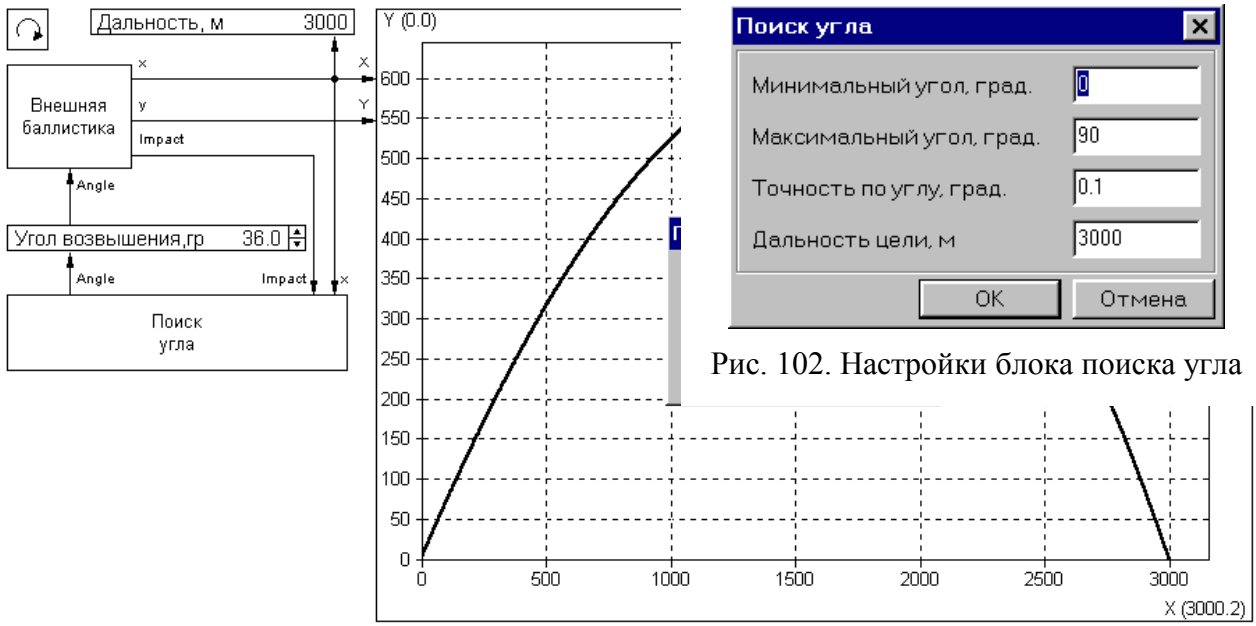


Рис. 102. Настройки блока поиска угла

Рис. 101. Схема поиска угла возвышения для заданной дальности

В рассмотренном примере очень важно расположить блок поиска угла в одной подсистеме со всеми блоками, ответственными за расчет (в нашем случае это блок моделирования полета снаряда и планировщик вычислений). Это связано с тем, что блок поиска угла сбрасывает свою родительскую подсистему со всеми подсистемами, вложенными в нее, поэтому, чтобы можно было сбрасывать расчет траектории и начинать его заново для другого значения угла возвышения, все блоки, участвующие в этом расчете, должны находиться в пределах действия этого сброса. Если мы вставим схему на рис. 101 в подсистему внутри другой подсистемы, “внешняя” подсистема в процессе работы алгоритма сбрасываться не будет. Этим можно воспользоваться, например, для построения графика зависимости дальности полета снаряда от угла возвышения. Для построения такого графика нужно создать блок, который будет изменять угол с заданным шагом, дожидаться конца расчета траектории и сбрасывать расчет. Этот блок нужно поместить в подсистему вместе с блоком расчета траектории, а сам график вместе с обслуживающими его блоками разместить снаружи этой подсистемы. При этом при сбросе внутренней подсистемы график сбрасываться не будет, и сможет, точка за точкой, построить кривую зависимости дальности от угла.

Прежде всего, нужно создать модель блока, который будет менять угол с заданным шагом и сбрасывать состояние подсистемы после каждого изменения. Эта модель будет гораздо проще модели блока поиска угла `ArtSearch` – здесь не нужно ни вычислять промах, ни изменять шаг по углу. Однако, в логике работы этого блока есть свои особенности, на которых нужно остановиться подробнее.

Блок поиска угла, рассмотренный ранее, получал от блока расчета траектории сигнал `Impact` вместе с новым значением дальности, вычислял промах и тут же сбрасывал подсистему в начальное состояние, начиная новый расчет. В новом блоке, который будет менять угол для построения графика, сбрасывать подсистему сразу же при получении сигнала `Impact` нельзя. Дело в том, что значение дальности полета здесь будет передаваться наружу подсистемы, в график, поэтому расчет можно сбрасывать только тогда, когда это значение будет принято и обработано блоком графика. Мы не знаем заранее, когда это случится: между подсистемой и графиком может оказаться длинная цепочка из блоков, которые как-то обрабатывают это значение. Фактически, нам нужно задержать сигнал `Impact`, сформированный блоком расчета траектории, на некоторое, заранее неизвестное,

число тактов. Если этого не сделать, подсистема вместе со всеми своими переменными, включая вычисленную дальность полета, может сброситься раньше, чем график успеет считать со своего входа очередное значение дальности. Конечно, можно подсчитать число простых блоков во всей схеме, и задержать сброс подсистемы на такое же число тактов, тогда график точно успеет сработать. Однако, есть более простое решение – можно заставить блок изменения угла ждать не только окончания расчета траектории, но и готовности графика и остальных блоков, находящихся снаружи подсистемы. Для этого нужно предусмотреть в блоке два сигнальных входа. На один из них будет подаваться сигнал окончания расчета траектории `Impact`, а на другой – сигнал готовности графика снаружи подсистемы. Только когда оба этих сигнала станут равными единице, блок сбросит подсистему. Формирование этого сигнала готовности снаружи подсистемы – отдельный вопрос, и мы займемся им, когда будем создавать схему. А пока можно заняться моделью блока изменения угла, который будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Назначение*
0	Start	Сигнал	1	Вход	✓	Стандартный сигнал запуска
1	Ready	Сигнал	1	Выход		Стандартный сигнал готовности
2	MinAngle	double	8	Внутр.		Минимально возможный угол возвышения, градусов
10	MaxAngle	double	8	Внутр.		Максимально возможный угол возвышения, градусов
18	Accuracy	double	8	Внутр.		Шаг изменения угла, градусов
26	Impact	Сигнал	1	Вход	✓	Сигнал о падении снаряда (от блока расчета траектории)
27	GraphReady	Сигнал	1	Вход	✓	Сигнал готовности графика и других блоков снаружи подсистемы
28	Angle	double	8	Выход		Текущий угол возвышения, градусов

Сама модель будет такой:

```
//=========================================
// Блок перебора углов для построения графика
// зависимости дальности от угла
//=========================================
// Личная область данных блока
class TArtWalkData
{ public:
    BOOL SelfReset;   // Блок сам сбросил подсистему
    double AngleToSet;// Угол, который нужно установить
                      // после сброса
    BOOL FirstStart;  // Начать построение графика с начала

    // Конструктор класса
    TArtWalkData(void)
      { SelfReset=FALSE; FirstStart=TRUE; };
};
//=========================================
// Функция модели блока
extern "C" __declspec(dllexport)
    int RDSCALL ArtWalk(int CallMode,
                        RDS_PBLOCKDATA BlockData,
```

```
                          LPVOID ExtParam)
{ TArtWalkData *data=(TArtWalkData*)(BlockData->BlockData);
// Макроопределения для переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))                          // 0
#define Ready (*((char *)(pStart+1)))                        // 1
#define MinAngle (*((double *)(pStart+2)))                   // 2
#define MaxAngle (*((double *)(pStart+10)))                  // 3
#define Accuracy (*((double *)(pStart+18)))                  // 4
#define Impact (*((char *)(pStart+26)))                      // 5
#define GraphReady (*((char *)(pStart+27)))                  // 6
#define Angle (*((double *)(pStart+28)))                     // 7
  // Массив описания параметров для универсальной функции настройки
  static char *setup[]={
    "2Минимальный угол, град.",
    "3Маскимальный угол, град.",
    "4Шаг по углу, град.",
    NULL};

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        BlockData->BlockData=new TArtWalkData();
        break;
      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;
      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSDDDSSD}")?
                          RDS_BFR_BADVARSMSG:RDS_BFR_DONE;
      // Вызов функции настройки
      case RDS_BFM_SETUP:
        return SetupDoubleVars(BlockData->Block,
                               "Перебор углов",setup);

      // Запуск расчета
      case RDS_BFM_STARTCALC:
        if(data->FirstStart)
          { // Самый первый запуск - начало графика
            Angle=MinAngle; // Начинаем с начала диапазона
            data->FirstStart=FALSE;
            Impact=GraphReady=0;
            Ready=1; // Для передачи по связям
          }
        break;

      // Сброс расчета
      case RDS_BFM_RESETCALC:
        if(data->SelfReset)
          { // Блок сам сбросил подсистему
            data->SelfReset=FALSE;
            // Устанавливаем новый угол
            Angle=data->AngleToSet;
            Ready=1; // Для передачи по связям
          }
```

```
            else // Расчет сброшен кем=то еще
              data->FirstStart=TRUE;
            break;

          // Один такт расчета
          case RDS_BFM_MODEL:
            if(!Impact) // Снаряд не долетел
              { // Сбрасываем сигнал готовности графика
                GraphReady=0;
                break;
              }
            // Снаряд долетел - проверяем готовность графика
            if(!GraphReady)
              break;
            // График готов
            Impact=GraphReady=0; // Сбрасываем оба сигнала
            if(Angle>=MaxAngle)
              { // Достигли конца диапазона - прекращаем расчет
                rdsStopCalc();
                break;
              }
            // Увеличиваем угол на один шаг
            data->AngleToSet=Angle+Accuracy;
            if(data->AngleToSet>MaxAngle) // Не допускаем выход
              data->AngleToSet=MaxAngle;  // за MaxAngle
            data->SelfReset=TRUE; // Сейчас блок сам сбросит подсистему
            rdsResetSystemState(BlockData->Parent);
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef Angle
    #undef GraphReady
    #undef Impact
    #undef Accuracy
    #undef MaxAngle
    #undef MinAngle
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Не будем подробно останавливаться на личной области данных этого блока: фактически, она представляет собой урезанный вариант класса `TArtSearchData`, в котором остался только флаг самостоятельного сброса `SelfReset` и поле `AngleToSet` для хранения значения угла, которое нужно установить после сброса. Вместо целого поля для режима работы блока в этом классе используется логическое `FirstStart` – блок может находиться всего в двух режимах: либо он ничего не делает в данный момент и готов к работе (`FirstStart` истинно), либо он работает, увеличивая угол шаг за шагом (`FirstStart` ложно).

Из всех реакций этой модели подробно рассмотрим только три: `RDS_BFM_STARTCALC`, `RDS_BFM_RESETCALC` и `RDS_BFM_MODEL`, поскольку остальные устроены очевидным образом и похожи на реакции других моделей, уже рассматривавшихся раньше. В реакции на запуск расчета `RDS_BFM_STARTCALC` модель проверяет состояние блока, и, если он до сих пор ничего не делал, инициализирует выход `Angle` началом диапазона углов `MinAngle`, сбрасывает флаг `FirstStart` и входные сигналы `Impact` и `GraphReady` – теперь блок установил на своем выходе самое первое значение угла, для

которого нужно вычислить дальность, и готов к приему сигнала конца расчета траектории Impact и сигнала готовности графика GraphReady. После этого взводится выходной сигнал Ready, чтобы значение Angle передалось по связям в блок расчета траектории.

Реакция на сброс расчета RDS_BFM_RESETCALC, в целом, похожа на реакцию предыдущей модели: если блок сам сбросил подсистему, на выход выдается новое значение угла из поля AngleToSet, если нет – переменной FirstStart присваивается значение TRUE, при следующем запуске расчета это приведет к тому, что блок снова начнет перебирать углы от начала диапазона с заданным шагом.

В реакции на такт расчета RDS_BFM_MODEL прежде всего проверяется, закончен ли расчет траектории, то есть взведен ли сигнал Impact. Если он сброшен, значит, снаряд еще летит, и нас не интересует сигнал готовности графика GraphReady (он сбрасывается), и функция модели на этом завершается. Если же сигнал Impact взведен, нужно ждать готовности графика: проверяется сигнал GraphReady, и, если он сброшен, модель завершается. Если оба сигнала взведены, значит, и расчет траектории уже закончен, и график уже считал нужные ему значения, в этом случае работа функции модели будет продолжена.

Может возникнуть вопрос: если нам нужно продолжать работу модели и изменять угол только после поступления обоих сигналов Impact и GraphReady, почему бы не проверить эти сигналы одновременно, написав проверку так:

```
if(Impact && GraphReady)
  { Impact=GraphReady=0;
    // ... дальнейшие действия ...
  }
```

Почему в модели сначала проверяется сигнал Impact, и, если он не взведен, GraphReady сбрасывается? Дело в том, что мы пока не знаем, как и какими блоками будет формироваться сигнал, поступающий на вход GraphReady. Если он будет взят с выхода готовности одного из блоков непосредственно перед графиком, не исключены ложные срабатывания сигнала, если этот блок по какой-либо причине запустится до завершения расчета траектории. Мы точно знаем только одно: при нормальной работе схемы сигнал готовности графика может прийти только после завершения расчета траектории, то есть сигнал GraphReady может быть взведен только после того, как взведется сигнал Impact. Таким образом, все срабатывания GraphReady до Impact можно считать ложными, и сбрасывать их. Без этого нам пришлось бы жестко требовать от пользователя, который будет собирать схему на основе нашего блока, исключения ложных срабатываний GraphReady, что, несомненно, усложнит его работу.

Вернемся к реакции модели на такт расчета. Когда оба входных сигнала взведены, модель сбрасывает их и сравнивает текущий угол с верхней границей диапазона. Если угол достиг конца диапазона, значит, все углы уже перебраны, и расчет завершается. В противном случае к текущему значению угла Angle добавляется шаг Accuracy, и сумма записывается в поле AngleToSet личной области данных – после сброса расчета это значение будет переписано в выход блока Angle и передастся в блок расчета траектории. Затем взводится флаг SelfReset и родительская подсистема сбрасывается в начальное состояние.

Теперь, используя созданный блок, можно собрать схему для построения графика, изображенную на рис. 103. В этой схеме блок-планировщик, блок перебора углов и блок расчета траектории помещены в подсистему Sys1, которая, в свою очередь, помещена в корневую подсистему. Устанавливаемое блоком перебора значение угла Angle подается на одноименный вход блока расчета траектории и выведено на внешний выход подсистемы. Сигнал Impact с блока расчета траектории подан на одноименный вход блока перебора углов и на внешний выход OK подсистемы. Горизонтальная координата снаряда x выведена наружу подсистемы через внешний выход L. Наконец, с внешнего входа подсистемы Ready

сигнал подается на вход GraphReady блока перебора углов – это и есть сигнал готовности графика, который формируется снаружи подсистемы Sys1.

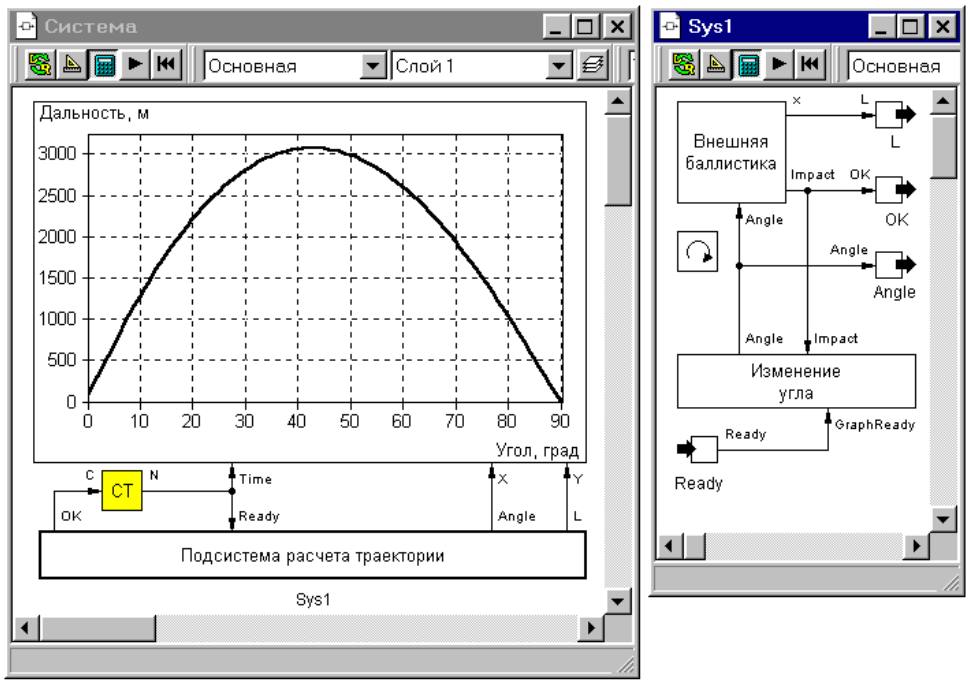


Рис. 103. Построение графика зависимости дальности от угла

Снаружи подсистемы Sys1 все устроено достаточно просто. Сигнал OK (к нему внутри подсистемы подключен сигнал Impact) подан на вход счетчика, который, таким образом, будет считать число расчетов траектории, увеличивая свой выход N каждый раз, как снаряд будет падать на землю. Выход N счетчика соединен со входом времени Time двухкоординатного графика. Чтобы схема работала, в параметрах этого графика нужно указать, что значение времени нужно брать не из динамической переменной “DynTime”, а с входа Time – в этом случае график будет запоминать очередной отсчет при изменении этого входа. К входу X графика присоединен выход подсистемы Angle, а к входу Y – выход L. Таким образом, на момент окончания расчета траектории, у графика на входе X окажется значение угла возвышения, а на входе Y – соответствующая ему дальность. В качестве сигнала готовности графика (внешний вход Ready подсистемы) используется выход счетчика N: в РДС с сигнальным входом можно соединять выход любого типа, при этом сигнал взведется при срабатывании связи, идущей от этого выхода, какое бы значение ни было передано по этой связи.

Схема будет работать следующим образом: при самом первом запуске расчета блок перебора выдаст на выход Angle начало диапазона углов, и блок расчета траектории начнет моделировать полет снаряда. На выходе счетчика в корневой подсистеме, а, значит, и на входе Time графика при этом будет начальное значение счетчика, то есть 0. На входе X графика установится начальное значение угла, а на вход Y будет постоянно передаваться меняющаяся горизонтальная координата снаряда. Как только снаряд упадет на землю, блок расчета траектории выдаст сигнал Impact, который попадет на вход блока перебора углов и, через выход подсистемы OK, на вход счетчика. Блок перебора углов не сделает ничего, поскольку он пока еще не получил сигнал готовности графика. Счетчик увеличит значение на 1, и это значение одновременно попадет на вход Time графика и взведет сигнал GraphReady у блока перебора углов через внешний вход Ready подсистемы. График при этом запомнит очередной отсчет “угол–дальность”, а блок перебора сбросит расчет. Затем все повторится снова: по окончании расчета траектории счетчик увеличит выходное

значение, график запишет очередной отсчет, расчет будет сброшен и т.д. Так будет продолжаться до тех пор, пока угол не достигнет верхней границы диапазона. В результате получится график зависимости дальности полета снаряда от угла возвышения для заданного диапазона углов.

График на рис. 103 построен для начальной скорости 200 м/с в диапазоне углов 0...90º с шагом по углу 0.1º. Поскольку ось нашей метательной машины поднята на 1 м от поверхности земли (см. рис. 100), при нулевом угле возвышения дальность полета получилась не нулевой. При угле возвышения 90º, то есть при выстреле вертикально вверх, снаряд, как и следовало ожидать, имеет нулевую дальность полета. Видно, что дальности 3000 м, для которой при тех же параметрах блоком поиска угла был найден угол возвышения 36º (рис. 101), соответствует еще и угол около 50º. Таким образом, в прошлый раз наш блок поиска угла нашел угол возвышения для настильной траектории.

Оба рассмотренных примера используют сброс подсистемы для повторного проведения расчета с новыми параметрами. В принципе, можно было бы написать блок расчета траектории снаряда так, чтобы его можно было перезапускать при помощи специального входного сигнала, тогда программный сброс расчета не понадобился бы. Однако, это несколько усложнило бы модель блока. Кроме того, многие стандартные блоки (например, графики) и блоки, написанные сторонними программистами, могут не иметь возможности перезапуска, поэтому при работе с ними программный сброс расчета – единственный выход. Использование программного сброса для многократного проведения расчета делает схему универсальной, поскольку она при этом не зависит от особенностей реализации моделей входящих в нее блоков: если они работают при сбросе и запуске расчета пользователем, они будут работать и при программном сбросе.

§2.14.3. Сохранение и загрузка состояния блоков

Описывается механизм возврата блоков в запомненное состояние. Рассматривается пример, в котором запоминание состояния подсистемы и возврат к этому состоянию производятся по команде пользователя.

Рассмотренная в предыдущем параграфе сервисная функция `rdsResetSystemState` позволяет вернуть всю схему или отдельную подсистему в исходное состояние, то есть в состояние, предшествовавшее первому запуску расчета. В РДС есть еще один механизм, позволяющий “вернуть в прошлое” какой-либо блок или подсистему: вызовом сервисной функции `rdsSaveSystemState` можно сохранить в памяти текущее состояние одного или нескольких блоков, а затем, вызвав `rdsLoadSystemState`, вернуть эти блоки в запомненное состояние. Этот механизм используется значительно реже, чем программный сброс, однако, в некоторых случаях он тоже может быть полезен. В системе может быть запомнено произвольное число состояний блоков, каждое из которых получает уникальный целый идентификатор.

Функция `rdsSaveSystemState` принимает следующие параметры:

```
int RDSCALL rdsSaveSystemState(
      RDS_BHANDLE Block,    // Блок или подсистема
      int num,              // Идентификатор сохраняемого состояния
      BOOL Recursive,       // Сохранять ли вложенные блоки
      RDS_BBhpB CallBack);  // Указатель на функцию проверки
                            // необходимости сохранения или NULL
```

В параметре `Block` передается идентификатор блока или подсистемы, чье состояние необходимо сохранить. Если в этом параметре передан идентификатор подсистемы, а в параметре `Recursive` – значение `TRUE`, то будет сохранено не только состояние самой подсистемы, но и состояния всех ее внутренних блоков и подсистем. В параметре `num` передается идентификатор запомненного состояния, которое нужно заменить на сохраняемое, или –1, если необходимо запомнить новое состояние. Функция возвращает

идентификатор запомненного состояния: если в параметре num был передан идентификатор существующего запомненного состояния, будет возвращено значение num, если же в num было передано значение –1 или любое другое целое число, не соответствующее ни одному из существующих идентификаторов, будет создан и возвращен новый уникальный идентификатор запомненного состояния.

В последнем параметре функции (CallBack) может быть передан указатель на функцию вида

```
BOOL func(RDS_BHANDLE block,BOOL *pContinue);
```

Если этот параметр не равен NULL, указанная функция будет вызываться для каждого блока, состояние которого должно быть сохранено. Состояние блока block будет сохранено только в том случае, если функция вернет TRUE. При этом, если функция запишет по указателю, переданному в параметре pContinue, значение FALSE, перебор блоков и сохранение их состояний будет остановлено. Таким образом, при сохранении состояния подсистем с вложенными блоками, можно гибко управлять тем, какие именно блоки должны сохраниться.

Для загрузки ранее запомненного состояния служит функция rdsLoadSystemState:

```
BOOL RDSCALL rdsLoadSystemState(
     int num,              // Идентификатор загружаемого состояния
     RDS_BBhpB CallBack);  // Указатель на функцию проверки
                           // необходимости загрузки или NULL
```

В параметре num передается уникальный идентификатор загружаемого состояния. Идентификатор подсистемы или блока, состояние которого будет загружено, не передается: загружаются состояния тех блоков, которые были запомнены с идентификатором num. Если после сохранения часть блоков была удалена, ничего страшного не произойдет – будут загружены состояния только тех блоков, которые остались в схеме. В параметре CallBack, если необходимо, передается указатель на функцию обратного вызова того же формата, который использовалась в rdsSaveSystemState, только теперь она определяет необходимость загрузки состояния конкретного блока.

При сохранении состояния блока запоминаются значения всех его статических переменных. Затем, поскольку РДС не может работать с личной областью данных блока, а в ней могут находиться какие-то параметры, связанные с состоянием этого блока, модель блока вызывается в режиме RDS_BFM_SAVESTATE. Внутри этого вызова модель может сохранить необходимые параметры при помощи уже знакомой нам функции записи в двоичном виде rdsWriteBlockData (см. стр. 154). Сохранение состояния блока в текстовом формате не предусмотрено, поскольку проблемы с совместимостью форматов здесь вряд ли возникнут: запомненное состояние хранится только в памяти и теряется при выгрузке схемы или при смене модели блока. При загрузке состояния блока значения статических переменных восстанавливаются, а затем модель блока вызывается в режиме RDS_BFM_LOADSTATE для загрузки параметров личной области данных, если это необходимо, при помощи функции rdsReadBlockData. Таким образом, следует помнить, что блоки, текущие параметры которых хранятся не только в статических переменных, должны поддерживать загрузку и сохранение этих параметров в режимах RDS_BFM_SAVESTATE/RDS_BFM_LOADSTATE.

В отличие от многих других вспомогательных объектов, создаваемых блоками, сохраненное состояние никак не привязано к блоку, вызвавшему функцию rdsSaveSystemState, и не будет удалено из памяти при его удалении. При выгрузке всей схемы оно, разумеется, будет удалено, чтобы не вызывать утечек памяти, но до этого момента оно будет оставаться в памяти и может быть в любой момент загружено любым блоком схемы, которому известен целый идентификатор этого состояния (сохранивший блок может передать его другому блоку по связи или через вызов функции блока). Если

сохраненное состояние больше не нужно, оно может быть принудительно удалено из памяти сервисной функцией `rdsDeleteSystemState`, в которую передается идентификатор удаляемого состояния.

В качестве примера рассмотрим простой блок, который будет сохранять и загружать состояние своей родительской подсистемы по щелчку мыши. Обработку щелчков мыши мы возьмем из модели блока, увеличивающего и уменьшающего значение выхода (см. стр. 262) – будем сохранять состояние при щелчке левой кнопкой мыши по верхней части изображения блока и загружать его при щелчке по нижней части. Кроме того, дадим пользователю возможность нарисовать кнопки сохранения и загрузки в редакторе картинки блока: если у блока есть картинка, будем определять идентификатор ее элемента под курсором мыши (см. пример на стр. 265), для положительных идентификаторов будем сохранять состояния, для отрицательных – загружать.

Нам потребуется где-то хранить целый идентификатор сохраненного состояния, чтобы его можно было потом загрузить. Статические переменные блока для этого не подходят – они изменяются при загрузке состояния родительской подсистемы, и мы потеряем этот идентификатор. Будем хранить его в личной обрасти данных блока, которая будет представлять собой единственное число типа `int`. Статические переменные нашему блоку не нужны (разумеется, у него будут два обязательных сигнала `Start` и `Ready`, но мы не будем ими пользоваться). Модель блока будет достаточно простой:

```
// Запись и загрузка состояния
extern "C" __declspec(dllexport)
    int RDSCALL SaveLoadState(int CallMode,
                              RDS_PBLOCKDATA BlockData,
                              LPVOID ExtParam)
{ RDS_PMOUSEDATA mouse;
  // Указатель на личную обрасть данных блока (int)
  int *pSaveId=(int*)(BlockData->BlockData);

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Создание личной области данных (одно число int)
        BlockData->BlockData=pSaveId=new int;
        // Исходное значение идентификатора: -1 (то есть нет)
        *pSaveId=-1;
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        // Удаление сохраненного состояния (если есть)
        rdsDeleteSystemState(*pSaveId);
        // Уничтожение личной области
        delete pSaveId;
        break;

      // Нажатие кнопки мыши
      case RDS_BFM_MOUSEDOWN:
        mouse=(RDS_PMOUSEDATA)ExtParam;
        if(mouse->Button==RDS_MLEFTBUTTON) // Левая кнопка
          { RDS_BLOCKDESCRIPTION descr;
            descr.servSize=sizeof(descr);
            BOOL save=FALSE,load=FALSE;
            // Есть ли у блока картинка?
            rdsGetBlockDescription(BlockData->Block,&descr);
```

```
                if(descr.Flags & RDS_BDF_HASPICTURE)
                  { // Картинка есть - смотрим идентификатор элемента
                    // под курсором
                    int pic_id=rdsGetMouseObjectId(mouse);
                    if(pic_id>0)
                      save=TRUE;
                    else if(pic_id<0)
                      load=TRUE;
                  }
                else if(mouse->y<mouse->Top+mouse->Height/2)
                  save=TRUE; // Картинки нет, верх блока
                else
                  load=TRUE; // Картинки нет, низ блока
                if(save) // Сохраняем состояние
                  *pSaveId=rdsSaveSystemState(BlockData->Parent,
                                *pSaveId,TRUE,NULL);
                if(load) // Загружаем состояние
                  { rdsLoadSystemState(*pSaveId,NULL);
                    // Необходимо обновить окно подсистемы
                    rdsRefreshBlockWindows(BlockData->Parent,TRUE);
                  }
              }
            break;
        }
      return RDS_BFR_DONE;
    }
    //=======================================
```

При инициализации модели мы создаем личную область данных блока размером в одно целое число оператором `new int`, а затем записываем в нее число –1, которое означает, что у нас пока нет сохраненного состояния. При очистке модели мы удаляем сохраненное состояние, идентификатор которого содержится в личной области и на которое указывает переменная `pSaveId`, при помощи функции `rdsDeleteSystemState`. Если мы ни разу не сохраняли состояние, и там осталось значение –1, ничего страшного не произойдет: если число, переданное в функцию `rdsDeleteSystemState`, не соответствует никакому идентификатору сохраненного состояния, функция не выполнит никаких действий. Затем оператором `delete` удаляется сама личная область данных.

Загрузка и сохранение состояний выполняется в реакции на нажатие кнопки мыши `RDS_BFM_MOUSEDOWN`. Прежде всего, нажатая кнопка сравнивается с константой `RDS_MLEFTBUTTON`, обозначающей левую кнопку мыши. Если они совпали, мы проверяем, есть ли у блока картинка. Для блоков с картинкой считывается идентификатор элемента под курсором мыши и, в зависимости от его идентификатора, значение `TRUE` присваивается вспомогательной переменной `load` или `save`. Для блоков без картинки значение `TRUE` присваивается переменной `save` при попадании в верхнюю часть блока, и переменной `load` – при попадании в нижнюю. Таким образом, после всех этих действий, переменная `save` будет истинной, если мы должны сохранить состояние подсистемы, а переменная `load` – если мы должны загрузить его. Если значение `save` истинно, вызывается функция `rdsSaveSystemState` для сохранения состояния родительской подсистемы этого блока (`BlockData->Parent`) со всеми вложенными блоками (в параметре `Recursive` передается `TRUE`). В качестве идентификатора сохраняемого состояния передается значение, хранящееся в личной области данных блока (`*pSaveId`), туда же записывается результат возврата функции. Таким образом, при самом первом сохранении состояния, когда в личной области данных находится значение –1, функция добавит состояние к набору уже сохраненных (если они есть) и создаст для него новый идентификатор, который запишется в

личную область блока. При всех последующих сохранениях функция будет заменять сохраненное состояние с этим идентификатором на новое.

Если истинно значение `load`, вызывается `rdsLoadSystemState` для загрузки состояния с идентификатором `*pSaveId`. Попытка загрузить состояние, не записывая его (при этом в личной области данных блока будет находиться –1) не приведет к ошибкам, функция просто не выполнит никаких действий. После загрузки состояния вызывается функция `rdsRefreshBlockWindows` для принудительного обновления окна подсистемы, поскольку параметры ее блоков изменились, и это могло отразиться на их внешнем виде.

Для проверки работы созданной модели следует создать новый блок и подключить к нему эту модель, разрешив ей реагировать на действия пользователя мышью. Внешним видом блока можно сделать прямоугольник со словами “Записать” и “Загрузить”, расположенными друг под другом. Рядом с ним следует собрать схему, подобную изображенной на рис. 104. В этой схеме генератор синусоидальных колебаний подключен к двум графикам: в верхней части схемы расположен созданный нами ранее блок-график (стр. 190), в нижней – стандартный график из библиотеки блоков РДС. Если запустить расчет, щелкнуть по верхней части созданного блока, а затем, подождав некоторое время, щелкнуть по его нижней части, можно будет увидеть, как блоки в подсистеме вернутся в состояние на момент первого щелчка.

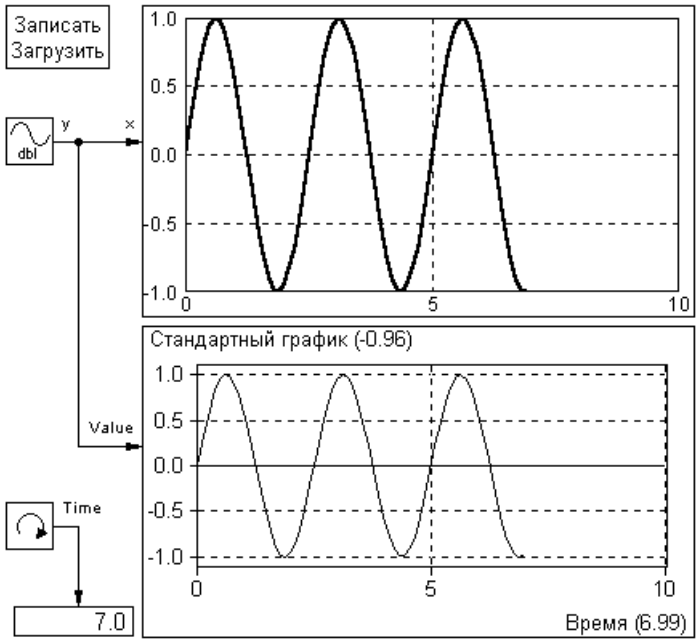


Рис. 104. Схема для проверки работы блока сохранения и загрузки состояния подсистемы

На самом деле, в это состояние вернутся все блоки, кроме верхнего, созданного нами, графика – он просто перестанет рисоваться. Дело в том, что все текущие параметры этого блока, включая массивы отсчетов, хранятся в личной области данных. Когда мы писали модель этого блока, мы не включили в нее поддержку режимов `RDS_BFM_SAVESTATE` и `RDS_BFM_LOADSTATE`, поэтому блок не реагирует на запись и загрузку своего состояния. Сейчас мы исправим эту ошибку.

Прежде всего, добавим в класс личной области данных блока-графика (см. стр. 190) описания двух новый функций для записи и загрузки состояния – мы будем вызывать их из функции модели блока:

```
//=======================================
// Простой график - личная область данных
//=======================================
class TSimplePlotData
{ // ...
  // ...
  // ...
    void Draw(RDS_PDRAWDATA DrawData);     // Функция рисования

    void SaveState(void);  // Функция записи состояния
    void LoadState(void);  // Функция загрузки состояния
```

```
    TSimplePlotData(void); // Конструктор класса
    ~TSimplePlotData();    // Деструктор класса
};
//=======================================
```

Эти функции будут устроены очень просто: при сохранении состояния мы должны записать все изменяющиеся параметры блока, включая массивы отсчетов, при помощи функции `rdsWriteBlockData`, а при загрузке – загрузить их при помощи `rdsReadBlockData` (при этом нужно будет заново отвести массивы отсчетов перед их непосредственной загрузкой, поскольку их размер мог измениться). Функция записи будет выглядеть так:

```
// Запись состояния блока
void TSimplePlotData::SaveState(void)
{
  // Макрос для записи одной переменной
  #define WRITEBLOCKDATAVAR(v) rdsWriteBlockData(&v,sizeof(v))

  WRITEBLOCKDATAVAR(TimeStep);
  WRITEBLOCKDATAVAR(Xmin);
  WRITEBLOCKDATAVAR(Xmax);
  WRITEBLOCKDATAVAR(XGridStep);

  // Запись массивов отсчетов
  rdsWriteBlockData(Times,Count*sizeof(double));
  rdsWriteBlockData(Values,Count*sizeof(double));

  WRITEBLOCKDATAVAR(NextIndex);
  WRITEBLOCKDATAVAR(NextTime);

  // Отмена макроса
  #undef WRITEBLOCKDATAVAR
}
//=======================================
```

В этой функции, чтобы для каждой переменной `A` не писать конструкцию вида

```
rdsWriteBlockData(&A,sizeof(A));
```

мы вводим макрос `WRITEBLOCKDATAVAR`. Сначала мы записываем параметры горизонтальной оси графика `TimeStep`, `Xmin` и `Xmax`, которые однозначно определяют размер массивов отсчетов `Count`. Затем, на всякий случай, сохраняется шаг сетки горизонтальной оси `XGridStep` – вдруг пользователь изменил его после записи состояния. Параметры вертикальной оси мы не записываем, хотя можно было бы записать и их: будем считать, что изменения, внесенные пользователем в настройки этой оси, отменять при загрузке состояния не нужно. После этого мы записываем оба массива отсчетов `Times` и `Values`, с которыми работает график, и переменные `NextIndex` и `NextTime`, определяющие текущую точку записи в эти массивы. На этом работа функции завершается – мы записали все важные или изменяющиеся со временем переменные из личной области данных блока.

Функция загрузки состояния будет очень похожа на функцию записи:

```
// Загрузка состояния блока
void TSimplePlotData::LoadState(void)
{
  // Макрос для загрузки одной переменной
  #define READBLOCKDATAVAR(v) rdsReadBlockData(&v,sizeof(v))

  READBLOCKDATAVAR(TimeStep);
  READBLOCKDATAVAR(Xmin);
  READBLOCKDATAVAR(Xmax);
```

```
    READBLOCKDATAVAR(XGridStep);

    // Отведение массивов перед загрузкой
    AllocateArrays();
    rdsReadBlockData(Times,Count*sizeof(double));
    rdsReadBlockData(Values,Count*sizeof(double));

    READBLOCKDATAVAR(NextIndex);
    READBLOCKDATAVAR(NextTime);

    // Отмена макроса
    #undef READBLOCKDATAVAR

    // Сброс параметров оптимизации рисования
    OldZoom=-1.0;
    LastDrawnIndex=-1;
  }
  //=======================================
```

В этой функции мы тоже вводим макрос, упрощающий загрузку одной переменной, а затем начинаем загружать переменные в том же порядке, в котором мы их записывали. Перед загрузкой массивов отсчетов вызывается функция `AllocateArrays`, которая отведет эти массивы заново согласно новым загруженным параметрам горизонтальной оси графика, и вычислит значение `Count`. В самом конце функции необходимо сбросить переменные, которые мы используем для ускорения работы функции рисования графика (см. §2.10.2) – параметры графика изменились, и прежние значения вспомогательных переменных рисования теперь не годятся, их нужно будет вычислить заново.

В оператор `switch(CallMode)` внутри функции модели `SimplePlot` необходимо внести два новых оператора `case` для вызова функций загрузки и записи состояния:

```
    // Запись состояния блока
    case RDS_BFM_SAVESTATE:
      data->SaveState();
      break;

    // Загрузка состояния блока
    case RDS_BFM_LOADSTATE:
      data->LoadState();
      break;
```

Это все изменения, которые нужно внести в модель блока-графика. Теперь при загрузке состояния подсистемы в схеме на рис. 104 в запомненное состояние будут возвращаться оба графика.

Механизм возврата в запомненное состояние может использоваться для проведения сложных вычислений (например, если какая-то переменная схемы отклонилась от заданного значения, можно “вернуться в прошлое” и скорректировать параметры, чтобы не допустить этого отклонения), для создания пользовательских точек остановки расчета, к которым можно возвращаться по команде, и т.п. Следует только помнить, что запомненные состояния существуют только в памяти и никак не могут быть сохранены для работы при следующей загрузке схемы.

§2.14.4. Отдельный расчет подсистемы

Рассматривается механизм временной остановки расчета всей схемы кроме одной единственной подсистемы. Приводится пример блока, переводящего родительскую подсистему в режим отдельного расчета для того, чтобы цепочка блоков в этой подсистеме не создавала паразитных задержек.

Собирая схемы в РДС, нужно всегда помнить, что в режиме расчета каждый блок срабатывает за один такт, поэтому значение на выходе цепочки из N последовательно соединенных блоков установится только через N тактов, даже если это алгебраические блоки, то есть блоки, выходы которых теоретически должны были бы вычисляться без какой-либо задержки. Мы уже сталкивались с задержками, которые возникают из-за цепочек блоков, в §2.14.2 при обсуждении установки начальных значений на входе блока расчета баллистики. В схемах с планировщиком вычислений, который управляет динамической переменной времени “DynTime”, это не так важно, поскольку в таких схемах основные вычисления производятся в моменты изменения времени, а в параметрах планировщика можно задать такое количество дополнительных тактов, за которое все цепочки алгебраических блоков успеют сработать. Однако, в логических схемах и схемах, работающих по тактам, зависимость времени установки значения от длины цепочки последовательно соединенных блоков может привести к временному появлению неверных значений на выходе некоторых блоков.

Рассмотрим, например, схему на рис. 105. В ней значение A с поля ввода подается на две параллельные ветви: в одной блок Sum1 прибавляет к этому значению число 6, в другой – блоки Sum2, Sum3 и Sum4 три раза последовательно прибавляют к нему число 2. Выходы обоих ветвей вычитаются. Все блоки в этой схеме взяты из библиотеки, и их модели запускаются при срабатывании любой из подключенных к входам связей.

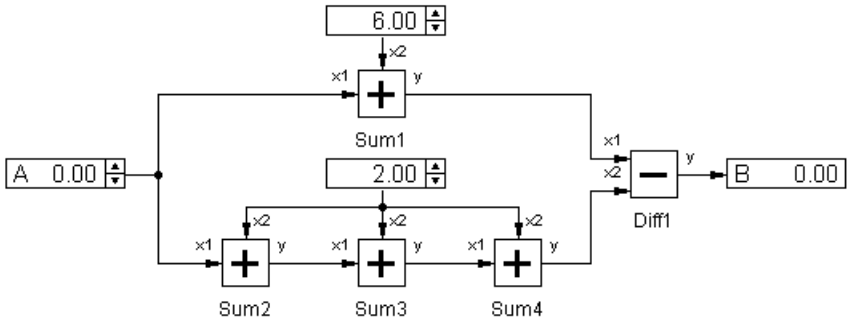


Рис. 105. Параллельные цепочки блоков разной длины

С точки зрения математики, значение B на выходе блока вычитания Diff1 всегда должно быть нулевым, поскольку верхняя ветвь реализует формулу A+6, а нижняя – A+2+2+2. Однако, если рассмотреть работу этой схемы по тактам, можно заметить, что нулевое значение на выходе Diff1 установится не сразу. Допустим, в исходном состоянии A=0 и на выходах всех блоков тоже нулевые значения. Изменим значение A на 1 и посмотрим, что будет происходить в схеме.

В первом после изменения A такте расчета запустятся модели блоков Sum1 и Sum2, поскольку сработала связь, соединяющая их входы с полем ввода A. Таким образом, в конце первого такта на выходе блока Sum1 появится число 7, а на выходе Sum2 – число 3.

В втором такте запустятся модели блоков Diff1 (сработала связь, соединяющая его с Sum1) и Sum3 (из-за связи с Sum2). На выходе Sum3 появится значение 5 (3+2), а на выходе Diff1 – значение 7 (7–0). Таким образом, из-за того, что верхняя ветвь сработала быстрее, на выходе схемы появилось ненулевое значение – блок Sum4 еще не успел сработать.

В третьем такте запустится модель блока Sum4, и на его выходе появится значение 7 (5+2). Модель блока Diff1 работать не будет, поэтому на его выходе останется значение 7.

Наконец, в четвертом такте модель блока Diff1, наконец, запустится из-за срабатывания связи, соединяющей его с Sum4, и на его выходе установится нулевое (7–7) значение.

В конце концов, когда все блоки схемы сработали, на ее выходе появилось правильное, нулевое, значение. Однако, в течение двух тактов расчета вместо нуля на выходе схемы находилось значение 7. Если бы к этой схеме была присоединена другая, это выброс мог бы повлиять на нее. Таким образом, при сборке схем нужно всегда иметь в виду, что правильные значения на выходах иногда устанавливаются не сразу, и принимать по этому поводу соответствующие меры (например, вводить сигналы готовности, привязанные к самой длинной цепочке, или задерживать работу блоков на некоторое количество тактов).

В качестве одного из способов борьбы с задержками можно использовать встроенный в РДС механизм отдельного расчета подсистемы, позволяющий временно останавливать расчет всех подсистем схемы, кроме одной. Это позволит поместить цепочку блоков в подсистему и остановить расчет остальной схемы до тех пор, пока вся цепочка не сработает. Конечно, этот механизм можно применять и с другими целями, но в качестве примера рассмотрим именно такое его использование.

Для включения и выключения отдельного расчета подсистемы используется сервисная функция `rdsSetExclusiveCalc`:

```
BOOL RDSCALL rdsSetExclusiveCalc(
      RDS_BHANDLE system,   // Идентификатор подсистемы
      BOOL on);             // Включить (TRUE) или выключить (FALSE)
```

В функцию передается идентификатор подсистемы, для которой нужно включить или выключить отдельный расчет, и логическое значение, определяющее собственно включение или выключение. В случае успешного включения отдельного расчета функция возвращает `TRUE`, при возникновении ошибок – `FALSE`. Вызовы этих функций могут быть вложены: включив отдельный расчет подсистемы, можно потом включить отдельный расчет другой подсистемы, вложенной в нее. При выключении отдельного расчета второй подсистемы РДС вернется к отдельному расчету первой, пока он тоже не будет выключен.

Отдельный расчет подсистемы можно включить, только если РДС находится в режиме расчета, в других режимах вызов этой функции будет проигнорирован. Следует также учитывать, что отдельный расчет включается не сразу, а только после того, как отработают все модели блоков, которые должны быть вызваны в текущем такте. Таким образом, он не может включиться в середине такта расчета – только после него или перед ним (для выполнения различных действий непосредственно перед тактом расчета предусмотрен специальный режим вызова модели `RDS_BFM_PREMODEL`, который будет рассмотрен в следующем параграфе).

Прежде чем разбираться с самим механизмом отдельного расчета, сделаем один вспомогательный блок, который поможет нам наблюдать отставание или опережение срабатывания параллельных ветвей схемы. Этот блок должен будет ждать изменения входной вещественной переменной, и, после него, начать считать такты расчета, выдавая количество прошедших тактов на свой выход. Таким образом, мы сможем построить график зависимости значения на выходе схемы (например, выхода блока Diff1 на рис. 105) от номера такта расчета. Кроме того, блок должен останавливать расчет после заданного числа тактов (вручную мы не успеем его остановить – такты следуют друг за другом слишком быстро) и иметь вход разрешения работы, чтобы можно было отключить его на время начальных переходных процессов в схеме при первом запуске расчета. Зададим для блока следующую структуру переменных (начальные значения всех переменных – нулевые):

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Назначение*
0	Start	Сигнал	1	Вход	✓	Стандартный сигнал запуска

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Пуск*	*Назначение*
1	Ready	Сигнал	1	Выход		Стандартный сигнал готовности
2	Count	int	4	Выход		Число тактов, прошедших с момента изменения входа x
6	Stop	int	4	Вход		Число тактов, после которого нужно остановить расчет (настроечный параметр)
10	Enabled	Логический	1	Вход	✓	Вход разрешения работы блока
11	x	double	8	Вход	✓	Вход, изменение которого отслеживается
19	xold	double	8	Внутр.		Предыдущее значение входа x для отслеживания его изменения

Модель блока будет иметь следующий вид:

```
// Счетчик тактов
extern "C" __declspec(dllexport)
    int RDSCALL ChgCalcTickCount(int CallMode,
                        RDS_PBLOCKDATA BlockData,
                        LPVOID ExtParam)
{
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Count (*((int *)(pStart+2)))
#define Stop (*((int *)(pStart+6)))
#define Enabled (*((char *)(pStart+10)))
#define x (*((double *)(pStart+11)))
#define xold (*((double *)(pStart+19)))
  switch(CallMode)
    { // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSIILDD}")?
                          RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Запуск расчета
      case RDS_BFM_STARTCALC:
        Start=1; // Принудительный запуск модели в следующем такте
        break;

      // Один такт моделирования
      case RDS_BFM_MODEL:
        if(!Enabled)  // Работа блока не разрешена
          { xold=x; // Запоминаем значение входа
            break;
          }
        // Работа блока разрешена
        if(xold==x && Count==0)
          break;      // Вход не изменился или счет не идет
        // Изменился вход (x!=xold) или уже считаем (Count!=0)
        Count++;            // Увеличиваем число тактов
        if(Count>Stop)      // Пора остановить расчет
          rdsStopCalc();
```

```
            Start=1;            // Принудительный перезапуск модели
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef xold
    #undef x
    #undef Enabled
    #undef Stop
    #undef Count
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

При запуске расчета (режим `RDS_BFM_STARTCALC`) эта модель на всякий случай взводит сигнал `Start`, чтобы запустить себя в первом же такте расчета. В такте расчета (`RDS_BFM_MODEL`) прежде всего проверяется состояние логического входа `Enabled`, разрешающего работу блока. Если он имеет нулевое значение, значит, работа запрещена. В этом случае внутренней переменной `xold` присваивается значение входа `x` (таким образом отключенный блок будет игнорировать изменения входа, поскольку текущее значение входа всегда будет равно запомненному) и работа модели завершается. Если же работа блока разрешена, модель проверяет, нужно ли увеличить на единицу счетчик тактов `Count`. Блок должен начать считать такты при изменении входа `x`, а затем продолжать счет независимо от изменений этого входа, поэтому проверка состоит из двух частей: во-первых, значение `x` сравнивается с запомненным `xold` (если они отличаются, значит, вход изменился), и, во-вторых, значение `Count` сравнивается с нулем (если `Count!=0`, значит, счет уже начался, и его нужно продолжать). Если вход не изменился и значение `Count` нулевое, модель завершается (она автоматически запустится снова при изменении `Enabled` или `x`, поскольку для этих переменных установлен флаг “Пуск”). В противном случае `Count` увеличивается на 1 и сравнивается с параметром `Stop`: как только `Count` превысит `Stop`, расчет будет остановлен функцией `rdsStopCalc`. В самом конце взводится сигнал `Start`, чтобы модель снова запустилась в следующем такте. Можно было бы просто установить для блока флаг “Запуск каждый такт”, но такое принудительное взведение `Start` улучшает надежность модели – она будет работать независимо от того, установлен ли этот флаг в параметрах блока.

Теперь можно собрать схему с двумя параллельными ветвями разной длины, подключенными к вычитающему блоку, и использовать созданный нами блок для построения графика зависимости выхода этого вычитающего блока от номера такта (рис. 106).

Верхняя ветвь схемы состоит из единственного блока “Kx+C”, созданного нами ранее (см. стр. 136). В нижнюю ветвь схемы включена подсистема, реализующая ту же самую формулу при помощи двух библиотечных блоков: блока умножения на константу и сумматора. Значения `x`, `K` и `C` подаются в обе ветви с одних и тех же полей ввода, поэтому выходы обеих ветвей должны быть одинаковыми. Выходы блока и подсистемы поданы на блок вычитания, выход которого, в свою очередь, выведен на график. На вход времени графика `Time` подан сигнал с выхода `Count` нашего блока-счетчика тактов, при этом в параметрах графика указано, что значения времени он должен брать не из динамической переменной “DynTime”, как он это делает по умолчанию, а со своего входа. Дополнительно, для улучшения внешнего вида графика, для него установлен тип линии “ступенька посередине” – процесс у нас дискретный, и кусочно-линейный график здесь ни к чему.

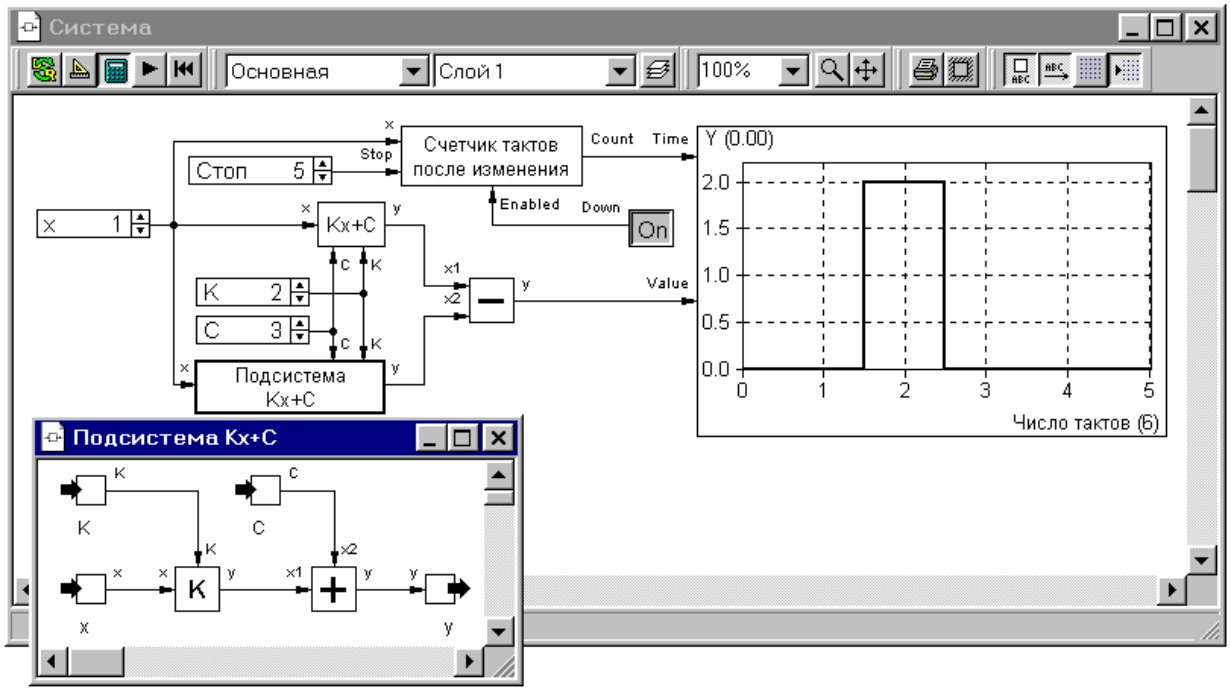


Рис. 106. Схема сравнения скоростей срабатывания одиночного блока и цепочки блоков

С входом `Enabled` блока-счетчика соединена кнопка “On”, к входу `Stop` подключено поле ввода со значением 5, а к входу `x` – связь от того же самого поля ввода, которое идет на вход обеих параллельных ветвей схемы. Таким образом, если запустить расчет, нажать кнопку “On”, а затем изменить значение поля ввода “x”, блок-счетчик начнет каждый такт увеличивать значение своего выхода `Count` на 1. При этом будет строиться график разности выходов двух ветвей схемы в каждом такте расчета. Отработав 5 тактов, блок остановит расчет, и мы сможем на графике увидеть выбросы, возникшие из-за разного времени срабатывания двух ветвей.

В данном случае мы видим выброс вверх длиной в один такт расчета. Из-за того, что верхняя ветвь состоит из одного блока, а нижняя – из двух, значение на выходе подсистемы запоздало на один такт, в течение которого вход `x1` блока вычитания уже имел правильное значение 2, а вход `x2` все еще оставался нулевым.

Попробуем избавиться от этого выброса, заставив подсистему в нижней ветви считаться отдельно. Как только сработает одна из входных связей подсистемы, мы будем останавливать расчет всей остальной схемы, возобновляя его только после срабатывания последнего блока в цепочке, то есть сумматора. В результате, с точки зрения остальной схемы, подсистема будет считаться за один такт, и задержки в двух ветвях окажутся одинаковыми.

Нам потребуется создать блок с двумя входными сигналами `Lock` и `Unlock`, который по сигналу `Lock` будет переводить родительскую подсистему в режим отдельного расчета, а по сигналу `Unlock` – возобновлять нормальный режим работы. Подключив к сигналу `Lock` сигналы с внешних входов подсистемы, а к сигналу `Unlock` – выход сумматора, мы добьемся нужного результата. Как только на вход подсистемы поступит новое значение, сработает связь, ведущая к сигналу `Lock`, и расчет всей остальной схемы остановится. Когда будет готово значение на выходе сумматора, сработает связь, ведущая к сигналу `Unlock`, и расчет остальной схемы продолжится. Помимо двух сигнальных входов в этом блоке нам потребуется еще и логическая переменная состояния `Locked` – она позволит игнорировать повторные сигналы `Lock` при включенном отдельном расчете и повторы `Unlock` при выключенном.

Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/ выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	0
1	Ready	Сигнал	1	Выход		0
2	Lock	Сигнал	1	Вход	✓	0
3	Unlock	Сигнал	1	Вход	✓	0
4	Locked	Логический	1	Внутр.		0

Теперь можно написать модель блока:

```
// Включение/выключение отдельного расчета
extern "C" __declspec(dllexport)
  int RDSCALL SetExclusiveCalc(int CallMode,
                      RDS_PBLOCKDATA BlockData,
                      LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Lock (*((char *)(pStart+2)))
#define Unlock (*((char *)(pStart+3)))
#define Locked (*((char *)(pStart+4)))

  switch(CallMode)
    { // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSSSL}")?
                                RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Такт расчета
      case RDS_BFM_MODEL:
        if(Lock) // Поступил сигнал включения отдельного расчета
          { if(Locked)     // Отдельный расчет уже включен
              { Lock=0;    // Сбрасываем сигнал, завершаем модель
                break;
              }
            // Включаем отдельный расчет родительской подсистемы
            if(rdsSetExclusiveCalc(BlockData->Parent,TRUE))
              { // Включить удалось
                Lock=0;    // Сбрасываем сигнал
                Locked=1;  // Запоминаем факт включения
              }
          }
        if(Unlock)    // Поступил сигнал выключения
          { if(Locked) // Отдельный расчет был включен - выключаем
              rdsSetExclusiveCalc(BlockData->Parent,FALSE);
            Locked=Lock=Unlock=0; // Сбрасываем сигналы и состояние
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Locked
#undef Unlock
```

```
#undef Lock
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Модель получилась достаточно простой. В такте расчета мы сначала проверяем, не пришел ли сигнал `Lock`. Если он взведен, мы проверяем, не включен ли уже отдельный расчет родительской подсистемы (для запоминания факта включения мы используем логическую переменную состояния `Locked`), и, если он включен, мы сбрасываем сигнал `Lock` и завершаем работу модели – повторные попытки включить режим отдельного расчета игнорируются блоком. Если же он не был включен, включаем его функцией `rdsSetExclusiveCalc`, в которую передаем идентификатор родительской подсистемы блока `BlockData->Parent`, сбрасываем `Lock` и взводим переменную состояния `Locked`.

В случае поступления сигнала `Unlock` мы выключаем отдельный расчет, если он был включен, и сбрасываем и поступивший сигнал, и переменную состояния `Locked`.

Изменим теперь схему на рис. 106 так, чтобы подсистема в нижней ветви считалась с приостановкой расчета остальной схемы. Для этого включим в нее созданный нами блок отдельного расчета, подключим к его входу `Lock` все входные связи подсистемы, а выход сумматора подадим на вход `Unlock` (рис. 107). В результате этого при срабатывании любой входной связи подсистемы сигнал `Lock` взведется, и подсистема перейдет в режим отдельного расчета. Как только сумматор сработает и данные его выхода передадутся по связи наружу подсистемы, взведется сигнал `Unlock`, и расчет остальной части схемы возобновится.

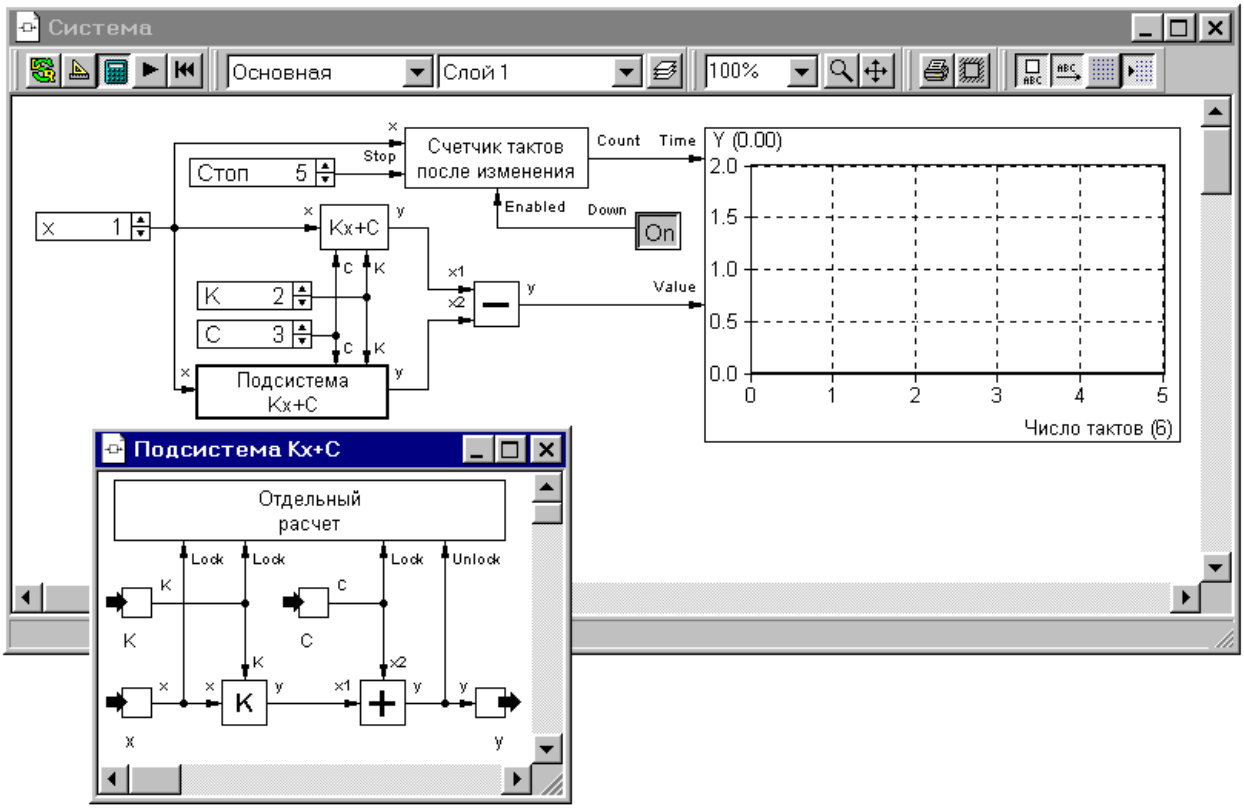


Рис. 107. Подключение блока для отдельного расчета подсистемы

Если теперь запустить расчет, нажать кнопку “On” и изменить значение в поле ввода “x”, можно будет увидеть, что на графике нет никаких выбросов: поскольку расчет всей

схемы останавливался на время расчета подсистемы, получилось, что обе параллельных ветви схемы сработали одновременно.

Важно помнить, что механизм отдельного расчета влияет только на блокировку вызовов моделей остальной схемы в режимах выполнения такта расчета (RDS_BFM_MODEL) и выполнения действий перед тактом расчета (RDS_BFM_PREMODEL, будет рассмотрен в следующем параграфе). Все остальные действия – реакции на мышь и клавиатуру, на изменение динамических переменных, на срабатывание таймеров, рисование и т.п. – модели остальной части схемы продолжают выполнять. Если, например, блок в подсистеме изменит какую-либо динамическую переменную и вызовет сервисную функцию rdsNotifyDynVarSubscribers для уведомления подписчиков этой переменной об изменениях, в режиме RDS_BFM_DYNVARCHANGE будут вызваны не только модели подсистемы, считающейся отдельно, но и модели блоков за ее пределами, несмотря на то, что расчет этой части схемы приостановлен. Как правило, это мало влияет на расчет, поскольку выходы этих блоков все равно не будут передаваться по связям, пока расчет остальной части схемы не будет возобновлен. Однако, если, по каким-либо причинам, нужно приостановить обработку и этих действий, об этом нужно позаботиться в программе модели.

Допустим, что у нас есть модель блока, делающая что-то по таймеру и при изменении динамической переменной. В функции этой модели, очевидно, будет находиться оператор switch следующего вида:

```
switch(CallMode)
  { // .....

    // Срабатывание таймера
    case RDS_BFM_TIMER:
      // РЕАКЦИЯ НА ТАЙМЕР
      break;

    // Изменение динамической переменной
    case RDS_BFM_DYNVARCHANGE:
      // РЕАКЦИЯ НА ДИНАМИЧЕСКУЮ ПЕРЕМЕННУЮ
      break;

    // Такт расчета
    case RDS_BFM_MODEL:
      // ДЕЙСТВИЯ В ТАКТЕ РАСЧЕТА
      break;
  } // switch
```

Для того, чтобы заблокировать эти две реакции на время приостановки расчета данного блока, нужно ввести в переменные этого блока пару логических переменных для фиксации произошедшего события (например, TimerOk для таймера и DynVarOk для изменения динамической переменной) с нулевыми начальными значениями, и изменить модель следующим образом:

```
switch(CallMode)
  { // .....

    // Срабатывание таймера
    case RDS_BFM_TIMER:
      TimerOk=1;      // Запоминаем факт срабатывания таймера
      Start=1;        // Запускаем модель в следующем такте
      break;

    // Изменение динамической переменной
    case RDS_BFM_DYNVARCHANGE:
      DynVarOk=1;     // Запоминаем факт изменения переменной
```

```
      Start=1;        // Запускаем модель в следующем такте
      break;

    // Такт расчета
    case RDS_BFM_MODEL:
      if(TimerOk)     // Было срабатывание таймера
        { // РЕАКЦИЯ НА ТАЙМЕР
          TimerOk=0;  // Реакция выполнена, больше помнить не нужно
        }
      if(DynVarOk)    // Было изменение переменной
        { // РЕАКЦИЯ НА ДИНАМИЧЕСКУЮ ПЕРЕМЕННУЮ
          DynVarOk=0; // Реакция выполнена, больше помнить не нужно
        }
      // ДЕЙСТВИЯ В ТАКТЕ РАСЧЕТА
      break;
  } // switch
```

В этой модели при срабатывании таймера или изменении динамической переменной взводятся соответствующие логические переменные и, вместе с ними, флаг запуска `Start`. Сами реакции на события при этом выполняются в реакции на такт расчета. В результате, если расчет блока будет приостановлен, реакция на таймер или динамическую переменную будет отложена до того момента, когда блок снова начнет участвовать в расчете и его модель будет запущена в режиме `RDS_BFM_MODEL`.

§2.14.5. Вызов модели блока перед тактом расчета

Рассматривается специальный режим вызова модели блока непосредственно перед тактом расчета (RDS_BFM_PREMODEL). Описанный в предыдущем параграфе пример блока, ускоряющего работу подсистемы, модифицируется так, чтобы блоки подсистемы срабатывали до блоков остальной части схемы.

При выполнении такта расчета блоки, которые должны сработать в этом такте, вызываются в случайном порядке, и разработчик модели блока никак не может повлиять на то, какой по счету будет вызвана его модель. Все события, произошедшие в одном такте расчета, считаются одновременными, поэтому порядок выполнения моделей на самом деле не важен. Однако, иногда возникает необходимость сделать так, чтобы модель какого-либо блока была вызвана до того, как начнется очередной такт расчета. Особенно это важно, если эта модель выполняет действия, влияющие на расчет всей схемы.

РДС позволяет вызвать модель блока перед тактом расчета – для этого существует специальный режим `RDS_BFM_PREMODEL`. Этот вызов будет производиться только в том случае, если модель блока при инициализации установила в поле `Flags` структуры данных блока `RDS_BLOCKDATA` битовый флаг `RDS_CTRLCALC`. По умолчанию этот флаг сброшен, поэтому без его явной установки модели блоков в режиме `RDS_BFM_PREMODEL` не вызываются. Флаг можно устанавливать не только при инициализации – главное установить его до перехода в режим моделирования. После перехода в режим моделирования (и, тем более, запуска расчета) установка этого флага будет отложена РДС до возврата в режим редактирования – это связано с внутренней логикой работы РДС.

Если в схеме имеются блоки, установившие флаг `RDS_CTRLCALC`, то перед каждым тактом расчета эти блоки вызываются в режиме `RDS_BFM_PREMODEL`, если у них взведен сигнал запуска (первая сигнальная переменная, `Start`) или установлен флаг “запуск каждый такт”. Порядок вызова блоков в этом режиме, как и при выполнении такта расчета, определяется внутренней логикой РДС и не может быть изменен программистом. Затем, как обычно, выполняется вызов моделей блоков в режиме `RDS_BFM_MODEL` и передача данных по связям. Фактически, вызов перед тактом расчета позволяет разбить действия блока в такте расчета на две части: сначала для всех блоков выполняется первая часть в режиме `RDS_BFM_PREMODEL`, затем – вторая, в режиме `RDS_BFM_MODEL`. Таким образом можно,

например, выполнить какие-либо действия с соседними блоками до того, как они выполнят такт расчета, или запретить блоку работу в такте расчета, принудительно сбросив его сигнал `Start`.

Важно помнить, что в режиме `RDS_BFM_PREMODEL` запускаются только блоки, готовые к срабатыванию в очередном такте. Если, например, блок работает по сигналу, и после очередного такта расчета его вход `Start` имеет нулевое значение, этот блок не будет вызван перед следующим тактом расчета, даже если его модель установила флаг `RDS_CTRLCALC`. Также следует учитывать, что вызов перед тактом расчета не будет производиться, если расчет блока приостановлен из-за вызова функции `rdsSetExclusiveCalc`, описанной в предыдущем параграфе.

Ранее мы создали блок, приостанавливающий расчет схемы до завершения расчета родительской подсистемы, и добились с его помощью того, что, с точки зрения схемы, подсистема со всем своим содержимым стала выполняться за один такт расчета, как и простой блок в параллельной ветви (см. рис. 107). Теперь, используя вызов модели перед тактом расчета, добьемся того, чтобы подсистема выполнялась быстрее простого блока, то есть мгновенно, за ноль тактов. Если мы приостановим расчет остальной схемы перед очередным тактом, дадим всем блокам подсистемы сработать, а потом возобновим расчет, получится, что вся подсистема сработала перед этим остановленным тактом, чего мы и добиваемся. Нам потребуется только сделать новую модель блока отдельного расчета, перенеся включение и выключение отдельного расчета в реакцию на вызов перед тактом. Новая модель будет выглядеть так:

```
// Включение/выключение отдельного расчета - вариант
extern "C" __declspec(dllexport)
  int RDSCALL SetExclusiveCalc0(int CallMode,
                      RDS_PBLOCKDATA BlockData,
                      LPVOID ExtParam)
{ // Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define Lock (*((char *)(pStart+2)))
#define Unlock (*((char *)(pStart+3)))
#define Locked (*((char *)(pStart+4)))

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        BlockData->Flags|=RDS_CTRLCALC;     // Разрешаем вызов
                                            // перед тактом
        break;

      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSSSL}")?
                                RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Вызов перед тактом расчета
      case RDS_BFM_PREMODEL:
        if(Lock) // Поступил сигнал включения отдельного расчета
          { if(Locked)      // Отдельный расчет уже включен
              { Lock=0;     // Сбрасываем поступивший сигнал
                Start=0;    // Не работаем в такте расчета
                break;
              }
```

```
            // Включаем отдельный расчет родительской подсистемы
            if(rdsSetExclusiveCalc(BlockData->Parent,TRUE))
              { // Включить удалось
                Lock=0;    // Сбрасываем поступивший сигнал
                Start=0;   // Не работаем в такте расчета
                Locked=1;  // Запоминаем факт включения
              }
          }
        if(Unlock)    // Поступил сигнал выключения
          { if(Locked) // Отдельный расчет был включен - выключаем
              rdsSetExclusiveCalc(BlockData->Parent,FALSE);
            Locked=Lock=Unlock=0; // Сбрасываем сигналы и состояние
            Start=0;  // Не работаем в такте расчета
          }
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Locked
#undef Unlock
#undef Lock
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Новая модель `SetExclusiveCalc0` мало отличается от старой `SetExclusiveCalc`. Во-первых, в нее добавлена реакция на инициализацию `RDS_BFM_INIT`, в которой взводится битовый флаг `RDS_CTRLCALC`. Во-вторых, все действия, раньше выполнявшиеся в такте расчета, теперь перенесены в реакцию `RDS_BFM_PREMODEL`. И, в-третьих, в эти действия добавлен сброс флага `Start`, чтобы после вызова перед тактом расчета блок не вызывался в самом такте – у него все равно нет реакции `RDS_BFM_MODEL`, и это было бы напрасной тратой процессорного времени.

Теперь, если заменить модель блока отдельного расчета в схеме на рис. 107 на новую, запустить расчет, нажать кнопку “On” и изменить значение в поле ввода “x”, на графике можно будет увидеть выброс вниз шириной в один такт (рис. 108). Это означает, что данные на вход `x2` блока вычитания пришли раньше, чем на вход `x1`, то есть что подсистема в нижней ветви схемы сработала раньше на один такт, чем простой блок в верхней ветви.

Вызов модели блока перед тактом расчета применяется не так уж часто, но он бывает полезен в тех случаях, когда необходимо заставить модели каких-либо блоков гарантированно выполняться раньше всех остальных моделей.

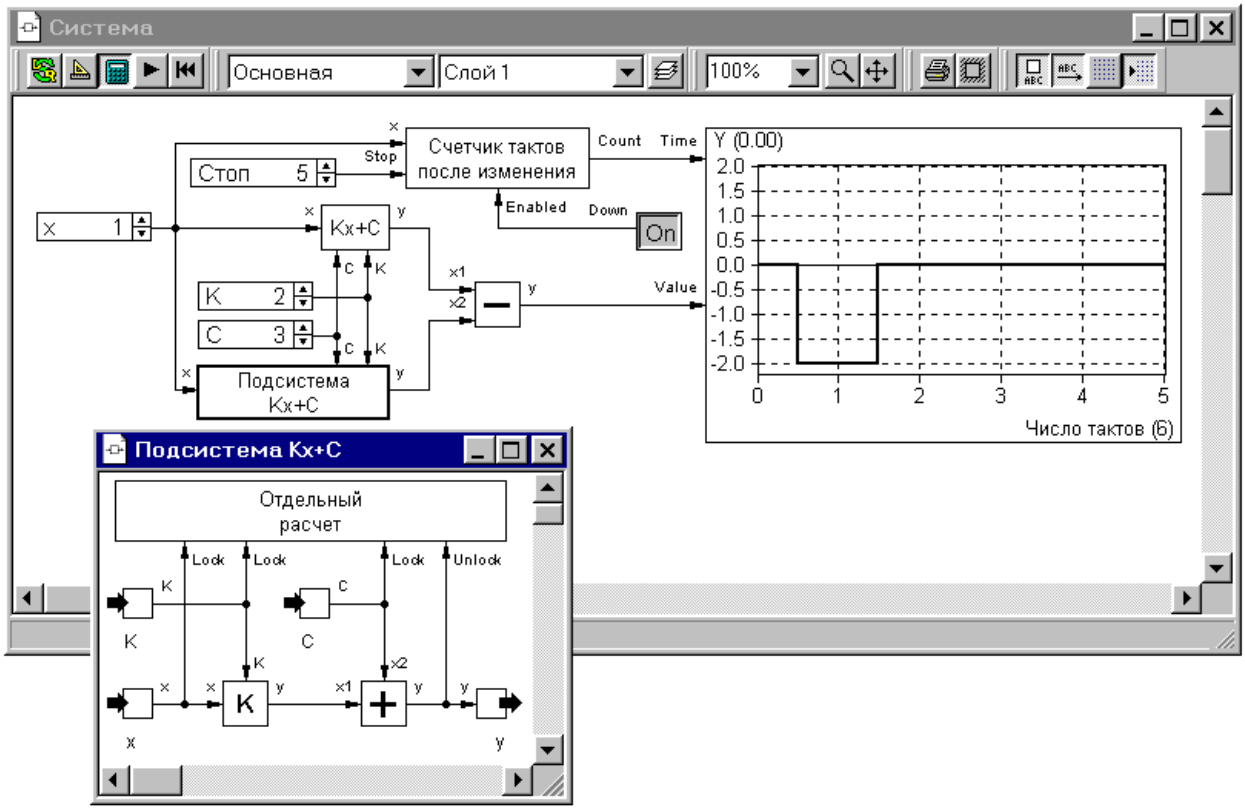


Рис. 108. Подсистема выполняется быстрее простого блока

§2.15. Обмен данными по сети

Рассматриваются способы обмена данными между схемами, работающими на разных машинах. Приводятся примеры моделей блоков, осуществляющих такой обмен.

§2.15.1. Общие принципы обмена данными по сети в РДС

Рассматривается механизм обмена данными по сети между блоками РДС. Описываются основные сервисные функции и структуры и способ их использования.

Поскольку модели блоков в РДС являются просто функциями в составе динамически подключаемой библиотеки (DLL), им доступны все функции Windows API, и они могут самостоятельно создавать сетевые соединения и обмениваться данными друг с другом, как это делают обычные программы. Тем не менее, в РДС включен специализированный механизм сетевого обмена, облегчающий работу программистам, которые не хотят писать сетевые функции самостоятельно. Конечно, этот механизм менее гибок, чем универсальные функции API, но свою задачу он выполняет: с его помощью блок может передать произвольные данные одному или нескольким блокам в схемах, запущенных на других машинах.

В РДС используется клиент-серверная модель обмена данными. Одна из машин в сети (точнее, одна из запущенных на ней копий “rds.exe”) должна стать сервером, через который будет проходить весь поток данных. Все остальные будут передавать ей данные и получать их от нее. РДС может работать в режиме выделенного сервера, то есть заниматься только организацией обмена данными между клиентами (для этого нужно запустить “rds.exe” с параметром командной строки “/server default” или “/server *номер_порта*”) или обслуживать работу какой-либо схемы, выполняя при этом еще и функции сервера. Если в сети есть свободная машина, можно использовать первый вариант, если нет – второй. Технически, первый вариант предпочтительнее, поскольку во втором варианте критические ошибки в

моделях блоков загруженной схемы могут привести к аварийному завершению РДС, при этом вся сеть лишится сервера.

Для того, чтобы блоки схемы могли обмениваться данными с другими схемами, в сетевых настройках РДС (рис. 109) должен быть разрешен сетевой обмен – в противном случае все попытки установить соединение с другими машинами или запустить сервер будут блокироваться. Разумеется, все это относится только к встроенному механизму сетевого обмена РДС: если модель блока установит соединение с помощью функций Windows API, никакие настройки РДС не смогут ей помешать. В настройках также обычно указывается имя или IP-адрес и порт сервера по умолчанию. Эти параметры будут использованы клиентом при создании соединения с сервером, если адрес сервера и порт не будут указаны моделью блока. Номер порта по умолчанию также используется при запуске РДС в режиме выделенного сервера с параметром командной строки "/server default". Для обмена данными между клиентом и сервером используются протоколы TCP и UDP: служебные данные всегда передаются по протоколу TCP, а данные, которыми обмениваются блоки, могут передаваться как по TCP, так и по UDP, в зависимости от настроек РДС (использование UDP должно быть разрешено) и флагов при вызове сервисных функций.

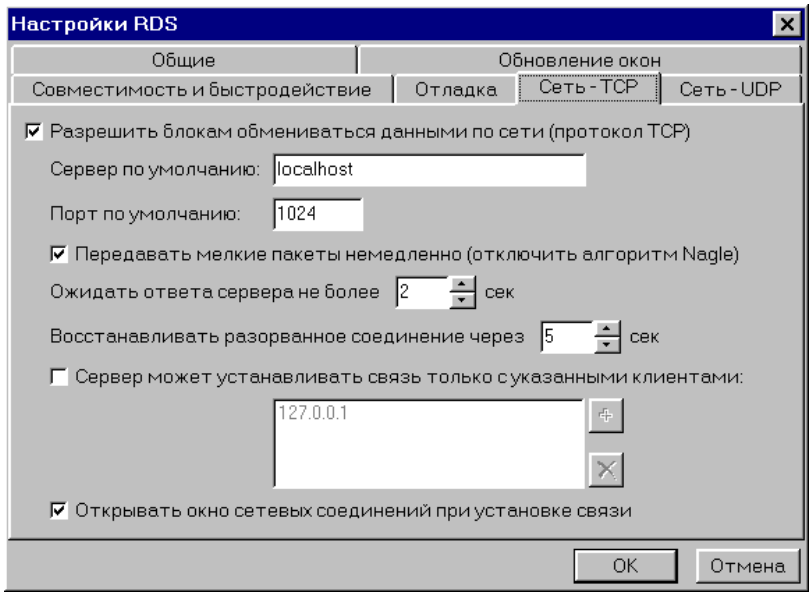


Рис. 109. Сетевые настройки РДС.

Обмен данными между блоками происходит через так называемые *каналы передачи данных* сервера, каждый из которых имеет текстовое имя, уникальное на данном сервере. Когда блок создает сетевое соединение с сервером при помощи сервисной функции РДС, он указывает IP-адрес и порт сервера (или разрешает использовать значения по умолчанию из настроек РДС), имя канала, с которым нужно установить соединение, а также хочет ли он получать данные из этого канала, или будет только передавать их. После этого блок может передавать в этот канал двоичные данные произвольного размера. Если при передаче данных не указана машина и блок, которые должны их получить, то сервер, приняв эти данные, отправляет их на все машины, блоки которых сообщили о желании получать данные из этого канала. Копии РДС, работающие на этих машинах, принимают данные от сервера и вызывают модели блоков-получателей, передавая им полученные данные. Если же при передаче данных указан конкретный блок-получатель на конкретной машине, сервер передаст данные только на эту машину, и на ней будет вызвана модель только одного блока.

Для установки соединения с каким-либо каналом сервера модель блока должна вызвать сервисную функцию `rdsNetConnect`:

```
int RDSCALL rdsNetConnect(
      LPSTR Host,      // IP-адрес или имя сервера, или NULL для
                       // сервера по умолчанию
      int Port,        // Порт сервера или -1 для порта по умолчанию
      LPSTR Channel,   // Имя канала передачи данных сервера
      BOOL Receive);   // Будет ли блок получать данные
```

В параметре Host передается строка с IP-адресом сервера (четыре числа, разделенные точками, например “192.168.1.1”) или его именем. Если вместо строки будет передано значение NULL, адрес сервера будет взят из сетевых настроек РДС (см. рис. 109). В параметре Port передается номер порта, используемого сервером РДС для обмена данными, при передаче значения –1 номер порта будет взят из сетевых настроек. Лучше всего использовать именно значения по умолчанию, то есть NULL и –1 – в этом случае при переносе сервера на другую машину достаточно будет изменить сетевые настройки РДС на каждой клиентской машине. Если же адрес сервера и номер порта будут храниться в настройках блока, при изменении адреса или порта сервера потребуется изменить настройки всех блоков на всех клиентских машинах. Тем не менее, если схеме необходимо устанавливать соединения с двумя разными серверами, адрес и порт одного из них придется хранить в настройках блоков.

В параметре Channel передается имя канала передачи данных, с которым устанавливается соединение. Имя канала может быть произвольной строкой символов. В параметрах сервера каналы никак не настраиваются, они создаются автоматически при первом запросе на подключение к каналу с указанным именем. Типа данных у канала тоже нет, блоки передают в него двоичные данные произвольного размера, и сервер пересылает их получателям без какой-либо обработки. Чаще всего имя канала хранят в настройках блока, чтобы пользователь мог ввести его самостоятельно, указывая, какие группы блоков связываются друг с другом через этот канал. Для обеспечения уникальности можно добавлять к имени, введенному пользователем или жестко указанному в программе, какой-нибудь префикс – например, имя библиотеки, в которой находится модель блока. Логический параметр Receive определяет, должен ли блок, вызвавший функцию, получать данные из канала (TRUE) или он будет только передавать их (FALSE). Если в модели блока не предусмотрен прием данных по сети, а в параметре Receive передано значение TRUE, ничего страшного не случится – при поступлении данных модель блока будет вызвана, но, поскольку в ней нет соответствующей реакции, принятые данные будут проигнорированы. Тем не менее, в блоках, которые только передают данные, лучше указывать в параметре Receive значение FALSE: это не только предотвратит потери процессорного времени на лишние вызовы модели блока, но и снизит нагрузку на сеть. Если в схеме на клиентской машине нет ни одного блока, получающего данные из канала, сервер вообще не будет передавать данные этого канала на эту машину.

Функция rdsNetConnect возвращает целое число, являющееся уникальным идентификатором созданного соединения. Это число в дальнейшем используется в сервисных функциях передачи данных и в реакциях блока на получение данных из сети. Если сетевое соединение установить невозможно (например, если обмен данными по сети запрещен в настройках РДС), функция вернет –1.

Если нужно не подключиться к другому серверу, а включить функции сервера в РДС на данной машине, необходимо вызвать функцию rdsNetServer. Она одновременно запускает сервер РДС, если он еще не запущен, и устанавливает соединение с указанным каналом в нем:

```
int RDSCALL rdsNetServer(
      int Port,       // Порт сервера или -1 для порта по умолчанию
      LPSTR Channel,  // Имя канала передачи данных сервера
      BOOL Receive);  // Будет ли блок получать данные
```

Параметры этой функции совпадают с параметрами rdsNetConnect, за исключением отсутствующего параметра Host: поскольку сервер запускается на той же машине, на которой загружена схема с вызвавшим эту функцию блоком, адрес сервера, очевидно, указывать не нужно. Для запуска сервера достаточно одного блока, вызвавшего эту функцию, все остальные блоки схемы могут использовать как rdsNetConnect, так и

rdsNetServer. Функция rdsNetServer тоже возвращает уникальный номер соединения, которым можно пользоваться при приеме и передаче данных.

После успешного соединении с сервером (не важно, на этой же машине, или на другой) модель блока вызывается в режиме RDS_BFM_NETCONNECT, при этом в параметре ExtParam передается указатель на структуру RDS_NETCONNDATA:

```
typedef struct
{ int ConnId;    // Уникальный идентификатор соединения
  LPSTR Host;    // Сервер (только для клиента)
  int Port;      // Порт сервера
  LPSTR Channel;// Имя канала
  BOOL ByServer;// Соединение разорвано сервером (только при
                // разрыве соединения)
} RDS_NETCONNDATA;
typedef RDS_NETCONNDATA *RDS_PNETCONNDATA;
```

Поля этой структуры повторяют параметры функций rdsNetConnect и rdsNetServer (кроме поля ByServer, которое в режиме вызова RDS_BFM_NETCONNECT не используется). Этот вызов информирует блок о том, что соединение с сервером успешно установлено, и теперь можно передавать данные в указанный канал, используя уникальный номер соединения ConnId.

Для передачи данных всем блокам канала используется сервисная функция rdsNetBroadcastData, возвращающая TRUE при успешной передаче:

```
BOOL RDSCALL rdsNetBroadcastData(
      int ConnId,      // Идентификатор соединения
      DWORD Flags,     // Флаги RDS_NETSEND_*
      int id,          // Передаваемое целое число
      LPSTR string,    // Передаваемая строка (или NULL)
      LPVOID buf,      // Указатель на передаваемый блока данных
                       // (или NULL)
      DWORD bufsize); // Размер передаваемого блока данных
```

В параметре ConnId передается уникальный номер соединения с каналом сервера, возвращенный функцией rdsNetConnect или rdsNetServer при создании этого соединения. Функция позволяет одновременно передать в канал целое число id, строку текста string и блок двоичных данных buf произвольного размера bufsize. Целое число передается всегда, а строку и блок данных можно не передавать – в этом случае в соответствующем параметре следует указать NULL. Разбиение передаваемых данных на три части сделано для удобства программиста: в принципе, строки и числа тоже можно передавать в виде двоичных данных. Однако, во многих случаях приходится передавать данные сложного формата, и тут отдельная передача целых чисел и строк облегчает жизнь программисту, ликвидируя необходимость разбирать принятые данные. Например, число id можно использовать в качестве типа передаваемых двоичных данных или номера фрагмента, если данные передаются последовательными блоками; строка string может содержать имя файла, содержимое которого передается в двоичном блоке, и т.д.

Способом передачи данных серверу управляют битовые флаги в параметре Flags. Можно использовать сочетание следующих флагов:

- RDS_NETSEND_UDP – использовать для передачи данных протокол UDP, если его использование разрешено в настройках РДС. По умолчанию все данные передаются по протоколу TCP. С использованием протокола UDP данные, как правило, передаются быстрее, но не всегда настройки и политика сети позволяют использовать этот протокол. Если при использовании этого флага данные по каким-либо причинам не могут быть переданы по протоколу UDP, они будут переданы по TCP – никаких дополнительных действий для этого не потребуется.

- RDS_NETSEND_UPDATE – можно заменять еще не отправленные данные новыми. Для передачи данных на сервер в РДС используется очередь: при вызове rdsNetBroadcastData данные ставятся в ее конец, а данные, находящиеся в ее начале, отправляются на сервер с той скоростью, которую может обеспечить сеть. Если блоки будут передавать данные быстрее, чем они будут уходить в сеть, очередь будет расти. В результате этого в некоторых случаях в очереди могут оказаться устаревшие данные, которые можно было бы и не передавать. Допустим, например, что в одной из схем данные в канал передачи поступают через какой-либо блок, берущий эти данные со своего вещественного входа. Если значение входа изменится, блок поставит это новое значение в очередь на передачу, вызвав rdsNetBroadcastData. Если затем, через очень небольшой промежуток времени, значение снова изменится, блок снова поставит в очередь новое значение. Если сеть перегружена и очередь достаточно длинная, в очереди окажется два числа для передачи в один и тот же канал: устаревшее (поставленное в очередь первым) и актуальное (поставленное вторым). Если при передаче использовать флаг RDS_NETSEND_UPDATE, устаревшее значение будет автоматически выбрасываться из очереди при поступлении новых данных для этого канала, таким образом, нагрузка на сеть несколько снизится (устаревшими будут считаться ранее поставленные в очередь данные ***с тем же значением Id***). Без этого флага передаются все поставленные в очередь данные – чаще всего, повторная передача данных в канал не означает, что предыдущая устарела, обычно это просто новая порция данных, которые должен обработать принимающий блок.
- RDS_NETSEND_NOWAIT – не ждать ответа от сервера, передавать следующие данные немедленно. По умолчанию после передачи данных на сервер клиент ждет от него сигнала о получении этих данных, и, пока он не придет, не передает следующую порцию данных из очереди. Это позволяет несколько снизить нагрузку на сеть, но уменьшает скорость передачи почти в два раза – к задержке при передаче данных на сервер добавляется задержка ответа от сервера. Если при передаче использовать этот флаг, для передаваемой порции данных сервер не будет отправлять подтверждение приема, а клиент не будет ждать этого подтверждения.
- RDS_NETSEND_SERVREPLY – получив данные, сервер должен сообщить об этом передавшему их блоку (нельзя использовать вместе с флагом RDS_NETSEND_NOWAIT). Если этот флаг включен, то, как только клиент получит подтверждение приема данных сервером, модель блока, передавшего данные, будет вызвана в режиме RDS_BFM_NETDATAACCEPTED. Это позволит, при необходимости, организовать в модели блока собственную очередь передачи данных, отправляя новую порцию на сервер только после подтверждения приема предыдущей.

При вызове модели в режиме RDS_BFM_NETDATAACCEPTED в параметре ExtParam передается указатель на структуру RDS_NETACCEPTDATA:

```
typedef struct
{ int ConnId;    // Идентификатор соединения
  LPSTR Host;    // Адрес сервера (только для клиента)
  int Port;      // Порт сервера
  LPSTR Channel;// Канал передачи

  int Id;        // Целое число (id) из принятого блока
} RDS_NETACCEPTDATA;
typedef RDS_NETACCEPTDATA *RDS_PNETACCEPTDATA;
```

В полях ConnId, Host, Port и Channel, как и в структуре RDS_NETCONNDATA, содержатся идентификатор соединения, адрес сервера, порт и имя канала соответственно. В поле Id записано целое число, которое было передано в параметре id при вызове rdsNetBroadcastData. Таким образом, реагируя на вызов в режиме

RDS_BFM_NETDATAACCEPTED, модель блока сможет понять, прием какой именно порции данных подтвердил сервер.

Получив данные для какого-либо канала передачи, сервер рассылает их всем подключенным к нему клиентам, блоки которых создали соединение с этим каналом и изъявили желание получать данные (параметр Receive при вызове rdsNetConnect или rdsNetServer был равен TRUE). Приняв данные, клиент вызывает модели всех этих блоков в режиме RDS_BFM_NETDATARECEIVED, передавая в параметре ExtParam указатель на структуру RDS_NETRECEIVEDDATA:

```
typedef struct
{ // Параметры соединения
  int ConnId;    // Идентификатор соединения
  LPSTR Host;    // Адрес сервера (только для клиента)
  int Port;      // Порт сервера
  LPSTR Channel;// Имя канала передачи данных
  // Принятые данные
  int Id;              // Принятое целое число
  LPSTR Str;           // Принятая строка
  LPVOID Buffer;       // Указатель на буфер с принятыми двоичными
                       // данными или NULL
  DWORD BufferSize;    // Размер принятого буфера
  // Идентификаторы отправителя
  RDS_NETSTATION SenderStation; // Идентификатор пердавшей машины
  RDS_NETBLOCK SenderBlock;     // Идентификатор передавшего блока
} RDS_NETRECEIVEDDATA;
typedef RDS_NETRECEIVEDDATA *RDS_PNETRECEIVEDDATA;
```

Первые четыре поля структуры, как всегда, описывают параметры соединения, по которому пришли данные, для обработки которых вызвана модель. Если блок установил сразу несколько соединений, по этим параметрам можно понять, какое из них приняло данные. В поле Id содержится принятое целое число, переданное другим блоком в параметре id при вызове rdsNetBroadcastData. В поле Str – указатель на принятую строку во внутренней памяти РДС (ей соответствует параметр string в вызове rdsNetBroadcastData). Этот указатель никогда не будет равен NULL: если строка не передавалась, поле String будет указывать на пустую строку. Наконец, поле Buffer указывает на принятый блок двоичных данных (при этом в BufferSize записан размер этого блока), или содержит NULL, если двоичные данные не передавались. Таким образом, есть однозначное соответствие между параметрами функции rdsNetBroadcastData и полями структуры RDS_NETRECEIVEDDATA: то, что один блок передал в параметрах сервисной функции, другой блок получает в полях структуры при вызове модели в режиме RDS_BFM_NETDATARECEIVED.

Два последних поля структуры, SenderStation и SenderBlock, содержат уникальные идентификаторы машины и блока в схеме на этой машине, отправившего данные в канал. Эти поля могут использоваться для того, чтобы получивший данные блок мог передать ответ только блоку, пославшему эти данные, а не всем блокам, подключенным к каналу передачи. Для передачи данных единственному блоку используется сервисная функция rdsNetSendData:

```
BOOL RDSCALL rdsNetSendData(
     int ConnId,      // Идентификатор соединения
     DWORD Flags,     // Флаги RDS_NETSEND_*
     int id,          // Передаваемое целое число
     LPSTR string,    // Передаваемая строка (или NULL)
     LPVOID buf,      // Указатель на передаваемый блока данных
                      // (или NULL)
     DWORD bufsize,   // Размер передаваемого блока данных
```

```
    RDS_NETSTATION station,     // Машина-получатель
    RDS_NETBLOCK block);        // Блок-получатель
```

Первые шесть параметров функции в точности совпадают с параметрами `rdsNetBroadcastData` – они определяют передаваемые данные и соединение, через которое их нужно передать. Последние два параметра указывают машину-клиент, на которую нужно передать эти данные, и блок в схеме на этой машине, который должен их получить. Эти два идентификатора блок-отправитель данных может узнать, только приняв данные из структуры `RDS_NETRECEIVEDDATA`, поэтому функцию `rdsNetSendData` можно использовать только для ответа на переданные данные. Нет никакой возможности узнать идентификатор машины-клиента и блока, не приняв от них какие-либо данные. Обычно это и не требуется: если нужно передавать данные конкретному блоку конкретной схемы, лучше всего выделить ему отдельный канал передачи данных, в котором он будет единственным получателем.

Таким образом, вызывая функции `rdsNetBroadcastData` и `rdsNetSendData` и реагируя на вызовы в режиме `RDS_BFM_NETDATARECEIVED`, блоки на разных машинах могут обмениваться произвольными данными до тех пор, пока сетевое соединение не будет разорвано. Для завершения сетевого соединения модель блока должна вызвать сервисную функцию `rdsNetCloseConnection`, принимающую единственный параметр: уникальный идентификатор соединения, которое нужно закрыть. После этого, как только соединение будет разорвано, модель блока будет вызвана в режиме `RDS_BFM_NETDISCONNECT`, и в параметре `ExtParam` ей будет передан указатель на уже знакомую нам структуру `RDS_NETCONNDATA`, содержащую параметры конкретного разорванного соединения. В этом режиме модель вызывается не только при самостоятельном завершении соединения, но и при его разрыве по инициативе сервера, из-за отключения сети или по другим причинам. При разрыве соединения из-за отключения сервера в поле `ByServer` будет содержаться значение `TRUE`. В этом случае, а также при разрыве соединения по любым причинам, отличным от вызова `rdsNetCloseConnection`, РДС будет пытаться самостоятельно восстановить связь до тех пор, пока эти попытки не увенчаются успехом, или пока для этого соединения не будет вызвана функция `rdsNetCloseConnection`.

В случае возникновения ошибок при приеме или передаче данных модель блока вызывается в режиме `RDS_BFM_NETERROR`, при этом в параметре `ExtParam` передается указатель на структуру ошибки `RDS_NETERRORDATA`, описанную в приложении А. Поскольку РДС самостоятельно восстанавливает оборванные соединения, заново посылает пропавшие данные и решает прочие возникающие проблемы, модели редко реагируют на этот вызов. Чаще всего реакция на `RDS_BFM_NETERROR` используется при отладке моделей блоков, когда обмен данными по сети не работает, и нужно выяснить, почему именно.

§2.15.2. Пример использования функций передачи и приема данных

Приводятся примеры моделей блоков, позволяющих организовать передачу вещественного значения между схемами, работающими на разных машинах. В дополнение к ним описывается модель блока, добавление которого в схему включает серверные функции в РДС.

Для иллюстрации работы с сетевыми сервисными функциями создадим два блока: один будет передавать в выбранный пользователем канал данных значение своего вещественного входа, другой будет принимать это значение и выдавать его на выход. С помощью этих блоков можно будет организовать связь между схемами, работающими на разных машинах. Но сначала, чтобы не запускать на третьей машине сервер РДС, сделаем вспомогательный блок, который будет запускать серверные функции РДС в схеме, в которой он находится. Таким образом, для проверки работы сетевой связи между двумя схемами не нужно будет запускать отдельный сервер: РДС на одной из машин будет не только работать

со схемой, но и выполнять функции сервера. Разумеется, в качестве сервера лучше выбрать машину побыстрее.

Модель блока для включения сервера будет очень простой: при инициализации будет вызываться функция `rdsNetServer` с указанием какого-нибудь имени канала, а при очистке – `rdsNetCloseConnection`. Мы не будем ни передавать данные в этот канал, ни получать их из него – он нужен только для запуска сервера, поскольку функцию `rdsNetServer` нельзя вызвать с пустым именем канала. Назовем этот фиктивный канал “ProgrammersGuide.Server”. Если другие блоки будут использовать его для передачи своих данных, ничего страшного не случится, поскольку наш блок не вмешивается в обмен данными по этому каналу.

Модель блока будет выглядеть следующим образом:

```
// Включение сервера
extern "C" __declspec(dllexport)
  int RDSCALL Server(int CallMode,
                     RDS_PBLOCKDATA BlockData,
                     LPVOID ExtParam)
{ int *pConnId;

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Отводим место под личную область данных размером в int
        BlockData->BlockData=pConnId=new int;
        // Запускаем сервер и соединяемся с ним
        *pConnId=rdsNetServer(-1,     // Порт по умолчанию
                              "ProgrammersGuide.Server", // Канал
                              FALSE); // Не принимаем данные
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        // Личная область данных – целое число
        pConnId=(int*)(BlockData->BlockData);
        // Разрываем связь с сервером (он завершится сам)
        rdsNetCloseConnection(*pConnId);
        // Удаляем личную область
        delete pConnId;
        break;
    }
  return RDS_BFR_DONE;
}
//=======================================
```

При инициализации блока мы отводим место в памяти под личную область данных размером в одно целое число – в ней мы будем хранить идентификатор соединения. Затем мы вызываем `rdsNetServer` для соединения с каналом “ProgrammersGuide.Server” на локальном сервере, то есть сервере в исполняемом файле РДС, который работает с данной схемой. В качестве номера порта мы передаем –1, чтобы был использован порт по умолчанию, указанный в настройках РДС. Конечно, лучше было бы дать пользователю выбор: использовать порт по умолчанию или задать свое значение, но мы не будем этого делать, чтобы не усложнять пример. Принимать данные из канала нам не нужно, поэтому в параметре `Receive` функции `rdsNetServer` передается `FALSE`.

При первом вызове `rdsNetServer` с конкретным номером порта сервер запускается и начинает ждать внешних соединений с этим портом, а функция возвращает уникальный внутренний идентификатор соединения, который можно использовать для передачи данных

на этот сервер напрямую, минуя сетевые протоколы. Нам этот идентификатор нужен для того, чтобы при удалении блока из схемы закрыть соединение с сервером, поэтому мы сохраняем его в личной области данных. Таким образом, после добавления этого блока в схему, машина, на которой эта схема работает, становится сервером РДС, принимающим данные через порт, указанный в сетевых настройках (см. рис. 109).

При удалении блока из схемы его модель будет вызвана в режиме RDS_BFM_CLEANUP. При этом мы закрываем соединение, идентификатор которого хранится в личной области данных блока. Если это соединение было единственным, сервер будет автоматически отключен.

Теперь, когда у нас есть блок, превращающий машину в сервер РДС, можно заняться блоками приема и передачи данных. У этих блоков будет много общего: в настройках обоих необходимо будет вводить имя канала, оба будут подключаться к серверу, и т.д. Чтобы не писать один и тот же код два раза, мы сделаем для этих двух блоков общий класс личной области данных с полным набором функций и для приема, и для передачи данных, а из функции модели каждого блока будем вызывать только те из них, которые нужны данному блоку. Начнем с описания класса:

```
// Личная область данных блоков приема и передачи по сети
class TNetSendRcvData
{ public:
    int Mode;   // Режим данного блока: прием или передача
      #define NETSRMODE_SENDER        0     // Передатчик
      #define NETSRMODE_RECEIVER      1     // Приемник
    char *ChannelName;     // Имя канала

    int ConnId; // Идентификатор соединения

    // Переменные состояния блока-передатчика
    BOOL Connected;   // Соединение установлено
    BOOL DataWaiting; // Передача данных отложена

    // Функции класса
    void Connect(void);    // Установить соединение
    void Disconnect(void); // Разорвать соединение

    void SendValue(double value);     // Передать число в канал
    BOOL ReceiveValue(RDS_NETRECEIVEDDATA *rcv,  // Реакция на
                      double *pOut);       // пришедшие данные

    int Setup(char *title);      // Функция настройки блока
    void SaveText(void);         // Сохранить параметры
    void LoadText(char *text);   // Загрузить параметры

    // Конструктор класса
    TNetSendRcvData(int mode)
      { ConnId=-1;    // Нет соединения
        Connected=DataWaiting=FALSE;
        ChannelName=NULL;
        Mode=mode;};  // Режим передается в параметре конструктора
    // Деструктор класса
    ~TNetSendRcvData()
      { Disconnect();            // Разорвать соединение
        rdsFree(ChannelName);    // Освободить строку имени канала
      };
};
//=========================================
```

Самое первое поля класса, Mode, указывает на то, принадлежит ли эта область данных блоку-передатчику (константа NETSRMODE_SENDER) или приемнику (константа NETSRMODE_RECEIVER). Значение этому полю присваивается в конструкторе и далее не меняется на протяжении всего существования блока: в блоке-передатчике конструктор будет вызываться с параметром NETSRMODE_SENDER, в приемнике – с параметром NETSRMODE_RECEIVER.

Мы не будем делать имя сервера и его порт параметрами блока – для простоты всегда будем использовать значения по умолчанию из сетевых настроек РДС. Поэтому соответствующие поля в классе не предусмотрены. А вот имя канала нужно обязательно сделать настраиваемым, для этого используется поле ChannelName. Оно содержит указатель на динамически отведенную строку с именем канала, к которому подключен блок, это имя будет вводиться пользователем в функции настройки блока. В конструкторе ему присваивается значение NULL: место под строку будет отведено позднее, при настройке блока или загрузке его параметров. В деструкторе класса память, занятая строкой, освобождается при помощи функции rdsFree (отводиться она тоже будет сервисными функциями РДС).

В следующем поле, ConnId, будет храниться идентификатор соединения с сервером, когда оно будет установлено. В конструкторе ему присваивается значение –1 (соединение в конструкторе установить нельзя – еще не известно имя канала), в деструкторе соединение разрывается функцией класса Disconnect.

Логические поля Connected и DataWaiting используются только в блоке-передатчике. Connected используется как флаг наличия связи с сервером – при установке связи ему будет присваиваться значение TRUE, при разрыве – FALSE. DataWaiting служит для запоминания попытки передачи данных, пока связь еще не установлена. Если значение входа блока-передатчика изменится, а связи с сервером пока нет, полю DataWaiting будет присвоено значение TRUE. Как только связь будет установлена, модель передаст данные в канал и сбросит DataWaiting.

Функции класса Connect и Disconnect предназначены для установки связи с сервером и ее разрыва соответственно. Они будут использоваться и блоками-приемниками, и блоками-передатчиками. Функция SendValue передает в канал вещественное число, она будет использоваться только в блоках-передатчиках. Функция ReceiveValue будет вызываться в реакции модели блока-приемника на поступившие данные, анализировать эти данные и, если размер переданного буфера равен размеру вещественного числа, копировать их на выход блока (указатель на переменную выхода передается в параметре функции). Наконец, функции Setup, LoadText и SaveText предназначены для настройки параметров блока (у нас только один параметр – имя канала), их загрузки и записи. В функцию Setup будет передаваться заголовок окна настройки – у передатчика и приемника он будет различаться.

Напишем функцию установки соединения с сервером. Она будет использовать имя и порт сервера по умолчанию, а имя канала будет брать из поля ChannelName, добавляя к нему префикс “ProgrammersGuide.” для того, чтобы каналы, используемые нашими блоками, не пересеклись по именам с каналами блоков других разработчиков (точно так же мы поступали, когда давали имена функциям блоков, см. стр. 314).

```
// Установка соединения
void TNetSendRcvData::Connect(void)
{ char *PrefixedName; // Полное имя канала

  // Если имя канала пустое, соединение невозможно
  if(ChannelName==NULL || (*ChannelName)==0)
    return;
```

```
    // Добавляем префикс к имени канала
    PrefixedName=rdsDynStrCat("ProgrammersGuide.",ChannelName,FALSE);
    // Устанавливаем соединение с сервером
    ConnId=rdsNetConnect(NULL,     // Сервер по умолчанию
                         -1,       // Порт по умолчанию
                         PrefixedName,// Имя канала с префиксом
                         Mode==NETSRMODE_RECEIVER); // Прием данных
    // Освобождаем динамически отведенную строку
    rdsFree(PrefixedName);
  }
  //=======================================
```

Прежде всего, мы проверяем, введено ли имя канала, то есть не пуста ли строка `ChannelName`. Если имя канала не введено (поле `ChannelName` содержит `NULL` или указывает на пустую строку), соединение с сервером установить невозможно – функция завершается. В противном случае при помощи сервисной функции `rdsDynStrCat` к имени канала, заданному пользователем, добавляется префикс “ProgrammersGuide.”, и результат записывается в переменную `PrefixedName`. `rdsDynStrCat` возвращает указатель на динамически отведенную строку, поэтому ее нужно будет освободить перед завершением функции `Connect`.

Теперь можно устанавливать соединение с сервером: вызывается функция `rdsNetConnect`, и возвращенный ей идентификатор соединения записывается в поле `ConnId`. Мы подключаемся к серверу по умолчанию, указанному в настройках РДС, поэтому вместо адреса сервера передается `NULL`, а вместо номера его порта – –1. Имя канала с префиксом сформировано в переменной `PrefixedName`, а последний параметр функции `rdsNetConnect`, определяющий, должен ли наш блок получать данные по сети, зависит от значения поля класса `Mode`. Блок будет принимать данные только в том случае, если в этом поле находится константа `NETSRMODE_RECEIVER`, то есть если эта личная область данных принадлежит блоку-приемнику. Таким образом, сервер не будет отправлять данные блокам-передатчикам, что снизит нагрузку на сеть.

Следует помнить, что при вызове `rdsNetConnect` соединение с сервером устанавливается не мгновенно. Эта функция только запрашивает соединение и создает внутренний объект РДС, идентификатор которого возвращается. После этого клиент отправит запрос серверу, дождется ответа от него, обменяется с ним служебной информацией и т.п. – на все это нужно время. После фактической установки соединения модель блока будет вызвана в режиме `RDS_BFM_NETCONNECT`, до этого момента передаваемые данные отправляться на сервер не будут. Лучше всего отложить вызов функций передачи данных до фактической установки соединения, чтобы не загружать очередь. По этой причине мы не взводим логический флаг наличия соединения `Connected` в функции `Connect`: он будет установлен только после вызова модели в режиме `RDS_BFM_NETCONNECT`.

Функция разрыва соединения будет короткой:

```
// Разорвать соединение
void TNetSendRcvData::Disconnect(void)
{ if(ConnId!=-1) // Соединение было создано
    rdsNetCloseConnection(ConnId); // Разорвать
  // Сбрасываем переменные состояния
  ConnId=-1;          // Соединения больше нет
  Connected=FALSE;    // Связи тоже больше нет
}
//=======================================
```

Если соединение было создано (`ConnId` не равно –1), мы разрываем его функцией `rdsNetCloseConnection`, после чего присваиваем `ConnId` значение –1, поскольку

соединения уже нет. Логический флаг Connected мы тоже сбрасываем – хотя физически соединение, может быть, еще не разорвано, данные передавать уже нельзя.

Теперь напишем функцию передачи вещественного числа – она будет вызываться при изменении входа блока-передатчика:

```
// Передать данные
void TNetSendRcvData::SendValue(double value)
{
  if(!Connected)      // Нет связи с сервером
    { // Взводим флаг наличия данных, ожидающих передачи
      DataWaiting=TRUE;
      return;
    }

  // Связь есть - передаем всем блокам канала
  rdsNetBroadcastData(ConnId,    // Соединение
                      RDS_NETSEND_UPDATE|RDS_NETSEND_UDP, // Флаги
                      0,NULL, // Не передаем целое число и строку
                      &value,         // Указатель на данные
                      sizeof(value)); // Размер данных
  // Сбрасываем флаг ожидания - мы только что передали данные
  DataWaiting=FALSE;
}
//=======================================
```

Сначала мы проверяем, взведен ли логический флаг Connected, то есть пришло ли подтверждение установки связи с сервером. Если он сброшен, связи еще нет, и передавать данные не следует. В этом случае взводится логический флаг DataWaiting и функция завершается – как только связь будет установлена, мы проверим этот флаг, и, если он взведен, передадим в канал значение входа блока.

Если флаг взведен, вызывается функция rdsNetBroadcastData, передающая через соединение ConnId данные всем блокам, подключившимся к этому же каналу сервера. При передаче используются флаги RDS_NETSEND_UPDATE (если в очереди на передачу уже стоят данные для этого канала, они будут выброшены) и RDS_NETSEND_UDP (использовать при передаче протокол UDP, если возможно). Вещественное число мы отправляем в канал как восемь байтов двоичных данных – в качестве указателя на буфер с данными мы передаем в функцию указатель на параметр value, а в качестве длины буфера – размер этого параметра sizeof(value) (value имеет тип double, поэтому sizeof(value) будет равно восьми). Функция rdsNetBroadcastData одновременно с блоком двоичных данных может передать целое число и строку, но это нам не нужно, поэтому вместо строки передается NULL, а вместо числа – 0. На самом деле, число 0 все равно будет передано, но блок-приемник никак не будет на него реагировать.

Сразу после вызова rdsNetBroadcastData мы сбрасываем флаг DataWaiting – теперь данные уже не ожидают передачи, они только что были переданы.

Функция ReceiveValue будет вызываться в реакции модели на поступление данных из сети (RDS_BFM_NETDATARECEIVED). В нее будет передаваться указатель на структуру RDS_NETRECEIVEDDATA, описывающую принятые данные (в функцию модели этот указатель передается в параметре ExtParam) и указатель на вещественный выход блока-приемника, в который нужно записать принятое значение. Функция будет возвращать TRUE, если размер принятых данных соответствует типу double, то есть равен восьми.

```
// Прием данных
BOOL TNetSendRcvData::ReceiveValue(
  RDS_NETRECEIVEDDATA *rcv, // Указатель на структуру с данными
  double *pOut)             // Указатель на выход блока
{
```

```
  if(rcv==NULL || pOut==NULL)    // Нет одного из указателей
    return FALSE;

  // Проверяем, есть ли среди принятых данных двоичные,
  // и равен ли размер этих данных размеру double
  if(rcv->Buffer==NULL || rcv->BufferSize!=sizeof(double))
    return FALSE; // Нет данных или не совпал размер

  // Копируем принятое числов pOut
  memcpy(pOut,rcv->Buffer,sizeof(double));
  return TRUE;  // Приняты правильные данные
}
//=======================================
```

В начале функции мы, на всякий случай, проверяем, не переданы ли в нее случайно нулевые указатели на структуру с принятыми данными и выход блока – без этих указателей функция не сможет работать. Затем мы проверяем, какие именно данные приняты. В канал можно одновременно передать целое число, строку и буфер с двоичными данными, но нас интересуют только двоичные данные, причем их размер должен совпадать с размером вещественного числа двойной точности (`double`). Если в принятых данных нет двоичных (указатель `rcv->Buffer` равен `NULL`) или размер данных неправильный (`rcv->BufferSize` не равно `sizeof(double)`), функция возвращает `FALSE` – блок не может обработать такие данные. В противном случае весь принятый двоичный буфер копируется в выход блока, указатель на который передан в параметре `pOut`, и функция возвращает `TRUE`.

Осталось написать три вспомогательные функции. Начнем с функций сохранения и загрузки параметров блока. В данном случае, параметр у нас один – строка имени канала. Сохранять параметры мы будем в текстовом формате, аналогичном формату INI-файлов Windows (см. стр. 168):

```
// Сохранение параметров блока
void TNetSendRcvData::SaveText(void)
{ RDS_HOBJECT ini;    // Вспомогательный объект
  // Создаем вспомогательный объект
  ini=rdsINICreateTextHolder(TRUE);
  // Создаем в объекте секцию "[General]"
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"General");
  // Записываем в эту секцию имя канала
  rdsINIWriteString(ini,"Channel",ChannelName);
  // Сохраняем текст, сформированный объектом, как параметры блока
  rdsCommandObject(ini,RDS_HINI_SAVEBLOCKTEXT);
  // Удаляем вспомогательный объект
  rdsDeleteObject(ini);
}
//=======================================
```

Функция загрузки параметров будет аналогична функции сохранения:

```
// Загрузка параметров блока
void TNetSendRcvData::LoadText(char *text)
{ RDS_HOBJECT ini;    // Вспомогательный объект
  char *str;
  // Создаем вспомогательный объект
  ini=rdsINICreateTextHolder(TRUE);
  // Записываем в объект полученный текст с параметрами блока
  rdsSetObjectStr(ini,RDS_HINI_SETTEXT,0,text);
  // Начинаем чтение секции "[General]", если она есть
  if(rdsINIOpenSection(ini,"General")) // Секция есть
```

```
    { // Освобождаем старое имя канала
      rdsFree(ChannelName);
      ChannelName=NULL;
      // Получаем у объекта указатель на строку с именем
      str=rdsINIReadString(ini,"Channel","",NULL);
      // Если такая строка есть в тексте, копируем ее в
      // ChannelName
      if(str)
        ChannelName=rdsDynStrCopy(str);
    }
  // Удаляем вспомогательный объект
  rdsDeleteObject(ini);

  // Поскольку имя канала могло измениться, соединяемся с
  // сервером заново
  Disconnect(); // Разрываем старое соединение
  Connect();    // Создаем новое
}
//=======================================
```

В этой функции следует обратить внимание на последние два вызова: после загрузки параметров блока мы разрываем связь с сервером и устанавливаем ее заново. Это связано с тем, что в результате загрузки параметров имя канала в `ChannelName` могло измениться, и мы должны присоединиться к каналу с новым именем, предварительно разорвав связь со старым.

Теперь напишем функцию настройки, которая позволит пользователю задавать имя канала для блока-приемника и блока-передатчика. Будем использовать для ввода строки сервисную функцию `rdsInputString` (см. стр. 119):

```
// Настройка параметров
int TNetSendRcvData::Setup(char *title)
{ char *NewName;

  // Запрос строки у пользователя
  NewName=rdsInputString(
     title,          // Заголовок окна
     "Имя канала:",  // Текст перед полем
     ChannelName,    // Исходное значение
     200);           // Ширина поля

  if(NewName==NULL)   // Нажата кнопка "Отмена"
    return 0;

  if(ChannelName!=NULL && strcmp(NewName,ChannelName)==0)
    { // Имя канала не изменилось
      rdsFree(NewName); // Освобождаем вовращенную строку
      return 0;
    }

  // Освобождаем старое имя канала
  rdsFree(ChannelName);
  // Запоминаем новое
  ChannelName=NewName;
  // Имя канала изменилось - устанавливаем связь заново
  Disconnect();
  Connect();
```

```
  return 1; // Параметры блока изменены
}
//=======================================
```

Заголовок окна, которое откроет функция `rdsInputString`, передается в параметре функции `Setup`, он будет зависеть от того, какому типу блока принадлежит объект этого класса. Функция вернет нам управление только после того, как пользователь закроет окно кнопкой “ОК” или “Отмена”. При нажатии кнопки “Отмена” функция возвращает `NULL`, при этом мы завершаем функцию настройки, возвращая ноль — параметры блока не изменились. Если же пользователь нажмет “ОК”, функция вернет указатель на динамически отведенную строку с текстом, введенным в окне. В этом случае имеет смысл проверить, изменил ли пользователь имя канала, или оно осталось прежним. Если строки `ChnName` и `NewName` совпадают, то есть пользователь не изменил имя канала, возвращенная функцией `rdsInputString` строка `NewName` освобождается, и мы завершаем функцию настройки. В противном случае мы освобождаем строку со старым именем канала и копируем в `ChnName` новое из переменной `NewName`. Затем, как и после загрузки параметров, мы разрываем соединение с сервером и устанавливаем его заново, но уже с новым именем канала.

Теперь, когда все вспомогательные функции готовы, можно приступать к написанию моделей блока приемника и блока-передатчика. Начнем с модели передатчика – она будет чуть более сложной, поскольку, в отличие от модели приемника, ей нужно следить за наличием связи с сервером и передавать данные только тогда, когда эта связь есть.

В блоке-передатчике нам нужен единственный вещественный вход, значение которого будет передаваться в канал, поэтому его структура переменных будет следующей:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/ выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	0
1	Ready	Сигнал	1	Выход		0
2	x	double	8	Вход	✓	0

Для входа блока `x` установлен флаг “пуск”, чтобы при срабатывании связи, подключенной к этому входу, модель блока автоматически запускалась в следующем такте расчета – передавать данные на сервер блок будет именно в такте расчета. Для правильной работы модели необходимо будет также установить в параметрах блока флаг “запуск по сигналу”, иначе модель будет передавать данные не при изменении входа, а в каждом такте расчета, что приведет к неоправданно большой нагрузке на сеть.

Модель блока будет следующей:

```
// Блок-передатчик
extern "C" __declspec(dllexport)
  int RDSCALL NetSend(int CallMode,
                      RDS_PBLOCKDATA BlockData,
                      LPVOID ExtParam)
{ // Указатель на личную область данных, приведенный к нужному типу
  TNetSendRcvData *data=(TNetSendRcvData*)(BlockData->BlockData);
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define x (*((double *)(pStart+2)))

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
```

```
    // Создаем объект класса TNetSendRcvData с
    // Mode==NETSRMODE_SENDER (передатчик)
    BlockData->BlockData=new TNetSendRcvData(NETSRMODE_SENDER);
    break;

  // Очистка
  case RDS_BFM_CLEANUP:
    delete data;
    break;

  // Проверка типов переменных
  case RDS_BFM_VARCHECK:
    return strcmp((char*)ExtParam,"{SSD}")?
                 RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

  // Связь с сервером установлена
  case RDS_BFM_NETCONNECT:
    // Взводим флаг наличия связи
    data->Connected=TRUE;
    // Если были данные, ожидающие отправки,
    // посылаем значение входа блока
    if(data->DataWaiting)
      data->SendValue(x);
    break;

  // Связь с сервером разорвана
  case RDS_BFM_NETDISCONNECT:
    // Сбрасываем флаг наличия связи
    data->Connected=FALSE;
    break;

  // Запуск расчета
  case RDS_BFM_STARTCALC:
    // Если это - первый запуск после сброса,
    // передаем значение входа
    if(((RDS_PSTARTSTOPDATA)ExtParam)->FirstStart)
      data->SendValue(x);
    break;

  // Такт расчета
  case RDS_BFM_MODEL:
    data->SendValue(x); // Передаем значение входа
    break;

  // Вызов настройки
  case RDS_BFM_SETUP:
    return data->Setup("Передача double");

// Сохранение параметров в текстовом виде
case RDS_BFM_SAVETXT:
    data->SaveText();
    break;

// Загрузка параметров в текстовом виде
case RDS_BFM_LOADTXT:
    data->LoadText((char*)ExtParam);
    break;
```

```
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef x
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

При инициализации модели мы создаем личную область данных блока – объект класса `TNetSendRcvData`. В параметре конструктора класса мы передаем константу `NETSRMODE_SENDER` – она будет записана в поле `Mode`, чтобы все функции класса знали, что объект принадлежит блоку-передатчику.

Как только связь с сервером будет установлена (а устанавливается она после загрузки параметров или ввода имени канала пользователем в функциях, которые мы уже написали), модель будет вызвана в режиме `RDS_BFM_NETCONNECT`. При этом будет взведен флаг наличия связи `data->Connected`, и, если есть данные, ожидающие передачи (взведен `data->DataWaiting`), значение входа блока `x` будет передано на сервер функцией `SendValue`. Флаг `DataWaiting` взводится внутри функции `SendValue` при попытке передать данные в отсутствие связи с сервером, и сбрасывается там же после успешной передачи. Если на момент установления связи `DataWaiting` будет истинным, значит, до этого были неудачные попытки передачи, и вход блока нужно передать на сервер.

При разрыве связи модель будет вызвана в режиме `RDS_BFM_NETDISCONNECT`, при этом будет сброшен флаг `data->Connected`. Начиная с этого момента все вызовы функции `SendValue` будут приводить не к передаче данных, а к взведению флага `DataWaiting`. Как только РДС сможет восстановить связь, данные немедленно будут переданы.

При самом первом запуске расчета (сразу после загрузки схемы или после сброса) блок отправляет значение своего входа на сервер – таким образом мы передаем всем блокам-приемникам начальное значение входа блока передатчика. Текущее значение входа `x` при его изменениях передается в такте расчета (`RDS_BFM_MODEL`). В трех оставшихся реакциях модели вызываются функции класса для настройки, сохранения и загрузки параметров, которые мы уже написали.

Теперь напишем модель блока-приемника. У него будет очень похожая структура переменных, только место входа `x` займет выход `y`:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/ выход*	*Начальное значение*
0	Start	Сигнал	1	Вход	0
1	Ready	Сигнал	1	Выход	0
2	y	double	8	Выход	0

Для этого блока тоже желательно установить в параметрах блока флаг “запуск по сигналу”, чтобы он не тратил зря процессорное время: в его модели не будет реакции на выполнение такта расчета.

Модель блока-приемника будет выглядеть так:

```
// Блок-приемник
extern "C" __declspec(dllexport)
  int RDSCALL NetReceive(int CallMode,
                         RDS_PBLOCKDATA BlockData,
                         LPVOID ExtParam)
```

```
{ // Указатель на личную область данных, приведенный к нужному типу
  TNetSendRcvData *data=(TNetSendRcvData*)(BlockData->BlockData);
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define y (*((double *)(pStart+2)))

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Создаем объект класса TNetSendRcvData с
        // Mode==NETSRMODE_RECEIVER (приемник)
        BlockData->BlockData=
                          new TNetSendRcvData(NETSRMODE_RECEIVER);
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;

      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSD}")?
                     RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // По сети получены данные
      case RDS_BFM_NETDATARECEIVED:
        if(data->ReceiveValue((RDS_NETRECEIVEDDATA*)ExtParam,&y))
          Ready=1;    // Если данные верны, взводим флаг готовности
                      // для передачи выхода по связям
        break;

      // Вызов настройки
      case RDS_BFM_SETUP:
        return data->Setup("Прием double");

     // Сохранение параметров в текстовом виде
     case RDS_BFM_SAVETXT:
        data->SaveText();
        break;

     // Загрузка параметров в текстовом виде
     case RDS_BFM_LOADTXT:
        data->LoadText((char*)ExtParam);
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef y
#undef Ready
#undef Start
#undef pStart
}
//=========================================
```

В этой модели, как и в модели передатчика, при инициализации тоже создается объект класса TNetSendRcvData, но в его конструктор передается константа

NETSRMODE_RECEIVER, указывающая на то, что этот объект принадлежит блоку-приемнику. Реакции на установку и разрыв связи в модели отсутствуют: этому блоку не нужно знать, есть ли в данный момент связь с сервером. Он реагирует на пришедшие от сервера данные – если связи нет, данные просто не будут приходить.

Полученные по сети данные обрабатываются в реакции модели на событие RDS_BFM_NETDATARECEIVED. При этом вызывается функция ReceiveValue класса TNetSendRcvData, которую мы написали ранее. В функцию передается указатель на структуру, описывающую принятые данные (для этого ExtParam приводится к типу RDS_NETRECEIVEDDATA*) и указатель на выход блока y, в который она запишет принятое вещественное число. Если ReceiveValue вернет TRUE, то есть если принятые данные соответствуют размеру числа double, будет взведен сигнал готовности блока Ready, чтобы новое значение выхода было передано по связям.

Все остальные реакции модели приемника совпадают с реакциями модели передатчика: их личная область данных описывается одним классом, поэтому при настройке, загрузке и сохранении параметров вызываются одни и те же функции этого класса. В тех случаях, когда для работы функции важно, обслуживает она блок-передатчик или приемник, внутри функции анализируется поле класса Mode.

Для проверки работы созданных блоков потребуется две машины, соединенные сетью. В сетевых настройках РДС (см. рис. 109) на каждой из них в поле “сервер по умолчанию” необходимо указать IP-адрес той машины, которая будет играть роль сервера, и задать один и тот же номер порта. На сервере следует собрать схему, изображенную на рис. 110 слева: в ней должен присутствовать блок, запускающий сервер (модель Server, стр. 448), блок-передатчик (модель NetSend) и блок-приемник (модель NetReceive). В параметрах всех трех блоков необходимо установить флаг “запуск по сигналу”, к входу блока-передатчика подключить поле ввода, а к выходу блока-приемника – индикатор. В параметрах передатчика и приемника нужно ввести разные имена каналов (например, “Channel1” и “Channel2”).

На машине-клиенте следует собрать аналогичную схему (рис. 110, справа), но без блока включения сервера. Имена каналов у приемника и передатчика нужно задать такими, чтобы приемник клиента работал с каналом передатчика сервера, а передатчик клиента – с каналом приемника сервера. Например, если в блоке-передатчике на сервере задан канал “Channel1”, это же имя канала нужно ввести в настройках блока-приемника клиента.

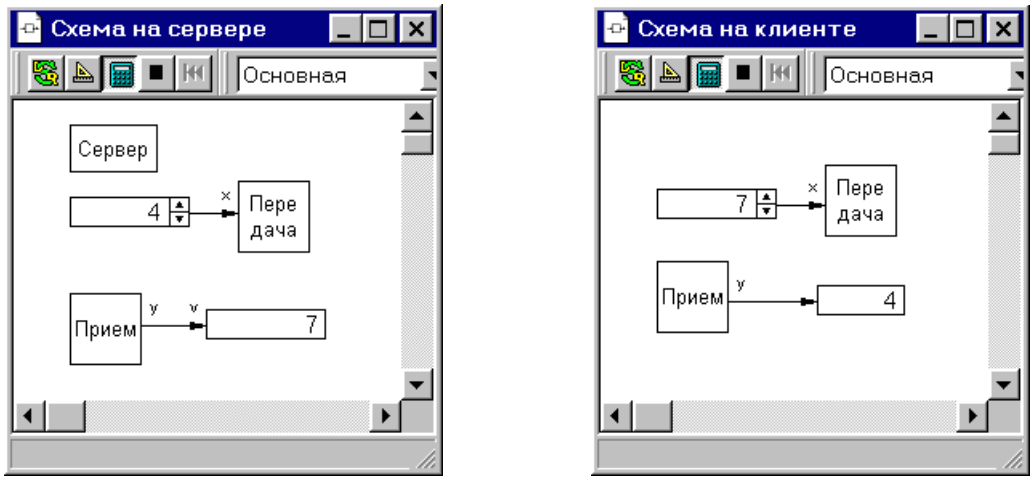


Рис. 110. Схемы для тестирования блоков сетевого обмена

Теперь нужно запустить расчет на обеих машинах. Если имена каналов заданы правильно, значение, введенное в поле ввода в одной схеме, должно отображаться на индикаторе в другой.

Если открыть на сервере окно сетевых соединений (пункт главного меню РДС “Окна | Сетевые соединения”, рис. 111), можно будет увидеть два IP-адреса подключенных клиентов. Один из них, “127.0.0.1”, это адрес этой же машины – она выступает сервером и для блоков в своей собственной схеме. Второй – адрес машины, на которой работает схема-клиент. Можно также видеть имена трех созданных блоками каналов, один из которых используется только для включения сервера, а два других связывают передатчики и приемники в двух схемах.

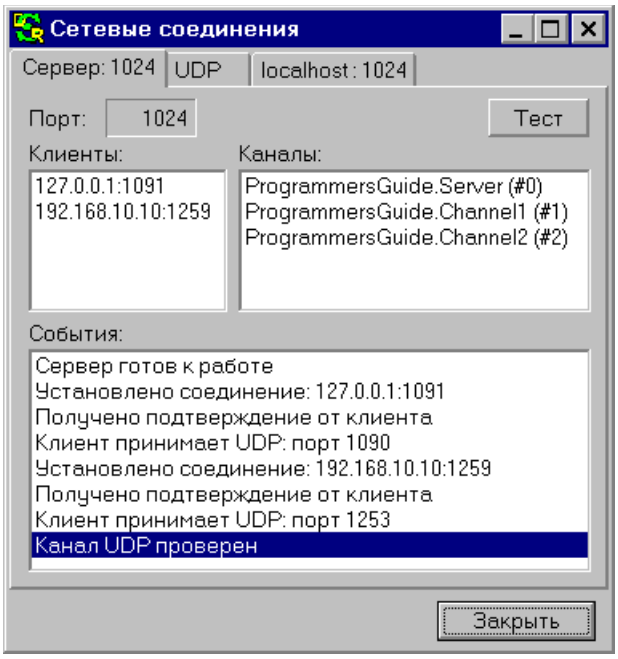


Рис. 111. Окно сетевых соединений

§2.15.3. Способы снижения нагрузки на сеть

Рассматриваются два способа снижения нагрузки на сеть: ограничение минимального интервала между моментами передачи данных и объединение передаваемых данных в один блок. Приводятся примеры реализации этих способов.

В схемах, собранных в предыдущем параграфе, входы блоков-передатчиков были соединены с полями ввода. Когда пользователь меняет значения в одном из этих полей, это значение поступает по связям на вход передатчика, его модель запускается, и значение передается на сервер. Как бы быстро ни менял пользователь значения в полях ввода, его скорость не может сравниться со скоростью передачи данных по сети, поэтому нагрузка на сеть от этих схем незначительна. Ситуация изменится, если к входу блока-передатчика подключить выход блока, выдающего новое значение в каждом такте расчета. Такты расчета следуют друг за другом очень быстро, и, если несколько блоков-передатчиков будут отправлять данные на сервер в каждом такте, это может привести к перегрузке сети.

Как правило, при обмене данными между схемами не требуется особенно высокая скорость передачи данных. Чаще всего достаточно поддерживать данные в блоках-приемниках в актуальном состоянии, передавая их один раз в фиксированный интервал времени. Этот интервал подбирают исходя из производительности сети и требований к работе схемы. Если данные приходят на вход блока слишком быстро, их отправку на сервер можно отложить до тех пор, пока с момента предыдущей отправки не пройдет заданное время. Это позволит существенно снизить нагрузку на сеть.

Изменим предыдущий пример таким образом, чтобы в параметрах блока-передатчика можно было задать минимально допустимый интервал времени между передачами данных. Для этого, прежде всего, нужно добавить в класс TNetSendRcvData несколько новых полей и функций (выделены жирным):

```
// Личная область данных блоков приема и передачи по сети
class TNetSendRcvData
{ public:
    int Mode;   // Режим данного блока: прием или передача
      #define NETSRMODE_SENDER        0     // Передатчик
      #define NETSRMODE_RECEIVER      1     // Приемник
    char *ChannelName;     // Имя канала
    BOOL LimitSpeed; // Задан минимальный интервал передачи
    DWORD Delay;     // Минимальный интервал в мс

    int ConnId; // Идентификатор соединения

    // Переменные состояния блока-передатчика
    BOOL Connected;   // Соединение установлено
    BOOL DataWaiting; // Передача данных отложена
    RDS_TIMERID Timer;// Таймер для отсчета интервала
    BOOL WaitingForTimer;// Таймер запущен - ждем
    DWORD LastSendTime; // Время последней отправки

    // Функции класса
    void Connect(void);    // Установить соединение
    void Disconnect(void); // Разорвать соединение

    void SendValue(double value);     // Передать число в канал
    BOOL ReceiveValue(RDS_NETRECEIVEDDATA *rcv,  // Реакция на
                      double *pOut);       // пришедшие данные

    void CreateTimer(void);      // Создать таймер
    void DeleteTimer(void);      // Удалить таймер
    BOOL CheckSendTimer(void);   // Проверить, можно ли передавать,
                                 // и запустить таймер, если нельзя

    int Setup(char *title);      // Функция настройки блока
    void SaveText(void);         // Сохранить параметры
    void LoadText(char *text);   // Загрузить параметры

    // Конструктор класса
    TNetSendRcvData(int mode)
      { ConnId=-1;    // Нет соединения
        Connected=DataWaiting=FALSE;
        LimitSpeed=WaitingForTimer=FALSE;Timer=NULL;Delay=100;
        ChannelName=NULL;
        Mode=mode;};  // Режим передается в параметре конструктора
    // Деструктор класса
    ~TNetSendRcvData()
      { Disconnect();           // Разорвать соединение
        DeleteTimer();          // Удалить таймер
        rdsFree(ChannelName);   // Освободить строку имени канала
      };
};
//=========================================
```

Поля LimitSpeed и Delay – настроечные параметры блока. Логическое поле LimitSpeed будет содержать TRUE, если интервал между передачами данных на сервер должен быть ограничен, и FALSE, если блок, как и прежде, должен работать без ограничений. Поле Delay содержит минимально разрешенный интервал в миллисекундах между передачами данных.

Остальные добавленные поля нужны для создания задержки в передаче данных. В поле LastSendTime при каждой передаче будет записываться время в миллисекундах, прошедшее с момента старта операционной системы (его можно получить при помощи функции Windows API GetTickCount). В поле Timer будет находиться идентификатор таймера, который будет отвечать за задержку передачи. Если вход блока изменился, но с момента последней передачи (LastSendTime) прошло менее Delay миллисекунд, этот таймер будет запускаться, и одновременно с его запуском будет взводиться логический флаг WaitingForTimer. Когда таймер сработает, значение входа блока будет передано на сервер и флаг WaitingForTimer будет сброшен. Если до момента срабатывания таймера, то есть при взведенном WaitingForTimer, вход блока изменится еще раз, данные передаваться не будут – они отправятся на сервер только тогда, когда таймер сработает.

Три новых функции, добавленные в класс, отвечают за работу с таймером: функция CreateTimer будет создавать таймер, если в настройках блока разрешено ограничение интервала передачи, функция DeleteTimer – уничтожать созданный таймер, функция CheckSendTimer – проверять, можно ли передать данные прямо сейчас (при этом она вернет TRUE) или нужно ждать заданный интервал времени (при этом она запустит таймер и вернет FALSE). В конструктор и деструктор класса тоже внесены изменения: в конструкторе новым полям присваиваются начальные значения, а в деструктор добавлен вызов функции уничтожения созданного таймера.

Напишем функции создания и уничтожения таймера. Работа с таймерами была подробно рассмотрена в §2.9, поэтому в этих функциях не будет ничего нового: мы будем использовать однократно срабатывающий таймер, вызывающий модель блока в режиме RDS_BFM_TIMER. Разумеется, создавать таймер будет только блок-передатчик, блоку-приемнику таймер не нужен.

```
// Создать таймер
void TNetSendRcvData::CreateTimer(void)
{
  if(Mode!=NETSRMODE_SENDER || // Приемнику таймер не нужен
     (!LimitSpeed) ) // Интервал передачи не ограничивается
    { DeleteTimer(); // Удаляем таймер, если он создан
      return;
    }
  if(Timer) // Таймер уже создан
    return;
  // Создаем таймер
  Timer=rdsSetBlockTimer(
           NULL,      // Создается новый таймер
           0,         // Задержка задается при запуске
           RDS_TIMERM_STOP | RDS_TIMERS_TIMER, // Однократный
           FALSE);    // Создается остановленным
}
//=======================================

// Удалить таймер
void TNetSendRcvData::DeleteTimer(void)
{
  if(Timer) // Таймер есть
    { rdsDeleteBlockTimer(Timer);
```

```
      Timer=NULL;
    }
  WaitingForTimer=FALSE; // Сбрасываем флаг ожидания
}
//=======================================
```

В функции создания таймера `CreateTimer` сначала проверяется, нужен ли вообще этому блоку таймер. Если объект этого класса принадлежит блоку-приемнику (поле `Mode` не равно константе `NETSRMODE_SENDER`) или если ограничение интервала передачи выключено (поле `LimitSpeed` имеет значение `FALSE`), то таймер блоку не нужен: если он есть, он уничтожается вызовом `DeleteTimer` и работа функции завершается. Если же таймер блоку необходим, мы проверяем, нет ли уже идентификатора таймера в поле `Timer`. Если его значение – не `NULL`, значит, таймер уже был создан, и создавать его повторно не нужно. В противном случае вызовом сервисной функции РДС `rdsSetBlockTimer` создается новый таймер, который будет срабатывать однократно (флаг `RDS_TIMERM_STOP`) и, при срабатывании, вызывать модель блока в режиме `RDS_BFM_TIMER` (флаг `RDS_TIMERS_TIMER`).

Функция уничтожения таймера `DeleteTimer` удаляет созданный таймер вызовом `rdsDeleteBlockTimer` и присваивает полю `Timer` значение `NULL`, которое будет указывать на то, что таймер не создан. Кроме того, она сбрасывает логический флаг `WaitingForTimer`, поскольку теперь мы не ждем срабатывания таймера, и поступившие на вход блока данные нужно будет передавать на сервер немедленно.

Теперь напишем функцию `CheckSendTimer`, которая будет проверять, можно ли передать данные немедленно, при невозможности немедленной передачи, запускать таймер:

```
// Проверить, можно ли передать данные немедленно, и запустить
// таймер, если нельзя
BOOL TNetSendRcvData::CheckSendTimer(void)
{ DWORD interval;

  if(!Connected)      // Нет связи с сервером
    return TRUE;      // Разрешаем отправку, чтобы попытка отправки
                      // была зафиксирована в функции SendValue

  if(!LimitSpeed)     // Интервал не ограничивается
    return TRUE;

  if(WaitingForTimer) // Уже запустили таймер и ждем срабатывания
    return FALSE;

  // Интервал передачи ограничен, таймер сейчас выключен

  interval=GetTickCount()-LastSendTime; // Время с прошлой отправки
  if(interval>=Delay) // Прошло много времени - можно передавать
    return TRUE;
  // С прошлой отправки прошло менее Delay мс
  // Нужно подождать (Delay-interval)
  WaitingForTimer=TRUE;    // Взводим флаг ожидания таймера
  rdsRestartBlockTimer(Timer,Delay-interval); // Запускаем таймер
  return FALSE; // Передавать сейчас нельзя - ждем таймера
}
//=======================================
```

Сначала мы проверяем, взведен ли флаг `Connected`, то есть есть ли у блока связь с сервером. Если связи нет, функция возвращает `TRUE`, разрешая немедленную передачу данных. Это сделано для того, чтобы вызванная для передачи функция `SendValue` зафиксировала попытку отправить данные на сервер в отсутствие связи с ним, и взвела флаг

DataWaiting. Можно было бы просто взвести этот флаг в функции CheckSendTimer, но, поскольку функция SendValue уже написана, лучше оставить работу с этим флагом ей.

Если в настройках блока не ограничен интервал передачи данных (LimitSpeed равно FALSE), функция возвращает TRUE – данные нужно передать немедленно. Если таймер уже запущен, и мы ждем его срабатывания, чтобы передать данные (WaitingForTimer равно TRUE), функция возвращает FALSE – данные передавать нельзя. Если же ни одна из этих проверок не выполнилась, значит, в настройках блока задано ограничение интервала передачи, и таймер не запущен. В этом случае необходимо проверить, сколько времени прошло с момента последней передачи, и либо разрешить немедленную передачу данных, если прошло не менее Delay миллисекунд, либо запретить передачу и запустить таймер.

В переменную interval записывается интервал времени в миллисекундах, прошедший с момента последней передачи данных на сервер. Он вычисляется как разность между текущим значением времени, получаемым при помощи функции GetTickCount, и запомненным временем передачи LastSendTime. Если этот интервал больше или равен параметру Delay, функция возвращает TRUE – данные можно передавать немедленно. В противном случае нужно дождаться конца интервала: необходимо подождать еще Delay–interval миллисекунд. Таймер Timer программируется на это значение задержки вызовом rdsRestartBlockTimer, взводится флаг WaitingForTimer, и функция возвращает FALSE, запрещая передачу – данные будут переданы тогда, когда таймер сработает.

Для того, чтобы созданная нами конструкция работала, необходимо внести несколько изменений в написанные ранее функции класса. Прежде всего, функция SendValue теперь должна запоминать время последней передачи данных:

```
// Передать данные
void TNetSendRcvData::SendValue(double value)
{ if(!Connected)      // Нет связи с сервером
    { // Взводим флаг наличия данных, ожидающих передачи
      DataWaiting=TRUE;
      return;
    }
  // Связь есть - передаем всем блокам канала
  rdsNetBroadcastData(ConnId,    // Соединение
                      RDS_NETSEND_UPDATE|RDS_NETSEND_UDP, // Флаги
                      0,NULL, // Не передаем целое число и строку
                      &value,         // Указатель на данные
                      sizeof(value)); // Размер данных
  // Сбрасываем флаг ожидания - мы только что передали данные
  DataWaiting=FALSE;
  // Запоминаем время последней передачи
  LastSendTime=GetTickCount();
}
//=========================================
```

Создавать таймер, вызывая функцию CreateTimer, мы будем в функции установки соединения Connect: мы вызываем ее после загрузки параметров блока и изменения пользователем его настроек, а это как раз те случаи, когда нужно создать таймер, если он нужен, и удалить его, если необходимость в нем исчезла. Функция CreateTimer написана так, что она может не только создать таймер, но и уничтожить его, если, например, пользователь отменил ограничение интервала передачи в настройках блока.

```
// Установка соединения
void TNetSendRcvData::Connect(void)
{ char *PrefixedName; // Полное имя канала
```

```
  // Если имя канала пустое, соединение невозможно
  if(ChannelName==NULL || (*ChannelName)==0)
    return;
  // Добавляем префикс к имени канала
  PrefixedName=rdsDynStrCat("ProgrammersGuide.",ChannelName,FALSE);
  // Устанавливаем соединение с сервером
  ConnId=rdsNetConnect(NULL,     // Сервер по умолчанию
                       -1,       // Порт по умолчанию
                       PrefixedName,// Имя канала с префиксом
                       Mode==NETSRMODE_RECEIVER); // Прием данных
  // Освобождаем динамически отведенную строку
  rdsFree(PrefixedName);
  // Создаем или уничтожаем таймер
  CreateTimer();
}
//=======================================
```

В блоке появилось два новых настроечных параметра, `LimitSpeed` и `Delay`, поэтому нам нужно внести изменения в функции сохранения, загрузки и настройки параметров. Функция сохранения теперь будет выглядеть следующим образом:

```
// Сохранение параметров блока
void TNetSendRcvData::SaveText(void)
{ RDS_HOBJECT ini;    // Вспомогательный объект
  // Создаем вспомогательный объект
  ini=rdsINICreateTextHolder(TRUE);
  // Создаем в объекте секцию "[General]"
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"General");
  // Записываем в эту секцию имя канала
  rdsINIWriteString(ini,"Channel",ChannelName);
  if(Mode==NETSRMODE_SENDER) // Передатчик
    { // Создаем новую секцию
      rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"Timer");
      // Записываем параметры
      rdsINIWriteInt(ini,"On",LimitSpeed);
      rdsINIWriteInt(ini,"Delay",Delay);
    }
  // Сохраняем текст, сформированный объектом, как параметры блока
  rdsCommandObject(ini,RDS_HINI_SAVEBLOCKTEXT);
  // Удаляем вспомогательный объект
  rdsDeleteObject(ini);
}
//=======================================
```

Значения параметров `LimitSpeed` и `Delay` записываются только для блоков-передатчиков, поскольку в приемниках они не используются. Функция загрузки тоже меняется соответствующим образом:

```
// Загрузка параметров блока
void TNetSendRcvData::LoadText(char *text)
{ RDS_HOBJECT ini;    // Вспомогательный объект
  char *str;
  // Создаем вспомогательный объект
  ini=rdsINICreateTextHolder(TRUE);
  // Записываем в объект полученный текст с параметрами блока
  rdsSetObjectStr(ini,RDS_HINI_SETTEXT,0,text);
  // Начинаем чтение секции "[General]", если она есть
  if(rdsINIOpenSection(ini,"General")) // Секция есть
    { // ....
      // без изменений
```

```
    // ....
  }

  if(Mode==NETSRMODE_SENDER && // Передатчик
     rdsINIOpenSection(ini,"Timer")) // Есть секция "[Timer]"
  { LimitSpeed=rdsINIReadInt(ini,"On",LimitSpeed)!=0;
    Delay=rdsINIReadInt(ini,"Delay",Delay);
  }

  // Удаляем вспомогательный объект
  rdsDeleteObject(ini);
  // Поскольку имя канала могло измениться, соединяемся с
  // сервером заново
  Disconnect(); // Разрываем старое соединение
  Connect();    // Создаем новое
}
//=======================================
```

Здесь мы читаем параметры из секции “[Timer]” только в том случае, если данный блок – передатчик, и такая секция есть в тексте параметров блока.

Самые серьезные изменения необходимо внести в функцию настройки блока. Раньше у него был единственный параметр – имя канала `ChannelName`, поэтому для настройки мы пользовались функцией ввода строки `rdsInputString`. Теперь для передатчика нам необходимо вводить еще два параметра, для чего придется создавать полноценное модальное окно. Фактически, функцию `Setup` придется переписать заново:

```
// Настройка параметров блока
int TNetSendRcvData::Setup(char *title)
{ RDS_HOBJECT win;    // Идентификатор вспомогательного объекта
  BOOL ok;            // Пользователь нажал "OK"
  // Создание окна
  win=rdsFORMCreate(FALSE,-1,-1,title);
  // Поле ввода имени канала
  rdsFORMAddEdit(win,0,1,RDS_FORMCTRL_EDIT,
                 "Имя канала:",200);
  rdsSetObjectStr(win,1,RDS_FORMVAL_VALUE,ChannelName);

  if(Mode==NETSRMODE_SENDER)
    { // Для передатчика - ввод интервала
      rdsFORMAddEdit(win,0,2,
           RDS_FORMCTRL_EDIT | RDS_FORMFLAG_CHECK,
           "Интервал передачи, мс:",80);
      // Значение интервала
      rdsSetObjectInt(win,2,RDS_FORMVAL_VALUE,Delay);
      // Разрешающий флаг поля
      rdsSetObjectInt(win,2,RDS_FORMVAL_CHECK,LimitSpeed);
    }

  // Открытие окна
  ok=rdsFORMShowModalEx(win,NULL);
  if(ok)
    { // Пользователь нажал OK
      char *NewName=rdsGetObjectStr(win,1,RDS_FORMVAL_VALUE);
      if(ChannelName==NULL || strcmp(NewName,ChannelName)!=0)
        { // Имя канала изменилось - запоминаем новое
          rdsFree(ChannelName);
          ChannelName=rdsDynStrCopy(NewName);
        }
```

```
      // Флаг ограничения интервала и сам интервал
      LimitSpeed=rdsGetObjectInt(win,2,RDS_FORMVAL_CHECK)!=0;
      Delay=rdsGetObjectInt(win,2,RDS_FORMVAL_VALUE);
      // Устанавливаем соединение с сервером заново и создаем
      // или удаляем таймер, если необходимо
      Disconnect();
      Connect();
    }
  // Уничтожаем вспомогательный объект-окно
  rdsDeleteObject(win);
  // Возвращаемое значение
  return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
}
//=======================================
```

В этой функции, как и во многих других примерах, окно для ввода параметров создается с помощью вспомогательного объекта РДС. Функция будет вызываться и для блока-передатчика, и для блока-приемника, поэтому внутри нее анализируется поле `Mode`: если в нем находится константа `NETSRMODE_RECEIVER`, в окне будет только поле ввода для имени канала, если `NETSRMODE_SENDER` – дополнительное поле для ввода интервала передачи `Delay`, совмещенное с разрешающим флагом `LimitSpeed` (рис. 112). Если пользователь закроет окно кнопкой “OK”, строка имени канала `ChannelName` будет освобождена вызовом `rdsFree`, и новое имя будет скопировано из объекта в новую динамическую строку, созданную вызовом `rdsDynStrCopy`. Разрешающий флаг поля ввода интервала и значение этого поля будут записаны в поля класса `LimitSpeed` и `Delay` соответственно. Это будет сделано не только для блока-передатчика, но и для блока-приемника, которому эти поля не нужны – значение `Mode` здесь не проверяется. Хотя поля ввода с идентификатором 2, из которого читаются эти значения, в окне настроек приемника не будет, ничего страшного не случится: при попытке чтения данных из несуществующего поля будут возвращены нули.

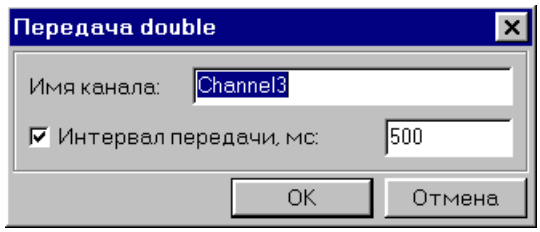


Рис. 112. Настройка передатчика

После чтения всех параметров из окна настроек в поля класса, как и раньше, последовательно вызываются функции `Disconnect` и `Connect`. Это приведет к тому, что соединение с сервером будет разорвано, а затем установлено заново, уже с новым именем канала. При этом внутри функции `Connect`, в зависимости от нового значения поля `LimitSpeed`, будет создан или удален таймер.

Теперь нужно внести изменения в функцию модели блока-передатчика `NetSend`: необходимо добавить реакцию на срабатывание таймера и проверку возможности передачи в реакцию на такт расчета:

```
    // Срабатывание таймера
    case RDS_BFM_TIMER:
      // Сбрасываем флаг ожидания таймера
      data->WaitingForTimer=FALSE;
      // Передаем значение входа блока
      data->SendValue(x);
      break;

    // Такт расчета
    case RDS_BFM_MODEL:
      if(data->CheckSendTimer()) // Можно передавать немедленно
```

```
            data->SendValue(x);    // Передаем значение входа
          break;
```

Если таймер сработал, значит, интервал ожидания кончился, и значение входа блока необходимо передать на сервер. Для этого вызывается функция `SendValue`, а флаг ожидания таймера `WaitingForTimer` сбрасывается – теперь его можно запустить снова, если значение на входе изменится слишком быстро. В реакцию на такт расчета добавлен вызов функции проверки `CheckSendTimer`. Если с момента последней передачи прошло слишком мало времени, она запустит таймер и вернет `FALSE` – функция `SendValue` при этом вызвана не будет. Если интервал передачи отключен, или уже прошло достаточно времени, функция вернет `TRUE`, и значение входа блока будет отправлено на сервер немедленно.

Для проверки работы измененной модели соберем две схемы, подобных показанным на рис. 113, на разных машинах (сервером можно сделать одну из них, или запустить отдельный сервер на третьей). В настройках блока-передатчика зададим минимальный интервал передачи 250 мс.

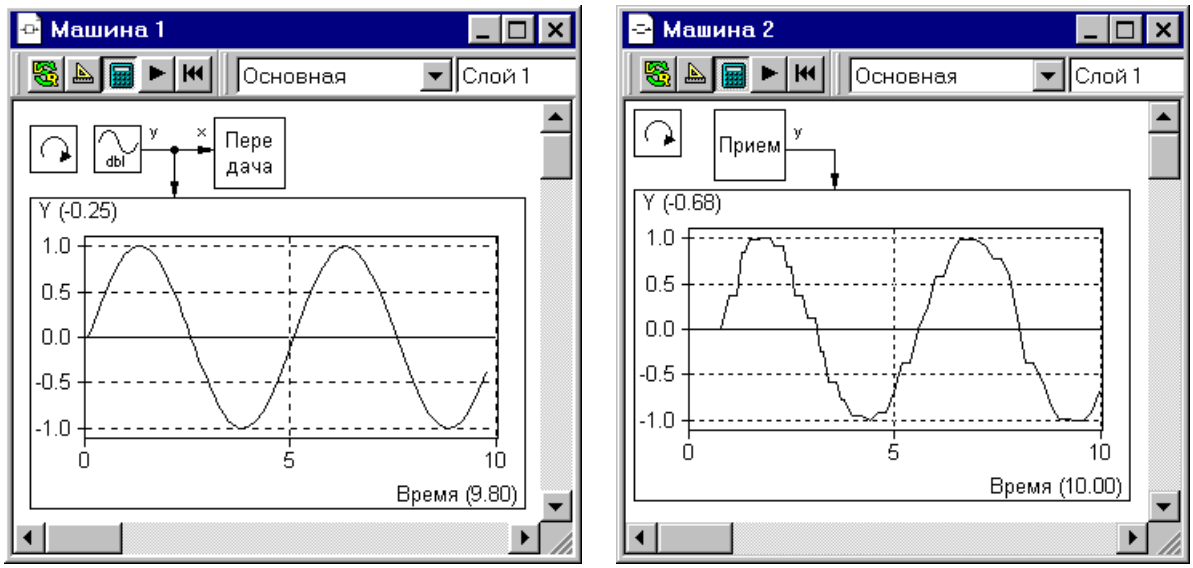


Рис. 113. Передача с ограничением интервала

В первой схеме выход генератора синусоиды соединен с входом блока-передатчика и графиком. Во второй схеме выход приемника, работающего с тем же каналом, что и передатчик в первой схеме, подан на график. Если одновременно запустить расчет в обеих схемах (для этого, при желании, можно использовать блок синхронизации запуска по сети из стандартной библиотеки блоков РДС), можно увидеть, что сигнал на принимающей машине, во-первых, отстает от переданного из-за задержек в сети, и, во-вторых, стал ступенчатым из-за неравномерности передачи данных и ограничения интервала передачи. Если увеличивать интервал, длина ступенек тоже будет увеличиваться. Если же отключить минимальный интервал в настройках передатчика, в загруженной сети принятый сигнал может потерять всякое сходство с синусоидой из-за перегрузки.

Другой способ уменьшить нагрузку на сеть – объединение различных передаваемых данных в один блок, что позволяет уменьшить накладные расходы на передачу. К каждому передаваемому блоку данных, независимо от его размера, добавляется служебная информация, позволяющая серверу понять, какому каналу адресована передача, что в ней содержится и т.п. Поэтому, если между двумя схемами необходимо передавать десять вещественных чисел двойной точности (`double`), каждое из которых занимает 8 байтов, значительно выгоднее отправить один восьмидесятибайтовый блок данных, чем десять

восьмибайтовых – общий объем служебной информации в этом случае будет в десять раз меньше.

Создадим два блока для передачи и приема массива вещественных чисел произвольного размера. И на передающей, и на принимающей стороне можно будет подключать связи к отдельным элементам этого массива, таким образом один блок сможет передать в другую схему сразу несколько чисел за один раз. Мы не будем создавать эти блоки "с нуля", вместо этого изменим все тот же класс `TNetSendRcvData`, добавив в него функции для передачи и приема массива, и сделаем две новых функции модели, которые будут пользоваться этим классом.

В описании класса, кроме добавления описаний двух функций `SendArray` и `ReceiveArray`, ничего не изменится:

```
// Личная область данных блоков приема и передачи по сети
class TNetSendRcvData
{ public:
    int Mode;   // Режим данного блока: прием или передача
      #define NETSRMODE_SENDER        0     // Передатчик
      #define NETSRMODE_RECEIVER      1     // Приемник

    // ... без изменений ...

    void SendValue(double value);     // Передать число в канал
    BOOL ReceiveValue(RDS_NETRECEIVEDDATA *rcv,  // Реакция на
                      double *pOut);       // пришедшие данные

    void SendArray(void *input); // Передать массив в канал
    BOOL ReceiveArray(RDS_NETRECEIVEDDATA *rcv,  // Реакция на
                      void *output);       // пришедшие данные

    int Setup(char *title);     // Функция настройки блока

    // ... без изменений ...

};
//=======================================
```

В функции `SendArray` и `ReceiveArray` будут передаваться указатели на вход блока `input` и выход `output` соответственно. Выход и вход должны быть массивами чисел типа `double` (строка типа "MD"), указатели на них, в данном случае, передаются как универсальные указатели `void*`. В `ReceiveArray`, как и в `ReceiveValue`, будет также передаваться указатель `rcv` на структуру с принятыми данными.

Функция `SendArray` будет очень похожа на `SendValue`, только она будет отправлять не одно вещественное число, а несколько чисел сразу. Количество передаваемых чисел будет определяться текущим размером массива. Поскольку функция `rdsNetBroadcastData` одновременно с блоком двоичных данных может передавать целое число и строку, размер массива будет передаваться в этом целом числе.

```
// Передать массив
void TNetSendRcvData::SendArray(void *input)
{ int N;

  if(!Connected)      // Нет связи с сервером
    { // Взводим флаг наличия данных, ожидающих передачи
      DataWaiting=TRUE;
      return;
    }
```

```
  // Связь есть - определяем размер массива input
  N=RDS_ARRAYEXISTS(input)?
      (RDS_ARRAYROWS(input)*RDS_ARRAYCOLS(input)):0;
  if(N==0) // Массив пуст - передавать нечего
    return;

  // Передаем N чисел double всем блокам канала
  rdsNetBroadcastData(
           ConnId,    // Соединение
           RDS_NETSEND_UPDATE|RDS_NETSEND_UDP, // Флаги
           N,         // Целое число - размер массива
           NULL,      // Строка не передается
           RDS_ARRAYDATA(input), // Начало данных массива
           N*sizeof(double));    // Размер массива в байтах
  // Сбрасываем флаг ожидания - мы только что передали данные
  DataWaiting=FALSE;
  // Запоминаем время передачи
  LastSendTime=GetTickCount();
}
//=======================================
```

Сначала мы, как и в `SendValue`, проверяем наличие связи с сервером, и, если ее нет, взводим флаг `DataWaiting` и завершаем функцию. Если связь есть, мы определяем размер массива. Для проверки массива на пустоту используется макрос `RDS_ARRAYEXISTS`, для определения числа строк и столбцов – `RDS_ARRAYROWS` и `RDS_ARRAYCOLS` (работа с массивами в РДС и использование этих макросов подробно рассмотрены в §2.5.3). Хотя массив в РДС представляет собой матрицу из одной строки, мы не можем гарантировать, что эта модель будет подключена к блоку, вход которого является именно массивом, а не матрицей вещественных чисел – оба они имеют строку типа “MD”. На всякий случай, мы вычисляем число элементов массива `N` как произведение числа строк на число столбцов, в этом случае размер будет определен правильно, даже если вход будет матрицей.

Если в массиве 0 элементов, функция завершается, в противном случае вызывается `rdsNetBroadcastData`, передающая содержимое массива как блок двоичных данных и его размер как целое число. Указатель на самый первый элемент массива мы получаем с помощью макроса `RDS_ARRAYDATA`, а размер массива в байтах вычисляется как произведение числа элементов `N` на размер элемента `sizeof(double)`. После передачи, как и в функции `SendValue`, сбрасывается флаг `DataWaiting` и запоминается время последней передачи `LastSendTime`.

Функция `ReceiveArray` тоже будет очень похожа на `ReceiveValue`:

```
// Прием массива
BOOL TNetSendRcvData::ReceiveArray(
     RDS_NETRECEIVEDDATA *rcv, // Указатель на структуру с данными
     void *output)   // Указатель на выход блока (массив)
{ int N;
  if(rcv==NULL||output==NULL) // Нет одного из указателей
    return FALSE;

  // Проверяем, есть ли среди принятых данных двоичные,
  // и кратен ли размер блока данных размеру double
  if(rcv->Buffer==NULL ||  // Нет буфера с данными
     rcv->BufferSize%sizeof(double)!=0) // Размер не кратен 8
    return FALSE;

  // Вычисляем число элементов в принятом массиве
  N=rcv->BufferSize/sizeof(double);
```

```
    // Вычисленное число элементов должно совпасть с переданным
    if(N!=rcv->Id)
      return FALSE;
    // Принято N чисел double - отводим массив под них
    if(!rdsResizeVarArray(output,1,N,FALSE,NULL))
      return FALSE;
    // Копируем принятые данные в отведенный массив выхода
    memcpy(RDS_ARRAYDATA(output),rcv->Buffer,rcv->BufferSize);
    return TRUE;
  }
  //=======================================
```

В этой функции необходимо проверить, есть ли в принятых данных двоичный блок, и кратен ли размер этого блока размеру числа `double`. Если остаток от деления `rcv->BufferSize` на `sizeof(double)` не будет равен нулю, значит, в принятом блоке не содержится целое число элементов типа `double`. Это говорит о том, что принятые данные не соответствуют формату, с которым работает наш блок, и функция завершается, возвращая `FALSE` – данные не приняты.

Если принятые данные прошли эту проверку, вычисляется число элементов в принятом массиве `N`: оно равно отношению размеру всего массива в байтах (`rcv->BufferSize`) к размеру одного элемента (`sizeof(double)`) – мы уже выяснили, что результат этого деления будет целым числом. Затем `N` сравнивается с принятым целым числом `rcv->Id`: если они не равны, значит, данные отправлены не блоком-передатчиком массива через функцию `SendArray`, в этом случае мы возвращаем `FALSE` – принятые данные не подходят нашему блоку. Если же два этих числа равны, мы даем выходу блока размерность `1`**`x`**`N` при помощи функции `rdsResizeVarArray`, после чего копируем принятые данные в полученный массив функцией `memcpy` из стандартной библиотеки языка C. Для получения указателя на первый элемент массива, то есть начального адреса области, куда копируются данные, используется макрос `RDS_ARRAYDATA`. После копирования функция `ReceiveArray` возвращает `TRUE` – данные приняты успешно.

Теперь нужно написать модели блоков, которые будут передавать и принимать массивы. Они будут практически точными копиями моделей `NetSend` и `NetReceive`, за исключением того, что в них будут использоваться другие макроопределения и строка для проверки типа переменных (входы и выходы будут массивами, а не числами, как раньше), у них будут другие заголовки окон настройки, а везде, где вызывались функции `SendValue` и `ReceiveValue`, будут вызываться `SendArray` и `ReceiveArray` соответственно.

Структура переменных блока-передатчика массива будет следующей:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/ выход*	*Пуск*	*Начальное значение*
0	Start	Сигнал	1	Вход	✓	0
1	Ready	Сигнал	1	Выход		0
2	X	Массив double	8	Вход	✓	[ ] 0

Как всегда, для блока нужно будет задать запуск по сигналу. Модель блока-передатчика будет такой (отличия от `NetSend` выделены жирным):

```
  // Блок-передатчик массива
  extern "C" __declspec(dllexport)
    int RDSCALL NetSendArray(int CallMode,
                             RDS_PBLOCKDATA BlockData,
                          LPVOID ExtParam)
  { // Указатель на личную область данных, приведенный к нужному типу
    TNetSendRcvData *data=(TNetSendRcvData*)(BlockData->BlockData);
```

```
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define pX ((void **)(pStart+2))

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Создаем объект класса TNetSendRcvData с
        // Mode==NETSRMODE_SENDER (передатчик)
        BlockData->BlockData=new TNetSendRcvData(NETSRMODE_SENDER);
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;

      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSMD}")?
                     RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Связь с сервером установлена
      case RDS_BFM_NETCONNECT:
        // Взводим флаг наличия связи
        data->Connected=TRUE;
        // Если были данные, ожидающие отправки,
        // посылаем значение входа блока
        if(data->DataWaiting)
          data->SendArray(pX);
        break;

      // Связь с сервером разорвана
      case RDS_BFM_NETDISCONNECT:
        // Сбрасываем флаг наличия связи
        data->Connected=FALSE;
        break;

      // Запуск расчета
      case RDS_BFM_STARTCALC:
        // Если это - первый запуск после сброса,
        // передаем значение входа
        if(((RDS_PSTARTSTOPDATA)ExtParam)->FirstStart)
          data->SendArray(pX);
        break;

      // Срабатывание таймера
      case RDS_BFM_TIMER:
        // Сбрасываем флаг ожидания таймера
        data->WaitingForTimer=FALSE;
        // Передаем значение входа блока
        data->SendArray(pX);
        break;
```

```
        // Такт расчета
        case RDS_BFM_MODEL:
          if(data->CheckSendTimer()) // Можно передавать немедленно
            data->SendArray(pX);  // Передаем значение входа
          break;

        // Вызов настройки
        case RDS_BFM_SETUP:
          return data->Setup("Передача массива");

       // Сохранение параметров в текстовом виде
       case RDS_BFM_SAVETXT:
          data->SaveText();
          break;

       // Загрузка параметров в текстовом виде
       case RDS_BFM_LOADTXT:
          data->LoadText((char*)ExtParam);
          break;
      }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef pX
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Блоку-приемнику потребуется следующая структура переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/ выход*	*Начальное значение*
0	Start	Сигнал	1	Вход	0
1	Ready	Сигнал	1	Выход	0
2	Y	Массив double	8	Выход	[ ] 0

В параметрах этого блока тоже нужно будет задать запуск по сигналу. Его модель будет мало отличаться от NetReceive:

```
// Блок-приемник массива
extern "C" __declspec(dllexport)
  int RDSCALL NetReceiveArray(int CallMode,
                              RDS_PBLOCKDATA BlockData,
                              LPVOID ExtParam)
{ // Указатель на личную область данных, приведенный к нужному типу
  TNetSendRcvData *data=(TNetSendRcvData*)(BlockData->BlockData);
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
#define pY ((void **)(pStart+2))

  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        // Создаем объект класса TNetSendRcvData с
        // Mode==NETSRMODE_RECEIVER (приемник)
```

```
        BlockData->BlockData=
                            new TNetSendRcvData(NETSRMODE_RECEIVER);
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;

      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSMD}")?
                     RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // По сети получены данные
      case RDS_BFM_NETDATARECEIVED:
        if(data->ReceiveArray((RDS_NETRECEIVEDDATA*)ExtParam,pY))
          Ready=1;    // Если данные верны, взводим флаг готовности
                      // для передачи выхода по связям
        break;

      // Вызов настройки
      case RDS_BFM_SETUP:
        return data->Setup("Прием массива");

     // Сохранение параметров в текстовом виде
     case RDS_BFM_SAVETXT:
        data->SaveText();
        break;

     // Загрузка параметров в текстовом виде
     case RDS_BFM_LOADTXT:
        data->LoadText((char*)ExtParam);
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef pY
#undef Ready
#undef Start
#undef pStart
}
//=======================================
```

Для проверки работы блоков нужно собрать схемы, подобные изображенным на рис. 114. В данном случае первая машина играет роль сервера – в схеме, собранной на ней, находится блок запуска сервера, созданный нами в §2.15.2. Если модели работают правильно, при запущенном расчете в обеих схемах значения из полей ввода в первой схеме должны попадать в соответствующие индикаторы во второй.

Для передачи массива по сети мы сделали две отдельных модели. Можно было бы вместо этого сделать модели универсального приемника и универсального передатчика, которые, анализируя структуру переменных блока, к которому подключены, сами разбирались бы, работать им с числом или массивом, и вызывали бы соответствующие функции класса `TNetSendRcvData`. Мы не стали делать этого, чтобы не загромождать пример, поскольку принцип передачи при этом не изменился бы.

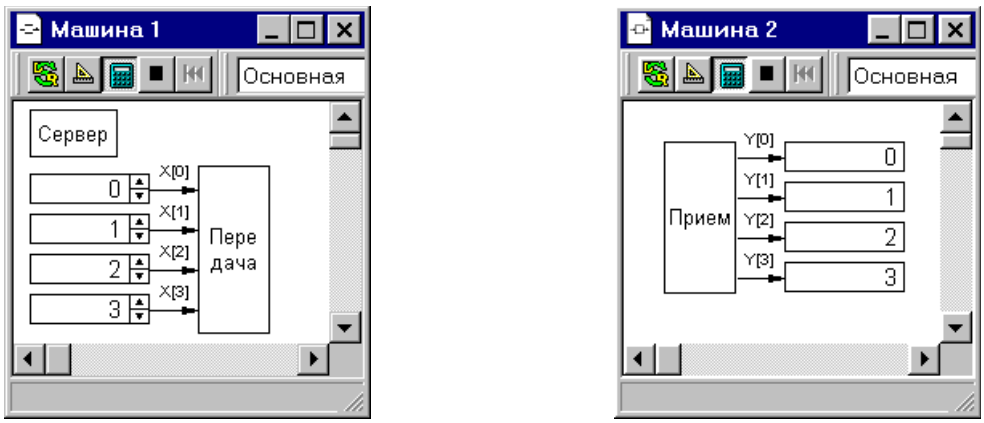


Рис. 114. Передача массива по сети

Рассмотренные способы снижения нагрузки на сеть позволяют несколько улучшить ситуацию при передаче данных и уменьшить вероятность перегрузки сети пакетами, которая обычно ведет к росту задержек при передаче. Однако, особенно высокой скорости при передаче данных от РДС ждать не следует. Если для работы блока нужна высокоскоростная передача, лучше реализовать ее в модели самостоятельно.

§2.16. Программное изменение схемы

Рассматриваются способы программной модификации схемы. Приводятся примеры блоков, изменяющих свою структуру переменных, а также добавляющих в схему и удаляющих из нее другие блоки и связи.

§2.16.1. Изменение структуры переменных блока

Рассматриваются функции для программного изменения структуры статических переменных блока из его модели. Модель одного из ранее рассмотренных блоков изменяется так, чтобы она сама корректировала переменные, если они ей не подходят. Описывается модель блока, предоставляющего пользователю интерфейс для ограниченного редактирования своих переменных.

Структура статических переменных простого блока обычно задается один раз при создании этого блока (см. рис. 8, 9), а модель только проверяет допустимость этой структуры в вызове `RDS_BFM_VARCHECK` и запрещает работу блока, если структура его переменных не соответствует ожиданиям модели (см. §2.5.1). Тем не менее, РДС позволяет модели, при необходимости, изменять структуру переменных блока или параметры этих переменных.

Одним из самых простых примеров из рассмотренных ранее был блок с моделью `TestSub`, выдававший на выход разность двух своих вещественных входов (см. стр. 41). В режиме `RDS_BFM_VARCHECK` его модель проверяла наличие двух обязательных сигналов `Start` и `Ready` и трех вещественных переменных, возвращая константу `RDS_BFR_BADVARSMSG`, если переданная ей строка типа отличалась от “{SSDDD}”. Таким образом, для того, чтобы этот блок мог работать, пользователь после подключения к новому блоку модели `TestSub` обязательно должен вручную задать ему правильную структуру переменных. Изменим модель так, чтобы при подключении к блоку с неподходящими переменными она автоматически меняла их на нужные ей. На самом деле, большой практической ценности этот пример иметь не будет: как правило, после создания блока его сразу записывают в библиотеку, откуда он добавляется в новые схемы уже с правильной структурой переменных. Однако, модель к блоку может быть подключена не только пользователем, но и, например, другим блоком при вызове сервисной функции `rdsSetBlockModel` – в этом случае автоматическое создание правильной структуры переменных может оказаться полезным.

Сначала напишем функцию, которая устанавливает нужную структуру переменных блока, а именно:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*
0	Start	Сигнал	1	Вход
1	Ready	Сигнал	1	Выход
2	x1	double	8	Вход
10	x2	double	8	Вход
18	y	double	8	Выход

Мы будем вызывать эту функцию из модели в режиме `RDS_BFM_VARCHECK` , чтобы установить правильную структуру переменных, если это необходимо.

```
// Создать структуру переменных из двух входов x1,x2 и выхода y
BOOL CreateVarStruct_x1x2y(RDS_BHANDLE Block)
{ // Строка описания структуры переменных
  static char str[]=
    "struct\nbegin\n"
    "signal name \"Start\" in run default 1\n"
    "signal name \"Ready\" out default 0\n"
    "double name \"x1\" in menu run default 0\n"
    "double name \"x2\" in menu run default 0\n"
    "double name \"y\" out menu default 0\n"
    "end";
  RDS_HOBJECT dv; // Вспомогательный объект
  BOOL ok;
  // Создаем объект для работы со структурой переменных
  dv=rdsVSCreateEditor();
  // Создаем в объекте набор переменных по строке описания
  ok=rdsVSCreateByDescr(dv,str);
  if(ok) // Переписываем структуру переменных из объекта в блок
    ok=rdsVSApplyToBlock(dv,Block,NULL);
  // Удаляем вспомогательный объект
  rdsDeleteObject(dv);
  // Возвращаем успешность операции
  return ok;
}
//=======================================
```

Большая часть операций над структурой переменных блока в РДС осуществляется через вспомогательный объект, который создается сервисной функцией `rdsVSCreateEditor`. Сначала в таком объекте подготавливается нужная структура, а затем вызовом функции `rdsVSApplyToBlock` эта структура записывается в конкретный блок. В данном случае идентификатор блока, которому нужно установить новую структуру переменных, передается в параметре нашей функции `CreateVarStruct_x1x2y`. Эта функция будет возвращать `TRUE`, если изменение структуры удалось, и `FALSE` в противном случае – мы не сможем установить такую структуру переменных, например, во внешнем входе или выходе, у которого может быть только одна переменная, или в блоке, модель которого требует другой структуры.

В самом начале функции вводится статическая переменная `str`, которой присваивается строка описания нашей структуры переменных в том же виде, в каком описываются переменные при сохранении блока или схемы в текстовом формате. В этой строке перечислены все имена и типы переменных, включая обязательные для простого блока сигналы `Start` и `Ready`. Такую строку описания для любой структуры переменных получить достаточно просто: нужно создать новую схему, добавить в нее единственный блок

с нужной структурой переменных, сохранить этот блок в файл, а затем открыть этот файл в любом текстовом редакторе, найти в нем описание переменных блока, начинающееся со слова “vars”, и скопировать это описание вплоть до ближайшего слова “end”, заменив слово “vars” на слово “struct” (пример текста, получающегося при сохранении блока, приведен на стр. 159).

Для того, чтобы создать структуру переменных по этой строке описания, мы сначала вызовом rdsVSCreateEditor создаем объект для манипуляций с переменными, и присваиваем его идентификатор переменной dv типа RDS_HOBJECT (идентификаторы всех вспомогательных объектов РДС имеют этот тип). Затем при помощи функции rdsVSCreateByDescr мы создаем в этом объекте структуру переменных, соответствующую строке str. Логическое значение, возвращенное функцией, записывается в переменную ok: если по какой-либо причине создать структуру не удастся (например, в строке описания будет синтаксическая ошибка), функция вернет FALSE. В случае успешного создания структуры мы копируем переменные из объекта в блок, идентификатор которого передан в параметре нашей функции, при помощи сервисной функции rdsVSApplyToBlock. Эта функция тоже возвращает логическое значение, свидетельствующее об успешности копирования – не всякому блоку можно назначить любую структуру переменных. В последнем параметре rdsVSApplyToBlock можно передать указатель на целую переменную, в которую функция запишет код ошибки, но нам это не нужно, поэтому мы передаем в нем NULL.

После того, как структура переменных установлена (или, в случае ошибок, ее установка не удалась), мы удаляем объект dv и возвращаем значение ok: если какая-то из вызванных сервисных функций вернула FALSE, ok тоже будет содержать FALSE.

Теперь нужно вставить вызов этой функции в модель блока TestSub в реакцию на вызов в режиме RDS_BFM_VARCHECK. Может возникнуть соблазн написать такую реакцию:

```
// Проверка типа переменных
case RDS_BFM_VARCHECK: // ОШИБКА!
  return CreateVarStruct_x1x2y(BlockData->Block)?
                  RDS_BFR_DONE:RDS_BFR_BADVARSMSG;
```

Однако, ни к чему хорошему такая запись не приведет. Рассмотрим работу такой модели подробнее. Допустим, модель TestSub подключается к только что созданному блоку. После инициализации она будет вызвана в режиме RDS_BFM_VARCHECK, что приведет к вызову функции CreateVarStruct_x1x2y, которая изменит структуру переменных блока функцией rdsVSApplyToBlock. Функция rdsVSApplyToBlock опять вызовет модель в режиме RDS_BFM_VARCHECK для проверки правильности новой структуры, что опять приведет к новому вызову rdsVSApplyToBlock и новому вызову модели в режиме RDS_BFM_VARCHECK, и т.д. – эта последовательность вызовов будет продолжаться, пока не переполнится стек, после чего РДС завершится с сообщением об ошибке.

Чтобы не допустить возникновения этого бесконечного цикла, изменять структуру переменных нужно только тогда, когда текущая структура не подходит модели. В этом случае второй вызов модели в режиме RDS_BFM_VARCHECK не приведет к новому вызову rdsVSApplyToBlock, поскольку структура переменных в этот момент уже правильная. Таким образом, цикл не возникнет, и реакция модели благополучно завершится.

Изменим реакцию модели, вернув в нее удаленную ранее проверку строки типа переменных:

```
// Проверка типа переменных
case RDS_BFM_VARCHECK:
  if(strcmp((char*)ExtParam,"{SSDDD}")==0)
    return RDS_BFR_DONE; // Тип правильный
```

```
        // Тип неверный – меняем структуру (ОШИБКА!)
        return CreateVarStruct_x1x2y(BlockData->Block)?
                        RDS_BFR_DONE:RDS_BFR_BADVARSMSG;
```

Однако, тут нас подстерегает еще одна опасность: что произойдет, если, по каким-либо причинам, функции `rdsVSApplyToBlock` не удастся установить нужную нам структуру переменных, или если строка описания переменных, используемая в функции `CreateVarStruct_x1x2y`, не будет соответствовать строке типа "{SSDDD}" из-за ошибки в исходном тексте программы? В этом случае проверка строки типа функцией `strcmp`, которую мы вернули в модель, укажет на несовпадение, и функция `CreateVarStruct_x1x2y` будет вызвана снова, что, как и в предыдущем варианте, приведет к бесконечной рекурсии. Чтобы избежать этого, нам нужно каким-то образом отличать вызовы модели в режиме `RDS_BFM_VARCHECK` из-за ее собственной попытки изменить структуру переменных блока от всех прочих ее вызовов в том же режиме. Если модель вызвана из-за того, что она сама меняет структуру переменных блока, и при этом структура переменных оказывается неверной, модель не должна снова пытаться изменить структуру – она только что уже попробовала, и это не дало результата. В этом случае модель должна просто завершиться, вернув константу `RDS_BFR_ERROR`, сигнализирующую об ошибке. В данном случае логично использовать `RDS_BFR_ERROR` вместо `RDS_BFR_BADVARSMSG`, поскольку нам не нужно выводить сообщение о неверной структуре переменных: нам нужно просто сообщить вызвавшей модель функции `rdsVSApplyToBlock` о том, что структура неправильная.

Чтобы отличить вызов модели при программном изменении структуры переменных от других вызовов, нужно использовать какой-либо флаг: перед вызовом функции `CreateVarStruct_x1x2y` мы будем взводить его, после возвращения из нее – сбрасывать. Таким образом, если при вызове модели в режиме `RDS_BFM_VARCHECK` флаг взведен, значит, мы попали сюда из-за программного изменения переменных, если сброшен – по какой-то другой причине. В качестве флага можно использовать целое поле `Tag` структуры данных блока `RDS_BLOCKDATA` (эта структура описывалась на стр. 36). Это поле никак не используется РДС и предназначено для хранения пользовательских данных. Конечно, нам придется добавить в модель инициализацию этого поля нулем при вызове модели в режиме `RDS_BFM_INIT`, поскольку РДС его инициализировать не будет, а в исходном состоянии наш флаг должен быть сброшен. Таким образом, новая модель блока будет выглядеть так (изменения выделены жирным):

```
extern "C" __declspec(dllexport)
  int RDSCALL TestSub(int CallMode,
                      RDS_PBLOCKDATA BlockData,
                      LPVOID ExtParam)
{ BOOL ok;
// Макроопределения для статических переменных
#define pStart  ((char *)(BlockData->VarData))
#define Start   (*((char *)(pStart)))
#define Ready   (*((char *)(pStart+1)))
#define x1      (*((double *)(pStart+2)))
#define x2      (*((double *)(pStart+10)))
#define y       (*((double *)(pStart+18)))
  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        BlockData->Tag=0; // Сброс флага
        break;

      // Проверка типа переменных
      case RDS_BFM_VARCHECK:
```

```
            if(strcmp((char*)ExtParam,"{SSDDD}")==0)
              return RDS_BFR_DONE; // Тип переменных правильный
            // Тип переменных неправильный
            if(BlockData->Tag) // Флаг взведен - программное изменение
              return RDS_BFR_ERROR; // Ошибка
            // Модель вызвана не из-за программного изменения
            // структуры переменных
            // Взводим флаг на время программного изменения структуры
            BlockData->Tag=1;
            // Пытаемся изменить структуру переменных блока
            ok=CreateVarStruct_x1x2y(BlockData->Block);
            // Сбрасываем флаг обратно
            BlockData->Tag=0;
            // Возвращаем результат попытки изменения
            return ok?RDS_BFR_DONE:RDS_BFR_BADVARSMSG;

          // Выполнение такта расчета
          case RDS_BFM_MODEL:
            // Вычисление значения выхода
            if(x1==DoubleErrorValue || x2==DoubleErrorValue)
              y=DoubleErrorValue;
            else
              y=x1-x2;
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef y
    #undef x2
    #undef x1
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=========================================
```

Эта модель защищена от бесконечной рекурсии, она будет менять структуру переменных блока только в том случае, если эта структура не соответствует строке типа “{SSDDD}”. Кроме решения технической проблемы с рекурсией, эта проверка позволяет пользователю переименовывать переменные блока, как ему вздумается – строка типа при этом не изменится, и модель не будет пытаться вернуть блоку жестко прописанную в функции `CreateVarStruct_x1x2y` структуру переменных с именами `x1`, `x2` и `y`.

В рассмотренном примере модель устанавливала в блоке жесткую, заранее заданную структуру переменных. Иногда блоку необходимо менять свою структуру переменных в зависимости от каких-либо других параметров, или даже из-за действий пользователя. Ранее (стр. 449) мы создали блоки, способные передавать по сети вещественные числа и массивы вещественных чисел. Передача массивов позволяет снизить нагрузку на сеть, однако для пользователя это может оказаться не слишком удобным: ему нужно помнить, что передается в каждом элементе массива. Например, если в схеме на одной машине он подаст на пятый элемент `X[4]` входного массива передающего блока значение скорости какого-либо объекта, на другой машине он будет получать его на выходе `Y[4]` принимающего блока. Было бы гораздо удобнее, если бы пользователь мог назвать этот вход и соответствующий ему выход “Speed”, тогда у него не возникло бы вопросов, с какого выхода принимающего блока снимать сигнал.

Сделать это достаточно просто: нужно создать блок, который будет передавать по сети всю свою структуру переменных, причем этот блок должен уметь делать свои переменные входами, если он передает данные, и выходами, если принимает. В настройках блока необходимо предусмотреть отдельный интерфейс, чтобы пользователь мог задавать имена и типы переменных блока. Конечно, можно воспользоваться встроенным в РДС редактором переменных, который можно вызвать из окна параметров любого простого блока, однако, в этом случае пользователь сможет указывать, какая переменная будет входом, а какая – выходом. Поскольку блок будет сам определять роли своих переменных, лучше не давать пользователю вмешиваться в этот процесс.

Для создания нового блока мы в очередной раз изменим класс `TNetSendRcvData` (см. стр. 449), добавив в него новое поле и пару новых функций (выделены жирным):

```
// Личная область данных блоков приема и передачи по сети
class TNetSendRcvData
{ public:
    int Mode;    // Режим данного блока: прием или передача
      #define NETSRMODE_SENDER        0     // Передатчик
      #define NETSRMODE_RECEIVER      1     // Приемник
    char *ChannelName;     // Имя канала
    BOOL LimitSpeed;  // Задан минимальный интервал передачи
    DWORD Delay;      // Минимальный интервал в мс
    BOOL StructInOut; // Этот блок передает или принимает структуру

    int ConnId; // Идентификатор соединения

    // Переменные состояния блока-передатчика
    BOOL Connected;   // Соединение установлено
    BOOL DataWaiting; // Передача данных отложена
    RDS_TIMERID Timer;// Таймер для отсчета интервала
    BOOL WaitingForTimer;// Таймер запущен - ждем
    DWORD LastSendTime; // Время последней отправки

    // Функции класса
    void Connect(void);    // Установить соединение
    void Disconnect(void); // Разорвать соединение

    void SendValue(double value);     // Передать число в канал
    BOOL ReceiveValue(RDS_NETRECEIVEDDATA *rcv,  // Реакция на
                      double *pOut);       // пришедшие данные
    void SendArray(void *input); // Передать массив в канал
    BOOL ReceiveArray(RDS_NETRECEIVEDDATA *rcv,  // Реакция на
                      void *output);       // пришедшие данные
    void SendStruct(RDS_BHANDLE Block); // Передать структуру
    BOOL ReceiveStruct(RDS_NETRECEIVEDDATA *rcv, // Реакция на
                       RDS_BHANDLE Block);// пришедшую структуру

    void CreateTimer(void);      // Создать таймер
    void DeleteTimer(void);      // Удалить таймер
    BOOL CheckSendTimer(void);   // Проверить, можно ли передавать,
                                 // и запустить таймер, если нельзя

    int Setup(char *title);      // Функция настройки блока
    void SaveText(void);         // Сохранить параметры
    void LoadText(char *text);   // Загрузить параметры
```

```
    // Конструктор класса
    TNetSendRcvData(int mode)
      { ConnId=-1;    // Нет соединения
        Connected=DataWaiting=FALSE;
        LimitSpeed=WaitingForTimer=FALSE;Timer=NULL;Delay=100;
        ChannelName=NULL; StructInOut=FALSE;
        Mode=mode;};  // Режим передается в параметре конструктора
    // Деструктор класса
    ~TNetSendRcvData()
      { Disconnect();           // Разорвать соединение
        DeleteTimer();          // Удалить таймер
        rdsFree(ChannelName);   // Освободить строку имени канала
      };
};
//=======================================
```

Логическое поле `StructInOut` будет иметь значение `TRUE` только в том случае, если эта личная область данных принадлежит блоку, который передает или принимает по сети всю свою структуру переменных. В конструкторе класса этому полю присваивается `FALSE`, поэтому все написанные ранее модели, использующие этот класс, будут работать по-старому. Даже если мы изменим некоторые функции класса, введя в них какие-либо действия при истинном `StructInOut`, в этих блоках значение этого поля будет ложным, и новые действия выполняться не будут.

Для передачи и приема структуры мы включили в класс две новых функции: `SendStruct` и `ReceiveStruct`. При наличии в блоке сложных переменных (матриц, строк и т.п.) в структуре будут содержаться указатели на другие области памяти, поэтому, в общем случае, структура переменных блока не является набором последовательных байтов. Для того, чтобы можно было передать ее по сети единым блоком, воспользуемся сервисной функцией `rdsBlockVarToMem`, которая записывает все дерево переменных блока в последовательный динамически отведенный буфер и возвращает указатель на него и его длину. Кроме того, эта функция может добавить к этому буферу строку типа переменных блока, что пригодится нам при приеме: успешно принять структуру переменных мы сможем только в том случае, если их типы в точности соответствуют типам переменных блока-приемника, поэтому, приняв из канала двоичные данные, блок-приемник должен будет проверить, соответствуют ли они его структуре переменных.

Начнем с функции `SendStruct` – в ее параметре мы будем передавать идентификатор блока, переменные которого нужно передать по сети:

```
// Передать структуру переменных блока
void TNetSendRcvData::SendStruct(RDS_BHANDLE Block)
{ void *buf;
  int len;

  // Является ли данный блок передатчиком?
  if(Mode!=NETSRMODE_SENDER)
    return; // Не является - это ошибка

  if(!Connected)      // Нет связи с сервером
    { // Взводим флаг наличия данных, ожидающих передачи
      DataWaiting=TRUE;
      return;
    }
  // Связь с сервером есть
```

```
  // Формируем буфер со всеми переменными блока
  buf=rdsBlockVarToMem(Block,-1,TRUE,&len);
  if(buf==NULL) return; // Не удалось сформировать буфер

  // Передаем по сети сформированных буфер
  rdsNetBroadcastData(ConnId,
                      RDS_NETSEND_UPDATE|RDS_NETSEND_UDP,
                      0,NULL,
                      buf,len);
  // Освобождаем буфер - он уже передан
  rdsFree(buf);
  // Сбрасываем флаг ожидания - мы только что передали данные
  DataWaiting=FALSE;
  // Запоминаем время последней передачи
  LastSendTime=GetTickCount();
}
//=======================================
```

Эта функция очень похожа на функции `SendValue` и `SendArray`, которые уже есть в этом классе. У нее только два отличия: во-первых, в самом ее начале значение поля `Mode` сравнивается с константой `NETSRMODE_SENDER`, чтобы проверить, является ли блок, для которого вызвана функция, передатчиком. Раньше у нас были отдельные модели блоков для приемника и передатчика, теперь же мы делаем универсальный блок, который будет принимать и передавать данные в зависимости от настроек. Если блок используется как приемник, важно не дать ему передать данные, если, например, пользователь зачем-то подаст единицу на его сигнальный вход `Start`, из-за чего модель блока вызовется в такте расчета (до сих пор мы передавали данные именно в такте расчета, при срабатывании какой-либо подключенной ко входу связи). Проще всего заблокировать передачу непосредственно внутри функции `SendStruct`, прервав ее выполнение, если, согласно настройкам, блок не является передатчиком.

Во-вторых, данные для передачи теперь подготавливаются во вспомогательном буфере при помощи функции `rdsBlockVarToMem`. В первом ее параметре передается идентификатор блока, переменные которого нужно записать в буфер, во втором – номер переменной в блоке (значение –1 указывает на запись всех переменных блока как единой структуры). Значение `TRUE` в третьем параметре заставит функцию записать в буфер не только значения переменных, но и строку типа, чтобы блок-приемник смог проверить совместимость принятых данных со своей структурой переменных. Наконец, в последнем параметре передается указатель на целую переменную, в которую функция запишет длину получившегося буфера. После того, как содержимое буфера будет передано в канал функцией `rdsNetBroadcastData`, он будет освобожден функцией `rdsFree`.

Функция приема структуры `ReceiveStruct` будет выглядеть так:

```
// Принять структуру переменных
BOOL TNetSendRcvData::ReceiveStruct(RDS_NETRECEIVEDDATA *rcv,
                                    RDS_BHANDLE Block)
{
  // Проверяем первый параметр и является ли блок приемником
  if(rcv==NULL || Mode!=NETSRMODE_RECEIVER)
    return FALSE;
  // Есть ли среди принятых данных двоичные?
  if(rcv->Buffer==NULL)
    return FALSE;
  // Пытаемся записать принятые данные в структуру переменных блока
  return rdsBlockVarFromMem(Block,-1,rcv->Buffer,rcv->BufferSize);
}
//=======================================
```

В этой функции тоже сначала проверяется, может ли этот блок принимать данные (поле Mode должно содержать константу NETSRMODE_RECEIVER). Если это не так, значит, объект класса принадлежит блоку-передатчику, и принятые данные нужно игнорировать. Затем функция обрабатывает принятые двоичные данные, если они есть, функцией rdsBlockVarFromMem. Эта функция – обратная к rdsBlockVarToMem, она переписывает данные из буфера в памяти в структуру переменных блока согласно ее формату. Если в этом буфере, кроме значений переменных, записана и строка типа, функция сама проверит соответствие этой строки типам переменных блока и вернет FALSE, если они не совпадут. В первом параметре функции передается идентификатор блока, в переменные которого будут записаны значения, во втором – номер переменной в блоке (мы передаем –1, чтобы считалась вся структура переменных), в третьем и четвертом – указатель на буфер и его размер соответственно.

Теперь, в очередной раз, нам нужно переписать функцию настройки блока Setup. В нее необходимо внести возможности переключения блока на прием или передачу и задания структуры переменных блока пользователем. Причем обе этих возможности должны присутствовать только в настройках блока, принимающего и передающего структуры – в настройках старых блоков, которые используют этот же класс, их быть не должно.

Для вызова редактора переменных из окна настроек мы будем использовать специальную кнопку, а для реакции на нажатие этой кнопки нам придется использовать функцию обратного вызова с расширенными возможностями (см. §2.7.3). Сначала мы напишем прототип этой функции, ее тело напишем позже, после самой функции Setup:

```
// Функция обратного вызова окна настройки блока
void RDSCALL TNetSendRcvData_Setup_Check(
     RDS_HOBJECT window,              // Объект-окно
     RDS_PFORMSERVFUNCDATA data);     // Описание события
```

В самой функции настройки мы не имеем права менять структуру переменных блока до тех пор, пока пользователь не нажмет кнопку “OK”. Значит, нам нужно куда-нибудь скопировать текущую структуру переменных, и дать пользователю редактировать эту копию. Эта копия будет переписана в блок только при нажатии “OK”. Чтобы не создавать никаких дополнительных объектов, указатели на которые достаточно затруднительно передавать в функцию обратного вызова, вызываемую при нажатии на кнопку редактирования переменных, воспользуемся специальным невидимым полем ввода объекта-окна: RDS_FORMCTRL_NONVISUAL. Это поле ввода предназначено для временного хранения параметров, для которых не предусмотрен пользовательский интерфейс, к его содержимому легко получить доступ как из функции настройки, так и из функции обратного вызова. Перед открытием окна мы запишем в это поле текст, описывающий все переменные блока. При нажатии кнопки редактирования переменных в функции обратного вызова мы будем восстанавливать переменные по этому тексту и предоставлять их пользователю для редактирования. Когда он закроет редактор, мы снова будем записывать в это поле текстовое описание отредактированных переменных. Наконец, когда пользователь нажмет “OK”, мы снова восстановим переменные по тексту и запишем их в блок.

Таким образом, в функцию Setup нужно внести следующие дополнения:

```
// Функция настройки блока
int TNetSendRcvData::Setup(char *title)
{ RDS_HOBJECT win;    // Вспомогательный объект-окно
  BOOL ok;            // Пользователь нажал "OK"
  char *str;

  // Создаем окно
  win=rdsFORMCreate(FALSE,-1,-1,title);
```

```
if(StructInOut) // Блок работает со структурами
  { // Выпадающий список "прием/передача"
    rdsFORMAddEdit(win,0,10,RDS_FORMCTRL_COMBOLIST,
              "Действие:",150);
    rdsSetObjectStr(win,10,RDS_FORMVAL_LIST,
              "Передача данных\nПрием данных");
    rdsSetObjectInt(win,10,RDS_FORMVAL_VALUE,
              (Mode==NETSRMODE_RECEIVER)?1:0);

    // Получение строки описания переменных блока
    str=rdsCreateVarDescriptionString(
              rdsGetBlockVar(NULL,-1,NULL),TRUE,0,NULL);
    // Установка этой строки как невидимого поля ввода 1000
    rdsFORMAddEdit(win,0,1000,RDS_FORMCTRL_NONVISUAL,NULL,0);
    rdsSetObjectStr(win,1000,RDS_FORMVAL_VALUE,str);
    rdsFree(str); // Строка больше не нужна

    // Кнопка редактирования переменных
    rdsFORMAddEdit(win,0,11,
              RDS_FORMCTRL_BUTTON | RDS_FORMFLAG_LINE,
              "Переменные:",150);
    rdsSetObjectStr(win,11,RDS_FORMVAL_VALUE,"Изменить...");
  }

// Поле ввода имени канала
rdsFORMAddEdit(win,0,1,RDS_FORMCTRL_EDIT,
               "Имя канала:",200);
rdsSetObjectStr(win,1,RDS_FORMVAL_VALUE,ChannelName);

if(Mode==NETSRMODE_SENDER || StructInOut)
  { // Для передатчика - ввод интервала
    rdsFORMAddEdit(win,0,2,
              RDS_FORMCTRL_EDIT | RDS_FORMFLAG_CHECK,
              "Интервал передачи, мс:",80);
    rdsSetObjectInt(win,2,RDS_FORMVAL_VALUE,Delay);
    rdsSetObjectInt(win,2,RDS_FORMVAL_CHECK,LimitSpeed);
  }

// Открытие окна
ok=rdsFORMShowModalServ(win,TNetSendRcvData_Setup_Check);
if(ok)
  { // Пользователь нажал OK - запись параметров в блок
    char *NewName=rdsGetObjectStr(win,1,RDS_FORMVAL_VALUE);
    if(ChannelName==NULL || strcmp(NewName,ChannelName)!=0)
      { // Имя канала изменилось - запоминаем новое
        rdsFree(ChannelName);
        ChannelName=rdsDynStrCopy(NewName);
      }
    // Флаг ограничения интервала и сам интервал
    LimitSpeed=rdsGetObjectInt(win,2,RDS_FORMVAL_CHECK)!=0;
    Delay=rdsGetObjectInt(win,2,RDS_FORMVAL_VALUE);

    if(StructInOut) // Блок работает со структурами
      { RDS_HOBJECT dv; // Объект для работы с переменными
        DWORD flags,mask;
        RDS_VARDESCRIPTION vdescr;
```

```
            // Прием или передача
            Mode=rdsGetObjectInt(win,10,RDS_FORMVAL_VALUE)?
                              NETSRMODE_RECEIVER:NETSRMODE_SENDER;
            // Чтение строки описания переменных из невидимого поля
            str=rdsGetObjectStr(win,1000,RDS_FORMVAL_VALUE);
            // Создаем объект для работы с переменными
            dv=rdsVSCreateEditor();
            // Создаем структуру переменных по строке описания
            if(rdsVSCreateByDescr(dv,str))
              { // Установка флагов переменных - входы или выходы
                if(Mode==NETSRMODE_SENDER) // Передатчик (входы)
                  flags=RDS_VARFLAG_INPUT|RDS_VARFLAG_RUN|
                        RDS_VARFLAG_MENU|RDS_VARFLAG_SHOWNAME;
                else // Приемник (выходы)
                  flags=RDS_VARFLAG_OUTPUT|RDS_VARFLAG_MENU|
                        RDS_VARFLAG_SHOWNAME;
                // Маска изменяемых флагов
                mask=RDS_VARFLAG_INPUT|RDS_VARFLAG_OUTPUT|
                        RDS_VARFLAG_RUN|RDS_VARFLAG_MENU|
                        RDS_VARFLAG_SHOWNAME;
                // Получение числа переменных в объекте dv
                vdescr.servSize=sizeof(vdescr);
                rdsVSGetVarDescription(dv,-1,&vdescr);
                // Установка флагов начиная со второй переменной
                for(int i=2;i<vdescr.StructFields;i++)
                  rdsVSSetVarFlags(dv,i,flags,mask);
                // Запись переменных из объекта в блок
                if(!rdsVSApplyToBlock(dv,NULL,NULL))
                  rdsMessageBox("Невозможно установить переменные "
                        "блока","Ошибка",MB_OK | MB_ICONWARNING);
              }
            // Удаление вспомогательного объекта
            rdsDeleteObject(dv);
          }

        Disconnect();
        Connect();
      }
    // Уничтожение окна
    rdsDeleteObject(win);
    // Возвращаемое значение
    return ok?RDS_BFR_MODIFIED:RDS_BFR_DONE;
  }
  //=======================================
```

Дополнительные поля вводятся в окно настроек, только если `StructInOut` имеет значение `TRUE`, то есть только для новых блоков. Прежде всего, вводится выпадающий список (`RDS_FORMCTRL_COMBOLIST`) с идентификатором 10. В нем доступно для выбора всего два варианта: “Передача данных” и “Прием данных”. Текущий вариант устанавливается по значению поля класса `Mode`. Затем вызывается сервисная функция `rdsCreateVarDescriptionString`, которая формирует в динамически отведенной области памяти текст с описанием переменных блока, аналогичный тексту в функции `CreateVarStruct_x1x2y`, рассмотренной в предыдущем примере (стр. 476). Эта функция принимает четыре параметра:

```
LPSTR RDSCALL rdsCreateVarDescriptionString(
      RDS_VHANDLE Var,      // Идентификатор переменной
```

```
        BOOL StructFields,    // Раскрывать поля структуры
        int Indent,           // Число пробелов перед каждой строкой
        int *pLength);        // Возвращаемая длина строки
```

В первом параметре передается идентификатор переменной, текстовое описание которой нужно получить. Мы передаем в нем идентификатор всей структуры переменных блока, возвращаемый функцией `rdsGetBlockVar` (ее мы рассмотрим ниже). Второй параметр управляет описанием структур: при передаче `TRUE` каждое поле структуры будет описано на отдельной строке текста, при передаче `FALSE` описание структуры будет состоять из одной строки с указанием имени типа этой структуры. Мы описываем структуру переменных блока, которая не имеет имени типа, поэтому во втором параметре мы передаем `TRUE`. В третьем параметре указывается число пробелов, которое необходимо добавить в начале каждой строки (нам это не нужно – передаем 0), и в четвертом – указатель на целую переменную, в которую нужно записать длину сформированного текста (нам эта длина не нужна, поэтому передаем `NULL`). Функция возвращает указатель на сформированный текст, который записывается во вспомогательную переменную `str`.

Для получения идентификатора структуры переменных блока мы использовали сервисную функцию `rdsGetBlockVar`. Она принимает три параметра:

```
RDS_VHANDLE RDSCALL rdsGetBlockVar(
      RDS_BHANDLE Block,          // Идентификатор блока или NULL
      int num,                    // Номер переменной или -1
      RDS_PVARDESCRIPTION descr);// Указатель на структуру описания
```

В первом параметре функции передается идентификатор блока, переменные которого нас интересуют. Мы передаем `NULL`, что означает, что нам нужна переменная блока, модель которого выполняется в данный момент. Во втором параметре передается номер переменной в блоке – мы передаем –1, поскольку нас интересует не какая-то конкретная переменная, а вся структура переменных блока как единое целое. И, наконец, в третьем параметре можно передать указатель на структуру описания переменной, которую функция должна заполнить. Нам это описание не нужно – мы передаем в последнем параметре `NULL`.

Теперь, когда у нас есть текст, описывающий все переменные блока, мы создаем в объекте-окне `win` для его хранения невидимое поле с идентификатором 1000 и записываем в него значение переменной `str`. Теперь функция обратного вызова `TNetSendRcvData_Setup_Check`, прототип которой мы описали ранее, сможет считать этот текст, создать по нему структуру переменных и открыть отдельное окно для их редактирования, а потом записать измененный текст обратно в невидимое поле. Как она будет это делать – пока не важно, мы разберемся с этим, когда приступим к ее написанию. После того, как значение `str` записано в поле, эта динамически отведенная строка больше не нужна, и она освобождается вызовом `rdsFree`.

Кроме невидимого поля, в котором хранится текстовое описание переменных, нам нужна кнопка, которая будет вызывать их редактор. Кнопка – это поле ввода типа `RDS_FORMCTRL_BUTTON`. У этого поля нет значения как такового: функции установки значения задают надпись на кнопке. Нажатие кнопки передается в функцию обратного вызова, а она уже выполняет соответствующие действия. Созданная нами кнопка будет иметь идентификатор 11 и, поскольку она создана с флагом `RDS_FORMFLAG_LINE`, от остальной части окна она будет отделяться горизонтальной линией (рис. 115).

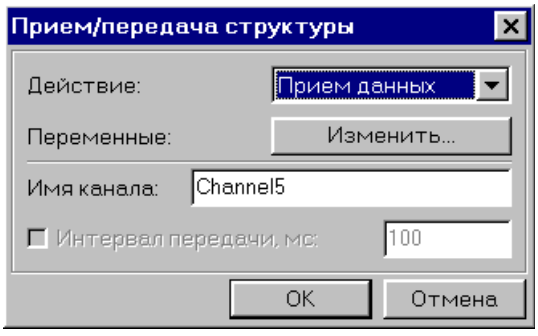


Рис. 115. Окно настроек блока приема/передачи структуры

Все остальные поля ввода мы оставили без изменения: как и раньше, в окне будет поля ввода для имени канала и интервала передачи. Единственное изменение, внесенное в эту часть, связано с тем, что раньше блок у нас был или приемником, или передатчиком, поэтому поле ввода интервала передачи создавалось только для передатчика. Теперь у нас единая модель, которая будет приемником или передатчиком по желанию пользователя, поэтому при истинном `StructInOut` это поле ввода создается в любом случае, независимо от значения `Mode`. Мы будем запрещать работу этого поля в функции обратного вызова, если пользователь выберет в выпадающем списке пункт “прием данных”, и разрешать ее при выборе пункта “передача данных”, таким образом, пользователь получит визуальное подтверждение своих действий.

Поскольку для реакции на нажатие кнопки нам нужна функция обратного вызова с расширенными возможностями, окно теперь открывается не функцией `rdsFORMShowModalEx`, а функцией `rdsFORMShowModalServ`, в которую передается указатель на функцию `TNetSendRcvData_Setup_Check` (нам еще предстоит ее написать, пока мы описали только ее прототип). После закрытия окна кнопкой “OK” мы сначала, как и раньше, считываем из полей ввода новое имя канала `ChannelName`, флаг ограничения интервала передачи `LimitSpeed` и значение этого интервала `Delay`. Затем, если значение `StructInOut` истинно (то есть если этот объект принадлежит блоку, принимающему и передающему структуры), мы должны считать из окна настроек режим работы блока `Mode` и измененную пользователем структуру переменных. Режим работы считывается из выпадающего списка: если в нем выбран вариант с индексом 0 (“прием данных”), в `Mode` записывается константа `NETSRMODE_RECEIVER`, если вариант с индексом 1 (“передача данных”) – константа `NETSRMODE_SENDER`. Структура переменных (возможно, измененная пользователем) хранится в виде текстового описания в невидимом поле с идентификатором 1000. Прежде всего, мы, как обычно, получаем указатель на строку с этим текстовым описанием в памяти объекта-окна при помощи вызова `rdsGetObjectStr` и записываем его в переменную `str` (это не динамическая строка, ее не нужно будет потом освобождать). Функцией `rdsVSCreateEditor` мы создаем вспомогательный объект для работы с переменными и записываем его идентификатор в переменную `dv`. Затем мы, точно так же, как и в предыдущем примере с программным изменением структуры переменных (стр. 476), создаем в этом объекте структуру переменных по текстовому описанию при помощи функции `rdsVSCreateByDescr`. Теперь в объекте есть структура, однако ее еще рано записывать в блок: нужно сделать переменные входами, если блок будет передавать структуру по сети, и выходами, если он будет ее принимать. При этом мы не имеем права трогать первые две переменных: это обязательные сигналы `Start` и `Ready`, первый из них обязательно должен быть входом, а второй – выходом.

В зависимости от значения поля `Mode` (оно уже получило свое значение из выпадающего списка в окне) в переменную `flags` записываются флаги переменных, которые будут установлены для каждой переменной в объекте `dv`, за исключением первых двух. Если блок передает структуру (`Mode` равно `NETSRMODE_SENDER`), переменная получит сочетание флагов `RDS_VARFLAG_INPUT` (вход), `RDS_VARFLAG_RUN` (запускать модель при срабатывании связи), `RDS_VARFLAG_MENU` (имя переменной присутствует в меню соединений) и `RDS_VARFLAG_SHOWNAME` (показывать имя переменной рядом с точкой подключения связи). Если же блок будет принимать структуру, у переменной будут установлены флаги `RDS_VARFLAG_OUTPUT` (выход), `RDS_VARFLAG_MENU` и `RDS_VARFLAG_SHOWNAME`. В переменную `mask` записывается маска флагов, которые мы будем изменять у переменной, то есть объединение битовым ИЛИ всех перечисленных выше флагов (в маске должны быть взведены биты, соответствующие флагам, которые мы меняем, и обнулены биты флагов, которые мы не трогаем). Чтобы определить общее число переменных в объекте, мы вызываем функцию `rdsVSGetVarDescription`, которая

заполняет структуру vdescr описанием структуры переменных объекта (вместо номера переменной во втором параметре функции передается –1, чтобы получить описание структуры переменных как единого целого). Предварительно, как обычно, в поле servSize структуры vdescr мы записываем размер этой структуры, чтобы функция смогла проверить правильность переданного параметра. После этого вызова в поле StructFields будет содержаться общее число переменных в объекте dv. Теперь мы можем в цикле установить всем переменным объекта, начиная с третьей (то есть с индекса 2, поскольку переменные нумеруются начиная с нуля), флаги из переменной flags при помощи функции rdsVSSetVarFlags. В эту функцию передается идентификатор объекта dv, индекс переменной i, битовые флаги переменной flags и битовая маска устанавливаемых флагов mask. После того, как флаги установлены, мы копируем структуру переменных из объекта в блок при помощи уже рассматривавшейся ранее функции rdsVSApplyToBlock. Если копирование не удалось, выводится сообщение об ошибке. В самом конце объект dv уничтожается вызовом rdsDeleteObject.

В конце функции настройки у нас записаны последовательные вызовы функций Disconnect и Connect, поэтому какую-либо дополнительную реакцию на изменение режима работы блока Mode, который раньше не менялся на протяжении всего существования блока, писать не нужно. Если пользователь изменит режим работы, это будет учтено функцией Connect, которая проверяет Mode при соединении с сервером.

Теперь нужно написать функцию обратного вызова, которая будет делать две вещи: открывать окно редактора переменных при нажатии кнопки “Изменить...” и управлять разрешенностью поля ввода интервала передачи в зависимости от выбранного пользователем режима работы:

```
// Функция обратного вызова окна настройки
void RDSCALL TNetSendRcvData_Setup_Check(RDS_HOBJECT window,
                          RDS_PFORMSERVFUNCDATA data)
{ RDS_HOBJECT dv=NULL;
  char *descr;
  RDS_VARDESCRIPTION vdescr;
  BOOL sender;

  // Реагируем на событие, из-за которого вызвана функция
  switch(data->Event)
    { case RDS_FORMSERVEVENT_CLICK: // Нажатие кнопки
        if(data->CtrlId!=11) // Это не кнопка "Изменить"
          break;
        // Создаем объект для манипуляций с переменными
        dv=rdsVSCreateEditor();
        // Считываем текст описания переменных из поля 1000
        descr=rdsGetObjectStr(window,1000,RDS_FORMVAL_VALUE);
        // Заполняем объект dv описанием переменных descr
        if(!rdsVSCreateByDescr(dv,descr))
          break;
        // Удаляем из объекта две первых переменных
        rdsVSDeleteVar(dv,0); // Start
        rdsVSDeleteVar(dv,0); // Ready
        // Открываем окно редактора переменных
        if(!rdsVSExecuteEditor(dv,FALSE,RDS_HVAR_FALLNS|
                RDS_HVAR_FNOSTRUCTNAME|RDS_HVAR_FNOOFFSET,-1,
                "Переменные"))
          break; // Пользователь нажал кнопку "Отмена”
        // Добавляем сигналы Start и Ready
        // Ready
        rdsVSAddVar(dv,0,"Ready",RDS_VARTYPE_SIGNAL,NULL,
```

```
                RDS_VARFLAG_OUTPUT|RDS_VARFLAG_SHOWNAME|
                RDS_VARFLAG_EXT_CHGNAME,0,"0");
              // Start
              rdsVSAddVar(dv,0,"Start",RDS_VARTYPE_SIGNAL,NULL,
                      RDS_VARFLAG_INPUT|RDS_VARFLAG_RUN|
                      RDS_VARFLAG_SHOWNAME|RDS_VARFLAG_EXT_CHGNAME,
                      0,"0");
              // Формируем текст описания получившихся переменных
              vdescr.servSize=sizeof(vdescr);
              if(!rdsVSGetVarDescription(dv,-1,&vdescr))
                break;
              descr=rdsCreateVarDescriptionString(vdescr.Var,
                                    TRUE,0,NULL);
              if(descr)
                { // Записываем новый текст в невидимое поле
                  rdsSetObjectStr(win,1000,RDS_FORMVAL_VALUE,descr);
                  rdsFree(descr);
                }
              break;

            case RDS_FORMSERVEVENT_CHANGE: // Изменилось поле
              // Считываем режим работы блока из выпадающего списка
              sender=rdsGetObjectInt(window,10,RDS_FORMVAL_VALUE)==0;
              // sender истинно, если блок - передатчик
              // Ограничение интервала разрешено только для передатчика
              rdsSetObjectInt(window,2,RDS_FORMVAL_ENABLED,sender);
              break;
          }

        // Если в процессе работы функции был создан объект, удаляем его
        if(dv)
          rdsDeleteObject(dv);
      }
      //=======================================
```

В параметре `window` в эту функцию передается идентификатор объекта-окна, для которого она вызвана, в параметре `data` – указатель на структуру, описывающую произошедшее событие. Нас интересуют два события: нажатие кнопки и изменение поля ввода, которые анализируются в операторе `switch`.

При нажатии какой-либо кнопки поле `Event` структуры описания события принимает значение `RDS_FORMSERVEVENT_CLICK`. При этом мы проверяем идентификатор поля ввода, выдавшего сообщение: он должен быть равен 11, поскольку такой идентификатор мы дали кнопке “Изменить”. Если нажата наша кнопка, вызовом `rdsVSCreateEditor` создается уже знакомый нам объект для работы с переменными, в который записывается текстовое описание переменных блока из невидимого поля с идентификатором 1000. Теперь нужно удалить из объекта две первых переменных: это сигналы `Start` и `Ready`, мы не будем показывать их пользователю в редакторе переменных. Для этого мы два раза вызываем функцию `rdsVSDeleteVar` с нулевым номером переменной – после этого в объекте останутся только те переменные, которые можно изменять пользователю. Теперь можно открыть окно редактора переменных функцией `rdsVSExecuteEditor`. Она принимает следующие параметры:

```
BOOL RDSCALL rdsVSExecuteEditor(
      RDS_HOBJECT Object,   // Объект с переменными
      BOOL Extended,        // Редактор в обычном (FALSE) или
                            // расширенном (TRUE) режиме
      DWORD Flags,          // Флаги, управляющие редактором
```

```
        int MaxDepth,          // Максимальная вложенность матриц
        LPSTR Caption);        // Заголовок окна
```

Окно редактора может быть открыто в обычном или расширенном режиме, за это отвечает параметр `Extended`. В обычном режиме (в РДС он используется при редактировании полей структур) пользователь может задавать для каждой переменной только имя, тип и значение по умолчанию. В расширенном он также может сделать ее входом, выходом или внутренней, включить запуск модели при срабатывании связи, соединенной с входом и т.п. (в РДС этот режим используется при задании переменных простых блоков, см. рис. 9). Нам не нужен расширенный режим, поэтому мы передаем в этом параметре `FALSE`.

В параметре `Flags` передается набор битовых флагов, управляющих поведением и внешним видом редактора. Полный список этих флагов приведен в приложении А, нам же нужны три из них: `RDS_HVAR_FALLNS` (пользователь может давать переменным все типы, кроме сигналов), `RDS_HVAR_FNOSTRUCTNAME` (имя редактируемой структуры не отображается – в нашем случае его просто нет) и `RDS_HVAR_FNOOFFSET` (колонка смещения переменной от начала дерева не выводится – пользователю она, в данном случае, не нужна). Параметр `MaxDepth` определяет максимально возможную вложенность матриц: при значении 1 пользователь может задать матрицу, при значении 2 – матрицу матриц и т.д. Значение –1, передаваемое нами, указывает на максимально возможную вложенность (в РДС она равна пяти). Наконец, в параметре `Caption` передается заголовок открываемого окна редактора. При указанных параметрах функции `rdsVSExecuteEditor` окно редактора будет выглядеть как на рис. 116.

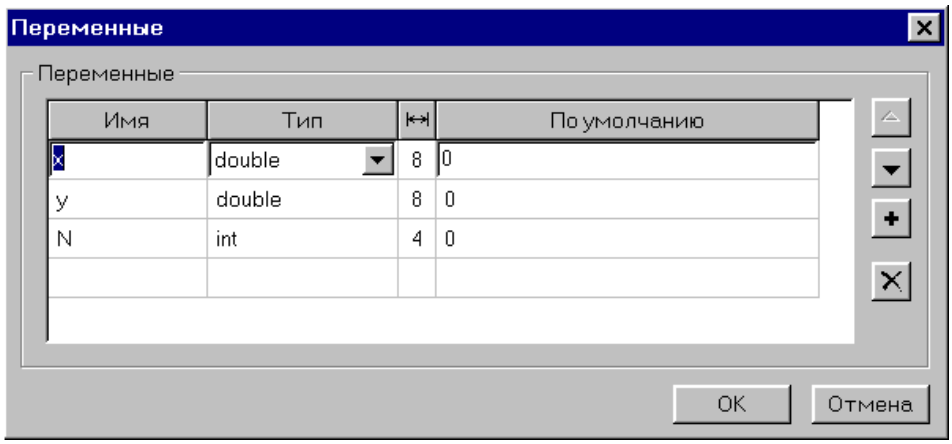


Рис. 116. Окно редактора переменных блока приема/передачи структуры

Если пользователь закроет окно редактора кнопкой “Отмена”, функция `rdsVSExecuteEditor` вернет значение `FALSE`, и выполнение оператора `switch` прервется оператором `break`. В противном случае мы добавляем в объект, переменные которого только что изменены пользователем, сигналы `Start` и `Ready`, которые мы стерли из него перед вызовом редактора. Для добавления переменных в объект мы используем функцию `rdsVSAddVar`:

```
int RDSCALL rdsVSAddVar(
     RDS_HOBJECT Object,   // Объект с переменными
     int Index,            // Номер добавляемой переменной
     LPSTR VarName,        // Имя переменной
     char BaseVarType,     // Базовый тип переменной
     LPSTR StructType,     // Имя структуры (если это структура)
     DWORD Flags,          // Флаги переменной
     int ArrayDepth,       // Вложенность матриц (если есть)
     LPSTR DefVal);        // Значение по умолчанию
```

В качестве номера переменной мы оба раза передаем 0, поэтому обе добавляемые переменные вставляются в начало списка. Сначала мы добавляем сигнал `Ready`, и он становится самой первой переменной в структуре. Затем мы добавляем `Start`, из-за чего `Ready` сдвигается на одну позицию вниз и становится вторым сигналом, а `Start` – первым, то есть оба сигнала теперь занимают правильное положение в списке. Параметр `VarName` указывает имя добавляемой переменной, а сочетание параметров `BaseVarType`, `StructType` и `ArrayDepth` – ее тип. В параметре `BaseVarType` передается символ из строки типа, соответствующий типу этой переменной или элемента матрицы, если это матрица или массив (все эти символы описаны в "RdsDef.h" как константы `RDS_VARTYPE_*`). В параметре `StructType` передается имя структуры из общего списка структур, если добавляемая переменная – структура, а в параметре `ArrayDepth` – глубина вложенности матриц, или 0, если добавляемая переменная – не матрица. Использование этих параметров может показаться несколько запутанным, поэтому приведем несколько примеров:

Тип переменной	***BaseVarType***	***StructType***	***ArrayDepth***
double	'D'	NULL	0
Матрица double	'D'	NULL	1
Логический	'L'	NULL	0
Матрица матриц double	'D'	NULL	2
Структура "Complex"	'{'	"Complex"	0
Матрица структур "Complex"	'{'	"Complex"	1

Мы добавляем одиночные сигналы, поэтому в параметре `BaseVarType` мы в обоих случаях передаем константу `RDS_VARTYPE_SIGNAL` (она имеет привычное нам значение 'S'), в параметре `StructType` – `NULL`, а в параметре `ArrayDepth` – 0.

В параметре `Flags` в функцию `rdsVSAddVar` передаются уже знакомые нам по приведенной выше функции настройки флаги переменной: `Start` должен быть входом (`RDS_VARFLAG_INPUT`), `Ready` – выходом (`RDS_VARFLAG_OUTPUT`). Эти флаги объединены с флагом `RDS_VARFLAG_SHOWNAME` (показывать имя переменной) и новым флагом `RDS_VARFLAG_EXT_CHGNAME`, который указывает на то, что добавляемую переменную нужно переименовать, если переменная с таким именем уже есть в объекте. Если среди добавленных пользователем переменных будет, например, переменная с именем "Start", добавляемый нами сигнал будет автоматически переименован в "Start1" – модели блока все равно, как называется ее сигнал запуска, главное, чтобы это была первая переменная в структуре.

После того, как в объект добавлены два обязательных для простого блока сигнала, мы заполняем описанием его переменных структуру `vdescr` – нам нужно ее поле `Var`, в котором содержится уникальный идентификатор переменной, в данном случае – структуры переменных объекта `dv` как единого целого. Этот идентификатор мы передаем в уже знакомую нам функцию `rdsCreateVarDescriptionString`, чтобы получить новый текст описания структуры переменных и записать его обратно в невидимое поле 1000. На этом реакция на нажатие кнопки "Изменить..." завершается: мы преобразовали текст из невидимого поля в набор переменных в объекте, дали пользователю отредактировать эти переменные, наложив на редактор некоторые ограничения, и записали текст описания новых переменных обратно в невидимое поле.

Вторая задача, решаемая функцией обратного вызова `TNetSendRcvData_Setup_Check` – управление разрешенностью поля ввода интервала

передачи. Это поле должно быть разрешено только в том случае, когда в выпадающем списке режима работы блока выбран пункт “передача данных”.

При открытии окна, а также при изменении пользователем любого поля ввода, в поле `Event` структуры описания события `data` будет находиться константа `RDS_FORMSERVEVENT_CHANGE`. В данном случае мы не проверяем идентификатор изменившегося поля ввода – при открытии окна он не передается, а нам в этот момент тоже нужно установить разрешенность поля интервала. Мы будем управлять этим полем при любом изменении в окне – так функция получится проще, а задержки из-за лишних вызовов здесь не важны. Сначала мы считываем значение поля ввода с идентификатором 10 (это выпадающий список режимов) и сравниваем его с нулем – если они равны, значит, блок настроен на передачу данных. Результат сравнения записывается в логическую переменную `sender` и используется для установки разрешенности поля с идентификатором 2 (это поле ввода интервала передачи) при вызове `rdsSetObjectInt` с константой `RDS_FORMVAL_ENABLED`. Если `sender` истинно, поле интервала будет разрешено, если ложно – запрещено.

Осталось внести еще пару небольших изменений в другие функции класса. Поскольку у блоков, работающих со структурами, режим работы может изменяться пользователем, нам необходимо добавить сохранение и загрузку этого параметра в функции `SaveText` и `LoadText` соответственно:

```
// Сохранение параметров блока
void TNetSendRcvData::SaveText(void)
{ RDS_HOBJECT ini;    // Вспомогательный объект
  // Создаем вспомогательный объект
  ini=rdsINICreateTextHolder(TRUE);
  // Создаем в объекте секцию "[General]"
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"General");
  // Записываем в эту секцию имя канала
  rdsINIWriteString(ini,"Channel",ChannelName);
  // Записываем режим для приемопередатчика структур
  if(StructInOut)
    rdsINIWriteInt(ini,"Mode",Mode);
  if(Mode==NETSRMODE_SENDER) // Передатчик
    { // Создаем новую секцию
      rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"Timer");
      // Записываем параметры
      rdsINIWriteInt(ini,"On",LimitSpeed);
      rdsINIWriteInt(ini,"Delay",Delay);
    }
  // Сохраняем текст, сформированный объектом, как параметры блока
  rdsCommandObject(ini,RDS_HINI_SAVEBLOCKTEXT);
  // Удаляем вспомогательный объект
  rdsDeleteObject(ini);
}
//=========================================

// Загрузка параметров блока
void TNetSendRcvData::LoadText(char *text)
{ RDS_HOBJECT ini;    // Вспомогательный объект
  char *str;
  // Создаем вспомогательный объект
  ini=rdsINICreateTextHolder(TRUE);
  // Записываем в объект полученный текст с параметрами блока
  rdsSetObjectStr(ini,RDS_HINI_SETTEXT,0,text);
  // Начинаем чтение секции "[General]", если она есть
  if(rdsINIOpenSection(ini,"General")) // Секция есть
```

```
    { // Освобождаем старое имя канала
      rdsFree(ChannelName);
      ChannelName=NULL;
      // Получаем у объекта указатель на строку с именем
      str=rdsINIReadString(ini,"Channel","",NULL);
      // Если такая строка есть в тексте, копируем ее в
      // ChannelName
      if(str)
        ChannelName=rdsDynStrCopy(str);
      // Считываем режим для приемопередатчика структур
      if(StructInOut)
        Mode=rdsINIReadInt(ini,"Mode",Mode);
    }
  if(Mode==NETSRMODE_SENDER && // Передатчик
     rdsINIOpenSection(ini,"Timer")) // Есть секция "[Timer]"
    { LimitSpeed=rdsINIReadInt(ini,"On",LimitSpeed)!=0;
      Delay=rdsINIReadInt(ini,"Delay",Delay);
    }
  // Удаляем вспомогательный объект
  rdsDeleteObject(ini);
  // Поскольку имя канала и режим работы могли измениться,
  // соединяемся с сервером заново
  Disconnect(); // Разрываем старое соединение
  Connect();    // Создаем новое
}
//=======================================
```

Параметр `Mode` в этих функциях записывается и считывается только при истинном `StructInOut`, то есть только для блоков, принимающих и передающих структуры.

Теперь мы, наконец, можем написать модель нашего нового блока, передающего и принимающего структуры:

```
// Прием/передача структуры
extern "C" __declspec(dllexport)
  int RDSCALL NetSendRcvStruct(int CallMode,
                RDS_PBLOCKDATA BlockData,
                LPVOID ExtParam)
{ TNetSendRcvData *data=(TNetSendRcvData*)(BlockData->BlockData);
// Макроопределения для первых двух переменных блока
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))
#define Ready (*((char *)(pStart+1)))
  switch(CallMode)
    { // Инициализация
      case RDS_BFM_INIT:
        BlockData->BlockData=data=
                new TNetSendRcvData(NETSRMODE_SENDER);
        // Взводим флаг работы со структурами
        data->StructInOut=TRUE;
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;

      // Установлено соединение с сервером
      case RDS_BFM_NETCONNECT:
        data->Connected=TRUE;
```

```
        // Если были данные, ожидающие передачи - передаем их
        if(data->DataWaiting)
          data->SendStruct(BlockData->Block);
        break;

      // Соединение разорвано
      case RDS_BFM_NETDISCONNECT:
        data->Connected=FALSE;
        break;

      // Запуск расчета
      case RDS_BFM_STARTCALC:
        // При первом запуске передаем данные
        if(((RDS_PSTARTSTOPDATA)ExtParam)->FirstStart)
          data->SendStruct(BlockData->Block);
        break;

      // Срабатывание таймера
      case RDS_BFM_TIMER:
        data->WaitingForTimer=FALSE;
        // Передаем данные
        data->SendStruct(BlockData->Block);
        break;

      // Такт расчета
      case RDS_BFM_MODEL:
        if(data->CheckSendTimer())
          data->SendStruct(BlockData->Block);
        break;

      // Вызов функции настройки
      case RDS_BFM_SETUP:
        return data->Setup("Прием/передача структуры");

     // Сохранение параметров
     case RDS_BFM_SAVETXT:
        data->SaveText();
        break;

     // Загрузка параметров
     case RDS_BFM_LOADTXT:
        data->LoadText((char*)ExtParam);
        break;

      // По сети получены данные
      case RDS_BFM_NETDATARECEIVED:
        // Если данные совместимы с блоком - принимаем и
        // взводим сигнал готовности
        Ready=data->ReceiveStruct((RDS_NETRECEIVEDDATA*)ExtParam,
                BlockData->Block)?1:0;
        // Сбрасываем сигнал запуска
        Start=0;
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Ready
#undef Start
```

```
#undef pStart
}
//=======================================
```

Фактически, эта модель является объединением моделей приемника и передатчика с небольшими отличиями. Прежде всего, в ней используются макроопределения только для двух обязательных сигналов `Start` и `Ready` – структура переменных этого блока произвольно задается пользователем, поэтому мы не можем ввести никаких макросов (на самом деле, в этой модели они нам и не нужны). По этой же причине в модели нет проверки типов переменных – блок будет работать с любой их структурой, и эта проверка не нужна.

При инициализации, как и в других моделях сетевых блоков, создается объект класса `TNetSendRcvData`, и указатель на него записывается в личную область данных блока. В параметре конструктора класса передается константа `NETSRMODE_SENDER`, поэтому исходно, при подключении модели, блок будет передатчиком. Позже, при загрузке параметров блока или при вызове функции настройки, его роль может измениться. После создания объекта полю `StructInOut` присваивается значение `TRUE`, чтобы функции класса знали, что этот блок передает и принимает всю свою структуру переменных. Все остальные реакции блока, кроме `RDS_BFM_NETDATARECEIVED`, отличаются от рассмотренных ранее моделей блоков-передатчиков только тем, что для передачи данных используется новая функция `SendStruct`. Эта функция сама проверит, настроен ли блок на передачу данных, и, если она по какой-либо причине будет вызвана для блока, настроенного на прием, она не выполнит никаких действий.

При получении данных по сети в реакции `RDS_BFM_NETDATARECEIVED` вызывается функция `ReceiveStruct`, которая вернет значение `TRUE`, если принятые данные совместимы со структурой переменных блока. Эта функция тоже проверяет текущие настройки блока, и, если сейчас он является передатчиком, принятые данные будут проигнорированы, и функция вернет `FALSE`. По результатам работы функции устанавливается сигнал готовности блока `Ready`: если данные успешно приняты, он будет взведен, и связи, подключенные к его выходам, сработают в ближайшем такте расчета. Сигнал `Start` при приеме данных сбрасывается – дело в том, что блок считывает из принятых данных значения всех своих переменных, включая и этот сигнал. Независимо от того, какое значение было передано по сети, запускать этот блок в следующем такте расчета не нужно (в режиме приемника блок вообще не работает в тактах расчета), поэтому этот сигнал принудительно обнуляется.

Теперь можно поместить в схемы на двух соединенных сетью машинах такие блоки, включить один из них в режим приема, а другой – в режим передачи, и задать им одинаковые структуры переменных. В режиме расчета данные, поступившие на входы блока-передатчика, будут появляться на одноименных выходах блока-приемника (рис. 117). Единственная сложность в сборке таких схем – задать в блоках на двух разных машинах одинаковую структуру переменных и ничего при этом не перепутать. Проще всего сначала создать блок в одной схеме и задать ему нужную структуру переменных, а затем сохранить его в отдельный файл, перенести на другую машину и там загрузить в схему, после чего установить одному из блоков режим передачи, а другому – режим приема.

Модель блока можно было бы улучшить, добавив в нее возможность дистанционной установки структуры переменных по сети, чтобы изменение структуры переменных в одном блоке приводило бы к автоматическому изменению переменных во всех блоках, подключенных к тому же каналу передачи данных. Желающие могут проделать это самостоятельно – все функции, необходимые для такой операции, уже описаны.

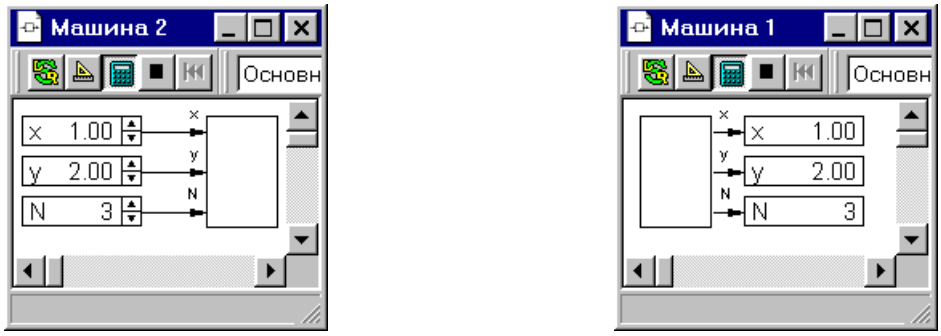


Рис. 117. Передача структуры по сети

§2.16.2. Добавление и удаление блоков и связей

Рассматриваются функции для программного добавления в схему новых блоков и связей и для их удаления. Приводится пример блока, который по двум текстовым файлам, содержащим список блоков и соединений между ними, создает в подсистеме новый фрагмент схемы.

Сервисные функции РДС позволяют моделям блоков как создавать в схеме новые блоки и связи, так и удалять их. Таким образом, можно создавать схемы, которые будут модифицировать сами себя. На практике это применяется довольно редко – в основном, для автоматического создания схем по каким-либо данным, созданным сторонними программами.

Рассмотрим следующий пример: допустим, у нас есть два текстовых файла, в одном из которых перечислены блоки с их координатами, а в другом – соединения между переменными этих блоков. Создадим модель блока, который, используя информацию из этих файлов, добавит в подсистему новые блоки и соединит их связями. Этот блок также будет уметь удалять все добавленные им в схему блоки и связи по запросу пользователя.

Сначала определимся с форматом текстовых файлов. Будем использовать формат CSV – comma separated values (“значения, разделенные запятой”). Это очень простой формат, в нем на каждой строчке текстового файла перечисляются какие-либо значения, разделенные запятыми. Он часто используется в разных программах, и в РДС есть вспомогательный объект для его автоматического разбора.

Строки файла со списком блоков будут иметь следующий вид:

имя_блока, *x*, *y*

где *x* и *y* – координаты точки привязки блока (для блоков с векторной картинкой – начало координат этой картинки, для остальных – верхний левый угол описывающего прямоугольника). Например, строчка “Block1, 10, 50” означает добавление в подсистему блока с именем “Block1” и координатами (10, 50). Здесь указывается только имя добавляемого блока, и нет никакой информации о том, что это за блок. Для простоты мы будем добавлять в схему одинаковые блоки из какого-либо файла. Имя файла с описанием блока нужно будет задавать в настройках нашего изменяющего схему блока.

В строках файла со списком связей будут указываться имена соединяемых блоков и переменных:

имя_блока_1, *имя_переменной_1*, *имя_блока_2*, *имя_переменной_2*

Например, строка “Block1, x, Block2, y” описывает связь, соединяющую переменную “x” блока “Block1” с переменной “y” блока “Block2”. Чтобы не усложнять пример, мы будем соединять блоки прямыми связями, проходящими через их геометрические центры. При этом мы будем начинать и заканчивать связь не в самом геометрическом центре, а на границе блока (рис. 118), иначе схема будет выглядеть неряшливо.

Для того, чтобы вычислить координаты точек начала и конца связи, зная координаты центров блоков и их размеры, нам придется написать вспомогательную функцию. Вычисления внутри этой функции будут достаточно простыми, тем не менее, мы рассмотрим

их подробно (те, кто считает эти вычисления элементарными, могут пропустить несколько следующих абзацев).

Допустим, (x_1, y_1) – координаты центра блока, описывающий прямоугольник которого имеет ширину w и высоту h. Линия связи будет идти от этого блока в направлении точки (x_2, y_2) – это координаты центра второго блока, но сейчас нам это не важно. Необходимо найти координаты точки (x_T, y_T), в которой отрезок (x_1, y_1)–(x_2, y_2) пересекает границу нашего блока.

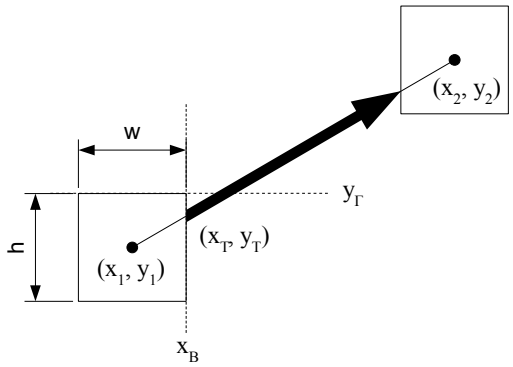


Рис. 118. Соединение блоков прямой связью

В зависимости от того, как точка (x_2, y_2) расположена относительно (x_1, y_1), отрезок пересечет одну из четырех сторон прямоугольника блока. Очевидно, что если x_2 больше x_1, то есть (x_2, y_2) находится правее (x_1, y_1), то отрезок не может пересекать левую сторону прямоугольника – он уходит вправо. Аналогично, если x_2 меньше x_1, отрезок не может пересекать правую сторону. Таким образом, сравнив x_2 и x_1, мы можем сразу выбросить из рассмотрения одну из двух вертикальных сторон и не искать пересечения с ней отрезка, соединяющего (x_1, y_1) и (x_2, y_2). Обозначим горизонтальную координату оставшейся вертикальной стороны через x_B ($x_B = x_1 \pm w/2$).

Точно так же мы можем выбросить из рассмотрения и одну из горизонтальных сторон: если отрезок уходит вверх (y_2 меньше y_1, не забываем о том, что вертикальная ось оконных координат перевернута), он не пересечет нижнюю сторону прямоугольника блока, если же он уходит вниз (y_2 больше y_1), можно не искать его пересечение с верхней стороной. Таким образом, из двух горизонтальных сторон мы будем проверять только одну – обозначим ее вертикальную координату через $y_Г$ ($y_Г = y_1 \pm h/2$).

Теперь нам нужно определить, с какой из двух оставшихся сторон прямоугольника (одной вертикальной $x=x_B$ и одной горизонтальной $y=y_Г$) пересекается наш отрезок, и найти точку этого пересечения (x_T, y_T). На рис. 118 отрезок пересекает вертикальную сторону, однако, если бы точка (x_2, y_2) находилась выше, он мог бы пересечь горизонтальную, или обе сразу, если бы прошел через угол прямоугольника. Проще всего определить пересечение отрезка с обеими прямыми $x=x_B$ и $y=y_Г$, и выбрать из двух полученных точек ближайшую к центру блока, то есть к точке (x_1, y_1) – она и будет точкой пересечения отрезка со стороной прямоугольника.

Прямая, проходящая через точки (x_1, y_1) и (x_2, y_2), описывается следующими параметрическими уравнениями:

$$\begin{cases} x = x_1 + t(x_2 - x_1) \\ y = y_1 + t(y_2 - y_1) \end{cases},$$

где t – параметр. Подстановка нулевого значения t в эти уравнения дает точку (x_1, y_1), подстановка $t=1$ – точку (x_2, y_2). Чем ближе значение t к нулю, тем ближе точка прямой будет к центру прямоугольника. Таким образом, если мы найдем точки пересечения отрезка с горизонтальной и вертикальной прямыми, нам будет нужна та из них, параметр t которой будет меньше.

Найдем параметр t_B точки пересечения отрезка (x_1, y_1)–(x_2, y_2) с вертикальной прямой $x=x_B$. В этой точке

$$x_B = x_1 + t_B(x_2 - x_1),$$

то есть

$$t_B = \frac{x_B - x_1}{x_2 - x_1}.$$

Аналогично, параметр $t_Г$ точки пересечения отрезка с горизонтальной прямой $y=y_Г$ будет равен

$$t_Г = \frac{y_Г - y_1}{y_2 - y_1}.$$

Интересующая нас точка пересечения $(x_Т, y_Т)$ будет иметь меньший из этих двух параметров:

$$t_T = min(t_B, t_Г),$$

и ее координаты будут вычисляться по уже приведенным выше уравнениям:

$$\begin{cases} x_T = x_1 + t_T(x_2 - x_1) \\ y_T = y_1 + t_T(y_2 - y_1) \end{cases}$$

Отдельно нужно рассмотреть случаи, когда $x_1=x_2$ (строго горизонтальный отрезок) и $y_1=y_2$ (строго вертикальный отрезок). В обоих этих случаях отрезок будет параллелен одной из двух прямых, и точка пересечения с ней не будет существовать (знаменатель в выражении для вычисления соответствующего t обращается в 0). При этом нам нужно просто взять в качестве $(x_Т, y_Т)$ точку пересечения с другой прямой – стороны перпендикулярны, поэтому эта точка гарантированно существует.

Наша вспомогательная функция будет выглядеть так:

```
// "Обрезание" отрезка по прямоугольнику
void ClipLineByRect(
           int x1,int y1,        // Центр прямоугольника
           int w,int h,          // Размеры прямоугольника
           int x2,int y2,        // Конец отрезка
           int *px,int *py)      // Координаты точки на границе
{ int xv,yg,xres,yres;
  double tv,tg,t;
  int status=0; // Наличие точек пересечения

  if(x2!=x1) // Отрезок не строго вертикален
    { if(x2>x1) // Отрезок уходит вправо
        xv=x1+w/2;
      else // Отрезок уходит влево
        xv=x1-w/2;
      // Параметр точки пересечения с вертикальной прямой
      tv=((double)(xv-x1))/((double)(x2-x1));
      status=1; // Есть точка пересечения с вертикалью
    }

  if(y2!=y1) // Отрезок не строго горизонтален
    { if(y2>y1) // Отрезок уходит вниз
        yg=y1+h/2;
      else // Отрезок уходит вверх
        yg=y1-h/2;
      // Параметр точки пересечения с горизонтальной прямой
      tg=((double)(yg-y1))/((double)(y2-y1));
      status+=10;     // Есть пересечение с горизонталью
    }

  switch(status)
    { case 0:   // Ошибка: (x1,y1)==(x2,y2)
        *px=x1; *py=y1; return;
```

```
        case 1:   // Строго горизонтальный отрезок
          t=tv; break;
        case 10:  // Строго вертикальный отрезок
          t=tg; break;
        default: // Диагональный отрезок
         t=(tv<tg)?tv:tg;
      }
    // Вычисление координат по параметру t
    *px=x1+t*(x2-x1);
    *py=y1+t*(y2-y1);
  }
  //=======================================
```

Названия параметров этой функции соответствуют рис. 118, через указатели `px` и `py` она возвращает координаты точки (x_T, y_T). Внутри функции выполняются описанные выше вычисления. Для отдельного рассмотрения горизонтальных и вертикальных отрезков в ней вводится вспомогательная переменная `status` с нулевым начальным значением. Если отрезок не вертикальный, переменной присваивается значение 1, если он не горизонтальный, к ней прибавляется 10. Таким образом, строго горизонтальному отрезку соответствует значение 1, строго вертикальному – 10, диагональному (ни горизонтальному, ни вертикальному) – 11. Нулевое значение `status` укажет на совпадение точек (x_1, y_1) и (x_2, y_2) – в этом случае отрезок вырождается в точку, и определить координаты (x_T, y_T) невозможно. При этом функция возвращает координаты точки (x_1, y_1) – связь будет начинаться в геометрическом центре блока, но для такого вырожденного случая это уже не важно.

Теперь займемся самим блоком. Для работы ему нужны имена трех файлов: файла со списком блоков, файла со списком связей и файла с описанием блока, копии которого мы будем вставлять по координатам из списка блоков. В качестве файла с описанием блока можно использовать любой блок, сохраненный из РДС в отдельный файл (обычно такие файлы имеют расширение “blk”). Все эти три имени мы будем хранить в значениях по умолчанию статических переменных блока (см. §2.7.4). В контекстном меню блока будет два дополнительных пункта: один для добавления в подсистему блоков и связей из файла, другой – для удаления добавленного. Чтобы блок мог удалить добавленные им объекты, мы предусмотрим в его личной области данных список блоков и связей, который будем заполнять по мере их добавления (в РДС есть специальный вспомогательный объект для хранения таких списков).

Личную область данных блока мы, как обычно, оформим в виде класса:

```
// Личная область данных блока
class TLoadGraphData
{ public:
    RDS_HOBJECT List; // Объект-список добавленных блоков и связей

    // Функция добавления блоков по списку из файла
    BOOL LoadBlocks(RDS_BHANDLE thisblock,char *blockfile,
                          char *blocklist);
    // Функция добавления связей по списку из файла
    void LoadConnections(RDS_BHANDLE thisblock,char *connlist);
    // Функция удаления добавленного
    void DeleteByList(void);

    // Функция настройки (в нее передаются номера статических
    // переменных, в которых хранятся параметры блока)
    int Setup(RDS_BHANDLE Block,int BlockFileVar,
                     int BlockLstVar,int ConnLstVar);
```

```
    // Конструктор класса
    TLoadGraphData(void){List=NULL;};
    // Деструктор класса
    ~TLoadGraphData(){rdsDeleteObject(List);};
};
//=======================================
```

В поле `List` этого класса будет храниться идентификатор вспомогательного объекта со списком добавленных блоков и связей. Модель блока будет заполнять этот список в процессе работы, и просматривать его при удалении добавленного по команде пользователя. В конструкторе класса мы присваиваем этому полю значение `NULL` (объект не создан), а в деструкторе – удаляем объект сервисной функцией `rdsDeleteObject` (ее можно безопасно вызывать и для нулевых идентификаторов).

Основные действия, выполняемые блоком, реализованы в трех функциях-членах класса. Функция `LoadBlocks` добавляет в родительскую подсистему нашего блока новые блоки по списку, функция `LoadConnections` создает между ними связи, а функция `DeleteByList` удаляет добавленные блоки и связи по команде пользователя. Для настройки параметров блока, а именно ввода трех имен файлов, в класс включена функция `Setup`, в нее будет передаваться идентификатор блока, параметры которого настраиваются, и номера статических переменных этого блока, в которых хранятся имена файлов.

В процессе чтения файлов со списками блоков и связей могут возникнуть различные ошибки: отсутствие файла с указанным именем, ссылка на несуществующий блок в списке связей и т.п. Об этих ошибках нужно сообщать пользователю, причем желательно указывать не только описание ошибки, но и место ее возникновения, то есть имя файла и номер строки в нем. Напишем для этого отдельную функцию, которая будет формировать текст сообщения об ошибке и показывать его пользователю (имя файла, номер строки и описание ошибки будут передаваться в ее параметрах):

```
// Вывод сообщения об ошибке в файле
void FileErrorMessage(int line,       // Номер строки или 0
                      char *file,     // Имя файла или NULL
                      char *message)  // Описание ошибки
{ char *msg=NULL;     // Здесь будет формироваться текст

  if(message)    // Есть текст описания
    rdsAddToDynStr(&msg,message,FALSE); // Копируем в msg
  if(file)       // Есть имя файла
    { // Добавляем к динамической строке msg
      rdsAddToDynStr(&msg,"\nФайл: ",FALSE);
      rdsAddToDynStr(&msg,file,FALSE);
    }
  if(line>0)     // Есть номер строки
    { char buf[80]; // Преобразуем в текст и добавляем к msg
      sprintf(buf,"\nСтрока: %d",line);
      rdsAddToDynStr(&msg,buf,FALSE);
    }
  // Выводим сообщение пользователю
  rdsMessageBox(msg,"Ошибка",MB_OK|MB_ICONWARNING);
  // Освобождаем динамическую строку msg
  rdsFree(msg);
}
//=======================================
```

В этой функции во вспомогательной переменной `msg` формируется динамическая строка с полным текстом сообщения об ошибке, состоящая из текста описания этой ошибки, имени файла, в котором она возникла, и номера строки в этом файле. Затем полученный текст демонстрируется пользователю при помощи функции `rdsMessageBox`, после чего строка

msg освобождается. Для постепенного добавления строк к тексту здесь используется функция rdsAddToDynStr, уже рассматривавшаяся ранее (см. стр. 256).

Теперь, прежде чем переходить в сложным функциям, напишем функцию настройки параметров Setup – она будет достаточно простой:

```
// Функция настройки параметров блока
int TLoadGraphData::Setup(
      RDS_BHANDLE Block,    // Идентификатор блока
      int BlockFileVar,     // Номер переменной с файлом блока
      int BlockLstVar,      // Номер переменной со списком блоков
      int ConnLstVar)       // Номер переменной со списком связей
{ RDS_HOBJECT window; // Идентификатор вспомогательного объекта
  BOOL ok;            // Пользователь нажал "ОК"
  char *defval;

  // Создаем окно
  window=rdsFORMCreate(FALSE,-1,-1,"Добавление блоков");

  //----- Файл с описанием блока -----
  // Значение по умолчанию переменной с номером BlockFileVar
  defval=rdsGetBlockVarDefValueStr(Block,BlockFileVar,NULL);
  // Поле ввода с возможностью выбора файла
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_OPENDIALOG,
          "Добавляемый блок:",300);
  // Фильтр имен файлов для поля
  rdsSetObjectStr(window,1,RDS_FORMVAL_LIST,
          "Файлы блоков (*.blk)|*.blk\nВсе файлы|*.*");
  // Значение поля ввода
  rdsSetObjectStr(window,1,RDS_FORMVAL_VALUE,defval);
  // Динамическая строка defval больше не нужна
  rdsFree(defval);

  // Список блоков
  defval=rdsGetBlockVarDefValueStr(Block,BlockLstVar,NULL);
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_OPENDIALOG,
          "Список блоков:",300);
  rdsSetObjectStr(window,2,RDS_FORMVAL_LIST,
          "Текстовые файлы (*.txt)|*.txt\nВсе файлы|*.*");
  rdsSetObjectStr(window,2,RDS_FORMVAL_VALUE,defval);
  rdsFree(defval);

  // Список связей
  defval=rdsGetBlockVarDefValueStr(Block,ConnLstVar,NULL);
  rdsFORMAddEdit(window,0,3,RDS_FORMCTRL_OPENDIALOG,
          "Список связей:",300);
  rdsSetObjectStr(window,3,RDS_FORMVAL_LIST,
          "Текстовые файлы (*.txt)|*.txt\nВсе файлы|*.*");
  rdsSetObjectStr(window,3,RDS_FORMVAL_VALUE,defval);
  rdsFree(defval);

  // Открытие окна
  ok=rdsFORMShowModalEx(window,NULL);
  if(ok)
    { // Запись значений в переменных блока
      defval=rdsGetObjectStr(window,1,RDS_FORMVAL_VALUE);
      rdsSetBlockVarDefValueStr(Block,BlockFileVar,defval);
      defval=rdsGetObjectStr(window,2,RDS_FORMVAL_VALUE);
      rdsSetBlockVarDefValueStr(Block,BlockLstVar,defval);
```

```
      defval=rdsGetObjectStr(window,3,RDS_FORMVAL_VALUE);
      rdsSetBlockVarDefValueStr(Block,ConnLstVar,defval);
    }
  // Уничтожение окна
  rdsDeleteObject(window);
  return ok?1:0;
}
//=======================================
```

В этой функции для ввода имен файлов используются поля с возможностью выбора файла (тип RDS_FORMCTRL_OPENDIALOG). Внешне они идентичны уже использовавшимся ранее полям RDS_FORMCTRL_SAVEDIALOG (см. стр. 384), но вместо стандартного диалога сохранения файла при нажатии на кнопку рядом с таким полем вызывается стандартный диалог открытия. Поля для выбора файлов автоматически заменяют стандартные пути принятыми в РДС сокращениями: например, в окне на рис. 119 в качестве добавляемого блока выбран стандартный блок умножения на константу (“K.blk”) с закладки “Алгебра” панели блоков, при этом вместо полного пути к папке панели блоков подставлено стандартное сокращение “$PANEL$”. Таким образом, при переносе схемы на другую машину, где папка панели блоков может размещаться в другом месте, никакой корректировки путей не потребуется. Файлы списка блоков и списка связей на рисунке не имеют путей – это значит, что они находятся в одной папке с загруженной схемой.

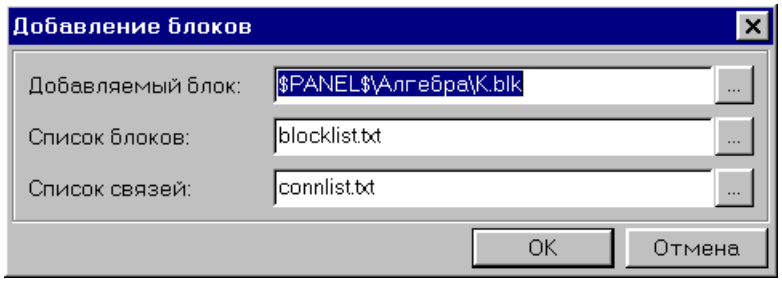


Рис. 119. Окно настроек блока создания блоков и связей

Теперь можно приступать к написанию основных функций блока. Начнем с функции создания блоков LoadBlocks, в параметрах которой передаются идентификатор нашего блока (thisblock), имя файла с описанием добавляемого блока (blockfile) и имя файла списка добавляемых блоков (blocklist). Функция будет возвращать TRUE, если в родительскую подсистему блока thisblock удалось добавить блоки из списка, и FALSE в противном случае.

```
// Создание блоков по списку
BOOL TLoadGraphData::LoadBlocks(
     RDS_BHANDLE thisblock,     // Идентификатор этого блока
     char *blockfile,           // Файл добавляемого блока
     char *blocklist)           // Файл списка блоков
{ RDS_BHANDLE block=NULL;
  RDS_HOBJECT csv;
  RDS_BLOCKDESCRIPTION descr;
  BOOL ok=TRUE;
  BOOL Modified=FALSE;     // Флаг внесения изменений в схему
  int line=0;              // Номер строки в списке блоков

  if(List) // Уже есть список объектов - очищаем его
    rdsCommandObject(List,RDS_HBCL_CLEAR);
  else // Списка нет - создаем новый (пустой)
    { List=rdsBCLCreateList(NULL,0,FALSE);
      // Включаем автоматическое обнуление удаленных
      // блоков и связей
      rdsSetObjectInt(List,RDS_HBCL_AUTODELETE,0,1);
    }
```

```
// Получаем описание нашего блока (нужны его родитель и имя)
descr.servSize=sizeof(descr);
rdsGetBlockDescription(thisblock,&descr);

// Создаем объект для разбора формата CSV
csv=rdsCSVCreate();

// Открываем файл со списком блоков для чтения в csv
rdsSetObjectStr(csv,RDS_CSV_OPENFILEREAD,0,blocklist);

// Проверяем, открылся ли файл
if(!rdsGetObjectInt(csv,RDS_CSV_FILEISOPEN,0))
  { FileErrorMessage(0,blocklist,
        "Невозможно открыть список блоков");
    rdsDeleteObject(csv);
    return FALSE;
  }

// Читаем строки файла в цикле
for(;;)
  { char *name;
    int x,y;
    // Читаем очередную строку из файла в строку объекта 0
    if(!rdsCommandObjectEx(csv,RDS_CSV_STRFROMFILE,0,NULL))
      break; // Строки в файле кончились
    line++;
    name=rdsCSVGetItem(csv,0,0); // Имя блока (элемент 0)
    if(*name==0) // Имя блока пустое
      { FileErrorMessage(line,blocklist,"Пустое имя блока");
        ok=FALSE;
        break;
      }
    // Читаем из объекта координаты блока (элементы 1 и 2)
    x=atoi(rdsCSVGetItem(csv,0,1));
    y=atoi(rdsCSVGetItem(csv,0,2));
    // Если имя блока, который нужно добавить, совпадает с именем
    // нашего блока - переименовываем наш
    if(strcmp(name,descr.BlockName)==0)
      { // Имена совпали - подбираем новое для нашего
        char *newname=rdsMakeUniqueBlockName(
                          descr.Parent,descr.BlockName);
       // Переименовываем наш блок и получаем его новое описание
        rdsRenameBlock(thisblock,newname,&descr);
        rdsFree(newname); // Освобождаем строку имени
        Modified=TRUE;
      }

    if(block==NULL) // Добавляем самый первый блок (из файла)
      block=rdsCreateBlockFromFile(blockfile,descr.Parent,
                                        x,y,NULL);
    else // Уже добавляли блок раньше - копируем добавленный
      block=rdsDuplicateBlock(block,descr.Parent,x,y,NULL);
    if(block==NULL) // Ошибка при добавлении блока
      { FileErrorMessage(line,blocklist,
                    "Не удалось добавить блок");
        ok=FALSE;
        break;
      }
```

```
        Modified=TRUE;  // Схема изменилась
        // Заносим добавленный блок в список List
        rdsBCLAddBlock(List,block,FALSE);
        // Даем добавленному блоку имя, считанное из списка
        if(!rdsRenameBlock(block,name,NULL))
          { FileErrorMessage(line,blocklist,"Повтор имени блока");
            ok=FALSE;
            break;
          }
      } // for(;;)

    // Удаляем объект csv (файл закроется автоматически)
    rdsDeleteObject(csv);
    if(Modified) // Добавлены блоки
      rdsSetModifiedFlag(TRUE);
    return ok;
  }
  //=======================================
```

Прежде всего нам нужно очистить список добавленных блоков и связей, если он уже создан (мы будем заполнять его заново) или создать его, если его еще нет. Если в поле `List` находится ненулевой идентификатор, значит, мы уже создали список, и нужно очистить его, передав ему команду `RDS_HBCL_CLEAR`. Если же значение `List` нулевое, список создается при помощи сервисной функции `rdsBCLCreateList`. Эта функция может создавать как пустой список, так и список, уже заполненный блоками и связями какой-либо подсистемы:

```
RDS_HOBJECT RDSCALL rdsBCLCreateList(
      RDS_BHANDLE sys, // Добавить блоки и связи подсистемы sys
      DWORD mask,      // Маска типов блоков и связей
      BOOL Recursive); // Добавлять из вложенных подсистем
```

Нам нужен именно пустой список, поэтому в первом параметре вместо идентификатора подсистемы мы передаем `NULL`. При этом остальные параметры, определяющие, какие именно блоки добавляются в список, могут быть любыми – в данном случае мы передаем 0 и `FALSE`.

Далее для переданного в параметрах функции идентификатора нашего блока `thisblock` вызывается функция `rdsGetBlockDescription`, которая заполняет описанием этого блока структуру `descr`. В этой структуре нас будет, прежде всего, интересовать идентификатор родительской подсистемы нашего блока – именно в нее мы будем добавлять новые блоки. Нам также понадобится имя блока: поскольку имена двух блоков в подсистеме не могут быть одинаковыми, при обнаружении такого же имени в списке блоков нам придется как-нибудь переименовать наш блок.

Теперь можно начинать читать файл со списком блоков. Мы уже решили, что этот список будет текстом в формате CSV, где первым значением в каждой строке будет имя блока, вторым – горизонтальная координата, третьим – вертикальная (см. стр. 496). Для работы с таким текстовым файлом мы будем использовать специальный объект РДС, создаваемый функцией `rdsCSVCreate`. Идентификатор этого объекта присваивается переменной `csv`. После создания мы сообщаем объекту, что файл с именем `blocklist` необходимо открыть для чтения – для объекта `csv` вызывается функция `rdsSetObjectStr` с константой `RDS_CSV_OPENFILEREAD`. Имя файла передается в объект в том виде, в котором оно записано в настройках блока, то есть у него может не быть полного пути, а стандартные пути могут быть заменены специальными сокращениями РДС (см. рис. 119). Объект самостоятельно добавит к имени файла путь к загруженной схеме, если в этом имени путь вообще не указан, и заменит сокращения на настоящие пути к стандартным папкам.

Функция rdsSetObjectStr, которая используется для передачи строк всем вспомогательным объектам РДС, не возвращает никаких значений, поэтому, вызвав ее, мы не можем быть уверены, что файл успешно открыт. Для проверки мы вызываем для объекта функцию rdsGetObjectInt с константой RDS_CSV_FILEISOPEN – если при открытии файла произошла какая-либо ошибка (например, файла с таким именем не существует), она вернет нулевое значение. В этом случае мы выводим пользователю сообщение об ошибке с указанием имени файла, используя для этого написанную ранее функцию FileErrorMessage, удаляем созданный объект, и возвращаем FALSE: добавление блоков не удалось.

Объект, создаваемый функцией rdsCSVCreate, позволяет работать с данными в формате CSV как с матрицей строк, обращаясь к каждому значению по номеру строки и номеру элемента в этой строке. В принципе, мы можем сразу считать в объект весь файл, а потом в цикле по строкам получать у объекта имя каждого блока и его координаты. Однако, поскольку мы добавляем блоки последовательно, нам нет нужды считывать в память сразу все строки файла – список блоков может оказаться достаточно длинным, и файл займет много место в памяти. Гораздо лучше считывать из файла по одной строке за раз, и, считав эту строку, тут же добавлять в подсистему новый блок. После этого считанная строка будет уже не нужна, и на ее место можно будет загрузить новую. Так мы и поступим – будем каждый раз считывать из файла одну строку и записывать ее в объект под номером 0.

Для чтения строк файла мы используем “бесконечный” цикл for(;;) – мы принудительно прервем его когда весь файл будет прочитан. В этом цикле для чтения очередной строки в объект мы вызываем для этого объекта функцию rdsCommandObjectEx, передавая ей константу RDS_CSV_STRFROMFILE и целое значение 0 (мы считываем строку из файла в нулевую строку внутренней памяти объекта). Если эта функция вернет нулевое значение, значит, строки в файле кончились, и мы прерываем цикл оператором break. В противном случае строка из файла считана успешно, и мы увеличиваем счетчик считанных строк line на единицу – номер строки нам понадобится для вывода сообщений об ошибках.

Теперь мы можем получить у объекта три значения, которые должны находиться в считанной строке. Для чтения строк из объекта используется функция rdsCSVGetItem, в которую передается идентификатор объекта (csv), номер строки (в нашем случае – 0, поскольку мы каждый раз считываем строку из файла в нулевую строку объекта) и номер элемента в строке. Функция возвращает указатель на строку, представляющую запрошенное значение и находящуюся во внутренней памяти объекта, поэтому после получения эти строки не нужно освобождать при помощи rdsFree. Однако, следует помнить, что возвращенный функцией указатель ссылается на значение только до тех пор, пока оно не изменится. После этого указатель на значение нужно получать заново. В нашем случае мы не меняем нулевую строку объекта csv до конца тела цикла, поэтому, получив указатель на значение в нулевой строке объекта, мы можем не заботиться о его правильности.

Самое первое значение (то есть значение с нулевым индексом) в считанной строке – это имя блока, мы получаем указатель на него вызовом rdsCSVGetItem(csv,0,0) и запоминаем его в переменной name. Функция rdsCSVGetItem устроена так, что она всегда возвращает указатель на строку, даже если мы попытаемся считать из объекта элемент, которого в нем нет. В случае каких-либо ошибок функция вернет указатель на пустую строку (то есть на нулевой байт), поэтому ее результат можно использовать в вызовах функций обработки строк без каких-либо дополнительных проверок. В данном случае мы проверяем полученное имя блока на пустоту: сравниваем первый символ строки (*name) с нулем. Имя блока не может быть пустым – в этом случае мы выводим сообщение об ошибке, прерываем цикл чтения файла и записываем FALSE в переменную ok – функция вернет это значение, указывая на то, что чтение списка блоков не удалось.

Если имя блока не пустое, мы считываем элементы строки с индексами 1 и 2 в целые переменные x и y, предварительно преобразовав их в целые числа функцией atoi – это координаты блока, который мы должны добавить в подсистему. Но прежде мы должны проверить, не совпадает ли имя добавляемого блока name с именем нашего блока, которое запомнено в поле BlockName структуры descr. Если они совпадают (функция strcmp возвращает 0), мы переименуем наш блок: имя блока из списка изменять нельзя (на него будут ссылки из списка связей), а как называется блок, выполняющий добавление блоков в подсистему, на самом деле, не важно.

Для того, чтобы переименовать блок мы, прежде всего, должны подобрать ему новое имя, которое не будет совпадать ни с одним из имен блоков, уже присутствующих в подсистеме. В РДС для этого предусмотрена специальная функция rdsMakeUniqueBlockName. В параметрах этой функции передается идентификатор подсистемы, внутри которой нужно подобрать уникальное имя блока, и исходное имя, на базе которого подбирается новое. Функция разбивает это исходное имя на буквенную и цифровую части, а затем меняет цифровую часть так, чтобы имя стало уникальным в подсистеме. Например, если в подсистеме есть блоки с именами “Block1”, “Block2” и “Block3”, при вызове этой функции с исходным именем “Block1” она вернет имя “Block4”. Если же вызвать ее с исходным именем “Block100”, она вернет это имя, не изменив его, поскольку оно и так не совпадает ни с одним из имен блоков. Возвращаемое функцией имя представляет собой динамически отведенную строку, поэтому после использования ее необходимо освободить при помощи rdsFree.

В данном случае мы подбираем новое имя для нашего блока на основе его же собственного старого имени. Сформированное функцией rdsMakeUniqueBlockName новое имя записывается в переменную newname и используется в вызове функции rdsRenameBlock, которая переименовывает блок и записывает его новое описание в структуру descr (нам важно все время хранить в этой структуре актуальное описание блока, поскольку она постоянно используется при проверке совпадения имен). Результат выполнения функции можно не проверять, поскольку, подобрав уникальное имя, мы гарантировали успешность переименования. После вызова rdsRenameBlock строка в переменной newname нам уже не нужна (новое имя блока теперь можно получить через структуру descr), и мы освобождаем эту динамическую строку функцией rdsFree.

Теперь можно создавать в подсистеме новый блок. Чтобы каждый раз не загружать блок из файла blockfile, мы сделаем это только один раз и запомним идентификатор загруженного блока, а затем будем просто копировать этот блок снова и снова. В начале функции мы ввели дополнительную переменную block и присвоили ей NULL – в ней мы будем хранить идентификатор загруженного блока. Если эта переменная сохранила нулевое значение, значит, мы еще не добавили в подсистему ни одного блока. В этом случае мы вызываем функцию создания блока из файла rdsCreateBlockFromFile, в которую передаем имя файла с описанием блока (blockfile), подсистему, в которой его нужно создать (descr.Parent, родительская подсистема нашего блока) и координаты, по которым он должен располагаться (x и y). В последнем параметре функции передается указатель на структуру, которая заполняется описанием загруженного блока, но нам это описание не нужно, поэтому мы передаем NULL. Функция возвращает идентификатор добавленного блока или NULL в случае ошибки, этот идентификатор записывается в переменную block.

Если же идентификатор в переменной block не равен NULL, значит, мы уже вызывали rdsCreateBlockFromFile по крайней мере один раз. В этом случае мы делаем копию блока block функцией rdsDuplicateBlock: в нее передается идентификатор копируемого блока (block), подсистема, в которую вставляется копия (descr.Parent) и координаты копии (x и y). В последнем параметре этой функции тоже можно передать

указатель на структуру, которая будет заполнена описанием созданной копии, но мы опять передаем NULL. Идентификатор созданной копии блока, который возвращает функция, мы записываем в ту же самую переменную block.

Теперь, независимо от того, каким именно образом мы создали в подсистеме новый блок, в переменной block содержится его идентификатор. Если вместо идентификатора там окажется NULL, это будет означать, что при создании блока произошла ошибка. В этом случае мы выводим пользователю сообщение об этом с указанием имени файла и номера строки в нем, заносим FALSE в ok (возникла ошибка) и прерываем цикл. Если же создание нового блока выполнено успешно, мы присваиваем переменной Modified значение TRUE (схема изменена) и заносим добавленный блок block в список List функцией rdsBCLAddBlock. В последнем параметре этой функции мы передаем FALSE, поскольку нам не нужно проверять, есть ли уже добавляемый блок в списке: при работе нашей функции не может возникнуть ситуации, в которой мы два раза добавим в список один и тот же блок.

Теперь осталось только дать блоку имя name: функции rdsCreateBlockFromFile и rdsDuplicateBlock автоматически обеспечивают создаваемый блок уникальным именем, но принудительно указать это имя в них нельзя. Для переименования блока мы снова вызываем rdsRenameBlock, но нам необходимо проверить результат ее выполнения: может оказаться, что блок с именем name уже есть в подсистеме. В этом случае мы выводим пользователю сообщение об ошибке и прерываем цикл – дальнейшее добавление блоков бессмысленно, пользователь должен поправить имена в списках блоков и связей или стереть лишние блоки в подсистеме.

На этом тело цикла чтения файла завершается. Перечисленные выше действия будут выполняться до тех пор, пока строки в файле не закончатся или пока не возникнет ошибка. После завершения цикла мы удаляем вспомогательный объект csv (открытый для чтения файл при этом автоматически закроется), взводим в РДС флаг наличия изменений в схеме, если мы добавили в нее блоки, вызовом rdsSetModifiedFlag (см. пример на стр. 307) и возвращаем значение ok. Таким образом, если в подсистему были добавлены все блоки из списка, функция вернет TRUE, если при добавлении возникли ошибки – FALSE.

Функция создания связей LoadConnections тоже будет разбирать файл формата CSV, поэтому она будет в целом похожа на LoadBlocks. В нее будет передаваться идентификатор нашего блока thisblock и имя файла со списком связей connlist:

```
// Функция создания связей по списку
void TLoadGraphData::LoadConnections(
     RDS_BHANDLE thisblock,     // Идентификатор этого блока
     char *connlist)            // Файл списка связей
{ RDS_HOBJECT csv;
  RDS_BLOCKDESCRIPTION descr;
  BOOL ok=TRUE;
  BOOL Modified=FALSE;
  RDS_HOBJECT editor=NULL;
  RDS_BLOCKDIMENSIONS dim1,dim2;
  int line=0;   // Счетчик считанных строк

  // Заполнение поля размера служебных структур
  dim1.servSize=sizeof(dim1);
  dim2.servSize=sizeof(dim2);

  // Получаем описание нашего блока (нужен его родитель)
  descr.servSize=sizeof(descr);
  rdsGetBlockDescription(thisblock,&descr);

  // Создаем объект для разбора формата CSV
  csv=rdsCSVCreate();
```

```
// Открываем файл со списком связей для чтения в csv
rdsSetObjectStr(csv,RDS_CSV_OPENFILEREAD,0,connlist);

// Проверяем, открылся ли файл
if(!rdsGetObjectInt(csv,RDS_CSV_FILEISOPEN,0))
  { FileErrorMessage(0,connlist,
              "Невозможно открыть список связей");
    rdsDeleteObject(csv);
    return;
  }

// Читаем строки файла в цикле
for(;;)
  { char *name1,*name2,*var1,*var2;
    RDS_BHANDLE block1,block2;
    int xc1,xc2,yc1,yc2,pnum1,pnum2,x1,y1,x2,y2;
    RDS_CHANDLE conn;
    // Читаем очередную строку из файла в строку объекта 0
    if(!rdsCommandObjectEx(csv,RDS_CSV_STRFROMFILE,0,NULL))
      break;
    line++;
    // Считываем из строки имена соединяемых блоков и переменных
    name1=rdsCSVGetItem(csv,0,0);   // Имя блока 1
    var1=rdsCSVGetItem(csv,0,1);    // Имя переменной 1
    name2=rdsCSVGetItem(csv,0,2);   // Имя блока 2
    var2=rdsCSVGetItem(csv,0,3);    // Имя переменной 2
    if(*name1==0 || *name2==0 || *var1==0 || *var2==0)
      { FileErrorMessage(line,connlist,
                    "Мало данных в строке связи");
        break;
      }
    // Ищем в подсистеме блоки с указанными именами
    block1=rdsGetChildBlockByName(descr.Parent,name1,NULL);
    block2=rdsGetChildBlockByName(descr.Parent,name2,NULL);
    if(block1==NULL || block2==NULL) // Какого-то нет
      { FileErrorMessage(line,connlist,
                         "Не найден один из блоков");
        break;
      }
    // Получаем координаты и размеры блоков
    rdsGetBlockDimensions(block1,&dim1,FALSE);
    rdsGetBlockDimensions(block2,&dim2,FALSE);
    // Вычисляем координаты центров блоков
    xc1=dim1.Left+dim1.Width/2;
    yc1=dim1.Top+dim1.Height/2;
    xc2=dim2.Left+dim2.Width/2;
    yc2=dim2.Top+dim2.Height/2;
    // "Обрезаем" прямую по границам первого блока
    ClipLineByRect(xc1,yc1,dim1.Width,dim1.Height,xc2,yc2,
                   &x1,&y1);
    // "Обрезаем" прямую по границам второго блока
    ClipLineByRect(xc2,yc2,dim2.Width,dim2.Height,xc1,yc1,
                   &x2,&y2);
    // Создаем или очищаем объект для редактирования связи
    if(editor==NULL) // Нужно создать
      editor=rdsCECreateEditor();
```

```
        else // Очищаем ранее созданный
          rdsCommandObject(editor,RDS_HCE_RESET);
        // Добавляем в объект editor две точки
        pnum1=rdsCEAddBlockPoint(editor,block1,var1,
                        x1-dim1.BlockX,y1-dim1.BlockY,FALSE);
        pnum2=rdsCEAddBlockPoint(editor,block2,var2,
                        x2-dim2.BlockX,y2-dim2.BlockY,FALSE);
        // Добавляем соединяющую их линию
        rdsCEAddLine(editor,pnum1,pnum2);
        // Создаем по данным объекта editor новую связь
        conn=rdsCECreateConnBus(editor,descr.Parent,
                                  RDS_CTCONNECTION,NULL);
        if(conn!=NULL) // Создание связи удалось
          { Modified=TRUE;
            // Добавляем связь в список
            rdsBCLAddConn(List,conn,FALSE);
          }
        else // Ошибка при создании связи
          { FileErrorMessage(line,connlist,
                  "Не удалось создать связь");
            break;
          }
      } // for(;;)

    // Удаляем объект csv (файл закроется автоматически)
    rdsDeleteObject(csv);
    // Удаляем объект-редактор
    rdsDeleteObject(editor);
    if(Modified) // Добавлены связи
      rdsSetModifiedFlag(TRUE);
  }
  //=======================================
```

В этой функции, как и в предыдущей, мы используем для чтения файла объект, создаваемый вызовом `rdsCSVCreate`, поэтому их начала практически совпадают. Точно так же, как и в `LoadBlocks`, мы в цикле считываем очередную строку файла в нулевую строку объекта, но после этого выполняются совершенно другие действия: теперь мы должны создать связи между блоками, которые, на данный момент, уже должны находиться в подсистеме (`LoadConnections` мы будем вызывать после того, как `LoadBlocks` успешно завершилась).

В списке связей каждая строка содержит четыре значения: две пары “имя блока, имя переменной” (см. стр. 496). Мы сразу считываем имена соединяемых блоков в `name1` и `name2`, а имена переменных в этих блоках – в `var1` и `var2`. Затем мы проверяем все эти четыре строки на пустоту: если хотя бы одна из них пуста, мы выводим сообщение об ошибке и прерываем цикл: ни имена блоков, ни имена переменных пустыми быть не могут.

Затем мы ищем идентификаторы обоих соединяемых блоков в родительской подсистеме нашего блока и присваиваем их переменным `block1` и `block2`. Для этого используется функция `rdsGetChildBlockByName`: она позволяет найти в подсистеме блок по его имени. Если вместо одного из идентификаторов функция вернет `NULL`, значит, такого блока в подсистеме нет – мы выводим сообщение об ошибке и прерываем цикл. Если же оба блока найдены, мы при помощи функции `rdsGetBlockDimensions` заполняем структуры `dim1` и `dim2` их геометрическими размерами. Размеры блоков нам нужны для вычисления координат точек создаваемой между ними связи (см. рис. 118). Затем мы вычисляем координаты геометрических центров блоков (`xc1`, `yc1`) и (`xc2`, `yc2`) – отрезок связи, который мы создаем, будет лежать на прямой, соединяющей эти точки. Чтобы

определить точки пересечения этой прямой с границами описывающего прямоугольника блока, мы вызываем написанную ранее вспомогательную функцию ClipLineByRect (см. стр. 496). В результате двух ее вызовов в переменных x1 и y1 окажутся координаты точки пересечения прямой с границами первого блока, в x2 и y2 – с границами второго. Эти точки будут началом и концом нашей связи.

Связь в РДС – сложная конструкция, она может состоять из множества точек и линий, как прямых, так и кривых, поэтому, в отличие от блока, ее нельзя создать одним вызовом. Для создания новых и изменения существующих связей в РДС предусмотрен специальный вспомогательный объект: сначала в него записываются координаты и параметры точек связей и указываются номера точек, соединенных линиями, а затем по данным этого объекта в подсистеме создается настоящая связь. Такой объект создается сервисной функцией rdsCECreateEditor. В данном случае в функции введена переменная editor с нулевым начальным значением для хранения идентификатора такого объекта-редактора связи. Если в editor находится значение NULL, значит, сейчас мы будем добавлять в подсистему самую первую связь из списка. В этом случае мы создаем новый объект вызовом rdsCECreateEditor. Если же в editor уже хранится какой-то идентификатор, значит, объект уже был создан ранее в цикле, и мы просто очищаем его, вызвав для него функцию rdsCommandObject с константой RDS_HCE_RESET.

Теперь мы должны заполнить объект editor точками связи и соединяющей их линией. Обе точки будут точками соединения связи с блоком, поэтому для их добавления мы вызываем функцию rdsCEAddBlockPoint:

```
int RDSCALL rdsCEAddBlockPoint(
      RDS_HOBJECT editor,   // Объект-редактор связи
      RDS_BHANDLE block,    // Блок, с которым связана точка
      LPSTR var,            // Имя переменной в блоке
      int x,int y,          // Координаты относительно блока
      BOOL displayname);    // Показывать ли имя переменной у точки
```

При вызове этой функции координаты точки x и y указываются не относительно верхнего левого угла рабочего поля подсистемы, как обычно, а относительно точки привязки блока, с которым эта точка соединяет связь. Координаты точек привязки обоих блоков находятся в полях BlockX и BlockY структур с их размерами dim1 и dim2, поэтому в вызове rdsCEAddBlockPoint для первой точки мы используем не ее абсолютные координаты x1 и y1, а x1-dim1.BlockX и y1-dim1.BlockY. Точно так же вычисляются относительные координаты второй точки. Функция rdsCEAddBlockPoint возвращает внутренний номер добавленной точки в объекте-редакторе, эти номера запоминаются в переменных pnum1 и pnum2, они потребуются нам при добавлении в объект линии, соединяющей эти точки.

В данном случае мы добавляем только точки соединения связи с блоком, поэтому нам достаточно функции rdsCEAddBlockPoint. Если бы мы захотели добавить в связь промежуточные точки, нам потребовалась бы функция rdsCEAddInternalPoint, точки связи с шиной – rdsCEAddBusPoint. Общий принцип добавления точек остается тем же: в параметрах функций передаются параметры точек, функции возвращают внутренние номера, под которыми эти точки добавлены в объект. Порядок добавления точек не важен – главное, при создании линий, соединяющих точки, указать правильные пары номеров этих точек.

Точки добавлены в объект, теперь нужно добавить в него соединяющую их линию. Поскольку мы решили, что связи у нас будут прямые, мы вызываем функцию rdsCEAddLine, в которую передаем внутренние номера соединяемых точек pnum1 и pnum2. Если бы мы вместо прямой линии захотели бы добавить кривую Безье, нужно было бы вызвать rdsCEAddBezier и указать, кроме номеров соединяемых точек, еще и координаты касательных кривой (см. рис. 88). Функция rdsCEAddLine (как и rdsCEAddBezier) возвращает внутренний номер добавленной линии, но, в данном случае, он нам не нужен, и мы не присваиваем его никакой переменной.

Теперь объект editor полностью описывает связь, которую нам нужно создать. Для создания по этому описанию связи в подсистеме вызывается функция rdsCECreateConnBus:

```
RDS_CHANDLE RDSCALL rdsCECreateConnBus(
      RDS_HOBJECT editor,   // Объект-редактор связи
      RDS_BHANDLE parent,   // Подсистема, в которой создается связь
      int type,             // Связь или шина (константа RDS_CT*)
      int *perror);         // Возвращаемый код ошибки
```

В первом параметре функции передается идентификатор объекта, по данным которого нужно создать связь, во втором – идентификатор подсистемы, в которой эта связь создается. Эта функция может создавать как связи, так и шины, поэтому в ее третьем параметре передается тип создаваемого объекта: RDS_CTCONNECTION для связи, RDS_CTBUS – для шины. В четвертом параметре можно передать указатель на целую переменную, в которую функция запишет код ошибки, но здесь нам это не нужно – мы передаем NULL. Функция возвращает идентификатор созданной связи или NULL при ошибке, мы записываем это значение в переменную conn. Если conn не равно NULL, значит, связь создана успешно, и мы добавляем ее в список List (он был создан в функции LoadBlocks) вызовом rdsBCLAddConn, в противном случае мы выводим сообщение пользователю и прерываем цикл.

Цикл for(;;) завершится тогда, когда в списке связей закончатся строки, или при возникновении какой-либо ошибки. После его завершения мы удаляем вспомогательные объекты csv и editor и взводим флаг наличия изменений в схеме, если в подсистему была добавлена хотя бы одна связь. После этого функция завершается.

Функция удаления добавленных объектов DeleteByList будет работать с полем класса List, в котором после вызова LoadBlocks и LoadConnections должен находиться идентификатор объекта со списком добавленных ими блоков и связей.

```
// Удаление добавленных блоков и связей
void TLoadGraphData::DeleteByList(void)
{ RDS_CHANDLE *conns;
  RDS_BHANDLE *blocks;
  int count;

  if(List==NULL) // Списка нет
    return;

  // Отключаем автоматическое обнуление удаляемых объектов
  rdsSetObjectInt(List,RDS_HBCL_AUTODELETE,0,0);

  // Получаем указатель на массив идентификаторов связей
  // и его размер
  conns=(RDS_CHANDLE*)rdsGetObjectArray(List,RDS_HBCL_CONNARRAY,
                                        0,&count);
  // Удаляем все связи из этого массива
  for(int i=0;i<count;i++)
    rdsDeleteConnection(conns[i]);

  // Получаем указатель на массив идентификаторов блоков
  // и его размер
  blocks=(RDS_BHANDLE*)rdsGetObjectArray(List,RDS_HBCL_BLOCKARRAY,
                                        0,&count);
  // Удаляем все связи из этого массива
  for(int i=0;i<count;i++)
    rdsDeleteBlock(blocks[i]);
```

```
    // Удаляем объект-список и обнуляем List
    rdsDeleteObject(List);
    List=NULL;
    // Взводим флаг измененности схемы
    rdsSetModifiedFlag(TRUE);
  }
  //=======================================
```

Прежде чем мы начнем удалять блоки и связи, нам лучше отключить в списке автоматическое обнуление удаляемых объектов. Мы включили его при создании списка в функции `LoadBlocks`, и теперь при удалении любого блока и связи, находящегося в этом списке, РДС автоматически будет заменять в этом списке его идентификатор на `NULL`. Включение этой функции гарантирует нам, что мы не попытаемся удалить блок или связь, которые уже удалены пользователем, поскольку РДС обнулит в списке их идентификаторы без нашего участия. Но теперь это обнуление нам не нужно: мы сами будем удалять все блоки и связи, находящиеся в списке, и пользователь не сможет вмешаться в этот процесс. В принципе, можно и не отключать автоматическое обнуление, просто в этом случае при удалении каждого блока или связи РДС будет искать их идентификаторы в списке, чтобы обнулить их, что приведет к замедлению работы.

Сначала мы удалим все связи. Для этого мы получим указатель на внутренний массив объекта, в котором содержатся идентификаторы связей, при помощи функции `rdsGetObjectArray`:

```
LPVOID RDSCALL rdsGetObjectArray(
      RDS_HOBJECT Object,   // Идентификаор объекта
      int ObjOp,            // Тип массива
      int OpParam,          // Дополнительный параметр
      int *pSize);          // Возвращаемый размер массива
```

Объект-список позволяет получить доступ к массиву блоков (тип `RDS_HBCL_BLOCKARRAY`) и массиву связей (тип `RDS_HBCL_CONNARRAY`), дополнительный параметр, который можно передать в функции, в этом объекте не используется. Нам нужен массив связей, поэтому мы используем константу `RDS_HBCL_CONNARRAY`, и приводим указатель, возвращенный функцией, к типу `RDS_CHANDLE*` (“массив идентификаторов типа `RDS_CHANDLE`”) и записываем его в переменную `conns`. Размер массива записывается в целую переменную `count`. Далее мы в цикле вызываем для каждого элемента массива функцию удаления связи `rdsDeleteConnection`. Эту функцию можно безопасно вызывать для нулевых идентификаторов, поэтому если в массиве встретятся элементы, обнуленные РДС из-за удаления соответствующих связей пользователем, ничего страшного не случится.

После удаления связей мы точно таким же образом удаляем все блоки, идентификаторы которых находятся в списке. Указатель на массив идентификаторов мы записываем в переменную `blocks`, а затем в цикле вызываем для каждого элемента этого массива функцию удаления блока `rdsDeleteBlock`. После этого мы удаляем список `List` (он нам больше не нужен) и взводим флаг наличия изменений в схеме функцией `rdsSetModifiedFlag`.

Все функции класса `TLoadGraphData` написаны, осталось написать модель блока, который будет их вызывать. Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Назначение*
0	Start	Сигнал	1	Вход	Стандартный сигнал запуска (здесь не используется)
1	Ready	Сигнал	1	Выход	Стандартный сигнал готовности (здесь не используется)

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Назначение*
2	BlockFile	Строка	4	Внутр.	Имя файла с описанием добавляемого блока
6	BlockList	Строка	4	Внутр.	Имя файла списка блоков
10	ConnList	Строка	4	Внутр.	Имя файла списка связей

Модель блока будет иметь следующий вид:

```
// Добавление в схему блоков и связей
extern "C" __declspec(dllexport)
  int RDSCALL LoadGraph(int CallMode,
                        RDS_PBLOCKDATA BlockData,
                        LPVOID ExtParam)
{ // Приведение указателя на личную область данных
  // к правильному типу
TLoadGraphData *data=(TLoadGraphData*)(BlockData->BlockData);
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))        // 0
#define Ready (*((char *)(pStart+1)))      // 1
#define BlockFile (*((char **)(pStart+2))) // 2
#define BlockList (*((char **)(pStart+6))) // 3
#define ConnList (*((char **)(pStart+10))) // 4
  switch(CallMode)
    { // Инициализация модели
      case RDS_BFM_INIT:
        BlockData->BlockData=new TLoadGraphData();
        break;

      // Очистка
      case RDS_BFM_CLEANUP:
        delete data;
        break;

      // Проверка типов переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSAAA}")?
                      RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Настройка параметров блока
      case RDS_BFM_SETUP:
        return data->Setup(BlockData->Block,
                           2,   // BlockFile
                           3,   // BlockList
                           4);  // ConnList

      // Вызов контекстного меню блока
      case RDS_BFM_CONTEXTPOPUP:
        rdsAdditionalContextMenuItemEx(
           "Добавить блоки и связи",0,1,0);
        rdsAdditionalContextMenuItemEx(
           "Удалить добавленное",
           data->List==NULL?RDS_MENU_DISABLED:0,2,0);
        break;

      // Выбор пункта в контекстном меню
      case RDS_BFM_MENUFUNCTION:
```

```
            switch(((RDS_PMENUFUNCDATA)ExtParam)->Function)
              { case 1:   // Добавить блоки и связи
                  // Подготовка к серьезным изменениям
                  rdsSetSystemUpdate(FALSE);
                  if(data->LoadBlocks(BlockData->Block,
                                      BlockFile,BlockList))
                    data->LoadConnections(BlockData->Block,ConnList);
                  // Серьезные изменения завершены
                  rdsSetSystemUpdate(TRUE);
                  break;
                case 2:   // Удалить добавленное
                  // Подготовка к серьезным изменениям
                  rdsSetSystemUpdate(FALSE);
                  data->DeleteByList();
                  // Серьезные изменения завершены
                  rdsSetSystemUpdate(TRUE);
                  break;
              }
            break;
        }
      return RDS_BFR_DONE;
    // Отмена макроопределений
    #undef ConnList
    #undef BlockList
    #undef BlockFile
    #undef Ready
    #undef Start
    #undef pStart
    }
    //=======================================
```

Реакции на события RDS_BFM_INIT, RDS_BFM_CLEANUP и RDS_BFM_VARCHECK в этой модели похожи на все ранее рассмотренные реакции моделей, в которых личная область данных представляет собой объект какого-либо класса. В реакции на событие RDS_BFM_SETUP вызывается функция класса Setup, в которую передаются номера переменных, в которых хранятся параметры блока. Остальные реакции нужно рассмотреть подробнее.

При открытии контекстного меню блока (реакция RDS_BFM_CONTEXTPOPUP) блок добавляет в него два пункта: “Добавить блоки и связи” с идентификатором 1 и “Удалить добавленное” с идентификатором 2, причем последний пункт будет разрешенным только в том случае, если поле List класса личной области данных блока не будет равно NULL, то есть если есть список добавленных блоков и связей.

При выборе пользователем одного из добавленных пунктов меню модель вызовется в режиме RDS_BFM_MENUFUNCTION. Если идентификатор выбранного пункта – 1, нужно добавить в подсистему блоки и связи из файлов, указанных в настроечных параметрах блока. Для этого нужно последовательно вызвать функции LoadBlocks и LoadConnections класса личной области данных. Однако, перед их вызовом лучше всего проинформировать РДС о том, что сейчас модель будет производить изменения в схеме. Дело в том, что добавление и удаление блоков и связей, а также изменение их параметров, приводит к изменениям во многих служебных, автоматически поддерживаемых РДС структурах, необходимых для работы схемы. Если в схему вносятся крупные изменения, например, добавляется и удаляется множество блоков и связей, целесообразно вносить изменения в эти структуры не после каждого небольшого изменения, а после завершения всех изменений. Перед добавлением блоков и связей лучше всего заблокировать изменения в служебных структурах РДС вызовом функции rdsSetSystemUpdate с параметром FALSE. Это не

обязательно, но в противном случае РДС будет обновлять свои структуры после добавления каждого блока и каждой связи, что приведет к существенному замедлению работы. При блокировке изменений функцией `rdsSetSystemUpdate` крайне важно помнить, что после каждого ее вызова с параметром `FALSE` обязательно должен следовать вызов с параметром `TRUE`, иначе РДС не сможет работать правильно.

Заблокировав изменения в служебных структурах РДС, мы вызываем функцию `LoadBlocks`, передавая ей идентификатор нашего блока `BlockData->Block`, имя файла с описанием блока из переменной `BlockFile` и имя списка блоков из переменной `BlockList`. Если функция вернет `TRUE` (блоки добавлены успешно), мы вызовем `LoadConnections`, передав ей идентификатор блока и имя списка связей из переменной `ConnList`. После этого мы снова разрешаем обновление служебных структур РДС вызовом `rdsSetSystemUpdate(TRUE)`.

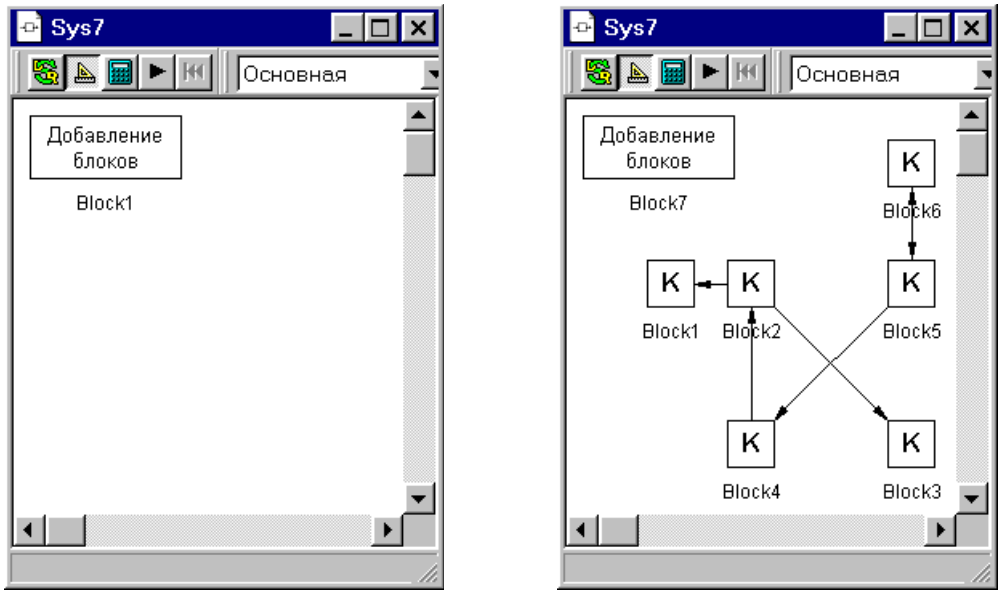
Если идентификатор выбранного пункта меню – 2, мы блокируем обновление служебных структур, удаляем добавленные блоки и связи вызовом написанной нами функции `DeleteByList`, а затем снова разрешаем обновление служебных структур.

Теперь можно протестировать созданную модель. Подключим ее к блоку – назовем его, например, “Block1” (рис. 120а) – и создадим текстовый файл “blocklist.txt” со списком блоков следующего вида:

```
Block1,    50,  100
Block2,    100, 100
Block3,    200, 200
Block4,    100, 200
Block5,    200, 100
Block6,    200, 25
```

Нам также потребуется текстовый файл “connlist.txt” со списком связей:

```
Block1,    x,    Block2,    y
Block2,    y,    Block3,    x
Block2,    x,    Block4,    y
Block5,    y,    Block4,    x
Block6,    x,    Block5,    x
```



а) до добавления б) после добавления

Рис. 120. Программное добавление блоков и связей

Теперь, если настроить блок, как показано на рис. 119, и выбрать в контекстном меню блока пункт “Добавить блоки и связи”, подсистема с этим блоком примет вид, показанный на

рис. 120б. Можно заметить, что созданный нами блок был автоматически переименован в “Block7”, поскольку имена, начиная с “Block1” и заканчивая “Block6”, встретились в списке блоков, и ему пришлось переименовываться, чтобы добавить их в подсистему. Связи между блоками, как мы и планировали, начинаются и заканчиваются на их границах.

Если теперь выбрать в меню блока пункт “Удалить добавленное”, все добавленные блоки и связи будут удалены.

В этом примере мы создавали простые связи, состоящие из единственного отрезка прямой. Объект для редактирования связи, создаваемый функцией `rdsCECreateEditor`, позволяет создавать и более сложные связи – разветвленные, состоящие из множества точек и линий, включая кривые Безье. Функции для работы с этим объектом подробно описаны в приложении А.

Глава 3. Управление РДС из других приложений

Рассматриваются методы выполнения в РДС различных действий по команде от внешнего приложения, передачи информации от внешнего приложения блокам схемы и обратно. Использование внешнего управления позволяет включать РДС в состав более сложных программных комплексов в качестве составной части.

§3.1. Общие принципы управления РДС

Рассматриваются общие принципы использования библиотеки RdsCtrl.dll для управления РДС из другого приложения Windows. Приводится исходный текст тестовой программы, в которую позже будут добавлены вызовы для управления РДС и для реакции на различные события, происходящие в загруженной схеме.

До сих пор мы рассматривали непосредственную работу пользователя с РДС: пользователь сам запускает программу “rds.exe”, сам загружает схему, сам переключает режимы работы, запускает и останавливает расчет. Однако, иногда возникает необходимость включить РДС в состав какого-либо программного комплекса для выполнения расчетов, демонстрации пользователю анимированной схемы соединения каких-либо объектов и т.п. (в частности, на базе РДС была создана подсистема визуального моделирования в составе комплексного ПО для управления муниципальным хозяйством [17]). В этом случае главная программа этого комплекса должна иметь возможность управлять работой РДС и обмениваться информацией с блоками схемы. Кроме того, при этом часто бывает нужно ограничить какие-либо действия пользователя, например, запретить редактирование схемы или переключение режимов.

В состав РДС входит библиотека дистанционного управления RdsCtrl.dll, позволяющая решать перечисленные выше задачи. Приложение должно загрузить эту библиотеку в свое адресное пространство, после чего все взаимодействие с РДС производится через ее экспортированные функции. Запуск и завершение главной программы РДС (“rds.exe”) при этом тоже производится через вызов соответствующих функций RdsCtrl.dll. Следует помнить, что в память управляющего приложения загружается только библиотека управления – РДС все равно запускается как отдельный процесс, с которым эта библиотека обменивается информацией. При этом процесс РДС может завершиться неожиданно для управляющего приложения (из-за какой-либо ошибки, или из-за того, что его завершит пользователь). Библиотека не будет автоматически перезапускать процесс РДС, но она может информировать об этом приложение, которое, при необходимости, перезапустит РДС самостоятельно. вызвав соответствующую функцию библиотеки.

Общая схема работы с библиотекой дистанционного управления такова: сначала приложение загружает эту библиотеку (например, функцией Windows API `LoadLibrary`) и получает доступ к экспортированным из нее функциям. Разумеется, для этого приложение должно знать, где именно находится эта библиотека. Обычно она располагается в папке установки РДС, однако, узнать путь к этой папке может оказаться не так просто. Дело в том, что РДС, как правило, ничего не записывает в системный реестр Windows при установке, поэтому обратиться к реестру и считать оттуда путь к РДС управляющее приложение не может. Это – плата за легкость переноса РДС с машины на машину: поскольку главная программа не зависит от реестра, можно просто скопировать все файлы РДС в произвольную папку (естественно, с сохранением структуры вложенных папок) и запускать “rds.exe” оттуда. Но этим же можно воспользоваться и при включении РДС в состав программного комплекса: можно разместить “rds.exe” и все необходимые для работы РДС файлы и папки в одной из внутренних папок самого комплекса, тогда главная программа всегда будет знать размещение “rds.exe” и RdsCtrl.dll.

После того, как библиотека RdsCtrl.dll загружена и доступ к ее функциям получен (например, при помощи функции Windows API `GetProcAddress`), приложение может сообщить библиотеке, где находится главный исполняемый файл РДС – “rds.exe”. Если

“rds.exe” и RdsCtrl.dll, как обычно, находятся в одной и той же папке, в этом нет необходимости, библиотека найдет этот файл сама. Однако, если библиотека включена в состав приложения, а “rds.exe”, вместе с остальными файлами РДС, располагается где-то в другом месте, приложение должно вызвать функцию `rdsctrlSetPath` (все функции RdsCtrl.dll имеют префикс “rdsctrl”), передав в ее параметре полный путь к главному исполняемому файлу РДС.

Если приложению нужно получать от РДС какую-либо информацию (а, чаще всего, это необходимо), оно должно зарегистрировать в библиотеке специальную функцию для возврата строк произвольной длины. Основная проблема со строками произвольной длины заключается в том, что при вызове какой-либо функции программа не может знать, сколько места в памяти займет возвращаемая этой функцией строка, поэтому она не может предоставить ей буфер (массив символов), размер которого будет заведомо достаточен для помещения в него этой строки. Существуют разные способы решения этой проблемы. Функция, возвращающая строку, может сравнить длину строки с размером переданного ей буфера и, если он слишком мал, вернуть вызвавшей программе специальный код ошибки, сообщив ей требуемый размер буфера (так устроены некоторые функции Windows API). Этот способ требует двойного вызова функции: сначала функции передается какой-то имеющийся по умолчанию буфер, а затем, если функция сообщила о недостаточности размера буфера, отводится новый буфер нужного размера и функция вызывается еще раз. Это – работающее, но несколько громоздкое решение. Другой способ заключается в том, что память под строку нужного размера можно динамически отводить внутри вызываемой функции и возвращать вызвавшей программе указатель на эту строку (так устроены сервисные функции РДС). При этом не требуется повторный вызов функции, поскольку она самостоятельно отводит буфер нужного размера, однако здесь могут возникнуть проблемы с освобождением отведенной памяти. Освобождать отведенную память нужно функцией или оператором, парным к тому, которым память была отведена: например, если память отведена стандартной функцией `malloc`, освобождать ее нужно функцией `free`, а если оператором `new`, освобождать нужно оператором `delete`. Ситуация усугубляется тем, что вызываемая функция может располагаться в библиотеке, которая написана на другом языке программирования с использованием механизмов работы с памятью, несовместимых с вызывающей программой. В этом случае вызвавшая программа вообще не сможет корректно освободить память отведенную в вызванной функции, и в библиотеку нужно будет включать специальную функцию для освобождения памяти. Например, в состав сервисных функций РДС включена функция `rdsFree` для освобождения памяти, отведенной другими сервисными функциями, и модели блоков должны вызывать именно ее (см. стр. 58). При этом, во избежание утечек памяти, вызвавшая программа не должна забывать освобождать возвращенные строки.

В RdsCtrl.dll используется третий способ возврата строк – может быть, несколько более запутанный, но более безопасный. Сразу после загрузки библиотеки приложение вызывает ее функцию `rdsctrlSetStringCallback`, в которую передает указатель на свою собственную функцию вида

```
void RDSCALL имя_функции(LPVOID ptr,LPSTR str);
```

Здесь `ptr` – указатель на какой-то объект, который вызывающая программа использует для хранения строк, а `str` – строка, которую нужно присвоить этому объекту. После этого во все вызываемые функции передается указатель на такой хранящий строку объект, а они, в свою очередь, будут вызывать зарегистрированную функцию для присвоения строки этому объекту. При этом все отведение и освобождение памяти производится в вызывающей программе.

Пусть, например, в качестве объекта для присвоения строк используется указатель `char*`, который будет указывать на строки, отводимые стандартной функцией `malloc` и освобождаемые функцией `free`. Функция для возврата строк будет выглядеть так:

```
void RDSCALL ReturnString(LPVOID ptr,LPSTR str)
{ // Приводим указатель ptr к типу "указатель на char*"
  char **pStr=(char**)ptr;
  // Если указатель не передан, не делаем ничего
  if(pStr==NULL) return;
  // Если там была строка – освобождаем ее функцией free
  if(*pStr) free(*pStr);
  // Отводим память нужного размера
  *pStr=malloc(strlen(str)+1);
  // Копируем туда полученную строку
  strcpy(*pStr,str);
}
```

Для регистрации этой функции после загрузки RdsCtrl.dll нужно сделать следующий вызов:

```
rdsctrlSetStringCallback(ReturnString);
```

Теперь представим себе, что в библиотеке RdsCtrl.dll есть некоторая функция `rdsctrlSomeFunction`, которая возвращает какую-нибудь строку – например, символьное представление целого числа (конечно, в библиотеке нет такой функции, она взята просто для примера):

```
void RDSCALL rdsctrlSomeFunction(int value,LPVOID retstr);
```

В эту функцию передается целое число `value` и указатель на объект, в который нужно передать возвращаемую строку `retstr`. Вызов этой функции будет выглядеть так:

```
char *buf=NULL; // Сюда запишется указатель на возвращаемую строку
rdsctrlSomeFunction(123,&buf);  // Вызов функции
// ... какие-то действия со строкой buf ...
if(buf) free(buf); // Освобождение памяти
```

В данном случае функция `rdsctrlSomeFunction` вызовет зарегистрированную функцию возврата строк `ReturnString`, передав ей указатель на объект `retstr`, равный `&buf`, и строку “123”. Таким образом, в переменной `buf` окажется указатель на копию переданной строки, которая потом, после использования, будет освобождена функцией `free`.

Теперь допустим, что вызывающая программа использует для работы со строками некоторый класс `String`, в котором определена операция присваивания для обычных строк `char*` (такие классы часто встречаются в различных библиотеках). В этом случае функция возврата строк будет выглядеть так:

```
void RDSCALL ReturnString(LPVOID ptr,LPSTR str)
{ // Приводим указатель ptr к типу "указатель на String"
  String *pS=(String*)ptr;
  // Если указатель не передан, не делаем ничего
  if(pS==NULL) return;
  // Присваиваем строку объекту
  *pS=str;
}
```

Вызов функции в этом случае будет выглядеть так:

```
String str;     // Объект для хранения строки
rdsctrlSomeFunction(123,&str);  // Вызов функции
// ... какие-то действия со строкой str ...
```

Здесь не нужно освобождать память, об этом должен позаботиться деструктор класса `String` (если, конечно, этот класс написан без ошибок).

Таким образом, библиотеке RdsCtrl.dll все равно, каким образом вызывающая программа хранит строки переменной длины. Программа должна предоставить библиотеке функцию для присваивания строк каким-то своим внутренним объектам, и передавать указатели на эти объекты при вызове функций библиотеки, которые возвращают строки.

Разобравшись с используемым способом возврата строк, вернемся собственно к управлению РДС. Для управления приложение должно создать одну или несколько связей с РДС при помощи функции `rdsctrlCreateLink`. Одна связь используется для запуска

одной копии “rds.exe” и управления ей. Таким образом, приложению необходимо создать столько связей, со сколькими схемами одновременно оно собирается работать. Чаще всего в каждый момент времени необходимо работать только с одной схемой – в этом случае достаточно создать одну единственную связь, и загружать в соответствующую ей копию РДС разные схемы по мере необходимости. Функция `rdsctrlCreateLink` возвращает целый неотрицательный идентификатор связи, который потом используется во всех остальных функциях библиотеки для обмена информацией с конкретной копией РДС, то есть с конкретной схемой. Создание связи не приводит к немедленному запуску главной программы РДС, вместо этого в памяти создаются структуры для хранения различных параметров, которые будут определять поведение и внешний вид РДС. Эти параметры хранятся до тех пор, пока связь не будет уничтожена вызовом `rdsctrlDeleteLink`, или до момента выгрузки RdsCtrl.dll из памяти. Таким образом, создав связь и настроив ее параметры, можно после этого много раз запускать и завершать “rds.exe” без необходимости повторной установки этих параметров. Например, одна из функций библиотеки позволяет отключить главное окно РДС. Если сначала вызвать эту функцию для созданной связи, а потом запустить РДС через эту связь, “rds.exe” сразу запустится без главного окна, и будет запускаться без него каждый раз при работе через эту связь. Если же сначала запустить РДС, а потом вызвать эту функцию, главное окно некоторое время (до срабатывания функции) будет находиться на экране.

После создания связи можно произвольное число раз запускать “rds.exe” функцией `rdsctrlConnect` и завершать функциями `rdsctrlDisconnect` и `rdsctrlClose`. Функция `rdsctrlDisconnect` передает РДС команду завершения и немедленно возвращает управление вызвавшей программе. Сразу после этого связь, для которой была вызвана `rdsctrlDisconnect`, можно использовать для запуска новой копии РДС, даже если предыдущая запущенная копия все еще чем-то занимается. Если приложению необходимо дождаться фактического завершения РДС, лучше использовать функцию `rdsctrlClose` – она вернет управление только после того, как “rds.exe” доложит о своем завершении.

Иногда возникает необходимость отключиться от управляемой копии РДС, оставив ее работать как самостоятельное приложение. В этом случае следует использовать функцию `rdsctrlLeave` – после ее вызова связь, через которую шла работа, можно использовать для запуска новой копии РДС, а старая лишается связи с управляющей программой и переходит под полный контроль пользователя, которому нужно будет закрыть ее самостоятельно. Следует помнить, что восстановить управление этой копией невозможно, как невозможно и получить управление копией РДС, запущенной пользователем вручную.

После того, как копия РДС запущена, управляющее приложение может отдавать ей различные команды (загружать и сохранять схемы, переключать режимы и т.п.), реагировать на различные события, произошедшие в РДС, а также обмениваться информацией с блоками загруженной схемы. Обмен информацией может происходить как по инициативе управляющего приложения (приложение передает данные блоку, блок их обрабатывает и возвращает результат), так и по инициативе блока (блок вызывает управляющее приложение и передает ему данные). Если приложению необходимо реагировать на события, происходящие в РДС, или получать данные, посланные блоками, оно должно разрешить для данной связи реакции на события и зарегистрировать в библиотеке функции, которые будут автоматически вызываться при наступлении этих событий. Примеры обмена информацией с блоками и реакции на события будут рассмотрены в §3.3 и §3.4.

Управляющее приложение может, при необходимости, отключить некоторые элементы интерфейса пользователя РДС. Например, если РДС используется для показа пользователю каких-либо анимированных мнемонических схем, не следует давать ему возможность редактировать эти схемы – для редактирования должен быть предусмотрен

отдельный режим. В этом случае целесообразно запретить РДС входить в режим редактирования, вызвав для связи, через которую приложение работает с этой копией РДС, функцию `rdsctrlEnableEditMode` с параметром `FALSE`. Для редактирования схемы (если, например, с приложением начал работать пользователь с более высокими полномочиями) можно вызвать `rdsctrlEnableEditMode` с параметром `TRUE`, или создать для редактирования схемы новую связь и запустить через нее другую копию РДС, не запрещая для нее вход в режим редактирования.

Подсистемы схем, загруженных в управляемые копии РДС, открываются, как обычно, в отдельных окнах. Если внешний вид и функции этих окон не подходят для пользовательского интерфейса приложения, оно может выделить одну или несколько прямоугольных областей своих собственных окон и приказать РДС отображать в этих областях (в терминологии RdsCtrl.dll они называются *портами вывода*) содержимое конкретных подсистем загруженной схемы. При этом, поскольку окна принадлежат управляющему приложению, блоки подсистем не смогут самостоятельно реагировать на мышь и клавиатуру и выводить всплывающие подсказки и контекстные меню, подсистемы не будут предоставлять пользователю кнопки изменения масштаба и переключения слоев, и т.п. Все это придется реализовывать в управляющем приложении: например, при щелчке кнопкой мыши в области порта вывода этот щелчок должен быть передан в РДС функцией `rdsctrlViewportMouse`, для получения текста всплывающей подсказки блока по заданным координатам приложение должно будет вызвать `rdsctrlVPPopupHint` и вывести подсказку самостоятельно, и т.д. Использование порта вывода будет подробно рассмотрено в §3.6.

Для того, чтобы привести пример работы с библиотекой RdsCtrl.dll, нам потребуется написать отдельное приложение Windows, которое и будет управлять РДС при помощи этой библиотеки. Сейчас такие приложения чаще всего создают в специализированных средах разработки, таких как Borland C++ Builder или Microsoft Visual C, с использованием библиотек (например, VCL или MFC), существенно облегчающих работу с окнами и прочими объектами Windows. Здесь мы не будем использовать среды разработки и специализированные библиотеки, чтобы рассматриваемые примеры были понятны всем программистам, независимо от того, чем они пользуются. Мы сделаем пример на чистом Windows API – эти функции можно использовать в любом приложении Windows. Пример можно будет компилировать любым компилятором с установленными библиотеками SDK – это стандартные библиотеки разработки, которые можно бесплатно загрузить с web-сайта Microsoft.

Прежде всего, нам потребуется “скелет” программы, которая будет открывать окно с необходимыми нам кнопками и полями ввода. Мы пока не будем загружать RdsCtrl.dll и управлять РДС, вместо этого мы введем в нашу программу функции, которые будут вызываться при нажатии кнопок и перед завершением программы, но пока оставим эти функции пустыми – сейчас наша задача просто создать работающее приложение. Текст программы будет выглядеть так (функции, которые мы допишем потом, выделены жирным):

```
// Описания, необходимые для используемого компилятора
// (в других компиляторах они не понадобятся или будут другими)
#define _WIN32_WINNT 0x0400
#define WINVER 0x0400
// Необходимые для приложения файлы заголовков
#include <windows.h>
#include <Commctrl.h>
#include <stdio.h>

// Идентификаторы кнопок и полей окна программы
#define IDC_OPENBUTTON     101
#define IDC_CLOSEBUTTON    102
```

```
#define IDC_LABEL1         103
#define IDC_LABEL2         104
#define IDC_LABEL3         105
#define IDC_BLKNAMEEDIT    106
#define IDC_VALUEEDIT      107
#define IDC_STRINGEDIT     108
#define IDC_CALLBUTTON     109
#define IDC_STARTBUTTON    110
#define IDC_STOPBUTTON     111
#define IDC_FINDBUTTON     112
//=========================================

// Буфер для индицируемого программой текста
char buffer[2000]="Программа запущена";
// Главное окно программы (для доступа к нему из функций)
HWND MainWin;
//=========================================

// Функция вывода текстового сообщения в окне программы
void DisplayText(char *text)
{ RECT rect;
  // Определяем размер клиентской области окна
  GetClientRect(MainWin,&rect);
  // Ограничиваем область снизу (ниже будут располагаться кнопки)
  rect.bottom=30;
  if(text) // Копируем текст в буфер
    { strncpy(buffer,text,sizeof(buffer)-1);
      buffer[sizeof(buffer)-1]=0; // Если строка слишком длинная
    }
  else
    strcpy(buffer,"(NULL)");
  // Указываем Windows, что область rect нужно перерисовать
  InvalidateRect(MainWin,&rect,TRUE);
}
//=========================================

// Функция, вызываемая перед завершением программы
void BeforeExit(void)
{ // Здесь мы будем выгружать RdsCtrl.dll
}
//=========================================

// Нажатие кнопки "Открыть"
void OpenButtonClick(void)
{ // Здесь мы будем загружать схему в РДС
}
//=========================================

// Нажатие кнопки "Закрыть"
void CloseButtonClick(void)
{ // Здесь мы будем завершать РДС
}
//=========================================

// Нажатие кнопки "Вызвать"
void CallBlockClick(void)
{ // Здесь мы будем передавать информацию блоку схемы
```

```
}
//=======================================

// Нажатие кнопки "Старт"
void StartClick(void)
{ // Здесь мы будем запускать расчет
}
//=======================================

// Нажатие кнопки "Стоп"
void StopClick(void)
{ // Здесь мы будем останавливать расчет
}
//=======================================

// Нажатие кнопки "Найти"
void FindFuncClick(void)
{ // Здесь мы будем искать блок, поддерживающий набор функций
}
//=======================================

// Процедура обработки сообщений главного окна
LRESULT CALLBACK MainWndProc(
                HWND hWindow,    // Окно
                UINT msg,        // Сообщение
                WPARAM wParam,LPARAM lParam) // Параметры
{ HINSTANCE app;
  PAINTSTRUCT ps;
  HDC hDC;
  RECT rect;

  switch(msg)
    { // Создание окна
      case WM_CREATE:
        // Запоминаем идентификатор окна в глобальной переменной
        MainWin=hWindow;
        // Получаем идентификатор приложения
        app=(HINSTANCE)GetWindowLong(hWindow,GWL_HINSTANCE);
        // Создаем кнопки и поля ввода
        // Кнопка "Открыть"
        CreateWindow("BUTTON","Открыть",   // Имя класса и текст
          WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON, // Стили
          0,30,100,24,                 // x,y,ширина,высота
          hWindow,                     // Родительское окно
          (HMENU)IDC_OPENBUTTON,       // Идентификатор кнопки
          app,NULL);
        // Кнопка "Закрыть"
        CreateWindow("BUTTON","Закрыть",
          WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
          101,30,100,24,hWindow,
          (HMENU)IDC_CLOSEBUTTON,app,NULL);
        // Кнопка "Старт"
        CreateWindow("BUTTON","Старт",
          WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
          222,30,90,24,hWindow,
          (HMENU)IDC_STARTBUTTON,app,NULL);
```

```
  // Кнопка "Стоп"
  CreateWindow("BUTTON","Стоп",
    WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
    313,30,90,24,hWindow,
     (HMENU)IDC_STOPBUTTON,app,NULL);
  // Надпись "Имя блока"
  CreateWindow("STATIC","Имя блока:",
   WS_VISIBLE | WS_CHILD | SS_LEFTNOWORDWRAP,
   0,55,100,24,hWindow,(HMENU)IDC_LABEL1,app,NULL);
  // Поле ввода для имени блока
  CreateWindow("EDIT",NULL,
    WS_VISIBLE | WS_CHILD | WS_BORDER | ES_AUTOHSCROLL,
    101,55,201,24,hWindow,(HMENU)IDC_BLKNAMEEDIT,app,NULL);
  // Надпись "Число"
  CreateWindow("STATIC","Число:",
    WS_VISIBLE | WS_CHILD | SS_LEFTNOWORDWRAP,
    0,80,100,24,hWindow,(HMENU)IDC_LABEL3,app,NULL);
  // Поле ввода числа
  CreateWindow("EDIT",NULL,WS_VISIBLE | WS_CHILD | WS_BORDER,
    101,80,201,24,hWindow,(HMENU)IDC_VALUEEDIT,app,NULL);
  // Надпись "Строка"
  CreateWindow("STATIC","Строка:",
    WS_VISIBLE | WS_CHILD | SS_LEFTNOWORDWRAP,
    0,105,100,24,hWindow,(HMENU)IDC_LABEL2,app,NULL);
  // Поле ввода строки
  CreateWindow("EDIT",NULL,
    WS_VISIBLE | WS_CHILD | WS_BORDER | ES_AUTOHSCROLL,
    101,105,201,24,hWindow,(HMENU)IDC_STRINGEDIT,app,NULL);
  // Кнопка вызова блока
  CreateWindow("BUTTON","Вызвать",
    WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
    303,55,100,24,hWindow,(HMENU)IDC_CALLBUTTON,app,NULL);
  // Кнопка поиска блока
  CreateWindow("BUTTON","Найти",
    WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
    303,105,100,24,hWindow,(HMENU)IDC_FINDBUTTON,app,NULL);
  break;

// Закрытие окна
case WM_DESTROY:
  // Выгрузка библиотеки, если она загружена
  BeforeExit();
  PostQuitMessage(0);
  return 0;

// Перерисовка окна
case WM_PAINT:
  // Получаем контекст для рисования
  hDC = BeginPaint(hWindow,&ps);
  // Получаем размеры внутреннего прямоугольника
  GetClientRect(hWindow,&rect);
  rect.bottom=30; // Обрезаем снизу
  // Выводим текст из buffer
  DrawText(hDC,buffer,-1,&rect,
          DT_SINGLELINE | DT_CENTER | DT_VCENTER);
  // Завершаем рисование
  EndPaint(hWindow,&ps);
  return 0;
```

```
      // Команда от органов управления
      case WM_COMMAND:
        if(HIWORD(wParam) == BN_CLICKED) // Нажатие кнопки
          { switch (LOWORD(wParam))
              { case IDC_OPENBUTTON:  OpenButtonClick();    break;
                case IDC_CLOSEBUTTON: CloseButtonClick();   break;
                case IDC_CALLBUTTON:  CallBlockClick();     break;
                case IDC_STARTBUTTON: StartClick();         break;
                case IDC_STOPBUTTON:  StopClick();          break;
                case IDC_FINDBUTTON:  FindFuncClick();      break;
              }
          }
        break;
    }
  // Вызов обработки сообщения по умолчанию
  return DefWindowProc(hWindow,msg,wParam,lParam);
}
//=======================================

// Главная функция приложения
int WINAPI WinMain(   HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      PSTR szCmdLine,
                      int iCmdShow)
{ static char appName[] = "Управление РДС",
              className[]="RDSControlTestWindow";
  WNDCLASSEX myWin;
  HWND hWindow;
  MSG msg;

  // Создание окна
  myWin.cbSize=sizeof(myWin);
  myWin.style=CS_HREDRAW | CS_VREDRAW;
  myWin.lpfnWndProc=MainWndProc;
  myWin.cbClsExtra=0;
  myWin.cbWndExtra=0;
  myWin.hInstance=hInstance;
  myWin.hIcon=0;
  myWin.hIconSm =0;
  myWin.hCursor=0;
  myWin.hbrBackground=(HBRUSH)(COLOR_WINDOW+1);
  myWin.lpszMenuName=0;
  myWin.lpszClassName=className;
  if (!RegisterClassEx(&myWin)) return 0;
  hWindow=CreateWindow(className,appName,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
    0,0,hInstance,0);
  // Открытие созданного окна
  ShowWindow(hWindow,iCmdShow);
  UpdateWindow(hWindow);

  // Инициализация стандартных компонентов
  INITCOMMONCONTROLSEX icc;
  icc.dwSize=sizeof(icc);
  icc.dwICC=ICC_WIN95_CLASSES;
  if(!InitCommonControlsEx(&icc))
```

```
    DisplayText("Ошибка InitCommonControlsEx");

  // Цикл обработки сообщений приложения
  while(GetMessage(&msg,0,0,0))
    { TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  return 0;
}
//=======================================
```

В начале приведенного выше текста мы вводим описания, необходимые для компиляции, и включаем стандартные файлы заголовков “windows.h” и “commctrl.h” из комплекта SDK Windows и “stdio.h” из стандартных библиотек C. Затем мы вводим `define`-константы для идентификаторов, которые мы присвоим кнопкам и полям ввода нашей программы, и описываем две глобальных переменных: массив символов `buffer`, из которого наша программа будет брать текстовые сообщения для вывода в окне, и переменная `MainWin`, в которую мы запишем дескриптор главного окна нашей программы после его создания, чтобы любая функция могла к нему обращаться. Далее описана функция `DisplayText`, которая помещает в массив `buffer` переданную строку сообщения и заставляет главное окно программы обновиться при первой возможности; за ней следует некоторое количество пустых функций, которые будут вызываться при нажатии кнопок в главном окне и перед завершением всего приложения – мы наполним их содержимым позднее. В конце текста описана функция обработки сообщений окна нашей программы `MainWndProc`, которая постоянно вызывается в процессе работы приложения, и главная функция приложения `WinMain`, которая создает главное окно и организует цикл обработки сообщений. Более подробно рассматривать описанные выше функции нашего приложения мы не будем – желающие могут обратиться к литературе по программированию с использованием Windows API. Для нас сейчас важно, что скомпилировав этот текст и запустив получившуюся программу, мы получим окно, изображенное на рис. 121: в верхней его части будет выводиться строка из массива `buffer`, в нижней будут располагаться различные кнопки и поля ввода. Теперь мы будем постепенно добавлять в это приложение команды управления РДС и реакции на события.

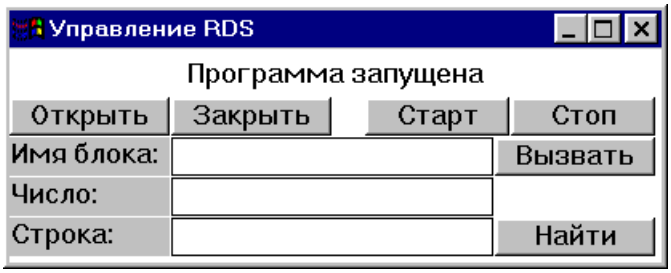


Рис. 121. Главное окно управляющего приложения

§3.2. Загрузка библиотеки и управление схемой

Рассматривается простой пример управления схемой из внешнего приложения. В программу, созданную в предыдущем параграфе, добавляются функции для запуска РДС, загрузки выбранной пользователем схемы и переключения режимов работы РДС.

В принципе, библиотеку RdsCtrl.dll можно загружать в память при старте приложения, однако, лучше делать это тогда, когда она действительно нужна. Сейчас мы сделаем так, чтобы при нажатии кнопки “Открыть” в нашей программе пользователю показывался стандартный диалог открытия файла. Если он выберет в нем какой-нибудь файл схемы, программа загрузит RdsCtrl.dll, запустит РДС и даст команду на открытие этой схемы.

Точно так же, как для написании моделей блоков мы должны были включить в наш исходный текст файлы заголовков с описаниями, необходимыми для этих моделей, и получить доступ к сервисным функциям РДС, здесь мы тоже должны включить в текст программы файл “RdsCtrl.h” и получить доступ к функциям библиотеки RdsCtrl.dll. Мы можем получать указатели на каждую функцию библиотеки вручную при помощи вызова `GetProcAddress`, но гораздо удобнее воспользоваться встроенной в “RdsCtrl.h” конструкцией для получения доступа ко всем функциям сразу, аналогичной использовавшейся нами при написании моделей файлу “RdsFunc.h” (см. стр. 34). Необходимые нам описания будут выглядеть так:

```
// Описания, необходимые для работы с RdsCtrl.dll
#define RDSCTRL_SERV_FUNC_BODY GetRdsCtrlFuncs
#include <RdsCtrl.h>
//=======================================
```

Перед включением файла заголовка мы разместили определение константы вида

```
#define RDSCTRL_SERV_FUNC_BODY имя_функции_пользователя
```

Это приведет к тому, что в месте включения файла “RdsCtrl.h” в текст программы, во-первых, будет создан полный список глобальных переменных-указателей на функции библиотеки (причем имена переменных будут совпадать с именами этих функций), и, во-вторых, вставлено тело функции с заданным пользователем именем, которая заполняет эти переменные указателями на функции. После загрузки библиотеки нужно вызвать эту функцию, передав ей идентификатор модуля загруженной библиотеки, после чего все функции этой библиотеки можно будет вызывать просто по именам. Например, для функции `rdsctrlCreateLink` будет автоматически создано описание глобальной переменной

```
RDSCTRL_IV rdsctrlCreateLink;
```

и внутри функции с заданным пользователем именем (в нашем случае – `GetRdsCtrlFuncs`) ей будет соответствовать строчка

```
rdsctrlCreateLink=
      (RDSCTRL_IV)GetProcAddress(dll,"rdsctrlCreateLink");
```

где `dll` – переданный в функцию идентификатор модуля загруженной библиотеки, а тип `RDSCTRL_IV` описан в “RdsCtrl.h” как указатель на функцию, не принимающую параметров, и возвращающую целое число.

Для создания этих глобальных переменных и функции получения указателей недостаточно просто включить в исходный текст программы файл “RdsCtrl.h”, нужно обязательно вставить перед ним описание константы `RDSCTRL_SERV_FUNC_BODY`. Без него программе будут доступны описания всех констант и типов, необходимых для работы с RdsCtrl.dll, но никаких описаний глобальных переменных и тела функции не будет.

Если исходный текст управляющего приложения состоит из нескольких файлов, включать файл “RdsCtrl.h” с описанной перед ним константой `RDSCTRL_SERV_FUNC_BODY` можно только в один из них, иначе будет создано два набора переменных и две функции с одинаковыми именами, что приведет к ошибкам при компиляции программы. Чтобы можно было вызывать функции RdsCtrl.dll из других модулей программы, нужно перед включением “RdsCtrl.h” описать константу `RDSCTRL_SERV_FUNC_EXTERNAL`:

```
#define RDSCTRL_SERV_FUNC_EXTERNAL
#include <RdsCtrl.h>
```

Этой константе не нужно давать какое-либо значение – сам факт ее описания приведет к тому, что в месте включения файла заголовка будет вставлен список внешних (`extern`) описаний глобальных переменных-указателей на функции. Поскольку сами эти переменные будут присутствовать в другом модуле, где есть описание `RDSCTRL_SERV_FUNC_BODY`, программа будет скомпилирована без ошибок. В нашем случае исходный текст приложения будет состоять из единственного файла, поэтому константа `RDSCTRL_SERV_FUNC_EXTERNAL` нам здесь не понадобится.

Разумеется, все приведенные выше описания можно использовать только в программах, написанных на C или C++. В других языках программирования придется получать доступ к функциям библиотеки вручную, вызовами `GetProcAddress`.

В нашей программе мы пока не будем реагировать на события, возникающие в РДС, но, чтобы потом эти реакции проще было добавить, мы создадим пустую функцию, которая будет регистрировать их в библиотеке:

```
// Разрешение событий и регистрация их функций
void RegisterEvents(void)
{ // Пока оставим эту функцию пустой
}
//=======================================
```

Еще мы опишем функцию возврата строки (см. стр. 518), но тоже оставим ее пустой. Пока нам не нужно получать какие-либо строки от РДС, тем не менее, в дальнейшем она нам понадобится:

```
// Функция возврата строки
void RDSCALL ReturnString(LPVOID ptr,LPSTR str)
{ // Пока пустая
}
//=======================================
```

Загрузку библиотеки и создание связи с одной копией РДС (нам потребуется только одна) мы оформим в виде отдельной функции. Для того, чтобы другие функции нашей программы могли работать с РДС через созданную связь, идентификаторы модуля загруженной библиотеки и связи мы сделаем глобальными переменными:

```
// Глобальные переменные для связи с РДС
HMODULE RdsCtrl=NULL; // Модуль библиотеки RdsCtrl.dll
int RdsLink=-1;       // Связь с РДС
//=======================================
```

Переменную `RdsCtrl` мы инициализируем нулем, после загрузки библиотеки в нее будет записан идентификатор модуля. Анализируя значение этой переменной, мы сможем загружать библиотеку только в том случае, если она еще не загружалась (`RdsCtrl` равна `NULL`). Идентификатор связи `RdsLink` имеет начальное значение –1 (созданная связь не может иметь отрицательный идентификатор), что позволяет нам проверять, создавалась ли уже связь с РДС: если она была создана, `RdsLink` будет иметь нулевое или положительное значение.

Теперь можно написать функцию, которая будет загружать RdsCtrl.dll и создавать связь с РДС (“rds.exe” загружаться при этом не будет, мы просто подготавливаем библиотеку к работе).

```
// Загрузка RdsCtrl.dll и создание связи
void InitRdsCtrl(void)
{
  if(RdsCtrl==NULL) // Библиотека еще не загружена
    { char rdsctrldll[MAX_PATH+1],*s;
      // Считаем, что наша программа находится в одной папке с РДС
      // Получаем путь к RdsCtrl.dll из пути к нашей программе
      GetModuleFileName(NULL,rdsctrldll,MAX_PATH);
      s=strrchr(rdsctrldll,'\\'); // Ищем последний '\'
      if(!s) // Ошибка
        { DisplayText("Библиотека не найдена");
          return;
        }
      // Заменяем имя файла в пути
      strcpy(s+1,"RdsCtrl.dll");
```

```
        // Загружаем библиотеку RdsCtrl.dll
        RdsCtrl=LoadLibrary(rdsctrldll);
        if(RdsCtrl==NULL) // Загрузка не удалась
          { DisplayText("Ошибка загрузки RdsCtrl.dll");
            return;
          }
        // Получаем доступ к функциям библиотеки
        if(!GetRdsCtrlFuncs(RdsCtrl))
          { // Ошибка
            DisplayText("Нет доступа к функциям RdsCtrl.dll");
            // Выгружаем библиотеку - она бесполезна
            FreeLibrary(RdsCtrl);
            RdsCtrl=NULL;
            return;
          }
        // Доступ к функциям получен - можно их вызывать

        // Установка функции возврата строки
        rdsctrlSetStringCallback(ReturnString);
        // Сброс идентификатора связи (если он почему-то не сброшен)
        RdsLink=-1;
      } // if(RdsCtrl==NULL)

    // Создание связи с РДС
    if(RdsLink<0) // Связь не создана
      { // Создаем связь (rds.exe пока не запускается)
        RdsLink=rdsctrlCreateLink();
        if(RdsLink<0) // Ошибка
          { DisplayText("Ошибка создания связи с РДС");
            return;
          }
        // Запрет главного окна РДС
        rdsctrlShowMainWindow(RdsLink,FALSE);
        // Регистрация откликов на события
        RegisterEvents();
      }
  }
  //=======================================
```

В этой функции сначала проверяется, не загружена ли уже RdsCtrl.dll – если это так, глобальная переменная `RdsCtrl` будет содержать идентификатор загруженного модуля. Если же в ней находится значение `NULL`, значит, нужно загрузить библиотеку и записать указатели на ее функции в автоматически вставленный в нашу программу набор глобальных переменных.

Чтобы загрузить библиотеку в память, мы должны знать полный путь к ее файлу. Для простоты будем считать, что и RdsCtrl.dll, и “rds.exe” находятся в одной папке с нашей программой. При создании “настоящего” приложения нужно либо включать РДС в его состав, размещая необходимые файлы в одной из внутренних папок, либо предусматривать в настройках приложения указание пути к РДС. В данном случае мы просто будем помещать исполняемый файл нашей программы в папку РДС, поэтому для поиска пути к библиотеке нам достаточно получить полный путь к нашей программе и заменить в нем имя файла на RdsCtrl.dll. Для получения полного пути к исполняемому файлу нашей программы используется функция Windows API `GetModuleFileName`, в которую вместо идентификатора модуля передается `NULL`, чтобы она вернула путь к файлу, из которого создан вызвавший ее процесс, то есть к нашей программе. Путь записывается в массив `rdsctrldll`, в котором затем функцией `strrchr` ищется последний символ обратной

косой черты – за ним идет имя файла нашей программы. Вместо этого имени файла мы записываем RdsCtrl.dll, в результате чего в rdsctrldll получается полный путь к библиотеке.

Для загрузки библиотеки мы вызываем функцию Windows API LoadLibrary, после чего записываем идентификатор загруженного модуля в глобальную переменную RdsCtrl. Затем этот идентификатор передается в автоматически вставленную в нашу программу функцию GetRdsCtrlFuncs, которая получит указатели на все функции загруженной библиотеки и запишет их в одноименные глобальные переменные (эта функция была вставлена в текст нашей программы в момент включения файла заголовков "RdsCtrl.h", поскольку перед ним мы описали константу RDSCTRL_SERV_FUNC_BODY). Функция вернет FALSE, если в библиотеке не окажется хотя бы одной из функций – так бывает, если версия файла "RdsCtrl.h" не соответствует версии самой библиотеки. В этом случае мы выгружаем библиотеку – мы не можем ей пользоваться.

После того, как доступ к функциям библиотеки получен, мы вызываем функцию rdsctrlSetStringCallback (теперь функции библиотеки можно вызывать просто по именам), чтобы зарегистрировать в библиотеке нашу функцию возврата строк ReturnString. Эта функция пока пуста, но мы все равно регистрируем ее, чтобы в дальнейшем, когда она нам понадобится, не переписывать функцию InitRdsCtrl. Затем мы на всякий случай присваиваем глобальной переменной RdsLink значение –1, указывающее на то, что связь с РДС не создана.

Теперь нужно создать связь с РДС. На всякий случай, мы проверяем, не создана ли она уже (это позволит нам безопасно вызывать InitRdsCtrl при уже созданной связи), сравнивая глобальную переменную RdsLink с нулем. Идентификатор связи не может быть отрицательным, поэтому если RdsLink меньше нуля, связь еще не создавалась. В этом случае мы вызываем функцию библиотеки rdsctrlCreateLink и записываем в RdsLink идентификатор, который она возвращает. Начиная с этого момента, мы можем управлять РДС через созданную связь. Мы пока не будем запускать "rds.exe", мы просто установим некоторые параметры связи. Во-первых, мы отключим главное окно РДС – для этого вызывается функция rdsctrlShowMainWindow с параметром FALSE. Запрет показа главного окна запомнится в параметрах связи и, при запуске РДС, библиотека отключит его. Во-вторых, мы вызываем нашу собственную функцию RegisterEvents для настройки реакций программы на события, возникающие в РДС. Мы пока оставили эту функцию пустой, поэтому наша программа никак не будет реагировать на события, но, в дальнейшем, мы введем эти реакции.

На этом функция InitRdsCtrl завершается. После ее вызова, если не возникло никаких ошибок, идентификатор загруженной библиотеки окажется в глобальной переменной RdsCtrl, а идентификатор созданной связи с РДС – в RdsLink.

Теперь напишем функцию BeforeExit, которую мы вызываем перед завершением нашей программы – она должна уничтожить связь и выгрузить библиотеку. Ранее мы оставили эту функцию пустой, сейчас мы ее заполним:

```
// Функция, вызываемая перед завершением программы
void BeforeExit(void)
{
  if(RdsCtrl!=NULL) // Библиотека загружена
    { if(RdsLink>=0) // Создана связь с РДС
        { // Завершаем РДС
          rdsctrlClose(RdsLink);
          // Удаляем связь
          rdsctrlDeleteLink(RdsLink);
          RdsLink=-1;
        }
```

```
        // Выгружаем библиотеку
        FreeLibrary(RdsCtrl);
        RdsCtrl=NULL;
      }
  }
  //=======================================
```

Перед уничтожением связи мы вызываем функцию `rdsctrlClose`, которая завершает запущенную копию РДС. В принципе, команда на завершение и так передалась бы РДС в момент уничтожения связи, но такой вызов надежнее – если при завершении произойдут какие-либо события, управляющая программа сможет на них отреагировать. Затем связь уничтожается функцией `rdsctrlDeleteLink`, после чего переменной `RdsLink` присваивается значение –1, говорящее об отсутствии связи. После этого мы выгружаем библиотеку функцией Windows API `FreeLibrary` и присваиваем глобальной переменной `RdsCtrl` значение `NULL`. Функция `BeforeExit` у нас вызывается только один раз перед завершением приложения, поэтому можно было бы и не сбрасывать переменные `RdsLink` и `RdsCtrl`. Однако, в таком виде ее можно вызвать в любой момент времени для разрыва связи и выгрузки библиотеки, с возможностью ее повторной загрузки функцией `InitRdsCtrl`, хотя в этом примере нам это и не нужно.

Теперь напишем функцию `LoadScheme`, которая загрузит в РДС схему, имя которой передается в ее параметре. Эта функция должна будет сама вызвать уже написанную нами `InitRdsCtrl`, если библиотека не загружена и связь не создана.

```
  // Открыть файл схемы (filename – имя файла)
  BOOL LoadScheme(char *filename)
  { if(RdsCtrl==NULL || RdsLink<0)
      { // Библиотека RdsCtrl.dll еще не загружена
        InitRdsCtrl(); // Загружаем
        if(RdsCtrl==NULL) // Ошибка
          return FALSE;
        // Запускаем rds.exe
        if(!rdsctrlConnect(RdsLink))
          { DisplayText("Ошибка запуска РДС");
            return FALSE;
          }
      }
    // Если rds.exe не работает (пользователь вышел из РДС),
    // перезапускаем РДС
    rdsctrlRestoreConnection(RdsLink);

    // РДС работает - загружаем схему
    if(!rdsctrlLoadSystemFromFile(RdsLink,filename,FALSE))
      { // Ошибка загрузки
        DisplayText("Ошибка загрузки схемы");
        return FALSE;
      }
    // Переходим в режим моделирования
    rdsctrlSetCalcMode(RdsLink);
    return TRUE;
  }
  //=======================================
```

Прежде всего мы проверяем, инициализированы ли глобальные переменные `RdsCtrl` и `RdsLink`, то есть загружена ли библиотека RdsCtrl.dll и создана ли связь с РДС. Если хотя бы одна из переменных имеет неправильное значение, мы вызываем функцию `InitRdsCtrl`, которая загрузит библиотеку и создаст связь, после чего функцией `rdsctrlConnect`, в которую передается идентификатор связи `RdsLink`, мы запускаем

“rds.exe” для работы с этой связью. Следует помнить, что во все функции, которые вызываются для управления РДС, передается идентификатор связи, чтобы библиотека могла понять, к какой из копий РДС (если их несколько) относятся эти вызовы.

Если библиотека загружена и связь создана (не важно, создана она только что или уже была до вызова функции `LoadScheme`), нам необходимо проверить, работает ли сейчас процесс “rds.exe”, которым управляет эта связь, и запустить РДС, если это не так. Для этого используется функция `rdsctrlRestoreConnection`, которая проверит наличие управляемого процесса и запустит РДС, если это необходимо. Ее можно безопасно вызывать и при работающем “rds.exe” – если он уже есть, она не будет запускать новую его копию. Может показаться, что проверять наличие процесса не нужно: сразу после загрузки библиотеки мы запустили его функцией `rdsctrlConnect`. Однако, пользователь может выйти из РДС и завершить этот процесс, при этом библиотека RdsCtrl.dll останется в памяти нашего приложения, и связь с идентификатором `RdsLink` тоже будет доступна для работы, просто ей не будет соответствовать работающая копия РДС. Поэтому перед тем, как загружать схему, нужно либо вызвать `rdsctrlRestoreConnection`, либо проверить наличие процесса РДС вручную, вызвав `rdsctrlIsConnected`, и повторить вызов `rdsctrlConnect`, если она вернула `FALSE`.

Теперь библиотека загружена, связь создана, и РДС работает – можно загружать схему, имя файла которой передано в параметре `filename`. Для этого используется функция `rdsctrlLoadSystemFromFile`, в которую, как всегда, передается идентификатор связи (`RdsLink`), имя файла и логический параметр, указывающий на необходимость предупреждать пользователя о наличии изменений в текущей схеме, вместо которой загружается новая. В данном случае не сохраненные изменения нас не волнуют, поэтому мы передаем `FALSE`. После того, как схема загрузится, мы переводим РДС в режим моделирования функцией `rdsctrlSetCalcMode` и возвращаем `TRUE` – загрузка схемы выполнена успешно.

Все вспомогательные функции готовы – осталось сделать так, чтобы при нажатии кнопки “Открыть” (см. рис. 121) пользователь мог выбрать схему и загрузить ее в РДС (при этом будет вызываться только что написанная нами функция `LoadScheme`), а нажатие кнопки “Закрыть” завершало бы РДС. При нажатии кнопок “Открыть” и “Закрыть” в нашей программе вызываются пустые функции `OpenButtonClick` и `CloseButtonClick` соответственно, теперь мы их заполним.

Функция, вызываемая при нажатии кнопки “Открыть”, будет выглядеть так:

```
// Нажатие кнопки "Открыть"
void OpenButtonClick(void)
{ char filename[MAX_PATH+1]=""; // Буфер для имени файла
  OPENFILENAME ofn;
  // Заполняем структуру OPENFILENAME
  ZeroMemory(&ofn,sizeof(ofn));
  ofn.lStructSize=sizeof(ofn);
  ofn.hwndOwner=MainWin;
  ofn.lpstrFile=filename;
  ofn.nMaxFile=sizeof(filename);
  ofn.lpstrFilter="Схемы (*.rds)\0*.rds\0Все файлы\0*.*\0";
  ofn.nFilterIndex=1;
  ofn.Flags=OFN_EXPLORER | OFN_FILEMUSTEXIST;
  // Вызываем стандартный диалог открытия файла
  if(GetOpenFileName(&ofn)) // Пользователь выбрал файл
    { // Загружаем схему
      if(LoadScheme(filename))
        DisplayText(filename);
    }
```

```
}
//=======================================
```

Для выбора файла пользователем мы используем стандартный диалог открытия файла Windows, который вызывается функцией `GetOpenFileName`. Для работы с этой функцией необходимо предварительно заполнить информацией структуру `OPENFILENAME`: в ней должен находиться указатель на массив для имени файла, дескриптор главного окна приложения (мы берем его из глобальной переменной `MainWin`), описание фильтров имен файлов диалога, флаги и т.п. Если пользователь выберет в диалоге какой-либо файл, функция вернет `TRUE`, и мы вызовем `LoadScheme` для запуска РДС и загрузки схемы, имя файла которой `GetOpenFileName` поместила в массив `filename`. Если схема будет загружена успешно, `LoadScheme` тоже вернет `TRUE`, и имя загруженного файла будет выведено в верхней части окна нашей программы функцией `DisplayText` (рис. 122).

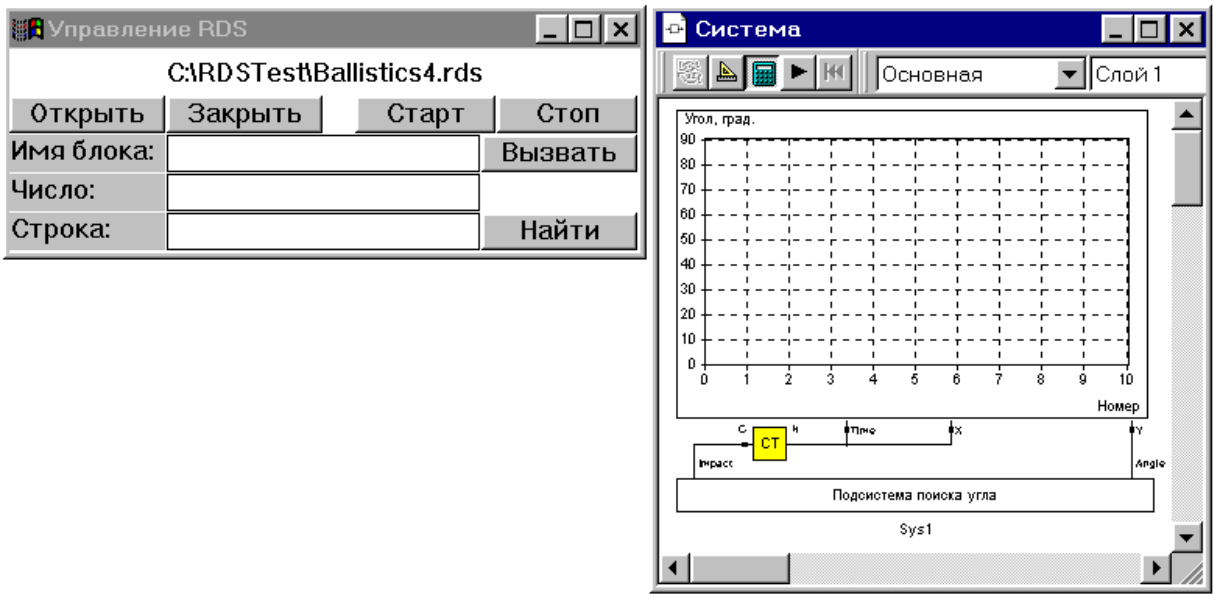


Рис. 122. Загрузка схемы из программы

Поскольку мы запретили показ главного окна РДС, на экране появятся только окна подсистем этой схемы, которые были открыты на момент ее сохранения. Кнопка вызова главного окна в окнах подсистем будет недоступна по этой же причине.

Для того, чтобы кнопкой “Закрыть” можно было завершить РДС, в функцию `CloseButtonClick` нужно вставить следующие операторы:

```
// Нажатие кнопки "Закрыть"
void CloseButtonClick(void)
{ if(RdsLink>=0) // Есть связь
    { rdsctrlClose(RdsLink); // Завершаем РДС
      DisplayText("Схема закрыта");
    }
}
//=======================================
```

Здесь, если существует связь с РДС (переменная `RdsLink` имеет неотрицательное значение), мы завершаем процесс РДС функцией `rdsctrlClose`. Связь при этом не удаляется, мы можем в любой момент снова запустить через нее РДС вызовом `rdsctrlConnect` или `rdsctrlRestoreConnection`. Если бы мы хотели вместе с завершением РДС выгрузить из памяти библиотеку RdsCtrl.dll (например, если мы знаем, что в следующий раз она понадобится нам не скоро), можно было бы вызвать вместо `rdsctrlClose` нашу функцию `BeforeExit`: она не только завершила бы “rds.exe”, но и уничтожила бы связь и выгрузила библиотеку из памяти. При следующем нажатии кнопки “Открыть” библиотека снова бы

загрузилась из-за вызова `InitRdsCtrl` внутри `LoadScheme`, так что работоспособность нашей программы при этом не пострадала бы.

Теперь заставим кнопки “Старт” и “Стоп” запускать и останавливать расчет в запущенной копии РДС. Для этого нужно наполнить содержимым функции `StartClick` и `StopClick`, которые мы оставили пустыми:

```
// Нажатие кнопки "Старт"
void StartClick(void)
{ if(RdsLink>=0) // Есть связь
    rdsctrlStartCalc(RdsLink);
}
//=======================================

// Нажатие кнопки "Стоп"
void StopClick(void)
{ if(RdsLink>=0) // Есть связь
    rdsctrlStopCalc(RdsLink);
}
//=======================================
```

Обе функции устроены одинаково: если в программе создана связь с РДС (значение `RdsLink` неотрицательно), вызывается функция запуска расчета `rdsctrlStartCalc` или функция остановки `rdsctrlStopCalc`. Точно так же можно было бы добавить кнопку сброса расчета, которая вызывала бы функцию `rdsctrlResetCalc`, но мы не будем делать этого в связи с очевидностью примера. Можно заметить, что в этих функциях, в отличие от функции `LoadScheme`, мы не проверяем, работает ли процесс РДС, а сразу пытаемся им управлять. Это безопасно – при отсутствии управляемого процесса вызовы управляющих функций просто игнорируются библиотекой. Если бы мы хотели автоматически перезапускать завершенный пользователем процесс РДС при нажатии на кнопку “Старт”, нам нужно было бы всегда помнить имя последней загруженной схемы (например, копировать его в глобальную переменную), и повторно вызывать `LoadScheme` с этим именем перед вызовом `rdsctrlStartCalc`, чтобы на момент вызова была загружена последняя выбранная пользователем схема.

Все остальные функции управления РДС (переключения режимов, разрешения и запрещения различных функций и т.п.) работают точно так же: при их вызове указывается идентификатор связи, соответствующей управляемой копии РДС, и передаваемые параметры. Все эти функции подробно описаны в приложении Б.

§3.3. Вызов функции блока загруженной схемы

Рассматривается передача информации конкретному блоку загруженной схемы и получение от него ответа, а также способ поиска блока по символическому имени выполняемой им операции. В программу, созданную в предыдущем параграфе, добавляются функции вызова и поиска блоков.

Управляющее приложение может вызвать модель любого блока загруженной схемы, если известно полное имя (см. §1.4) этого блока, и передать ей целое число и строку. Модель блока при этом вызывается в режиме `RDS_BFM_REMOTEMSG`. Реагируя на этот вызов, модель, в свою очередь, тоже может передать вызвавшему приложению целое число и строку. Передача данных других типов в RdsCtrl.dll не предусмотрена, поэтому обычно весь обмен информацией между приложением и блоком производится с преобразованием данных в строки, при этом передаваемое блоку целое число обычно трактуется как номер функции или операции, которую должен выполнить блок, а передаваемая строка – как параметры этой операции. Результат операции обычно возвращается приложению в виде строки, а возвращаемое целое число указывает на успешность выполнения операции.

Для передачи блоку числа и строки в библиотеке RdsCtrl.dll предусмотрена функция rdsctrlCallBlockFunctionEx:

```
int RDSCALL rdsctrlCallBlockFunctionEx(
      int link,             // Идентификатор связи с РДС
      LPSTR FullBlockName,  // Полное имя блока
      int MessageVal,       // Передаваемое целое число
      LPSTR MessageStr,     // Передаваемая строка
      LPVOID ReturnStr);    // Объект для возврата строки
```

В параметре link, как обычно, передается идентификатор связи с конкретной копией РДС, в которую загружена схема, блок которой вызывается. В параметре FullBlockName передается полное имя вызываемого блока, то есть строка, в которой через двоеточие перечисляются все подсистемы на пути от корневой подсистемы к блоку (например, ":Sys1:Sys10:Block1"). В параметрах MessageVal и MessageStr передаются целое число и строка, которые должен получить блок, соответственно. Строка, возвращаемая блоком, записывается зарегистрированной в библиотеке функцией возврата строки (см. стр. 518) в объект ReturnStr, а возвращаемое блоком целое число будет результатом самой функции rdsctrlCallBlockFunctionEx. Если блока с указанным полным именем нет в схеме, функция вернет значение –1.

Примером блоков, откликающихся на вызов внешних приложений, могут служить стандартные поля ввода вещественных чисел из библиотеки блоков РДС. При передаче полю ввода числа 0 оно возвращает число 1 и свое текущее значение, преобразованное в строку. При передаче числа 1 оно снова возвращает 1, преобразует полученную строку в вещественное число и устанавливает его в качестве своего текущего значения, при этом вызвавшей программе никаких строк не возвращается.

Добавим в нашу программу возможность вызова функции блока. Для этого сначала нам необходимо написать функцию возврата строки ReturnString, которую мы зарегистрировали, но оставили пустой. Для удобства работы со строками переменной длины введем вспомогательный класс TDynString (можно обойтись и без него, но так текст программы получится более наглядным):

```
// Вспомогательный класс для более удобного хранения
// строк произвольной длины
class TDynString
{ public:
    // Указатель на динамическую строку
    char *c_str;
    // Освободить память
    void Free(void)
      { if(c_str) delete[] c_str;
        c_str=NULL; };
    // Записать строку в объект
    void Set(char *s)
      { Free(); // Освободить старую
        if(s!=NULL) // Отвести память и скопировать новую
          { c_str=new char[strlen(s)+1];
            strcpy(c_str,s);
          }
      };
    // Строка пустая?
    BOOL IsEmpty(void)
      { return c_str==NULL || (*c_str)==0;};
    // Конструктор и деструктор
    TDynString(void){c_str=NULL;};
```

```
    ~TDynString(){Free();};
};
//=========================================
```

Этот класс сам отводит память под строку нужной длины и хранит указатель на нее в поле `c_str`. Функция класса `Free` служит для освобождения памяти, занятой строкой (она вызывается, в том числе, и в деструкторе класса), функция `IsEmpty` – для проверки строки на пустоту, а функция `Set` – для присвоения строки объекту с автоматическим отведением необходимого объема памяти. Чтобы присвоить, например, переменной `str` типа `TDynString` строку “123”, нужно записать `str.Set("123")`.

Функция возврата строки `ReturnString` будет присваивать строки объектам класса `TDynString`:

```
// Функция возврата строки
void RDSCALL ReturnString(LPVOID ptr,LPSTR str)
{ TDynString *pDS=(TDynString*)ptr;
  if(pDS) pDS->Set(str);
}
//=========================================
```

Переданный в функцию указатель на объект `ptr` приводится к типу “указатель на `TDynString`” и записывается во вспомогательную переменную `pDS`, после чего у этого объекта вызывается функция `Set` с переданной строкой `str`.

В окне управляющей программы мы предусмотрели три поля ввода: для имени блока, для числа и для строки (см. рис. 121), их мы и будем использовать при вызове блока. Вызывать блок мы будем кнопкой “Вызвать”, для этого нужно написать функцию `CallBlockClick`, которая в нашей программе вызывается при нажатии этой кнопки:

```
// Нажатие кнопки "Вызвать"
void CallBlockClick(void)
{ // Массивы для имени блока и передаваемой строки
  char blockname[1000],text[1000];
  // Передаваемое и возвращаемое числа
  int value,retcode;
  // Объект для возвращаемой строки
  TDynString retstr;

  if(RdsLink<0) // Связь не создана
    return;

  // Получаем содержимое полей ввода окна
  GetDlgItemText(MainWin,IDC_BLKNAMEEDIT,
                 blockname,sizeof(blockname)-1);
  value=(int)GetDlgItemInt(MainWin,IDC_VALUEEDIT,NULL,TRUE);
  GetDlgItemText(MainWin,IDC_STRINGEDIT,text,sizeof(text)-1);
  // Теперь имя вызываемого блока записано в массив blockname,
  // передаваемое число - в переменную value, передаваемая
  // строка - в массив text

  // Вызываем блок
  retcode=rdsctrlCallBlockFunctionEx(
     RdsLink,   // Связь с РДС
     blockname, // Полное имя блока
     value,     // Передаваемое целое число
     text,      // Передаваемая строка
     &retstr);  // Указатель на объект для возврата строки
  if(retcode==-1)
    DisplayText("Блок не найден");
```

```
    else // Выводим возвращенную строку в окне программы
      DisplayText(retstr.c_str);
  }
  //=======================================
```

В этой функции мы сначала считываем данные из полей ввода функциями Windows API `GetDlgItemText` и `GetDlgItemInt`: строка из поля с идентификатором `IDC_BLKNAMEEDIT` помещается в массив `blockname`, целое число из поля `IDC_VALUEEDIT` – в переменную `value`, строка из поля `IDC_STRINGEDIT` – в массив `text`. Дескриптор окна-владельца полей ввода, который нужно передать в эти функции, мы берем из глобальной переменной `MainWin` – мы записали его туда при создании главного окна программы. Затем мы вызываем блок загруженной схемы с именем `blockname`, передавая ему число `value` и строку `text`, при помощи функции `rdsctrlCallBlockFunctionEx`. В качестве указателя на объект для возврата строки мы передаем `&retstr` (`retstr` – переменная типа `TDynString`, описанная в начале функции), а результат вызова функции записываем в целую переменную `retcode`. Теперь в поле `c_str` объекта `retstr` должен содержаться указатель на динамически созданную копию возвращенной блоком строки (так работает созданный нами класс `TDynString`), а в переменной `retcode` – возвращенное блоком число. Если `retcode` не равно –1, мы выводим возвращенную блоком строку в верхней части окна программы функцией `DisplayText`.

Теперь, если загрузить в управляемую нашей программой копию РДС какую-нибудь схему с полями ввода, можно будет устанавливать и считывать значения этих полей. Чтобы занести значение в поле ввода, нужно указать его имя в поле “Имя блока”, число 1 в поле “Число”, ввести устанавливаемое значение в поле “Строка”, после чего нажать на кнопку “Вызвать” (рис. 123). Поскольку имена полей ввода обычно не отображаются в окне подсистемы, а для вызова блока нужно знать его имя, имя конкретного поля ввода можно узнать в окне его параметров или прочесть его в строке состояния, выделив это поле в режиме редактирования.

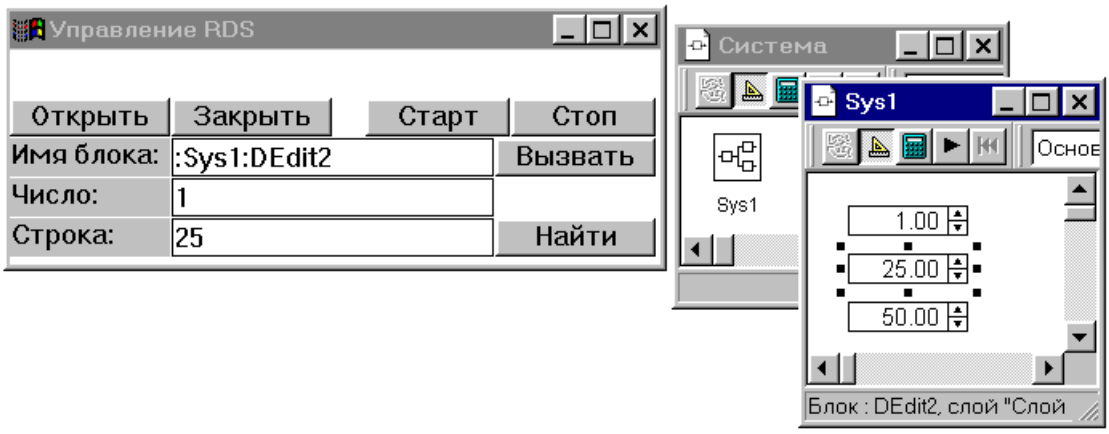


Рис. 123. Внешняя установка значения поля ввода (полю “:Sys1:DEdit2” передается значение 25)

Рассмотрим теперь модель блока, который может отвечать на вызовы внешних приложений. Пусть наш блок преобразует переданную ему строку в вещественное число, и вычисляет одну из четырех функций, в зависимости от полученного целого числа:

Полученное число	*Полученная строка*	*Функция*
0	Вещественное число	Синус переданного угла в градусах
1		Косинус переданного угла в градусах
2		*e* в степени переданного числа
3		Квадрат переданного числа

Результат вычислений блок должен преобразовать в строку и вернуть вызвавшему приложению. Кроме того, если функция вычислена, блок должен передать приложению число 1, если не вычислена (неверный номер функции, или полученная строка не преобразуется в вещественное число) – 0. Хотя этот пример и не имеет большой практической ценности, он вполне подходит для иллюстрации механизма обработки вызовов внешних приложений моделью блока.

Статические переменные в этом блоке использоваться не будут, поэтому его модель будет состоять только из реакции на вызов от внешнего приложения RDS_BFM_REMOTEMSG:

```
// Реакция на вызов внешнего приложения
extern "C" __declspec(dllexport)
  int RDSCALL RemoteFunction(int CallMode,
           RDS_PBLOCKDATA BlockData,
           LPVOID ExtParam)
{ RDS_REMOTEMSGDATA *msgdata;
  double val;
  char *str;

  switch(CallMode)
    { // Вызов от управляющей программы
      case RDS_BFM_REMOTEMSG:
        // Приведение ExtParam к правильному типу
        msgdata=(RDS_REMOTEMSGDATA*)ExtParam;
        // Преобразование полученной строки в число
        rdsAtoD(msgdata->String,&val);
        // Вычисление функции
        if(val!=DoubleErrorValue)
          switch(msgdata->Value)
            { case 0: val=sin(M_PI*val/180.0); break;
              case 1: val=cos(M_PI*val/180.0); break;
              case 2: val=exp(val); break;
              case 3: val=val*val; break;
              default: val=DoubleErrorValue;
            }
        // Преобразование результата в строку
        str=rdsDtoA(val,-1,NULL);
        // Запоминание этой строки в качестве ответа
        rdsRemoteReply(str);
        // Строка больше не нужна - освобождаем
        rdsFree(str);
        // Возвращаем 1, если функция выполнена
        return val!=DoubleErrorValue?1:0;
    }
  return RDS_BFR_DONE;
}
//=======================================
```

Когда управляющее приложение вызывает какой-либо блок схемы, его модель вызывается с параметром RDS_BFM_REMOTEMSG, а в ExtParam передается указатель на структуру RDS_REMOTEMSGDATA, содержащую полученные от внешнего приложения данные:

```
typedef struct
{ LPSTR String;            // Полученная от приложения строка
  int   Value;             // Полученное от приложения число
  LPSTR ControllerName;    // Название управляющего приложения,
                           // если оно его сообщило
} RDS_REMOTEMSGDATA;
typedef RDS_REMOTEMSGDATA *RDS_PREMOTEMSGDATA;
```

В этой структуре нас интересуют поля Value и String, содержащие полученные от внешнего приложения целое число и строку соответственно. В поле ControllerName содержится строка названия управляющего приложения, если это приложение сообщило свое название вызовом функции rdsctrlSetControllerName из библиотеки RdsCtrl.dll. Название приложения может быть произвольной строкой, обычно оно используется в тех случаях, когда один и тот же блок схемы должен выполнять разные действия в зависимости от того, какое именно приложение им управляет. В этом примере мы не даем управляющему приложению названия и не используем это поле.

Реагируя на вызов, модель блока приводит параметр ExtParam к типу “указатель на RDS_REMOTEMSGDATA” и записывает этот указатель в переменную msgdata, после чего преобразует полученную строку msgdata->String в вещественное число val сервисной функцией РДС rdsAtoD. Далее, если val не равно специальному значению DoubleErrorValue, использующемуся как индикатор ошибок математических функций (см. стр. 44), модель вычисляет одну из четырех заложенных в нее функций в зависимости от принятого целого числа msgdata->Value. Результат вычислений записывается в ту же самую переменную val. После этого вычисленное значение преобразуется в динамически отводимую строку str сервисной функцией rdsDtoA, и эта строка передается в функцию rdsRemoteReply.

Функция rdsRemoteReply служит для возврата строки управляющему приложению. Она не передает эту строку немедленно, вместо этого она запоминает ее во внутреннем буфере РДС. Когда функция модели завершится, эта строка вместе с возвращенным функцией модели целым значением будет передана управляющей программе. Таким образом, в реакции на событие RDS_BFM_REMOTEMSG модель блока устанавливает передаваемую управляющей программе строку функцией rdsRemoteReply, а передаваемое целое число возвращает оператором return. Вызов rdsRemoteReply в любых других реакциях модели игнорируется, его имеет смысл делать только в реакции на вызов управляющего приложения. Если в реакции RDS_BFM_REMOTEMSG эта функция будет вызвана несколько раз, внешняя программа получит только строку, переданную в самом последнем вызове. Если же rdsRemoteReply не будет вызвана ни разу, внешняя программа получит пустую строку.

После того, как строка str передана в функцию rdsRemoteReply и запомнена для возврата вызвавшему блок приложению, эта строка уничтожается функцией rdsFree (она была отведена динамически функцией rdsDtoA, поэтому ее обязательно нужно уничтожить вручную). Затем функция модели возвращает значение 1, если val отличается от индикатора ошибки DoubleErrorValue, и 0 в противном случае – это значение будет передано вызвавшей программе в качестве возвращенного блоком целого числа.

Теперь, если подключить эту модель к какому-либо блоку в схеме и загрузить эту схему в копию РДС, управляемую нашей программой, можно будет вызывать его по имени. Если ввести имя этого блока в поле ввода “Имя блока”, цифру 3 в поле ввода “Число”, число 12 в поле ввода “Строка”, и нажать кнопку “Вызвать”, в верхней части окна программы отобразится строка, возвращенная блоком – число 144, то есть квадрат двенадцати (рис. 124).

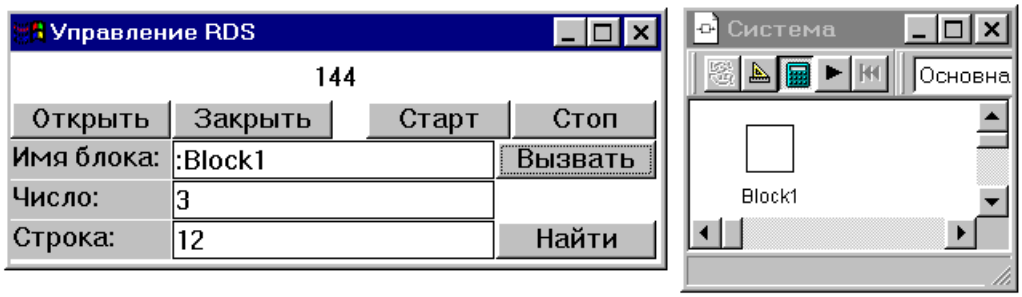


Рис. 124. Вызов блока из внешнего приложения

Для того, чтобы вызвать какой-либо блок схемы, управляющему приложению необходимо знать его полное имя. Это требует некоторой синхронизации между приложением и загружаемыми схемами (приложение должно помнить, в какой схеме какой блок за что отвечает), и при перемещении блоков из подсистемы в подсистему эта синхронизация может нарушиться, поскольку полное имя блока при этом меняется. Чтобы избежать путаницы с именами, можно дать каждой операции, выполняемой блоками по команде от внешнего приложения, какое-либо символическое имя. Если блок сообщит РДС, что он выполняет операцию с конкретным именем, управляющее приложение сможет найти его по имени этой операции независимо от того, в какой подсистеме он находится и как называется. При этом обычно выполняется следующая последовательность действий:

- Модель блока вызывает сервисную функцию `rdsExecutesRemoteOpsSet`, передавая ей строку с символическим именем выполняемой блоком операции и параметр `TRUE`. РДС запоминает этот факт, и будет помнить его до тех пор, пока блок не будет удален или пока модель не вызовет `rdsExecutesRemoteOpsSet` с тем же именем и параметром `FALSE`.
- Управляющая программа, загрузив схему, вызывает функцию `rdsctrlFindOpSetProviders` и передает ей символическое имя операции, получая в ответ список полных имен всех блоков, которые заявили о выполнении операции с этим именем.
- Выбрав из этого списка один или несколько блоков, управляющая программа вызывает их по именам.

Следует помнить, что поиск блоков, поддерживающих операцию с каким-либо именем, занимает некоторое время, поэтому не следует вызывать функцию `rdsctrlFindOpSetProviders` слишком часто. Лучше всего вызвать ее сразу после загрузки схемы или после переключения в режим моделирования, в котором изменения схемы маловероятны. Запомнив имя блока, поддерживающего операцию, можно вызывать его до тех пор, пока очередной вызов не вернет ошибочное значение (например, число –1, которое возвращается, когда блок не найден). При этом можно снова вызвать `rdsctrlFindOpSetProviders` и запомнить новое имя блока, и т.д.

Функция `rdsctrlFindOpSetProviders` позволяет искать блоки, поддерживающие операцию, во всей схеме или в конкретных подсистемах. Функция принимает следующие параметры:

```
int RDSCALL rdsctrlFindOpSetProviders(
      int link,             // Идентификатор связи с РДС
      LPSTR FullBlockName,  // Имя подсистемы, в которой искать
      LPSTR OpSetName,      // Символическое имя операции
      DWORD Flags,          // Флаги RDSCTRL_FOSP_*
      LPVOID ReturnStr);    // Объект для возврата имен блоков
```

В параметре `link` передается идентификатор связи с РДС. Строка с символическим именем операции передается в параметре `OpSetName`. В параметре `FullBlockName` передается полное имя подсистемы, начиная с которой нужно искать блоки. Если их нужно искать,

начиная с корневой подсистемы, в этом параметре передается пустая строка. В параметре `Flags` указываются битовые флаги, управляющие поиском блоков: если нужно искать блоки не только в подсистеме `FullBlockName`, но и во всех вложенных в нее подсистемах, указывается флаг `RDSCTRL_FOSP_RECURSIVE`. Если нужно проверить на поддержку искомой операции не только блоки подсистемы, но и модель самой подсистемы `FullBlockName` при ее наличии, указывается флаг `RDSCTRL_FOSP_SELF`. Наконец, в параметре `ReturnStr` передается указатель на объект, в который зарегистрированная в библиотеке RdsCtrl.dll функция возврата строк помещает список найденных блоков (имена в этом списке разделены кодом перевода строки "\n"). Сама функция возвращает число имен в списке – таким образом, если в схеме не найдено ни одного блока, выполняющего операцию с данным именем, функция вернет 0.

Назовем вычисления, выполняемые созданным нами блоком, "ProgrammersGuide.MathFunc1", и добавим в модель объявление о поддержке этой операции. Проще всего сделать это при инициализации блока, для этого в оператор `switch` в модели нужно добавить новый `case`:

```
// Инициализация
case RDS_BFM_INIT:
  rdsExecutesRemoteOpsSet("ProgrammersGuide.MathFunc1",TRUE);
  break;
```

Теперь внешняя программа сможет найти этот блок по имени выполняемой операции.

Поиск блока будем выполнять при нажатии на кнопку "Найти" (см. рис. 121), при этом в поле ввода "Строка" будем вводить имя искомой операции. Для этого перепишем ранее созданную пустую функцию, вызываемую при нажатии на эту кнопку, следующим образом:

```
// Нажатие кнопки "Найти"
void FindFuncClick(void)
{ char funcname[1000], // Массив для имени операции
       *s;
  TDynString list;    // Сюда запишется возвращаемый список имен

  if(RdsLink<0) // Нет связи с РДС
    return;

  // Копируем строку из поля ввода "Строка" в массив funcname
  GetDlgItemText(MainWin,IDC_STRINGEDIT,funcname,
                           sizeof(funcname)-1);

  // Ищем блоки, поддерживающие эту функцию
  if(!rdsctrlFindOpSetProviders(
          RdsLink,   // Связь с РДС
          "",        // Начиная с корневой подсистемы
          funcname,  // Имя операции
          RDSCTRL_FOSP_RECURSIVE, // С поиском во вложенных
          &list))    // Сюда запишется возвращаемый список
    { DisplayText("Функция не поддерживается");
      return;
    }

  // В строке list может быть список имен блоков.
  // Оставляем только первый его элемент
  if(list.c_str==NULL) // Нет строки
    return;
  // Ищем первый разделитель (перевод строки)
  s=strchr(list.c_str,'\n');
```

```
    // Заменяем найденный разделитель на 0 (конец текста)
    if(s) *s=0;

    // Заносим первое имя из списка в поле ввода имени блока
    SetDlgItemText(MainWin,IDC_BLKNAMEEDIT,list.c_str);
  }
  //=======================================
```

В этой функции мы читаем название операции из поля ввода “Строка” с идентификатором `IDC_STRINGEDIT` и записываем его в массив `funcname`. Затем мы вызываем функцию `rdsctrlFindOpSetProviders`, которая должна найти во всей схеме (вместо имени подсистемы передана пустая строка и указан флаг `RDSCTRL_FOSP_RECURSIVE`) блоки, выполняющие эту операцию, и записать список их имен в переменную `list` типа `TDynString`. Нас интересует только один блок, а в списке может быть много имен, поэтому мы ищем в списке первый разделитель “\n” и заменяем его нулем, тем самым отсекая все остальные имена в списке. Теперь в `list.c_str` находится указатель на имя первого блока из списка, его мы и копируем в поле ввода “Имя блока” функцией Windows API `SetDlgItemText`.

Если теперь ввести в поле “Строка” текст “ProgrammersGuide.MathFunc1” и нажать кнопку “Найти”, в поле ввода “Имя блока” должно появиться имя нашего блока (рис. 125). После этого мы сможем вызывать этот блок, поскольку теперь мы знаем его имя.

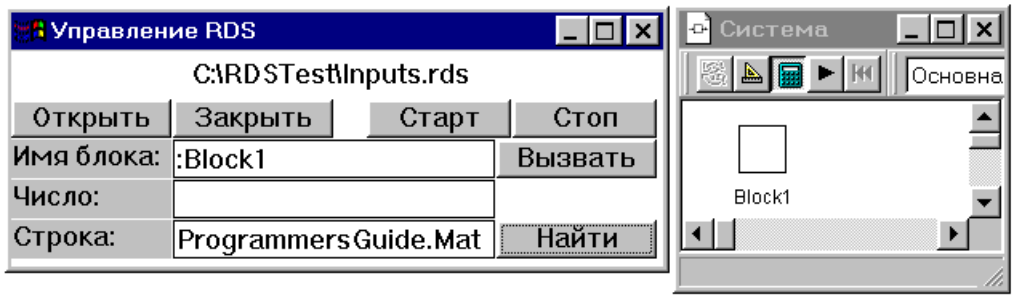


Рис. 125. Поиск блока, выполняющего операцию, имя которой указано в поле ввода “строка”

Есть еще один механизм взаимодействия управляющего приложения с блоками схемы, на котором мы не будем подробно останавливаться: приложение может передать в РДС произвольное количество строк с целыми идентификаторами при помощи функции `rdsctrlSetString`, а блоки в любой момент могут считать эти строки сервисной функцией `rdsGetRemoteControllerString`. Например, если наше приложение выполнит вызов `rdsctrlSetString(RdsLink,23,"abcd")`, модель блока, вызвав функцию `rdsGetRemoteControllerString(23)`, получит строку “abcd”. Этот механизм не предназначен для немедленной передачи информации блокам, поскольку модели блоков не вызываются при передаче новых строк. Обычно он используется для установки каких-либо глобальных параметров, влияющих на все блоки схемы. Функцию `rdsctrlSetString` можно вызывать независимо от того, работает ли сейчас процесс РДС. Как и многие другие параметры, переданные строки запоминаются во внутренних структурах библиотеки RdsCtrl.dll и передаются в РДС при запуске.

§3.4. Реакция на события и сообщения от блоков

Рассматривается вызов специальным образом зарегистрированных функций управляющей программы при возникновении различных событий в РДС и в ответ на вызовы от блоков загруженной схемы. В созданную ранее модель поиска угла возвышения метательной машины (§2.14.2) добавляются вызовы и реакции, позволяющие схеме с этой моделью обмениваться данными с управляющей программой. Дополнительно рассматривается пример блока-кнопки, сообщающего управляющей программе о щелчке на его изображении.

До сих пор мы рассматривали примеры, в которых управляющее приложение вызывало различные функции РДС (переключения режимов, загрузки схемы и т.п.) или моделей блоков. Теперь рассмотрим обратную ситуацию: часто требуется, чтобы в ответ на какие-либо события в РДС (например, действия пользователя, или завершение расчета) вызывалась функция в управляющем приложении.

Допустим, например, что управляющей программе нужно выполнить в РДС какой-либо расчет и получить его результаты. Для этого она должна запустить РДС, загрузить схему, запустить расчет, дождаться его конца и считать результаты из переменных каких-либо блоков. Основная проблема здесь заключается в том, что программа должна дождаться окончания расчета, то есть каким-то образом узнать о его остановке. Она могла бы вызывать в цикле функцию `rdsctrlGetMode` для получения текущего режима работы РДС, дожидаясь перехода из режима расчета в режим моделирования, однако это привело бы к постоянному обмену данными между процессами управляющего приложения и РДС, что отрицательным образом сказалось бы на производительности обеих программ. Кроме того, приложению пришлось бы сочетать выполнение цикла опроса режима РДС с обслуживанием собственного пользовательского интерфейса, иначе пользователю показалось бы, что программа “зависла”.

Библиотека RdsCtrl.dll предоставляет управляющей программе более удобную возможность: программа может зарегистрировать в библиотеке функцию специального вида и указать, при наступлении каких событий в РДС эта функция должна вызываться. В рассмотренном выше случае программа, ожидающая конца расчета, зарегистрировала бы функцию реакции на остановку расчета, после чего запустила бы расчет и продолжала заниматься своими делами. При остановке расчета эта функция вызвалась бы автоматически, и считывание результатов расчета можно было бы выполнять внутри нее.

Добавим в рассматриваемый нами в этой главе пример программы реакцию на три события: запуск расчета, остановку расчета и завершение РДС. Мы будем регистрировать реакции на эти события функцией `rdsctrlRegisterEventStdCallback` (это только одна из функций регистрации, но другие мы не будем здесь рассматривать):

```
void RDSCALL rdsctrlRegisterEventStdCallback(
      int Link,                  // Связь с РДС
      int Event,                 // Идентификатор события
      RDSCTRL_CALLBACK CallBack, // Указатель на функцию
      LPVOID AuxData);           // Дополнительные данные
```

В параметре `Link`, как обычно, передается идентификатор связи с управляемой копией РДС, в параметре `Event` – идентификатор события, на которое регистрируется реакция для данной связи (это одна из констант `RDSCTRLEVENT_*`, описанных в файле “RdsCtrl.h”). В параметре `CallBack` передается указатель на регистрируемую функцию, которая будет вызываться при получении от РДС информации о наступлении данного события для данной связи. И, наконец, в параметре `AuxData` вызвавшая программа может передать указатель на что угодно (например, на какую-либо структуру в памяти). Этот указатель просто запоминается вместе с указателем на зарегистрированную функцию, и передается ей при каждом вызове.

Функции реакции на стандартные события имеют следующий вид:

```
void RDSCALL имя_функции_реакции(
      int Link,        // Связь с копией РДС
```

```
        int Event,       // Идентификатор события
        LPVOID EvData,   // Данные события
        LPVOID AuxData);// Дополнительный параметр
```

В параметре `Link` функция получает идентификатор связи с копией РДС, в которой произошло событие, а в параметре `Event` – идентификатор этого события. Таким образом, одну и ту же функцию можно использовать для реакции на разные события в разных управляемых копиях РДС – функция сможет разобраться, где и что произошло. В параметре `EvData` функция получает указатель на структуру с описанием данных события. Поскольку разным событиям соответствуют разные структуры, а функция реакции для них имеет один и тот же формат, этот указатель приведен к общему типу `void*`, и функция должна самостоятельно привести его к нужному типу в зависимости от значения параметра `Event` (точно так же в РДС указатели на разные по типу данные передаются в функции моделей блоков через параметр `ExtParam` типа `LPVOID`). Наконец, в параметре `AuxData` функция получает тот самый указатель, который был передан в последнем параметре `rdsctrlRegisterEventStdCallback` при ее регистрации. Так можно давать функции реакции доступ к каким-либо данным управляющей программы, не делая их глобальными переменными: достаточно объединить их в структуру и передать указатель на нее при регистрации функции реакции на событие.

Мы опишем две функции реакции: одна из них будет вызываться при завершении РДС и выводить сообщение “Процесс РДС завершился”, другая – при запуске и остановке расчета, и выводить сообщения “Расчет запущен” и “Расчет остановлен” в зависимости от значения параметра `Event`. Дополнительный указатель `AuxData` мы использовать не будем, как и идентификатор связи с РДС `Link` – связь у нас единственная, и нам не нужно разбираться, по какой из работающих связей пришла информация о событии. Функция, которая будет вызываться при завершении РДС, выглядит так:

```
// Отклик на событие - завершение РДС
void RDSCALL RdsExitEvent(int Link,int Event,
                  LPVOID EvData,LPVOID AuxData)
{
  DisplayText("Процесс РДС завершился");
}
//=========================================
```

В этой функции мы вообще не используем переданные параметры, мы просто выводим сообщение вызовом `DisplayText`. Более сложное приложение могло бы в этой реакции, например, перезапустить РДС, но мы не будем этого делать.

Функция реакции на запуск и остановку расчета не будет существенно сложнее:

```
// Отклик на событие - запуск и остановка расчета
void RDSCALL CalcStartStopEvent(int Link,int Event,
                  LPVOID EvData,LPVOID AuxData)
{
  switch(Event)
    { case RDSCTRLEVENT_CALCSTART: // Запуск
        DisplayText("Расчет запущен");
        break;
      case RDSCTRLEVENT_CALCSTOP: // Остановка
        DisplayText("Расчет остановлен");
        break;
    }
}
//=========================================
```

Здесь, поскольку одна и та же функция будет запускаться и в ответ на запуск, и в ответ на остановку расчета, мы анализируем параметр `Event` и выводим сообщение, соответствующее произошедшему событию.

Функции реакций готовы – осталось зарегистрировать их. Ранее мы уже создали для этой цели пустую функцию RegisterEvents, которая вызывается сразу после загрузки библиотеки. Нужно добавить в нее следующие вызовы:

```
// Разрешение событий и регистрация функций реакций
void RegisterEvents(void)
{
  // Разрешить реакцию на события
  rdsctrlEnableEvents(RdsLink,TRUE);
  // Событие завершения РДС
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CONNCLOSED,RdsExitEvent,NULL);
  // Событие запуска расчета
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CALCSTART,CalcStartStopEvent,NULL);
  // Событие завершения расчета
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CALCSTOP,CalcStartStopEvent,NULL);
}
//=======================================
```

Сначала мы вызываем функцию rdsctrlEnableEvents с параметром TRUE, тем самым разрешая РДС передавать в наше приложение информацию о событиях. Если ее не вызвать, или вызвать с параметром FALSE, функции реакций не будут вызываться, несмотря на то, что мы их зарегистрировали – программа просто не узнает о событиях в РДС.

Затем мы три раза вызываем rdsctrlRegisterEventStdCallback для регистрации функций реакции на события RDSCTRLEVENT_CONNCLOSED (завершение процесса РДС), RDSCTRLEVENT_CALCSTART (запуск расчета) и RDSCTRLEVENT_CALCSTOP (остановка расчета). В последних двух случаях мы указываем одну и ту же функцию реакции CalcStartStopEvent.

Теперь, если мы запустим программу, загрузим в управляемую копию РДС какую-нибудь схему и будем запускать и останавливать расчет, в верхней части окна будут появляться сообщения, выведенные функцией реакции CalcStartStopEvent. При этом безразлично, будем ли мы запускать и останавливать расчет кнопками нашей программы, или кнопками РДС – информация о событиях будет все равно поступать в программу. Если закрыть окно главной подсистемы загруженной схемы, РДС завершится, и функция реакции RdsExitEvent выведет об этом сообщение.

Модели блоков могут сами генерировать события, вызывая зарегистрированную в управляющей программе функцию реакции и передавая ей целое число и строку. Такое событие называется “сообщением от блока”, и ему соответствует идентификатор RDSCTRLEVENT_BLOCKMSG. Реакция на сообщения от блоков используется очень часто. Фактически, если схемы РДС включаются в состав приложения, без таких сообщений бывает очень сложно обойтись. Через них можно передавать в управляющее приложение данные, полученные в процессе расчета (это удобнее, чем вручную считывать их из переменных разных блоков), нажатия на различные блоки-кнопки и т.д.

Для реакции на сообщение от блока можно использовать обычную функцию описанного выше вида, при этом имя вызвавшего событие блока и переданные им число и строка будут находиться в структуре RDSCTRL_BLOCKMSGDATA, указатель на которую передается в параметре EvData. Можно также зарегистрировать для этой реакции специализированную функцию следующего вида:

```
void RDSCALL имя_функции_реакции_на_сообщение(
     int Link,             // Связь с копией РДС
     LPSTR BlockName,      // Полное имя передавшего блока
     int IMsg,             // Переданное блоком целое число
```

```
        LPSTR SMsg,           // Переданная блоком строка
        LPVOID AuxData);      // Дополнительный параметр
```

Такая функция получает имя вызвавшего событие блока в параметре `BlockName`, а число и строку, которые он передал – в параметрах `IMsg` и `SMsg` соответственно. Как и обычная функция реакции, в параметре `AuxData` эта функция получает указатель, переданный при ее регистрации. Использование такой специализированной функции позволяет несколько упростить текст программы – здесь все необходимые данные сразу передаются в параметрах функции, и их не нужно "вытаскивать" из структуры, указатель на которую еще нужно привести к правильному типу.

Для регистрации специальной функции реакции на сообщение от блока используется функция `rdsctrlRegisterBlockMsgCallback`:

```
void RDSCALL rdsctrlRegisterBlockMsgCallback(
        int Link,        // Связь с копией РДС
        RDSCTRL_BMCALLBACK CallBack,    // Указатель на функцию
        LPVOID AuxData);// Дополнительные данные
```

В параметре `Link` передается идентификатор связи, для сообщений которой регистрируется функция, в параметре `CallBack` – указатель на регистрируемую функцию реакции, а в параметре `AuxData` – произвольный указатель, который будет передаваться в одноименном параметре зарегистрированной функции при каждом вызове.

Добавим в нашу программу реакцию на сообщения от блоков. Сначала напишем функцию реакции, которая будет помещать имя передавшего сообщение блока и полученные от него число и строку в одноименные поля ввода в окне программы:

```
// Отклик на событие - сообщение от блока
void RDSCALL BlockMsgEvent(int Link,LPSTR BlockName,
        int IMsg,LPSTR SMsg,LPVOID AuxData)
{
  // Заносим имя блока в одноименное поле ввода
  SetDlgItemText(MainWin,IDC_BLKNAMEEDIT,BlockName);
  // Заносим IMsg в поле ввода "число"
  SetDlgItemInt(MainWin,IDC_VALUEEDIT,IMsg,TRUE);
  // Заносим полученный текст в поле ввода "строка"
  SetDlgItemText(MainWin,IDC_STRINGEDIT,SMsg);

  DisplayText("Получено сообщение от блока");
}
//=======================================
```

Дескриптор окна, которому принадлежат поля ввода, в которые мы заносим информацию, как обычно, берется из глобальной переменной `MainWin`. Для помещения текста и числа в поля ввода используются функции Windows API `SetDlgItemText` и `SetDlgItemInt` соответственно.

Эту функцию необходимо зарегистрировать, добавив вызов `rdsctrlRegisterBlockMsgCallback` в уже написанную нами ранее функцию `RegisterEvents`:

```
// Разрешение событий и регистрация функций реакций
void RegisterEvents(void)
{
  // Разрешить реакцию на события
  rdsctrlEnableEvents(RdsLink,TRUE);
  // Событие завершения РДС
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CONNCLOSED,RdsExitEvent,NULL);
  // Событие запуска расчета
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CALCSTART,CalcStartStopEvent,NULL);
```

```
  // Событие завершения расчета
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CALCSTOP,CalcStartStopEvent,NULL);
  // Сообщение от блока
  rdsctrlRegisterBlockMsgCallback(RdsLink,BlockMsgEvent,NULL);
}
//=======================================
```

Теперь, если модель какого-либо блока в схеме, загруженной в управляемую копию РДС, передаст сообщение управляющей программе, данные этого сообщения появятся в полях ввода, а в верхней части окна программы будет выведен текст “Получено сообщение от блока”.

Мы ввели в нашу программу реакцию на сообщения от блоков, но у нас пока нет блоков, которые передавали бы такие сообщения. Исправим эту ситуацию. Ранее (§2.14.2) мы создали несколько блоков, с помощью которых мы моделировали метательную машину, выпускающую снаряд с заданной начальной скоростью и углом возвышения, и подбирали угол возвышения для заданной дальности полета снаряда. По окончании расчета одна из этих моделей сообщала пользователю найденное значение угла и соответствующую ему величину промаха. Изменим эту модель так, чтобы при управлении из внешнего приложения она, вместо вывода сообщения пользователю, генерировала бы в управляющей программе событие и передавала ей найденные значения. Это позволит использовать схему с такой моделью как составную часть какой-нибудь расчетной программы: программа загрузит схему, запустит расчет и через некоторое время получит его результаты через функцию реакции на сообщение от блока, в которой сможет их обработать.

Поиском угла возвышения занимается модель `ArtSearch` (стр. 407), личная область данных которой оформлена как класс `TArtSearchData`. За вывод сообщения пользователю отвечает функция `ShowResults` в этом классе, именно в нее мы вставим передачу сообщения управляющему приложению. Поскольку мы можем передать только целое число и строку, найденный угол возвышения и получившийся промах мы преобразуем в двухстрочный текст вида

```
Angle=значение_угла
Miss=значение_промаха
```

Строки текста будут отделяться друг от друга кодом перевода строки “\n”. Этот текст будет передаваемой строкой, а в качестве целого числа мы всегда будем передавать ноль. Функцию `ShowResults` необходимо изменить следующим образом:

```
// Вывод сообщения о результатах поиска
void ShowResults(void)
{ char *str,
       *angle=rdsDtoA(OptAngle,-1,NULL),
       *miss=rdsDtoA(OptMiss,0,NULL);
  if(rdsHasRemoteController()) // Управляется извне
    { // Формрование динамической строки с текстом
      str=rdsDynStrCat("Angle=",angle,FALSE);
      rdsAddToDynStr(&str,"\nMiss=",FALSE);
      rdsAddToDynStr(&str,miss,FALSE);
      // Передача сообщения управляющей программе
      rdsRemoteControllerCall(0,str);
    }
  else // Работает самостоятельно
    { // Формрование динамической строки с сообщением
      str=rdsDynStrCat("Угол возвышения: ",angle,FALSE);
      rdsAddToDynStr(&str," гр.\nПромах: ",FALSE);
      rdsAddToDynStr(&str,miss,FALSE);
      rdsAddToDynStr(&str," м",FALSE);
```

```
            // Вывод текста
            rdsMessageBox(str,"Поиск завершен",MB_OK);
          }
        // Освобождение всех динамических строк
        rdsFree(str);
        rdsFree(angle);
        rdsFree(miss);
      };
```

Чтобы проверить, управляется ли РДС внешним приложением или работает самостоятельно, мы вызываем сервисную функцию `rdsHasRemoteController`, которая вернет `TRUE`, если схема загружена в копию РДС, которая в данный момент управляется извне. В этом случае мы формируем динамическую строку `str` с текстом указанного формата, в который подставляются значения угла возвышения `angle` и промаха `miss`. Затем вызывается функция передачи сообщения управляющей программе `rdsRemoteControllerCall`, в которую передаются число 0 и сформированная строка: эти параметры программа получит в своей функции реакции.

Если же `rdsHasRemoteController` вернет `FALSE`, значит, РДС в данный момент работает самостоятельно. В этом случае, как и прежде, формируется сообщение для пользователя.

Если загрузить схему для поиска угла (рис. 101) в управляемую копию РДС и запустить расчет, через некоторое время наша программа получит сообщение от блока поиска угла, как на рис. 126.

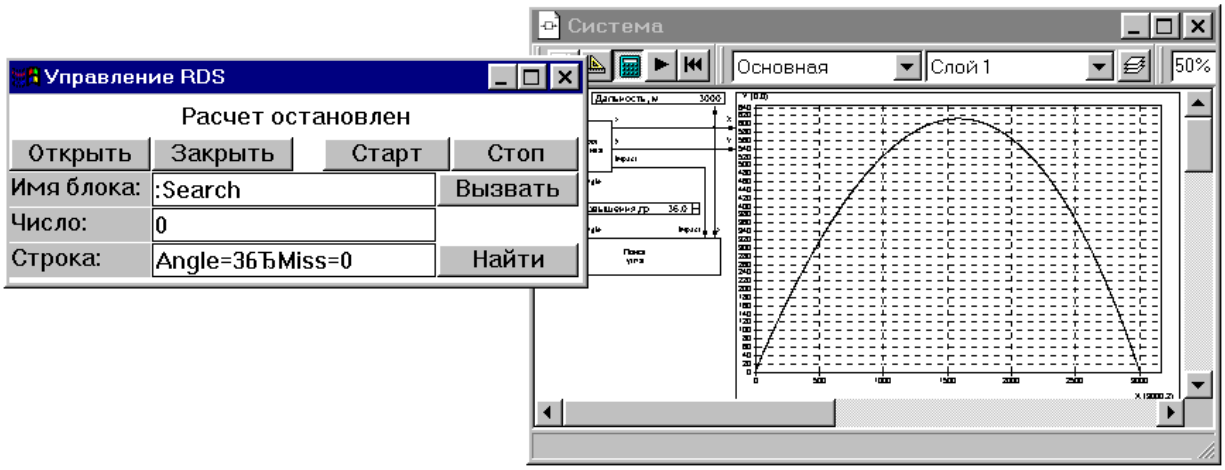


Рис. 126. Поиск угла возвышения с внешним управлением

В поле ввода имени блока окажется имя блока поиска угла, который передал сообщение (в данном случае – “:Search”), в поле “число” – полученное от блока целое число (ноль), а в поле “строка” – сформированный блоком текст с найденными значениями. Поля ввода, которые мы создали в нашей программе, не рассчитаны на многострочные тексты, поэтому на месте перевода строки отобразился специальный символ. В верхней части окна выведен текст “Расчет остановлен”, а не “Получено сообщение от блока”, поскольку после передачи сообщения блок поиска угла останавливает расчет, а в нашей программе отображается только самый последний текст из функции реакции.

Для того, чтобы сделать нашу схему полностью функциональной, необходимо добавить в блок поиска угла возможность установки параметров через внешнее управление в реакции на вызов `RDS_BFM_REMOTEMSG`. Чтобы можно было не привязываться к имени этого блока в схеме, назовем выполняемую им операцию “ProgrammersGuide.ArtSearch” и будем искать его функцией `rdsctrlFindOpSetProviders` (см. §3.3). Устанавливать

параметры блока будем функцией rdsctrlCallBlockFunctionEx, передавая блоку номер изменяемой переменной в целом параметре и ее значение в строковом.

Для того, чтобы объявить о поддержке блоком операции “ProgrammersGuide.ArtSearch”, нужно внести следующие изменения в конструктор класса TArtSearchData:

```
// Конструктор класса
TArtSearchData(void)
  { SelfReset=FALSE;
    Mode=ASMODE_READY;
    rdsExecutesRemoteOpsSet("ProgrammersGuide.ArtSearch",TRUE);
  };
```

В оператор switch в модели блока ArtSearch нужно добавить реакцию на вызов от управляющего приложения:

```
// Вызов от управляющей программы
case RDS_BFM_REMOTEMSG:
  // Устанавливаем значение переменной по умолчанию
  rdsSetBlockVarDefValueStr(
      BlockData->Block,     // Блок
      ((RDS_REMOTEMSGDATA*)ExtParam)->Value,  // Номер переменной
      ((RDS_REMOTEMSGDATA*)ExtParam)->String);// Значение
  break;
```

В этой реакции вызывается функция установки значения переменной по умолчанию (блок хранит свои параметры именно в них), в которой принятое от управляющей программы целое число трактуется как номер переменной, а строка – как ее значение. Таким образом, установке значения переменной MinAngle будет соответствовать число 2 (в списке переменных блока она третья, то есть имеет индекс 2 начиная с нуля), MaxAngle – 3, Accuracy – 4, Distance – 5. Теперь после загрузки схемы мы можем найти в ней блок, выполняющий операцию “ProgrammersGuide.ArtSearch”, и, вызывая его, установить в нем границы диапазона углов (переменные 2 и 3), точность установки угла (переменная 4) и требуемую дальность (переменная 5), запустить расчет и получить найденное значение угла возвышения для этой дальности через сообщение от блока. В таком виде модель блока поиска угла уже можно использовать для каких-либо расчетов.

Рассмотрим еще один пример: создадим блок, который будет сообщать вызывающей программе о щелчках мыши на его изображении. Такой блок можно использовать в качестве кнопки на каком-либо виртуальном пульте, работающем под управлением внешнего приложения. С каждым таким блоком будет связано целое число-идентификатор, которое он будет сообщать управляющей программе при щелчке – так программа сможет различать щелчки по разным блокам, не зная их имен. Чтобы программа могла понять, какая именно кнопка мыши была нажата, блок будет передавать строку “L” для левой кнопки, “R” для правой и “M” для средней.

Число-идентификатор мы будем хранить в переменной блока, и нам нужно будет предусмотреть функцию настройки, чтобы пользователь мог его вводить. Блок будет иметь следующую структуру переменных:

Смещение	*Имя*	*Тип*	*Размер*	*Вход/выход*	*Назначение*
0	Start	Сигнал	1	Вход	Стандартный сигнал запуска (здесь не используется)
1	Ready	Сигнал	1	Выход	Стандартный сигнал готовности (здесь не используется)
2	Value	int	4	Внутр.	Идентификатор блока

Модель блока будет иметь следующий вид:

```
// Нажатие кнопки мыши для ДУ
extern "C" __declspec(dllexport)
  int RDSCALL RemoteClick(int CallMode,
           RDS_PBLOCKDATA BlockData,
           LPVOID ExtParam)
{ char *str,*def;
  RDS_MOUSEDATA *mouse;
// Макроопределения для статических переменных
#define pStart ((char *)(BlockData->VarData))
#define Start (*((char *)(pStart)))   // 0
#define Ready (*((char *)(pStart+1))) // 1
#define Value (*((int *)(pStart+2)))  // 2
  switch(CallMode)
    { // Проверка типов статических переменных
      case RDS_BFM_VARCHECK:
        return strcmp((char*)ExtParam,"{SSI}")?
                RDS_BFR_BADVARSMSG:RDS_BFR_DONE;

      // Вызов функции настройки
      case RDS_BFM_SETUP:
        // Получаем значение Value по умолчанию в виде строки
        def=rdsGetBlockVarDefValueStr(BlockData->Block,2,NULL);
        // Вызываем окно ввода строки
        str=rdsInputString("Щелчок для ДУ",
                    "Идентификатор:",def,100);
        // Строка def больше не нужна
        rdsFree(def);
        if(str)
          { // Пользователь нажал "OK"
            rdsSetBlockVarDefValueStr(BlockData->Block,2,str);
            // Строка str больше не нужна
            rdsFree(str);
            return RDS_BFR_MODIFIED;
          }
        break;

      // Нажатие кнопки мыши
      case RDS_BFM_MOUSEDOWN:
        mouse=(RDS_MOUSEDATA*)ExtParam;
        // Определение нажатой кнопки
        switch(mouse->Button)
          { case RDS_MLEFTBUTTON:  str="L"; break;
            case RDS_MRIGHTBUTTON: str="R"; break;
            default: str="M";
          }
        // Передача сообщения управляющей программе
        rdsRemoteControllerCall(Value,str);
        break;
    }
  return RDS_BFR_DONE;
// Отмена макроопределений
#undef Value
#undef Ready
#undef Start
#undef pStart
}
//=========================================
```

Функция настройки в этой модели использует сервисную функцию ввода строки `rdsInputString` (см. стр. 119) – поскольку нам нужно вводить единственное значение, она нам вполне подходит. Целый идентификатор, который мы будем передавать управляющей программе при щелчке, хранится в значении по умолчанию переменной блока `Value` с индексом 2, для его чтения и записи используются функции `rdsGetBlockVarDefValueStr` и `rdsSetBlockVarDefValueStr` соответственно.

При щелчке на изображении блока вызывается реакция `RDS_BFM_MOUSEDOWN`. В ней значение идентификатора `Value` и строка, соответствующая нажатой кнопке, передается вызвавшему приложению функцией `rdsRemoteControllerCall` (при этом в приложении возникнет событие типа “сообщение от блока”). В отличие от предыдущего примера, здесь мы не вызываем функцию `rdsHasRemoteController` и не проверяем, работает ли РДС под управлением внешней программы. Функцию `rdsRemoteControllerCall` можно безопасно вызывать и при отсутствии связи с управляющей программой, в этом случае вызов будет проигнорирован.

Теперь можно создать новую схему, добавить в нее несколько блоков с моделью `RemoteClick`, разрешив для них реакцию на действия мышью (см. рис. 7), ввести в настройках этих блоков разные идентификаторы, и загрузить эту схему в управляемую нашей программой копию РДС. Теперь в режимах моделирования и расчета при щелчке на любом из этих блоков в верхней части окна программы будет появляться текст “Получено сообщение от блока”, в поле “число” – идентификатор блока, на котором щелкнул пользователь, а в поле “строка” – символ, соответствующий нажатой кнопке (рис. 127).

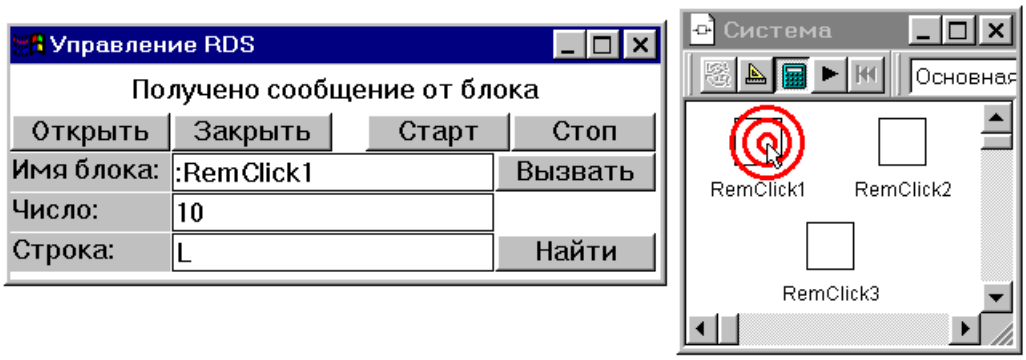


Рис. 127. Передача информации о щелчке мыши через событие

§3.5. Вмешательство в загрузку и сохранение схемы

Рассматриваются методы перехвата событий загрузки и сохранения схемы, которые позволяют управляющему приложению вмешаться в эти процессы для самостоятельной организации хранения схем (например, в базе данных). В пример программы, рассматриваемый в этой главе, добавляются функции, позволяющие сохранять схему в виде набора отдельных файлов, каждый из которых соответствует блоку или связи, и загружать схему из такого набора.

Управляющее приложение может отдать управляемой копии РДС команду загрузить схему из указанного файла – мы рассматривали функцию `rdsctrlLoadSystemFromFile` в §3.2. При необходимости, приложение может и сохранить загруженную схему в файл функцией `rdsctrlSaveSystemToFile`, например, если пользователь внес в нее какие-либо изменения. Кроме того, пользователь может сам загружать и сохранять схемы в управляемой копии РДС с помощью соответствующих пунктов главного меню. В рассматриваемом нами примере приложения показ главного окна РДС запрещен вызовом `rdsctrlShowMainWindow` с параметром `FALSE`, так что может показаться, что в нашем случае пользователь не сможет загружать и сохранять схемы, поскольку главное меню, как и

главное окно, ему недоступно. Однако, сочетания клавиш Ctrl+O и Ctrl+S, соответствующие этим пунктам главного меню, все равно будут работать даже при невидимом главном окне.

Тем не менее, разработчику управляющего приложения не всегда удобно хранить схемы в отдельно расположенных файлах, и, тем более, давать пользователю полную свободу загрузки и сохранения этих схем. Если, например, РДС в составе приложения используется для демонстрации схем соединения каких-либо объектов, все данные этих объектов (а, значит, и схемы их соединения) обычно хранятся в базе данных. Таким образом, если пользоваться файловыми функциями загрузки и сохранения, приложению придется выгружать схему из базы данных во временный файл, после чего давать РДС команду загрузить схему из этого файла. Если пользователю разрешено редактировать схему, приложению также придется записывать этот файл в базу данных после сохранения его пользователем. И, конечно, приложение должно будет запретить пользователю загружать в управляемую копию РДС “посторонние” схемы.

В библиотеке RdsCtrl.dll предусмотрено несколько способов облегчить жизнь разработчику управляющего приложения. Прежде всего, можно запретить сохранение схем и их загрузку из файлов функциями `rdsctrlNoDirectSave` и `rdsctrlNoDirectLoad` соответственно. В эти функции передается идентификатор связи с РДС и логический параметр, запрещающий (`TRUE`) или разрешающий (`FALSE`) работу с файлами схем. По умолчанию или после вызова `rdsctrlNoDirectLoad` с параметром `FALSE` РДС будет работать в обычном режиме: при выборе пункта меню “загрузить” или нажатии Ctrl+O появится стандартный диалог открытия файла, в котором пользователь сможет выбрать файл схемы для загрузки. Если же вызвать `rdsctrlNoDirectLoad` с параметром `TRUE`, вместо показа пользователю диалога открытия РДС будет вызывать в управляющем приложении событие `RDSCTRLEVENT_LOADREQ`. Реагируя на это событие, приложение сможет само показать диалог открытия или дать пользователю выбрать схему каким-либо другим способом (например, предъявив ему список схем, хранящихся в базе данных), после чего оно сможет передать выбранную схему в РДС через временный файл или одним из способов, которые будут описаны ниже.

Точно так же, после вызова `rdsctrlNoDirectSave` с параметром `TRUE`, вместо сохранения схемы РДС будет вызывать в управляющем приложении событие `RDSCTRLEVENT_SAVEFILE`, в ответ на которое приложение сможет получить содержимое схемы и записать его, например, в базу данных.

Передавать содержимое схемы из управляющего приложения в РДС и обратно можно не только через временный файл. Можно, например, использовать функцию `rdsctrlLoadSystemFromMem`, которая передает в РДС схему, которая находится в памяти приложения в текстовом формате, и парную к ней `rdsctrlGetSystemContent`, которая возвращает текст схемы, загруженной в данный момент в РДС, в виде строки. Эти функции несколько облегчают программирование, но, по своей сути, они мало отличаются от сохранения схемы в текстовом формате в одном приложении с последующей загрузкой этого текста в другом. Основной недостаток обоих этих способов загрузки и сохранения заключается в том, что при их использовании вся схема рассматривается как единый объект, разбить который на отдельные блоки и связи управляющее приложение сможет только с помощью разбора текстового формата РДС, что представляет собой отдельную, достаточно объемную, задачу. В то же время, разработчику приложения часто удобнее считать каждый блок схемы отдельным объектом, и хранить в базе данных не только описание этого блока, необходимое для его работы в схеме в РДС, но и различные дополнительные данные, к которым модель блока сможет обращаться при работе. В RdsCtrl.dll предусмотрен набор функций для поблочного сохранения схем, позволяющих получать информацию о загруженной в РДС схеме блок за блоком, связь за связью, при этом для каждого блока и каждой связи сообщается целый идентификатор, уникальный в пределах этой схемы. Этот

идентификатор можно использовать, например, в ключе какой-либо таблицы базы данных, в которой хранится информация, связанная с этим блоком. Идентификаторы ранее не упоминались в примерах, поскольку при написании моделей блоков они практически не используются. Основное их назначение – однозначная идентификация блока или связи в схеме при внешнем управлении. Идентификаторы блоков и связей в схеме не могут совпадать, то есть в схеме не может быть блока и связи с одинаковыми идентификаторами. Корневая подсистема схемы всегда имеет идентификатор 0.

Для того, чтобы начать поблочное сохранение схемы, управляющее приложение должно вызвать функцию `rdsctrlStartBlockByBlockSave`, как всегда указав в ее параметрах идентификатор связи с той копией РДС, откуда нужно будет получать данные. Эта функция подготовит RdsCtrl.dll к передаче информации о загруженной схеме и заполнит все внутренние структуры данных, необходимые для этой передачи. Результатом возврата функции будет общее число блоков данных (описаний блоков, описаний связей и вспомогательных), из которых состоит загруженная схема – таким образом, возврат нуля сигнализирует о невозможности получения информации о схеме (например, если схема не загружена).

Затем управляющее приложение должно в цикле вызывать функцию `rdsctrlGetBlockByBlockSavePiece`, каждый вызов которой будет возвращать описание очередного блока или связи схемы. Порядок следования этих описаний определяется РДС и не может быть изменен, задача приложения – обрабатывать полученное им описание необходимым ему способом (например, записывая его в таблицу базы данных). Функция `rdsctrlGetBlockByBlockSavePiece` принимает следующие параметры:

```
int RDSCALL rdsctrlGetBlockByBlockSavePiece(
      int link,             // Связь с РДС
      DWORD *pExtId,        // Возвращаемый идентификатор объекта
      DWORD *pParentId,     // Возвращаемый идентификатор родителя
      LPVOID text);         // Возвращаемый текст описания
```

В параметре `link` передается идентификатор связи с копией РДС, данные схемы которой запрашиваются. В параметре `pExtId` передается указатель на целую переменную типа `DWORD`, в которую функция запишет идентификатор возвращаемого объекта, если это блок или связь. В параметре `pParentId` передается такой же указатель на `DWORD`, через который функция возвращает идентификатор родительской подсистемы возвращаемого объекта – эта информация может потребоваться приложению для организации базы данных. Наконец, в параметре `text` передается указатель на объект для хранения строк, в который функция запишет текстовое описание возвращаемого объекта при помощи функции обратного вызова, ранее зарегистрированной вызовом `rdsctrlSetStringCallback` (см. стр. 518). Возвращает функция целый тип объекта, который указывает, описание чего именно функция только что передала приложению. Тип может принимать одно из следующих однобайтовых значений (эти константы описаны в “RdsDef.h”):

Константа	***Зна-чение***	***Объект***	****pExtId***	****pParentId***
`RDS_SFTAG_ROOT`	`0x15`	Корневая подсистема	0	0
`RDS_SFTAG_SIMPLEBLOCK`	`0x10`	Простой блок	Идентификатор объекта (*ИО*)	Идентифи-катор родительской подсистемы (*ИРП*)
`RDS_SFTAG_SYSTEM`	`0x11`	Подсистема	*ИО*	*ИРП*
`RDS_SFTAG_INPUTBLOCK`	`0x12`	Внешний вход	*ИО*	*ИРП*

Константа	*Значение*	*Объект*	**pExtId*	**pParentId*
RDS_SFTAG_OUTPUTBLOCK	0x13	Внешний выход	*ИО*	*ИРП*
RDS_SFTAG_BUSPORT	0x14	Ввод шины	*ИО*	*ИРП*
RDS_SFTAG_CONNECTION	0x20	Связь	*ИО*	*ИРП*
RDS_SFTAG_BUS	0x21	Шина	*ИО*	*ИРП*
RDS_SFTAG_CONNSTYLES	1	Стили связей	0	0
RDS_SFTAG_TYPES	2	Описание структур	0	0
RDS_SFTAG_EOF	0	Конец данных	0	0

Следует отметить, что для корневой подсистемы схемы предусмотрен отдельный тип RDS_SFTAG_ROOT. Это связано с тем, что обычно управляющее приложение обрабатывает корневую подсистему отдельно и связывает с ней информацию, относящуюся ко всей схеме в целом. Для корневой подсистемы не возвращается идентификатор родительской подсистемы (точнее, возвращается нулевое значение), поскольку родительская подсистема у нее отсутствует.

Типы RDS_SFTAG_CONNSTYLES и RDS_SFTAG_TYPES указывают на возврат вспомогательных данных – набора описаний стилей связей и набора структур соответственно. Приложение может не сохранять эти данные, поскольку вся информация о внешнем виде связей и структурах, используемых блоками, содержится в описаниях этих связей и блоков. Описания стилей связей и структур всей схемы имеет смысл сохранять, только если пользователь будет редактировать схему и создавать новые блоки и связи. Например, если пользователь захочет добавить в свой блок переменную типа “Complex”, описание структуры “Complex” должно быть в наборе структур схемы. Оно может оказаться там в двух случаях: либо если в схеме уже есть блок, использующий эту структуру (тогда она добавится в общий список структур схемы автоматически), либо если эта структуры когда-то была добавлена в общий набор структур схемы и этот набор был загружен вместе со схемой.

Управляющее приложение должно вызвать функцию получения описания очередного объекта схемы rdsctrlGetBlockByBlockSavePiece либо по числу блоков данных в схеме (это число возвращается функцией rdsctrlStartBlockByBlockSave), либо до тех пор, пока она вместо типа объекта не вернет константу RDS_SFTAG_EOF. После этого необходимо вызвать функцию rdsctrlEndBlockByBlockSave для завершения поблочного сохранения схемы и освобождения памяти, которую RdsCtrl.dll отвела под внутренние структуры данных (для больших схем эти структуры могут занимать достаточно много места, поэтому лучше вызывать эту функцию сразу после обработки последнего принятого блока данных). Таким образом, обычно процедура поблочного сохранения выглядит примерно следующим образом (считаем, что RdsLink – идентификатор связи с работающей копией РДС):

```
TDynString objtext;   // Объект для хранения текста из
                      // рассмотренных выше примеров
if(!rdsctrlStartBlockByBlockSave(RdsLink,0))
  { // ... ошибка – получение данных схемы невозможно ...
  }
else
  { DWORD id,parentid;
    for(;;)
      { // Получаем очередной блок данных
        int type=rdsctrlGetBlockByBlockSavePiece(
                    RdsLink,&id,&parentid,&objtext);
```

```
            if(type==0) // RDS_SFTAG_EOF – данные кончились
              break;
            // ... сохранение объекта типа type с идентификатором
            // id, идентификатором родителя parentid
            // и описанием objtext ...
          }
        // Завершаем прием данных из РДС
        rdsctrlEndBlockByBlockSave(RdsLink);
      }
```

Вместо сравнения полученного типа объекта с нулем (RDS_SFTAG_EOF) и прерывания бесконечного цикла for(;;) в случае совпадения, можно было бы использовать в качестве верхнего предела цикла число, возвращенное rdsctrlStartBlockByBlockSave – здесь разработчик приложения может поступать так, как ему удобнее.

Поблочная загрузка схемы в работающую копию РДС производится примерно так же, как и сохранение. Сначала нужно подготовить RdsCtrl.dll к приему данных о схеме вызовом rdsctrlStartBlockByBlockLoad, затем передать каждый объект по отдельности вызовами rdsctrlSetBlockByBlockLoadPiece (переданная информация запоминается во внутренней памяти RdsCtrl.dll), после чего завершить загрузку вызовом rdsctrlEndBlockByBlockLoad – в этот момент накопленная в памяти RdsCtrl.dll информация будет передана в РДС для формирования схемы.

Функция rdsctrlSetBlockByBlockLoadPiece принимает три параметра: идентификатор связи с РДС, тип передаваемого объекта (одну из констант RDS_SFTAG_*) и строку его описания. Порядок передачи описаний различных объектов может быть произвольным (можно передать описание блока до того, как передано описание его родительской подсистемы, и т.п.) Идентификатор объекта, как и идентификатор родительской подсистемы, не передается – вся эта информация содержится в тексте описания самого объекта. Таким образом, управляющая программа не может при передаче схемы в РДС поменять идентификаторы объектов – это может показаться искусственным ограничением, но разрешение смены идентификаторов могло бы привести к нарушению их уникальности и проблемам в работе схемы. Следует помнить, что идентификаторы блокам и связям присваиваются внутри РДС, и ни управляющая программа, ни модели блоков никак не могут повлиять на этот процесс.

Функция rdsctrlEndBlockByBlockLoad возвращает логическое значение, указывающее на успешность передачи схемы в РДС, и принимает два параметра:

```
BOOL RDSCALL rdsctrlEndBlockByBlockLoad(
      int link,       // Связь с РДС
      BOOL apply);    // Передать (TRUE) или отменить (FALSE)
```

В параметре apply передается TRUE, если из всех переданных объектов необходимо сформировать схему, или FALSE, если нужно отменить передачу и уничтожить все накопленные в памяти данные. Процедура поблочной загрузки выглядит примерно так (RdsLink – идентификатор связи с РДС):

```
// Начинаем передачу данных объектов
rdsctrlStartBlockByBlockLoad(RdsLink,0);
// Передаем данные всех объектов в цикле
for(...для всех объектов...)
  rdsctrlSetBlockByBlockLoadPiece(RdsLink,тип,строка_описания);
if(!rdsctrlEndBlockByBlockLoad(RdsLink,TRUE))
  { // ... ошибка сборки схемы ...
  }
```

Добавим в рассмотренный в этой главе пример управляющего приложения возможность поблочного сохранения и загрузки схемы. Мы не будем вводить в пример работу с базой данных – это неоправданно усложнило бы его. Вместо этого мы поступим следующим образом: каждый объект схемы мы будем сохранять в отдельный текстовый файл, и, кроме

того, список имен файлов всех объектов тоже запишем в файл, по одному имени файла на строку. Это даст представление о работе больших приложений, которые для хранения объектов схемы вместо файлов могут использовать записи в таблицах базы данных.

Чтобы не добавлять в программу новые кнопки для вызова функций поблочного сохранения и загрузки схемы, мы запретим в РДС сохранение и загрузку схем и введем в программе реакцию на на события RDSCTRLEVENT_LOADREQ и RDSCTRLEVENT_SAVEFILE. Таким образом, если пользователь нажмет Ctrl+O или Ctrl+S (мы запрещаем показ главного окна РДС, поэтому пункты меню “загрузить” и “сохранить” будут пользователю недоступны), вместо загрузки или сохранения схемы вызовется реакция в нашей программе, где мы и будем выполнять все действия для поблочной загрузки и сохранения.

Прежде всего, необходимо добавить в функцию RegisterEvents вызовы для запрещения загрузки и сохранения, а также регистрацию реакций на новые события:

```
// Разрешение событий и регистрация функций реакций
void RegisterEvents(void)
{
  // Разрешить реакцию на события
  rdsctrlEnableEvents(RdsLink,TRUE);
  // Событие завершения РДС
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CONNCLOSED,RdsExitEvent,NULL);
  // Событие запуска расчета
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CALCSTART,CalcStartStopEvent,NULL);
  // Событие завершения расчета
  rdsctrlRegisterEventStdCallback(RdsLink,
     RDSCTRLEVENT_CALCSTOP,CalcStartStopEvent,NULL);
  // Сообщение от блока
  rdsctrlRegisterBlockMsgCallback(RdsLink,BlockMsgEvent,NULL);
  // Запрет загрузки файла (вместо этого идет событие)
  rdsctrlNoDirectLoad(RdsLink,TRUE);
  // Запрет сохранения файла (вместо этого идет событие)
  rdsctrlNoDirectSave(RdsLink,TRUE);
  // Событие загрузки (нажатие "Ctrl+O")
  rdsctrlRegisterEventStdCallback(RdsLink,RDSCTRLEVENT_LOADREQ,
                LoadRequestCallback,NULL);
  // Событие сохранения (нажатие "Ctrl+S")
  rdsctrlRegisterEventStdCallback(RdsLink,RDSCTRLEVENT_SAVEFILE,
                SaveRequestCallback,NULL);
}
//=======================================
```

В функциях SaveRequestCallback и LoadRequestCallback, которые мы зарегистрировали в качестве реакций на новые события, мы будем показывать стандартный диалог выбора файла для сохранения или загрузки. Для этого весьма желательно сделать так, чтобы на момент появления этого диалога на экране наше приложение оказалось бы на переднем плане, иначе диалог окажется закрыт окнами подсистем РДС. Для этого мы напишем специальную функцию BringAppToTop:

```
// "Вытаскивание" приложения на передний план
void BringAppToTop(void)
{ HWND hCurrWnd;
  int iMyTID;
  int iCurrTID;
  hCurrWnd=GetForegroundWindow();
  iMyTID=GetCurrentThreadId();
```

```
  iCurrTID=GetWindowThreadProcessId(hCurrWnd,0);
  AttachThreadInput(iMyTID,iCurrTID,TRUE);
  SetForegroundWindow(MainWin);
  AttachThreadInput(iMyTID,iCurrTID,FALSE);
}
//=======================================
```

Эта функция обходит известную проблему Windows API, из-за которой в поздних версиях Windows неактивный (не обрабатывающий пользовательский ввод) процесс не может переместить свое окно на передний план. Здесь мы временно подключаем поток нашего процесса к обработке ввода вызовом функции `AttachThreadInput` с параметром `TRUE`, после чего перемещаем главное окно нашей программы (его дескриптор находится в глобальной переменной `MainWin`) на передний план функцией `SetForegroundWindow` – Windows уже не будет препятствовать этому. Затем мы снова отключаем наш поток от пользовательского ввода повторным вызовом `AttachThreadInput` с параметром `FALSE`. Более подробно останавливаться на этой функции мы не будем – желающие могут изучить описание функций Windows API и доступную информацию о проблемах с перемещением на передний план окна неактивного процесса.

Теперь можно написать функцию реакции на событие сохранения схемы `SaveRequestCallback`. Ее нужно разместить перед функцией `RegisterEvents`, чтобы на момент компиляции вызовов регистрации реакций на события `SaveRequestCallback` уже была известна компилятору.

```
// Реакция на событие - пользователь нажал "сохранить"
void RDSCALL SaveRequestCallback(int link,int event,
                      LPVOID edata,LPVOID aux)
{ // Массивы для работы с именами файлов
  char filename[MAX_PATH+1]="",
       dirname[MAX_PATH+1],
       auxname[MAX_PATH+1],
       temp[MAX_PATH+1],
       *s;
  OPENFILENAME ofn;
  DWORD attr;
  HANDLE flist;
  int dirlen;

  // Помещаем наше приложение на передний план
  BringAppToTop();

  // Заполняем структуру OPENFILENAME для диалога сохранения
  ZeroMemory(&ofn,sizeof(ofn));
  ofn.lStructSize=sizeof(ofn);
  ofn.hwndOwner=MainWin;
  ofn.lpstrFile=filename;
  ofn.nMaxFile=sizeof(filename);
  ofn.lpstrFilter="Спецформат (*.spc)\0*.spc\0Все файлы\0*.*\0";
  ofn.lpstrDefExt="spc";
  ofn.nFilterIndex=1;
  ofn.Flags=OFN_EXPLORER | OFN_OVERWRITEPROMPT;
  // Вызываем стандартный диалог сохранения Windows
  if(!GetSaveFileName(&ofn))
    return;
  // Пользователь выбрал в диалоге имя файла, оно записано в
  // массив filename. В этот файл мы запишем список всех
  // имен файлов объектов
```

```
// Файлы объектов будут записаны в подпапку filename+".files"
// (рядом с файлом filename)
if(strlen(filename)>MAX_PATH+8)
  { DisplayText("Слишком длинный путь");
    return;
  }
strcpy(dirname,filename);
strcat(dirname,".files");
// В массиве dirname теперь находится имя подпапки объектов
CreateDirectory(dirname,NULL); // Создаем эту папку
// Проверяем, создалась ли папка
attr=GetFileAttributes(dirname);
if(attr==0xFFFFFFFF || (attr & FILE_ATTRIBUTE_DIRECTORY)==0)
  { // Папка с именем dirname не существует
    DisplayText("Невозможно создать папку");
    return;
  }
// Папка есть (создана или существовала раньше)
strcat(dirname,"\\"); // Добавляем "\" в конец имени
dirlen=strlen(dirname); // Длина имени папки с завершающим "\"

// В файл filename мы будем записывать имена файлов объектов
// без пути - по одному на строчку

// Открываем файл списка для записи
flist=CreateFile(filename,GENERIC_WRITE,0,NULL,
              CREATE_ALWAYS,0,NULL);
if(flist==INVALID_HANDLE_VALUE) // Ошибка открытия файла
  { DisplayText("Невозможно записать главный файл");
    return;
  }

// Начинаем чтение содержимого схемы из РДС
if(!rdsctrlStartBlockByBlockSave(RdsLink,0))
  { DisplayText("Невозможно получить данные");
    CloseHandle(flist);
    return;
  }

// Читаем данные, пока функция не вернет нулевой тип блока
for(;;)
  { DWORD extid,parentid,res,size;
    TDynString text;
    HANDLE f;
    BOOL ok;
    // Получаем из РДС данные очередного объекта
    int tag=rdsctrlGetBlockByBlockSavePiece(RdsLink,
                         &extid,&parentid,&text);
    if(tag==0) // Данные кончились
      break;
    if(text.IsEmpty()) // Возвращенный текст описания пуст
      continue;
    // Формируем имя файла из типа tag и идентификатора extid
    sprintf(temp,"%d_%u.obj",tag,extid);
    if(strlen(temp)+dirlen>MAX_PATH)
      { DisplayText("слишком длинный путь");
        break;
      }
```

```
            // Добавляем к имени файла путь к папке dirname
            strcpy(auxname,dirname);
            strcat(auxname,temp);
            // Записываем текст text в файл auxname
            f=CreateFile(auxname,GENERIC_WRITE,0,NULL,CREATE_ALWAYS,
                         0,NULL);
            if(f==INVALID_HANDLE_VALUE)
              { DisplayText("Невозможно записать файл объекта");
                break;
              }
            size=strlen(text.c_str); // Размер текста
            ok=WriteFile(f,text.c_str,size,&res,NULL);
            CloseHandle(f);
            if((!ok) || res!=size)
              { DisplayText("Ошибка записи файла объекта");
                break;
              }
            // Дописываем имя файла (temp) в главный файл-список
            size=strlen(temp); // Длина имени файла
            ok=WriteFile(flist,temp,size,&res,NULL);
            if(res!=size) ok=FALSE;
            // После имени записываем перевод строки и возврат каретки
            ok=ok && WriteFile(flist,"\r\n",2,&res,NULL);
            if(res!=2) ok=FALSE;
            if(!ok)
              DisplayText("Ошибка записи имени в список");
          }
        // Закрываем файл списка
        CloseHandle(flist);
        // Завершаем чтение данных схемы
        rdsctrlEndBlockByBlockSave(RdsLink);
      }
      //=======================================
```

Хотя к получению данных из РДС в этой функции относятся всего три вызова, она получилась довольно длинной – в основном, из-за работы с диалогом и файловых операций. Сначала мы помещаем наше приложение на передний план вызовом написанной ранее вспомогательной функции `BringAppToTop`, чтобы диалог был виден пользователю, а затем заполняем структуру `OPENFILENAME ofn` и передаем ее в функцию Windows API `GetSaveFileName`, которая откроет стандартный диалог сохранения. Если пользователь закроет диалог, не выбрав имя файла для записи, функция вернет `FALSE`, и на этом наша реакция завершится. Если же пользователь выберет файл или введет его имя вручную, функция вернет `TRUE` и запишет имя этого файла в массив `filename` (указатель на этот массив мы передали через структуру `ofn`). В этот файл мы будем записывать список имен всех отдельных файлов объектов схемы.

Чтобы как-то упорядочить сохраняемую информацию, файлы объектов мы будем помещать в папку с именем выбранного пользователем файла, к которому добавлен суффикс ".files". Так часто делают web-браузеры при сохранении сложных документов: все вспомогательные файлы, связанные со страницей (например, графику), они помещают в отдельную папку с именем, близким к имени файла основного документа. В нашем случае, если, например, пользователь выберет файл "C:\Work\test.spc", файлы объектов будут размещаться в папке "C:\Work\test.spc.files\", а имена этих файлов без путей будут записаны в выбранный пользователем файл "test.spc".

Прежде чем начать запись, нам нужно создать папку файлов объектов, если она еще не существует. Для упрощения примера мы не будем проверять существование этой папки и создавать ее только в случае ее отсутствия. Вместо этого мы сразу попытаемся ее создать

(что не получится, если она уже есть), а затем проверим ее наличие – нам все равно пришлось бы проверять успешность создания, а так мы обойдемся единственной проверкой. Имя папки мы формируем в массиве `dirname`, добавляя к имени файла из массива `filename` строку ".files" (при этом следя за тем, чтобы полный путь к папке получился не длиннее стандартной константы `MAX_PATH`). Затем мы пытаемся создать папку вызовом `CreateDirectory`, после чего проверяем ее существование при помощи функции получения атрибутов файла (в данном случае – папки) `GetFileAttributes`. Если эта функция вместо атрибутов вернет `0xFFFFFFFF`, значит, файла или папки с таким именем нет, то есть создание папки по каким-то причинам не выполнено. Если среди атрибутов не будет `FILE_ATTRIBUTE_DIRECTORY`, значит, это не папка, а файл, то есть на момент нашей попытки создания папки уже существовал файл с тем же именем. В обоих случаях мы завершаем нашу функцию с сообщением об ошибке.

Далее мы добавляем в конец массива `dirname` обратную косую черту и запоминаем длину получившейся строки в переменной `dirlen` – она понадобится нам, когда мы будем формировать имена файлов объектов. Файл, выбранный пользователем, мы открываем для записи функцией `CreateFile` и помещаем его дескриптор в переменную `flist`. По мере записи объектов схемы мы будем записывать в него имена их файлов.

Теперь можно начинать чтение данных схемы из РДС. Мы вызываем `rdsctrlStartBlockByBlockSave` и, если она вернула ноль, завершаем функцию с сообщением об ошибке: данные получить невозможно. Затем мы начинаем "бесконечный" цикл `for(;;)` – когда мы запишем все объекты схемы, мы прервем его оператором `break`. В цикле мы описываем некоторое количество вспомогательных переменных, после чего вызываем функцию чтения данных очередного объекта схемы `rdsctrlGetBlockByBlockSavePiece`, которая запишет тип объекта в переменную `tag`, его идентификатор – в переменную `extid`, идентификатор родительской подсистемы – в `parentid` (здесь он нам не нужен, мы получаем его только для примера), а текст описания объекта – в переменную `text` типа `TDynString` (при создании нашей программы мы описали в ней класс `TDynString` для хранения строк произвольной длины и зарегистрировали в RdsCtrl.dll функцию записи строк в объекты этого класса, см. §3.3). Если возвращенный функцией тип объекта `tag` равен нулю, то есть константе `RDS_SFTAG_EOF`, мы прерываем цикл: все объекты схемы считаны. В противном случае мы проверяем возвращенное текстовое описание на пустоту (если описание объекта отсутствует, его незачем записывать в файл) и, если оно пустое, переходим в начало цикла оператором `continue`.

После того, как описание объекта считано, функцией `sprintf` мы формируем в массиве `temp` из типа объекта и его идентификатора уникальное имя файла вида "*тип_идентификатор*.obj" Может показаться, что для уникальности имени файла достаточно идентификатора объекта, но некоторые объекты схемы (стили связей и набор структур) не имеют идентификаторов – для них возвращаются нулевые значения, что совпадает с идентификатором корневой подсистемы. По этой причине мы добавляем к имени файла еще и тип объекта – это сочетание всегда будет уникально. Кроме того, так мы по имени файла сразу сможем понять, что за объект в нем находится. В массиве `auxname` мы добавляем к имени папки объектов `dirname` созданное имя файла `temp`, формируя, таким образом, имя файла объекта с полным путем, и открываем этот файл для записи функцией `CreateFile`, присваивая полученный дескриптор переменной `f`. Затем мы записываем в открытый файл текст описания объекта из переменной `text` функцией `WriteFile` и закрываем файл функцией `CloseHandle`. Затем мы проверяем успешность записи, сравнивая количество фактически записанных байтов с длиной строки и выводим сообщение об ошибке при их несовпадении.

Теперь нам нужно дописать имя файла объекта без пути (`temp`) в главный файл – он сейчас открыт, и его дескриптор находится в переменной `flist`. Мы записываем в него содержимое строки `temp` и символы перевода строки и возврата каретки "\r\n", которые обычно разделяют строки текстовых файлов. На этом тело цикла `for(;;)` заканчивается.

После завершения цикла (то есть после записи всех объектов схемы) мы закрываем файл списка `flist` вызовом `CloseHandle` и завершаем чтение данных схемы вызовом `rdsctrlEndBlockByBlockSave`. На этом наша реакция заканчивается.

Таким образом, теперь, если пользователь нажмет в РДС Ctrl+S, наша управляющая программа переместится на передний план и покажет диалог сохранения. Если пользователь выберет в нем файл для записи, все объекты схемы запишутся в отдельные файлы в папку с именем выбранного пользователем файла и суффиксом ".files", а в сам файл запишется список этих объектов.

Теперь нам нужно выполнить обратную операцию: собрать схему из отдельных файлов объектов при нажатии Ctrl+O. Мы зарегистрировали для этого функцию реакции `LoadRequestCallback`, но, прежде, чем мы ее напишем, нам нужно создать вспомогательную функцию, которая будет загружать в память текстовый файл с заданным именем – она потребуется нам и для загрузки файла со списком объектов, и для загрузки описаний самих объектов. Чтобы постоянно не отводить память под очередной считываемый файл, мы будем передавать в функцию указатель на переменную, в которой будет храниться указатель на динамически отведенный символьный буфер для файла, и указатель на переменную с текущим размером этого буфера. Если буфер не отведен или его размер недостаточен для файла, который нужно считать, функция будет самостоятельно отводить под него память оператором `new[]` и корректировать значения переданных ей переменных. Функция будет выглядеть так:

```
// Чтение текстового файла в память
BOOL ReadFileToBuffer(
           char *filename, // Имя файла
           char **pBuffer, // Указатель на указатель на буфер
           DWORD *pSize)   // Указатель на размер буфера
{ HANDLE f;
  DWORD size,actread;
  BOOL ok;

  // Проверяем переданные параметры
  if(pBuffer==NULL || pSize==NULL) // Некуда загружать
    return FALSE;

  // Открываем файл для чтения
  f=CreateFile(filename,GENERIC_READ,0,NULL,OPEN_EXISTING,0,NULL);
  if(f==INVALID_HANDLE_VALUE) // Ошибка
    return FALSE;

  // Определяем размер файла
  size=GetFileSize(f,NULL);
  if(size==0xFFFFFFFF) // Ошибка
    { CloseHandle(f);
      return FALSE;
    }
  // Переотводим буфер, если нужно
  if((*pBuffer)==NULL || (*pSize)<size+1)
    { if(*pBuffer) // Удаляем старый
        delete[] (*pBuffer);
      *pBuffer=new char[*pSize=size+1]; // Отводим новый
    }
```

```
  // Читаем файл, если он не пустой
  if(size)
    { if(ReadFile(f,*pBuffer,size,&actread,NULL))
       ok=(actread==size);
      else
       ok=FALSE;
    }
  CloseHandle(f); // Закрываем файл
  (*pBuffer)[size]=0; // Дописываем 0, завершающий строку
  return ok;
}
//=======================================
```

Эту функцию можно использовать следующим образом:

```
char *buffer=NULL;    // Буфер (пока не отведен)
DWORD bufsize;        // Размер буфера
BOOL ok;
...
ok=ReadFileToBuffer("file1.txt",&buffer,&bufsize);
...
ok=ReadFileToBuffer("file2.txt",&buffer,&bufsize);
...
if(buffer) delete[] buffer;
```

Сначала мы описываем указатель типа `char*` на динамически отводимый буфер и присваиваем ему `NULL` – это даст функции понять, что буфер еще не отведен. Для хранения текущего размера буфера мы вводим переменную `bufsize` типа `DWORD`. После этого мы можем произвольное число раз вызывать функцию `ReadFileToBuffer`, передавая ей указатели на `buffer` и `bufsize`. Если очередной считываемый файл умещается в текущий буфер, функция просто загрузит его туда. Если же файл не умещается или буфер еще не отводился (`buffer==NULL`), функция перед чтением отведет буфер заново и скорректирует `buffer` и `bufsize` так, что они будут соответствовать новому буферу. После того, как работа с файлами завершена, буфер нужно будет освободить оператором `delete[]`.

Теперь мы можем написать функцию реакции `LoadRequestCallback`. Как и `SaveRequestCallback`, ее нужно разместить перед функцией `RegisterEvents`.

```
// Реакция на событие - пользователь нажал "загрузить"
void RDSCALL LoadRequestCallback(int link,int event,
                      LPVOID edata,LPVOID aux)
{ // Массивы для работы с именами файлов
  char filename[MAX_PATH+1]="",
       temp[MAX_PATH+1],
       dirname[MAX_PATH+1],
       auxname[MAX_PATH+1];
  OPENFILENAME ofn;
  char *List=NULL,*Text=NULL,*fn;
  DWORD ListSize=0,TextSize=0;
  int dirlen,tag;
  DWORD attr;

  // Заполняем структуру OPENFILENAME
  ZeroMemory(&ofn,sizeof(ofn));
  ofn.lStructSize=sizeof(ofn);
  ofn.hwndOwner=MainWin;
  ofn.lpstrFile=filename;
  ofn.nMaxFile=sizeof(filename);
  ofn.lpstrFilter="Спецформат (*.spc)\0*.spc\0Все файлы\0*.*\0";
  ofn.nFilterIndex=1;
  ofn.Flags=OFN_EXPLORER | OFN_FILEMUSTEXIST;
```

```
// Помещаем наше приложение на передний план
BringAppToTop();

// Вызываем стандартный диалог открытия
if(!GetOpenFileName(&ofn))
  return;
// Пользователь выбрал файл filename

// Формируем из имени файла имя папки объектов
if(strlen(filename)>MAX_PATH+8)
  { DisplayText("Слишком длинный путь");
    return;
  }
strcpy(dirname,filename);
strcat(dirname,".files");
// Проверяем есть ли папка
attr=GetFileAttributes(dirname);
if(attr==0xFFFFFFFF || (attr & FILE_ATTRIBUTE_DIRECTORY)==0)
  { DisplayText("Отсутствует папка объектов");
    return;
  }
strcat(dirname,"\\"); // Добавляем "\" в конец имени
dirlen=strlen(dirname); // Длина пути к папке

// Считываем файл со списком объектов в буфер List
if(!ReadFileToBuffer(filename,&List,&ListSize))
  { DisplayText("Ошибка чтения списка файлов");
    if(List) delete[] List;
    return;
  }
// Список считан в буфер - разбираем его и грузим объекты

// Начинаем загрузку в РДС
rdsctrlStartBlockByBlockLoad(RdsLink,0);

// В цикле берем из списка имя за именем
fn=List;
for(;;)
  { int n;
    // Пропускаем пустые строки, если они есть
    fn+=strspn(fn,"\r\n");
    // Определяем длину до перевода строки
    n=strcspn(fn,"\r\n"); // В n - длина имени файла
    if(n==0) // Список кончился
      break;
    if(n+dirlen>MAX_PATH)
      { DisplayText("Слишком длинный путь");
        break;
      }
    // Копируем имя файла объекта (без пути) в temp
    strncpy(temp,fn,n);
    temp[n]=0;
    fn+=n; // Продвигаем указатель на следующее имя файла
    // Формируем в auxname полный путь к файлу объекта
    strcpy(auxname,dirname);
    strcat(auxname,temp);
```

```
        // Считываем файл объекта в буфер Text
        if(!ReadFileToBuffer(auxname,&Text,&TextSize))
          { DisplayText("Ошибка чтения файла объекта");
            break;
          }
        // Файл считан - передаем описание объекта в РДС
        tag=atoi(temp); // Тип объекта - первое число в имени файла
        rdsctrlSetBlockByBlockLoadPiece(RdsLink,tag,Text);
      }

    // Освобождаем буферы
    if(List) delete[] List;
    if(Text) delete[] Text;

    // Все считано - собираем схему из объектов
    if(!rdsctrlEndBlockByBlockLoad(RdsLink,TRUE))
      DisplayText("Ошибка сборки загруженных данных");
  }
  //=======================================
```

Начало этой функции очень похоже на `SaveRequestCallback`: здесь мы тоже помещаем окно нашей программы на передний план и открываем диалог, но только не диалог сохранения, а диалог открытия файла. Затем мы уже описанными способом формируем имя папки объектов из имени выбранного пользователем файла и проверяем существование этой папки – если ее нет, загрузка невозможна.

Далее мы загружаем файл, выбранный пользователем, в текстовый буфер `List` функцией `ReadFileToBuffer` и начинаем поблочную загрузку схемы функцией `rdsctrlStartBlockByBlockLoad`. Далее в цикле `for(;;)` мы берем из загруженного списка по одному имени, используя для поиска символов перевода строки, отделяющих одно имя от другого, стандартные функции работы со строками `strspn` и `strcspn`. Мы не будем подробно останавливаться на работе этих функций, желающие могут найти их подробные описания в справочниках по языку C. В результате этих манипуляций со строками в переменной `fn` окажется указатель на начало очередного имени файла, а в переменной `n` – длина этого имени. Если `n` равно нулю, значит, список кончился – в этом случае мы прерываем цикл оператором `break`. В противном случае мы копируем имя файла в массив `temp` и продвигаем `fn` на `n` символов вперед, то есть на начало следующего имени файла. Затем в массиве `auxname` мы формируем полный путь к файлу объекта, добавляя путь к папке объектов `dirname` к считанному из списка имени файла `temp`, и загружаем этот файл в буфер `Text` вызовом `ReadFileToBuffer`. Мы специально написали эту функцию так, чтобы она, при необходимости, увеличивала размер буфера, поэтому мы все время передаем в нее указатели на переменные `Text` (указатель на буфер) и `TextSize` (текущий размер буфера), а она сама изменит их значения, если очередной загружаемый файл не уместится в буфер.

После того, как файл объекта загружен, мы получаем тип этого объекта `tag` из имени файла при помощи стандартной функции преобразования строки в число `atoi`: имена файлов объектов мы формируем из типа и идентификатора, разделенных подчеркиванием, поэтому первое число в имени файла будет типом объекта. Тип загруженного объекта `tag` и его текстовое описание `Text` мы передаем в RdsCtrl.dll вызовом `rdsctrlSetBlockByBlockLoadPiece` и заканчиваем тело цикла, то есть переходим к загрузке следующего объекта из списка.

После завершения цикла `for(;;)` мы освобождаем оба использованных для загрузки файлов буфера операторами `delete[]` и, вызвав `rdsctrlEndBlockByBlockLoad` с

параметром `TRUE`, даем команду собрать схему из загруженных объектов. В случае ошибки функция вернет `FALSE`, и мы выведем сообщение пользователю функцией `DisplayText`.

Теперь при нажатии Ctrl+O в РДС наше приложение будет выходить на передний план и показывать диалог открытия файла. Если пользователь выберет в этом диалоге какой-нибудь из файлов, сохраненных ранее при помощи Ctrl+S (мы используем для них расширения “spc”), наша программа соберет в РДС схему из отдельных файлов объектов, список которых находится в выбранном пользователем файле.

Рассмотренный пример имеет множество недостатков. Например, если пользователь загрузит сохраненную таким образом схему, удалит какой-нибудь блок и снова сохранит ее, файл объекта, соответствующий удаленному блоку, не будет стерт. Тем не менее, он иллюстрирует основные методы перехвата загрузки и сохранения схемы, разборку схемы на отдельные объекты и ее повторную сборку из них, которые могут быть полезны при включении РДС в состав каких-либо программных комплексов.

Нужно отметить, что, помимо рассмотренного метода сохранения и загрузки схемы в виде набора объектов, можно также использовать функции `rdsctrlSaveSystemTagged`/ `rdsctrlLoadSystemTagged` и их расширенные версии с похожими названиями, которые подробно описаны в приложении Б. Эти функции позволяют сразу записать весь набор объектов в файл или разделяемую область памяти и загрузить их оттуда – в некоторых случаях это удобнее, чем получать из RdsCtrl.dll по одному объекту за вызов. Разработчик управляющего приложения может использовать любые из этих функций или их комбинации.

§3.6. Отображение схемы РДС в собственном окне приложения

Рассматривается возможность отображения подсистем загруженной в РДС схемы не в отдельных окнах, а внутри окна управляющего приложения. Приводится пример программы, реализующей эту возможность и основные функции пользовательского интерфейса для взаимодействия с блоками схемы.

§3.6.1. Общие принципы работы с портом вывода

Рассматривается общая структура программы, отображающей схему внутри своего окна. Приводится исходный текст такой программы, в которую позже будут добавлены необходимые для ее работы функции обмена данными с РДС.

Если разработчика приложения, управляющего РДС, не устраивает стандартный внешний вид окон подсистем, он может отображать содержимое подсистем в своих собственных окнах, объявив прямоугольный участок окна портом вывода и привязав к такому порту какую-либо подсистему загруженной схемы. Приложение может создать произвольное количество портов вывода в разных окнах и одновременно показывать в них разные подсистемы схемы или одну и ту же, но, например, в разных масштабах. Использование портов вывода может показаться заманчивым, поскольку дает программисту полный контроль над внешним видом приложения, однако, не стоит прибегать к этому без необходимости. Во-первых, это сильно усложняет управляющее приложение: подсистемы в портах вывода не могут самостоятельно реагировать на мышь и клавиатуру, менять масштаб, прокручивать содержимое и т.д., все эти функции и пользовательский интерфейс для них нужно реализовывать в управляющем приложении, транслируя команды и действия пользователя в РДС через вызовы RdsCtrl.dll. Во-вторых, в портах вывода не отображаются панели, создаваемые блоками в окнах подсистем (см. §2.10.4) – такие панели могут создаваться только в “настоящих” окнах подсистем. В-третьих, организовать редактирование схемы через порт вывода – очень сложная задача, требующая от разработчика практически полного повторения пользовательского интерфейса РДС в своем приложении. Тем не менее, если нужно замаскировать элементы интерфейса РДС и при этом, не предполагается работа со схемами, блоки которых пользуются панелями (а панели используются блоками не очень

часто), а также редактирование схем, работа с подсистемой через порт вывода может быть хорошим выходом из положения.

Для иллюстрации работы с портом вывода мы создадим новое приложение: старое, созданное в §3.1, рассчитано на работу с подсистемами, открывающимися в отдельных окнах. Сначала, как и в прошлый раз, напишем скелет программы, а потом допишем все необходимые для ее работы функции (функции-заглушки, которые мы пока вставим вместо них, их вызовы, а также некоторые важные изменения выделены жирным).

```
// Описания, необходимые для используемого компилятора
// (в других компиляторах они не понадобятся или будут другими)
#define _WIN32_WINNT 0x0400
#define WINVER 0x0400
// Необходимые файлы заголовков
#include <windows.h>
#include <Commctrl.h>
#include <stdio.h>

// Описания, необходимые для RdsCtrl.dll
#define RDSCTRL_SERV_FUNC_BODY GetRdsCtrlFuncs
#include <RdsCtrl.h>

// Вспомогательный класс для удобства работы со строками
// произвольной длины
class TDynString
{ public:
    // Указатель на динамическую строку
    char *c_str;
    // Освободить память
    void Free(void)
      { if(c_str) delete[] c_str;
        c_str=NULL; };
    // Записать строку в объект
    void Set(char *s)
      { Free(); // Освободить старую
        if(s!=NULL) // Отвести память и скопировать новую
          { c_str=new char[strlen(s)+1];
            strcpy(c_str,s);
          }
      };
    // Строка пустая?
    BOOL IsEmpty(void)
      { return c_str==NULL || (*c_str)==0;};
    // Конструктор и деструктор
    TDynString(void){c_str=NULL;};
    ~TDynString(){Free();};
};
//=======================================

// Буфер для индицируемого программой текста
char buffer[2000]="Программа запущена";
// Главное окно программы (для доступа к нему из функций)
HWND MainWin;
// Дескрипторы полос прокрутки главного окна
HWND HorzScroll,VertScroll;
// Дескриптор служебного окна всплывающей подсказки
HWND ToolTip;
//=======================================
```

```
// Последнее запомненное положение курсора мыши в окне
int LastX=0,LastY=0;
// Зона последней выведенной всплывающей подсказки
RECT LastToolTipRect;
// Флаг необходимости реакции на движение мыши без
// нажатых кнопок
BOOL FreeMouseMove;
// Строка для хранения полученного текста всплывающей подсказки
TDynString HintString;
//=========================================

// Отступ сверху до изображения подсистемы
#define VIEWPORTTOP 55

// Идентификаторы кнопок окна программы
#define IDC_OPENBUTTON  101
#define IDC_BACKBUTTON  102
#define IDC_ZOOMIN      103
#define IDC_ZOOMOUT     104
// Идентификатор всплывающей подсказки для порта вывода
#define VIEWPORTTIP_ID  105
// Начальный идентификатор пунктов контекстного меню блока
#define IDC_MENUSTART   200
//=========================================

// Функция вывода текстового сообщения в окне программы
void DisplayText(char *text)
{ RECT rect;
  // Определяем размер клиентской области окна
  GetClientRect(MainWin,&rect);
  // Ограничиваем область снизу (ниже будут располагаться кнопки)
  rect.bottom=30;
  if(text) // Копируем текст в буфер
    { strncpy(buffer,text,sizeof(buffer)-1);
      buffer[sizeof(buffer)-1]=0; // Если строка слишком длинная
    }
  else
    strcpy(buffer,"(NULL)");
  // Указываем Windows, что область rect нужно перерисовать
  InvalidateRect(MainWin,&rect,TRUE);
}
//=========================================

// Глобальные переменные для связи с РДС
HMODULE RdsCtrl=NULL; // Библиотека управления
int RdsLink=-1;       // Связь с РДС
int Viewport=-1;      // Порт вывода
//=========================================

// Функция возврата строки
void RDSCALL ReturnString(LPVOID ptr,LPSTR str)
{ TDynString *pDS=(TDynString*)ptr;
  if(pDS) pDS->Set(str);
}
//=========================================
```

```
// Получить текущие размеры области, доступной для порта вывода
BOOL GetAvailableRect(RECT *rect)
{ RECT ClientArea;
  // Получаем стандартную ширину и высоту полос прокрутки
  int w=GetSystemMetrics(SM_CXVSCROLL),
      h=GetSystemMetrics(SM_CYHSCROLL);
  if(rect==NULL) return FALSE;
  // Получаем размер клиентской (внутренней) области окна
  if(!GetClientRect(MainWin,&ClientArea))
    return FALSE;

  // Ограничиваем внутреннюю область сверху, снизу и справа
  rect->left=0;
  rect->top=VIEWPORTTOP;
  rect->right=ClientArea.right-w;
  rect->bottom=ClientArea.bottom-h;
  return TRUE;
}
//========================================

// Подстроить полосы прокрутки под текущий размер окна
void AdjustScrollBars(void)
{ RECT rect;
  // Получаем стандартную ширину и высоту полос прокрутки
  int w=GetSystemMetrics(SM_CXVSCROLL),
      h=GetSystemMetrics(SM_CYHSCROLL);
  // Получаем доступные размеры порта вывода
  if(!GetAvailableRect(&rect))
    return;

  // Горизонтальная полоса - по нижнему краю окна
  SetWindowPos(HorzScroll,NULL,
               rect.left,rect.bottom,
               rect.right-rect.left,h,
               SWP_NOZORDER);

  // Вертикальная полоса - по правому краю окна
  SetWindowPos(VertScroll,NULL,
               rect.right,rect.top,
               w,rect.bottom-rect.top,
               SWP_NOZORDER);
}
//========================================

// Установить параметры полос прокрутки
void SetScrollBarParams(void)
{
  // Здесь мы будем получать у РДС размеры рабочего поля
  // подсистемы и настраивать диапазон прокрутки
}
//========================================

// Нарисовать подсистему в порте вывода
void DrawViewport(void)
{
  // Здесь мы будем рисовать содержимое подсистемы
```

```
}
//=========================================

// Загрузка RdsCtrl.dll и создание связи
void InitRdsCtrl(void)
{
  if(RdsCtrl==NULL) // Библиотека еще не загружена
    { char rdsctrldll[MAX_PATH+1],*s;
      // Считаем, что наша программа находится в одной папке с РДС
      // Получаем путь к RdsCtrl.dll из пути к нашей программе
      GetModuleFileName(NULL,rdsctrldll,MAX_PATH);
      s=strrchr(rdsctrldll,'\\'); // Ищем последний '\'
      if(!s) // Ошибка
        { DisplayText("Библиотека не найдена");
          return;
        }
      // Заменяем имя файла в пути
      strcpy(s+1,"RdsCtrl.dll");

      // Загружаем библиотеку RdsCtrl.dll
      RdsCtrl=LoadLibrary(rdsctrldll);
      if(RdsCtrl==NULL) // Загрузка не удалась
        { DisplayText("Ошибка загрузки RdsCtrl.dll");
          return;
        }
      // Получаем доступ к функциям библиотеки
      if(!GetRdsCtrlFuncs(RdsCtrl))
        { // Ошибка
          DisplayText("Нет доступа к функциям RdsCtrl.dll");
          // Выгружаем библиотеку - она бесполезна
          FreeLibrary(RdsCtrl);
          RdsCtrl=NULL;
          return;
        }
      // Доступ к функциям получен - можно их вызывать

      // Установка функции возврата строки
      rdsctrlSetStringCallback(ReturnString);
      // Сброс идентификатора связи (если он почему-то не сброшен)
      RdsLink=-1;
    } // if(RdsCtrl==NULL)

  // Создание связи с РДС
  if(RdsLink<0) // Связь не создана
    { // Создаем связь (rds.exe пока не запускается)
      RdsLink=rdsctrlCreateLink();
      if(RdsLink<0) // Ошибка
        { DisplayText("Ошибка создания связи с РДС");
          return;
        }
      // Запрет главного окна РДС
      rdsctrlShowMainWindow(RdsLink,FALSE);
      // Запрет открытия окон подсистем
      rdsctrlEnableSubsystemWindows(RdsLink,FALSE);
      // Отключение пользовательского интерфейса
      rdsctrlEnableUI(RdsLink,FALSE);
```

```
    }
}
//=========================================

// Создать порт вывода и привязать к нему подсистему
void SetViewport(char *system)
{
  // Здесь мы будем создавать порт вывода (если он еще
  // не создан) и привязватьк нему прдсистему, имя которой
  // передано в параметре
}
//=========================================

// Функция, вызываемая перед завершением программы
void BeforeExit(void)
{
  if(RdsCtrl!=NULL) // Библиотека загружена
    { if(RdsLink>=0) // Создана связь с РДС
        { if(Viewport>=0) // Уничтожаем порт вывода
            { rdsctrlReleaseViewport(RdsLink,Viewport);
              Viewport=-1;
            }
          // Завершаем РДС
          rdsctrlClose(RdsLink);
          // Удаляем связь
          rdsctrlDeleteLink(RdsLink);
          RdsLink=-1;
        }
      // Выгружаем библиотеку
      FreeLibrary(RdsCtrl);
      RdsCtrl=NULL;
    }
}
//=========================================

// Открыть файл схемы (filename – имя файла)
BOOL LoadScheme(char *filename)
{
  if(RdsCtrl==NULL || RdsLink<0)
    { // Библиотека RdsCtrl.dll еще не загружена
      InitRdsCtrl(); // Загружаем
      if(RdsCtrl==NULL) // Ошибка
        return FALSE;
      // Запускаем rds.exe
      if(!rdsctrlConnect(RdsLink))
        { DisplayText("Ошибка запуска РДС");
          return FALSE;
        }
    }
  else if(Viewport>=0) // Уничтожаем порт вывода
    { rdsctrlReleaseViewport(RdsLink,Viewport);
      Viewport=-1;
    }

  // Если rds.exe не работает (пользователь вышел из РДС),
  // перезапускаем РДС
  rdsctrlRestoreConnection(RdsLink);
```

```
  // РДС работает - загружаем схему
  if(!rdsctrlLoadSystemFromFile(RdsLink,filename,FALSE))
    { // Ошибка загрузки
      DisplayText("Ошибка загрузки схемы");
      return FALSE;
    }
  // Переходим в режим моделирования
  rdsctrlSetCalcMode(RdsLink);
  // Создаем порт вывода и привязываем его к корневой подсистеме
  SetViewport("");
  return TRUE;
}
//=========================================

// Нажатие кнопки "Открыть"
void OpenButtonClick(void)
{ char filename[MAX_PATH+1]=""; // Буфер для имени файла
  OPENFILENAME ofn;
  // Заполняем структуру OPENFILENAME
  ZeroMemory(&ofn,sizeof(ofn));
  ofn.lStructSize=sizeof(ofn);
  ofn.hwndOwner=MainWin;
  ofn.lpstrFile=filename;
  ofn.nMaxFile=sizeof(filename);
  ofn.lpstrFilter="Схемы (*.rds)\0*.rds\0Все файлы\0*.*\0";
  ofn.nFilterIndex=1;
  ofn.Flags=OFN_EXPLORER | OFN_FILEMUSTEXIST;
  // Вызываем стандартный диалог открытия файла
  if(GetOpenFileName(&ofn)) // Пользователь выбрал файл
    { // Загружаем схему
      if(LoadScheme(filename))
        DisplayText(filename);
    }
}
//=========================================

// Изменение масштаба порта вывода
void ZoomButtonClick(double multiplier)
{
  // Здесь мы будем менять масштаб подсистемы в
  // multiplier раз
}
//=========================================

// Реакция на полосы прокрутки
void DoScroll(UINT msg,WORD btn)
{ int sx,sy,pagex,pagey,sx_track,sy_track;
  RECT rect;
  SCROLLINFO si;

  // Проверяем наличие связи с РДС
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;

  // Получаем текущую позицию горизонтальной полосы
  si.cbSize=sizeof(si);
```

```
  si.fMask=SIF_ALL;
  if(!GetScrollInfo(HorzScroll,SB_CTL,&si))
    return;
  sx=si.nPos;
  sx_track=si.nTrackPos;
  // Получаем текущую позицию вертикальной полосы
  if(!GetScrollInfo(VertScroll,SB_CTL,&si))
    return;
  sy=si.nPos;
  sy_track=si.nTrackPos;

  // Получаем доступный для порта вывода прямоугольник
  if(!GetAvailableRect(&rect))
    return;
  // Размер страницы прокрутки - половина прямокгольника
  pagex=(rect.right-rect.left)/2;
  pagey=(rect.bottom-rect.top)/2;

  // Разбираемся с поступившим сообщением
  if(msg==WM_HSCROLL) // Горизонтальная прокрутка
    { switch(btn)
        { case SB_LINELEFT:   sx-=5; break;
          case SB_LINERIGHT:  sx+=5; break;
          case SB_PAGELEFT:   sx-=pagex; break;
          case SB_PAGERIGHT:  sx+=pagex; break;
          case SB_THUMBTRACK: sx=sx_track; break;
        }
      // Устанавливаем новое положение полосы
      SetScrollPos(HorzScroll,SB_CTL,sx,TRUE);
      // Командуем обновить область порта вывода
      InvalidateRect(MainWin,&rect,FALSE);
    }
  else // Вертикальная прокрутка
    { switch(btn)
        { case SB_LINEDOWN:   sy+=5; break;
          case SB_LINEUP:     sy-=5; break;
          case SB_PAGELEFT:   sy-=pagey; break;
          case SB_PAGERIGHT:  sy+=pagey; break;
          case SB_THUMBTRACK: sy=sy_track; break;
        }
      // Устанавливаем новое положение полосы
      SetScrollPos(VertScroll,SB_CTL,sy,TRUE);
      // Командуем обновить область порта вывода
      InvalidateRect(MainWin,&rect,FALSE);
    }
}
//=======================================

// Передача в РДС нажатия/отпускания клавиши
BOOL RdsKeyboardOperation(BOOL down,int keycode,DWORD flags)
{
  // Здесь мы будем вызывать в схеме реакцию на клавиши.
  // Функция вернет TRUE, если блок среагировал
  return FALSE;
}
//=======================================
```

```
// Передача в РДС действия мышью
BOOL RdsMouseOperation(UINT msg,int X,int Y,WPARAM keys)
{
  // Здесь мы будем передавать в РДС информацию о нажатии и
  // отпускании кнопок мыши и о перемещении курсора. Функция
  // вернет TRUE, если действие обработано блоком схемы.
  return FALSE;
}
//=======================================

// Создание внутренних элементов окна
void CreateControls(HWND hWindow)
{ HINSTANCE app;
  HWND open;
  TOOLINFO ti;

  // Получаем идентификатор приложения
  app=(HINSTANCE)GetWindowLong(hWindow,GWL_HINSTANCE);

  // Кнопка "Открыть"
  open=CreateWindow("BUTTON","Открыть",    // Имя класса и текст
     WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON, // Стили
     0,30,100,24,              // x,y,ширина,высота
     hWindow,                  // Родительское окно
     (HMENU)IDC_OPENBUTTON,    // Идентификатор кнопки
     app,NULL);
  // Кнопка "Назад"
  CreateWindow("BUTTON","Назад",
    WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
    121,30,100,24,hWindow,
    (HMENU)IDC_BACKBUTTON,app,NULL);
  // Кнопка "Масшт +"
  CreateWindow("BUTTON","Масшт +",
    WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
    242,30,90,24,hWindow,
    (HMENU)IDC_ZOOMIN,app,NULL);
  // Кнопка "Масшт -"
  CreateWindow("BUTTON","Масшт -",
    WS_VISIBLE | WS_CHILD | BS_PUSHBUTTON,
    363,30,90,24,hWindow,
    (HMENU)IDC_ZOOMOUT,app,NULL);
  // Горизонтальная полоса прокрутки
  HorzScroll=CreateWindow("SCROLLBAR",NULL,
    WS_VISIBLE | WS_CHILD | SBS_HORZ,
    0,0,0,0,hWindow,NULL,app,NULL);
  // Вертикальная полоса прокрутки
  VertScroll=CreateWindow("SCROLLBAR",NULL,
    WS_VISIBLE | WS_CHILD | SBS_VERT,
    0,0,0,0,hWindow,NULL,app,NULL);
  // Создаем окно для всплывающей подсказки
  ToolTip=CreateWindow(TOOLTIPS_CLASS,NULL,
    TTS_ALWAYSTIP,0,0,0,0,hWindow,
    NULL,app,NULL);
  // Добавляем подсказку к кнопке "Открыть"
  ti.cbSize=sizeof(ti);
  ti.uFlags=TTF_IDISHWND|TTF_SUBCLASS;
  ti.uId=(UINT_PTR)open;
```

```
  ti.lpszText="Загрузить схему";
  ti.hwnd=MainWin;
  SendMessage(ToolTip,TTM_ADDTOOL,0,(LPARAM)(&ti));
  // Добавляем подсказку к зоне порта вывода
  ti.uFlags=TTF_SUBCLASS;
  ti.hwnd=MainWin;
  ti.uId=VIEWPORTTIP_ID; // Идентификатор зоны подсказки
  ti.lpszText=LPSTR_TEXTCALLBACK; // Запрос текста
  SetRectEmpty(&(ti.rect)); // Пока без размера
  SendMessage(ToolTip,TTM_ADDTOOL,0,(LPARAM)(&ti));
}
//=======================================

// Получение текста всплывающей подсказки к схеме
void GetTooltip(TOOLTIPTEXT *tiptext)
{
  // Здесь мы будем получать у РДС текст подсказки к
  // заданной точке подсистемы.
}
//=======================================

// Установка зоны всплывающей подсказки размером во все
// доступное место окна программы
void AdjustToolTipRect(void)
{ TOOLINFO ti;
  ti.cbSize=sizeof(ti);
  ti.hwnd=MainWin;
  ti.uId=VIEWPORTTIP_ID;
  GetAvailableRect(&(ti.rect));
  SendMessage(ToolTip,TTM_NEWTOOLRECT,0,(LPARAM)(&ti));
}
//=======================================

// Показать контекстное меню
void ShowPopupMenu(int screenx,int screeny)
{
  // Здесь мы будем выводить контекстное меню
}
//=======================================

// Реакция на выбор пункта меню
void PopupMenuClick(int id)
{
  // Здесь мы будем передавать в РДС идентификатор
  // выбранного пользователем пункта меню
}
//=======================================

// Открыть в порте вывода подсистему, изображение которой
// находится в заданной точке
void SubSystemToViewport(int x,int y)
{
  // Здесь мы будем привязывать к порту вывода новую
  // подсистему по двойному щелчку на ее изображении
}
//=======================================
```

```
// Реакция на кнопку "Назад"
void BackButtonClick(void)
{
  // Здесь мы будем привязывать к порту вывода
  // подсистему, родительскую по отношению к текущей
}
//=======================================

// Процедура главного окна
LRESULT CALLBACK MainWndProc(HWND hWindow,UINT msg,
                WPARAM wParam,LPARAM lParam)
{ PAINTSTRUCT ps;
  HDC hDC;
  RECT rect;
  POINT pt;
  NMHDR *nmhdr;

  switch(msg)
    { // Создание окна
      case WM_CREATE:
        // Запоминаем дескриптор окна в глобальной переменной
        MainWin=hWindow;
        // Создаем кнопки, полосы прокрутки и т.д.
        CreateControls(hWindow);
        // Очищаем запомненный прямоугольник всплывающей подсказки
        SetRectEmpty(&LastToolTipRect);
        break;

      // Закрытие окна
      case WM_DESTROY:
        BeforeExit(); // Выгружаем RdsCtrl.dll
        PostQuitMessage(0); // Завершаем программу
        return 0;

      // Рисование в окне
      case WM_PAINT:
        // Выводим текст из buffer в верхней части
        hDC = BeginPaint(hWindow,&ps);
        GetClientRect(hWindow,&rect);
        rect.bottom=30;
        DrawText(hDC,buffer,-1,&rect,
                DT_SINGLELINE|DT_CENTER|DT_VCENTER);
        EndPaint(hWindow,&ps);
        // Рисуем содержимое подсистемы
        DrawViewport();
        break;

      // Команда от органов управления
      case WM_COMMAND:
        if(LOWORD(wParam)>=IDC_MENUSTART) // Выбран пункт меню
          { PopupMenuClick(LOWORD(wParam));
            break;
          }
        if(HIWORD(wParam)==BN_CLICKED) // Нажата кнопка
          { switch (LOWORD(wParam))
              { case IDC_OPENBUTTON: OpenButtonClick(); break;
```

```
            case IDC_BACKBUTTON: BackButtonClick(); break;
            case IDC_ZOOMIN: ZoomButtonClick(2.0); break;
            case IDC_ZOOMOUT: ZoomButtonClick(0.5); break;
          }
      }
    break;

// Изменение размеров окна
case WM_SIZE:
  // Подстраиваем полосы прокрутки под новый размер
  AdjustScrollBars();
  SetScrollBarParams();
  // Увеличиваем зону вплывающей подсказки
  AdjustToolTipRect();
  break;

// Прокрутка (горизонтальная или вертикальная)
case WM_HSCROLL:
case WM_VSCROLL:
  DoScroll(msg,LOWORD(wParam));
  break;

// Действия мышью
case WM_MOUSEMOVE: // Перемещение
  pt.x=LOWORD(lParam);
  pt.y=HIWORD(lParam);
  // Если курсор вышел за пределы последней зоны всплывающей
  // подсказки, делаем эту зону размером в весь порт вывода
  if(!PtInRect(&LastToolTipRect,pt))
    AdjustToolTipRect();
  // Информируем РДС о перемещении мыши
  RdsMouseOperation(msg,pt.x,pt.y,wParam);
  // Запоминаем последние координаты курсора
  LastX=pt.x;
  LastY=pt.y;
  break;
case WM_LBUTTONDOWN: // Нажатие и отпускание кнопок
case WM_MBUTTONDOWN:
case WM_RBUTTONDOWN:
case WM_LBUTTONUP:
case WM_MBUTTONUP:
case WM_RBUTTONUP:
  // Информируем РДС
  if(RdsMouseOperation(msg,LOWORD(lParam),
                       HIWORD(lParam),wParam))
    return 0;
  break;
case WM_LBUTTONDBLCLK: // Двойной щелчок
  pt.x=LOWORD(lParam);
  pt.y=HIWORD(lParam);
  // Информируем РДС
  if(!RdsMouseOperation(msg,pt.x,pt.y,wParam))
    // Блок не среагировал - пытаемся открыть подсистему
    SubSystemToViewport(pt.x,pt.y);
  break;
```

```
      // Нажатие клавиш
      case WM_KEYDOWN:
      case WM_SYSKEYDOWN:
        // Информируем РДС
        if(RdsKeyboardOperation(TRUE,(int)wParam,(DWORD)lParam))
          return 0;
        break;

      // Отпускание клавиш
      case WM_KEYUP:
      case WM_SYSKEYUP:
        // Информируем РДС
        if(RdsKeyboardOperation(FALSE,(int)wParam,(DWORD)lParam))
          return 0;
        break;

      // Вызов контекстного меню
      case WM_CONTEXTMENU:
        if(((HWND)wParam)==MainWin) // В окне программы
          ShowPopupMenu(LOWORD(lParam),HIWORD(lParam));
        break;

      // Разные уведомления
      case WM_NOTIFY:
        nmhdr=(NMHDR*)lParam;
        switch(nmhdr->code)
          { // Запрос текста от всплывающей подсказки
            case TTN_NEEDTEXT:
              GetTooltip((LPTOOLTIPTEXT)lParam);
              break;
          }
        break;
    }
  return DefWindowProc(hWindow,msg,wParam,lParam);
}
//=======================================

// Главная функция приложения
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   PSTR szCmdLine,int iCmdShow)
{ static char appName[] = "Управление РДС - порт вывода",
              className[]="RDSControlTestWindow";
  WNDCLASSEX myWin;
  HWND hWindow;
  MSG msg;

  // Создание главного окна
  myWin.cbSize=sizeof(myWin);
  myWin.style=CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
  myWin.lpfnWndProc=MainWndProc;
  myWin.cbClsExtra=0;
  myWin.cbWndExtra=0;
  myWin.hInstance=hInstance;
  myWin.hIcon=0;
  myWin.hIconSm=0;
  myWin.hCursor=0;
  myWin.hbrBackground=(HBRUSH)(COLOR_WINDOW+1);
  myWin.lpszMenuName=0;
```

```
  myWin.lpszClassName=className;
  if(!RegisterClassEx(&myWin)) return 0;
  hWindow=CreateWindow(className,appName,
          WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
          0,0,hInstance,0);
  // Открытие созданного окна
  ShowWindow(hWindow,iCmdShow);
  UpdateWindow(hWindow);

  // Инициализация стандартных компонентов
  INITCOMMONCONTROLSEX icc;
  icc.dwSize=sizeof(icc);
  icc.dwICC=ICC_WIN95_CLASSES;
  if(!InitCommonControlsEx(&icc))
    DisplayText("Ошибка InitCommonControlsEx");

  // Цикл обработки сообщений приложения
  while(GetMessage(&msg,0,0,0))
    { TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  return 0;
}
//=======================================
```

Новая программа во многом похожа на предыдущую, и мы не будем подробно останавливаться на их общих функциях и на вызовах Windows API, необходимых для работы элементов пользовательского интерфейса (создание кнопок, работа полос прокрути и т.п.)

В верхней части окна новой программы (рис. 128), как и в предыдущем примере, будет выводиться строка из массива `buffer`. Ниже нее располагаются четыре управляющих кнопки. Кнопка “Открыть” будет вызывать стандартный диалог выбора файла и загружать в управляемую копию РДС выбранную пользователем схему. При ее нажатии вызывается

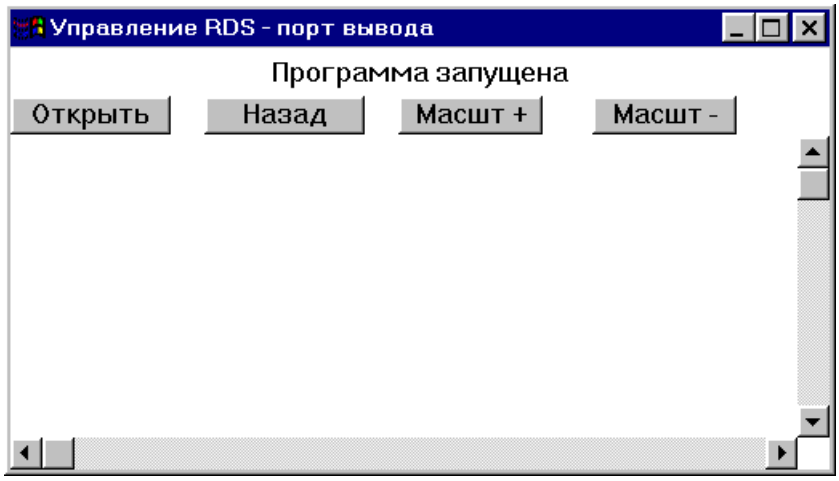


Рис. 128. Главное окно программы с пустым портом вывода

функция `OpenButtonClick` – точная копия одноименной функции из предыдущего примера. После выбора файла пользователем эта функция, в свою очередь, будет вызывать функцию `LoadScheme`, в которую мы добавили вызов еще не написанной функции `SetViewport` для создания порта вывода. Функция `BackButtonClick`, вызываемая по нажатию кнопки “Назад”, будет перенастраивать порт вывода на подсистему, родительскую по отношению к той, которая привязана к нему в данный момент: таким образом, нажатие

этой кнопки позволит перемещаться вверх по иерархии подсистем. Перемещение вниз по иерархии, то есть привязка к порту вывода подсистемы, находящейся внутри текущей отображаемой, будет осуществляться по двойному щелчку на изображении этой подсистемы в функции `SubSystemToViewport`. Нажатие кнопок “Масшт +” и “Масшт –” будет вызывать функцию `ZoomButtonClick` с параметром 2 и 0.5 соответственно, эта функция будет менять текущий масштаб отображения подсистемы в заданное число раз.

Всю оставшуюся площадь окна будет занимать область, отведенная под порт вывода, и две полосы прокрутки: горизонтальная (с дескриптором `HorzScroll`) и вертикальная (`VertScroll`). Настройкой полос прокрутки на размеры рабочего поля текущей отображаемой подсистемы будет заниматься функция `SetScrollBarParams`, которую нам предстоит написать, а привязкой полос к границам окна и реакцией на их прокрутку пользователем – уже написанные функции `AdjustScrollBars` и `DoScroll` соответственно. Для рисования в окне содержимого подсистемы с учетом текущего положения полос прокрутки мы вставили в реакцию главного окна программы на сообщение Windows `WM_PAINT` вызов функции `DrawViewport`. Отдельный набор функций будет заниматься передачей в РДС информации о перемещении курсора мыши и нажатии и отпускании клавиш в окне программы, запросом у блока текста всплывающей подсказки, выводом контекстного меню и т.д. Таким образом, нам предстоит написать следующие функции:

- `SetViewport` – создание порта вывода и привязка его к подсистеме, полное имя которой передается в параметре функции;
- `SetScrollBarParams` – настройка диапазона полос прокрутки окна на размер рабочего поля текущей отображаемой подсистемы;
- `DrawViewport` – рисование в окне программы содержимого подсистемы с учетом положения полос прокрутки;
- `ZoomButtonClick` – изменение масштаба отображаемой подсистемы в заданное число раз;
- `RdsKeyboardOperation` – вызов реакции блоков подсистемы на нажатие или отпускание клавиши;
- `RdsMouseOperation` – вызов реакции блоков подсистемы или самой подсистемы на перемещение курсора мыши и нажатие и отпускание ее кнопок;
- `GetTooltip` – получение текста всплывающей подсказки к блоку, находящемуся под курсором мыши;
- `ShowPopupMenu` – открытие контекстного меню в заданной точке окна;
- `PopupMenuClick` – передача в РДС информации о выборе пользователем конкретного пункта контекстного меню блока или подсистемы;
- `SubSystemToViewport` – привязка к порту вывода подсистемы, изображение которой находится в заданной точке окна;
- `BackButtonClick` – привязка к порту вывода родительской подсистемы.

Будем добавлять в программу эти функции одну за другой, но сначала разберемся с изменениями, которые внесены в некоторые функции из предыдущего примера для обеспечения работы порта вывода.

Прежде всего, мы вводим новую глобальную переменную `Viewport`: в ней будет храниться уникальный целый идентификатор порта вывода, созданного RdsCtrl.dll в главном окне нашей программы. Этот идентификатор не может быть отрицательным, поэтому для инициализации переменной `Viewport` мы используем значение –1 (порт не создан). Создавать порт мы будем только один раз, при первом открытии схемы. После этого мы будем менять параметры уже созданного порта с идентификатором `Viewport`.

Созданный порт необходимо удалить перед выгрузкой RdsCtrl.dll из памяти, поэтому в функцию `BeforeExit` мы вставили следующий оператор:

```
if(Viewport>=0) // Уничтожаем порт вывода
  { rdsctrlReleaseViewport(RdsLink,Viewport);
    Viewport=-1;
  }
```

Таким образом, если на момент завершения программы в переменной `Viewport` находится идентификатор порта вывода, мы уничтожаем его вызовом `rdsctrlReleaseViewport`. Далее, как и в прошлом примере, мы завершаем РДС, уничтожаем связь и выгружаем библиотеку RdsCtrl.dll.

Поскольку теперь мы работаем со схемой через порт вывода, нам нужно запретить РДС открывать подсистемы в отдельных окнах. Кроме того, мы запрещаем работу пользовательского интерфейса РДС – все действия теперь будут выполняться только по командам из нашей программы. С этой целью мы включаем в конец функции `InitRdsCtrl` два следующих вызова:

```
// Запрет открытия окон подсистем
rdsctrlEnableSubsystemWindows(RdsLink,FALSE);
// Отключение пользовательского интерфейса
rdsctrlEnableUI(RdsLink,FALSE);
```

В результате этого работа РДС окажется полностью скрыта от глаз пользователя – не будет ни окон подсистем (даже если модель блока попытается открыть окно подсистемы вызовом сервисной функции, у нее это не получится), ни кнопки РДС на панели задач Windows. У пользователя будет полное ощущение, что наша программа сама работает со схемой.

Сразу после загрузки новой схемы нам нужно создать порт вывода (если мы не создали его раньше) и настроить его на отображение корневой подсистемы схемы. Для этого в самый конец функции `LoadScheme` мы добавили вызов функции `SetViewport`:

```
// Создаем порт вывода и привязываем его к корневой подсистеме
SetViewport("");
```

В эту функцию, которую нам предстоит написать, передается полное имя подсистемы, привязываемой к порту вывода. Пустая строка в данном случае указывает на корневую подсистему схемы. Кроме того, в самое начало функции добавлено уничтожение порта вывода, если связь с РДС в данный момент создана, поскольку перед загрузкой новой схемы нужно уничтожить порт вывода, связанный со старой.

§3.6.2. Рисование и прокрутка изображения

Рассматриваются функции для создания и настройки порта вывода, привязки к нему подсистемы и рисования в нем изображения. Эти функции добавляются в программу, созданную в предыдущем параграфе.

Написание функций, которые будут непосредственно работать с РДС через порт вывода, начнем с функции привязки подсистемы к порту `SetViewport`:

```
// Привязка к порту вывода подсистемы system
void SetViewport(char *system)
{ int nvp;
  int ScrollX,ScrollY;

  // Проверяем, создана ли связь с РДС
  if(RdsCtrl==NULL || RdsLink<0)
    return;

  // Создаем новый порт или меняем параметры созданного
  nvp=rdsctrlSetViewport(RdsLink,Viewport,MainWin,system,0);
  if(nvp==-1) // Задать параметры порта не получилось
    return;
  Viewport=nvp;
```

```
    // Получаем запомненное положение прокрутки
    rdsctrlGetViewportParams(RdsLink,nvp,NULL,&ScrollX,&ScrollY);

    // Настраиваем полосы прокрутки по рабочему полю
    SetScrollBarParams();
    // Устанавливаем полосы прокрутки в запомненное положение
    SetScrollPos(HorzScroll,SB_CTL,ScrollX,TRUE);
    SetScrollPos(VertScroll,SB_CTL,ScrollY,TRUE);

    // Нужна ли какому-либо блоку подсистемы реакция на перемещение
    // мыши без нажатия кнопок
    FreeMouseMove=(rdsctrlGetVPMouseLevel(RdsLink,nvp)==2);

    // Командуем обновить окно программы
    InvalidateRect(MainWin,NULL,TRUE);
  }
  //=======================================
```

В этой функции мы вызываем функцию настройки порта вывода `rdsctrlSetViewport`, передавая ей идентификатор связи с РДС (`RdsLink`), идентификатор настраиваемого порта вывода (`Viewport`), дескриптор окна, в котором создается порт (`MainWin`), полное имя подсистемы, привязываемой к порту (`system`) и флаги порта (не используются). При самом первом вызове в переменной `Viewport` будет содержаться значение –1, поэтому функция создаст новый порт вывода. При всех последующих вызовах в ней будет находиться идентификатор порта, поэтому `rdsctrlSetViewport` вместо создания нового порта изменит параметры существующего. Идентификатор созданного (или измененного) порта вывода, возвращаемый функцией, присваивается переменной `nvp`. Если функция вернула –1, значит, настройка порта не удалась (например, передано несуществующее имя подсистемы), и никаких действий больше не производится. В противном случае `nvp` записывается в глобальную переменную `Viewport` – начиная с этого момента, в ней находится идентификатор успешно созданного порта вывода.

Теперь нужно получить из РДС текущее положение прокрутки схемы, то есть координаты, соответствующие верхнему левому углу видимой области рабочего поля (эти координаты сохраняются в файле схемы вместе с остальными параметрами подсистемы). Для этого мы вызываем функцию получения параметров порта вывода `rdsctrlGetViewportParams`, передавая в ее четвертом и пятом параметрах указатели на целые переменные `ScrollX` и `ScrollY`, в которые и будут записаны координаты верхнего левого угла этой области. В первом параметре функции, как обычно, передается идентификатор связи с РДС, во втором – идентификатор порта вывода, параметры которого запрашиваются, в третьем – указатель на вещественную переменную, через который функция вернет запомненный масштаб подсистемы (сейчас нам это не нужно, поэтому в третьем параметре мы передаем `NULL`).

Пока существует порт вывода, РДС запоминает в его внутренних параметрах положение прокрутки и масштаб для каждой подсистемы, которая когда-либо отображалась через этот порт. Если данная подсистема уже привязывалась к этому порту вывода ранее, функция `rdsctrlGetViewportParams` вернет не параметры, хранящиеся для этой подсистемы в файле схемы, а параметры, с которыми она рисовалась через этот порт в прошлый раз. Таким образом, возвращаясь к уже отображавшимся в данном порте подсистемам, мы будем наблюдать их в том же самом масштабе и положении, в котором мы оставили их. Следует помнить, что эта информация хранится именно в параметрах порта вывода и не передается в параметры самой подсистемы, поэтому она не сохраняется в файле схемы и теряется при уничтожении порта вывода: если, например, мы изменим масштаб какой-либо подсистемы через порт вывода и сохраним схему, изменение масштаба

запомнено не будет, и в следующий раз подсистема откроется в том масштабе, который установил пользователь при работе с ней в “нормальном” окне РДС.

Считав параметры прокрутки подсистемы в переменные `ScrollX` и `ScrollY`, мы настраиваем диапазон полос прокрутки на размер рабочего поля схемы функцией `SetScrollBarParams` (мы ее пока еще не написали) и устанавливаем в этих полосах текущую позицию `ScrollX` и `ScrollY` функцией Windows API `SetScrollPos`. Дескрипторы полос мы берем из глобальных переменных `HorzScroll` и `VertScroll`, в которые они были записаны при создании внутренних объектов окна. Мы также подстраиваем размер и положение полос под размеры окна программы функцией `AdjustScrollBars` – эта функция состоит только из вызовов Windows API, и мы не будем ее подробно разбирать.

Затем мы вызываем функцию `rdsctrlGetVPMouseLevel` чтобы узнать, нужна ли подсистеме в порте вывода или ее внутренним блокам реакция на действия пользователя мышью. Эта функция возвращает одно из трех значений: 0 – подсистема вообще не обрабатывает мышь, 1 – подсистема реагирует на кнопки мыши и на перемещения курсора при нажатых кнопках, 2 – подсистема реагирует на любые действия мышью. Если функция вернула значение 2, мы взводим глобальный логический флаг `FreeMouseMove`. Он поможет нам “разгрузить” взаимодействие нашей программы с РДС: если функция вернет значения 0 или 1 (то есть если `FreeMouseMove` ложно), нам не нужно при каждом перемещении курсора мыши без нажатия кнопок (а таких перемещений большинство) передавать информацию в RdsCtrl.dll.

В конце функции `SetViewport` мы указываем Windows на необходимость обновить окно нашей программы вызовом `InvalidateRect`: после того, как мы привязали подсистему к порту вывода, нужно нарисовать изображение. Можно заметить, что нигде в `SetViewport` мы не задавали координаты и размер порта вывода – для простоты примера мы будем делать это каждый раз перед рисованием содержимого подсистемы.

Теперь нужно написать функцию `SetScrollBarParams`, которую мы вызываем из `SetViewport` и из некоторых других функций нашей программы: она будет устанавливать диапазон полос прокрутки согласно размерам рабочего поля привязанной к порту вывода подсистемы.

```
// Установить параметры полос прокрутки
void SetScrollBarParams(void)
{ int w,h,m;
  RECT rect;
  SCROLLINFO si;

  // Есть ли связь с РДС и порт вывода?
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;

  // Размеры прямоугольника, доступного в окне для порта
  if(!GetAvailableRect(&rect))
    return;

  // Получаем размер рабочего поля подсистемы в текущем масштабе
  if(!rdsctrlGetViewportSysArea(RdsLink,Viewport,-1.0,&w,&h))
    return;

  // Заполняем общие поля структуры для установки параметров
  // полосы прокрутки
  si.cbSize=sizeof(si);
  si.fMask=SIF_RANGE|SIF_PAGE|SIF_DISABLENOSCROLL;
  si.nMin=0;
```

```
    // Максимум прокрутки по горизонтали
    m=w-(rect.right-rect.left);
    if(m<0) m=0;
    si.nMax=m;
    si.nPage=rect.right-rect.left;
    SetScrollInfo(HorzScroll,SB_CTL,&si,TRUE);

    // Максимум прокрутки по вертикали
    m=h-(rect.bottom-rect.top);
    if(m<0) m=0;
    si.nMax=m;
    si.nPage=rect.bottom-rect.top;
    SetScrollInfo(VertScroll,SB_CTL,&si,TRUE);
  }
  //=======================================
```

Нам нужно согласовать диапазон каждой из полос с размером порта вывода и размером рабочей области схемы. Допустим, например, что в текущем масштабе рабочее поле имеет по вертикали размер в тысячу точек экрана, а область порта вывода – высоту сто точек. Положение вертикальной полосы прокрутки, то есть вертикальная координата верхней границы отображаемой области, может изменяться от нуля (видны первые сто точек рабочего поля) до девятисот (видны последние сто точек). Таким образом, чтобы получить максимально возможное значение полосы прокрутки, нужно из размеров рабочего поля вычесть размер прямоугольника порта вывода.

Сначала мы вызываем функцию `GetAvailableRect`, которая записывает координаты и размеры прямоугольника, доступного для порта вывода при текущем размере окна, в переменную `rect` типа `RECT` (стандартный тип Windows API, описывающий прямоугольник). В этой функции, которую мы не будем разбирать подробно, из всей внутренней области окна сверху вычитается горизонтальная полоса высотой в константу `VIEWPORTTOP` (там размещаются кнопки), снизу – высота горизонтальной полосы прокрутки, и справа – ширина вертикальной полосы прокрутки. Все оставшееся пространство окна считается доступным для порта вывода и записывается в структуру типа `RECT`, указатель на которую передается в функцию. Таким образом, ширину порта вывода можно вычислить как разность правой и левой координат прямоугольника `rect`: `rect.right-rect.left`, а высоту – как разность его нижней и верхней координат: `rect.bottom-rect.top`.

Для получения размеров рабочего поля подсистемы, привязанной к порту вывода, мы вызываем функцию `rdsctrlGetViewportSysArea`. В ее параметрах передается идентификатор связи с РДС (`RdsLink`), идентификатор порта (`Viewport`), масштабный коэффициент, для которого мы ходим получить размеры поля (мы передаем –1, поскольку нас интересует размер рабочего поля в текущем масштабе подсистемы), и указатели на целые переменные, в которые функция запишет ширину и высоту рабочего поля (`&w` и `&h`). Теперь мы знаем размеры рабочего поля и размеры порта вывода – можно устанавливать параметры полос прокрутки.

Для каждой из полос мы будем устанавливать два параметра: диапазон изменения (размер рабочего поля минус размер порта вывода) и размер страницы, от которого зависит размер движка на полосе, символизирующего положение текущей видимой области во всем диапазоне прокрутки. В Windows для задания параметров полос прокрутки служит структура `SCROLLINFO` (в нашей функции описана переменная `si` этого типа), поле `fMask` которой содержит флаги, определяющие устанавливаемые параметры. Мы будем использовать три флага: `SIF_RANGE` (установка диапазона), `SIF_PAGE` (установка размера страницы) и

SIF_DISABLENOSCROLL (запрет исчезновения полосы с экрана). Диапазон в нашем случае всегда начинается с нуля, поэтому в поле nMin мы сразу записываем 0.

Сначала мы устанавливаем параметры горизонтальной полосы – ее дескриптор находится в глобальной переменной HorzScroll. Максимум диапазона прокрутки m мы вычисляем как w-(rect.right-rect.left): ширина рабочего поля подсистемы минус ширина порта вывода. Если рабочее поле меньше порта вывода, значение m получится отрицательным, в этом случае мы делаем его нулевым. В поле максимума диапазона nMax структуры si мы записываем m, в поле размера страницы nPage – ширину порта вывода, после чего записываем эти параметры в горизонтальную полосу прокрутки HorzScroll функцией Windows API SetScrollInfo. После этого мы аналогичным образом настраиваем параметры вертикальной полосы VertScroll.

Теперь наши полосы прокрутки настраиваются согласно параметрам привязанной к рабочему полю подсистемы и реагируют на щелчки и перетаскивание своих движков (за это отвечает функция DoScroll, которая целиком состоит из различных вызовов Windows API, поэтому мы не будем ее подробно рассматривать). Однако, изображение подсистемы пока не рисуется – нужно написать функцию DrawViewport, которая вызывается из процедуры нашего окна при обработке сообщения Windows WM_PAINT:

```
// Рисование подсистемы в порте вывода
void DrawViewport(void)
{ RECT rect;
  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;
  // Получаем прямоугольник, доступный для вывода
  if(!GetAvailableRect(&rect))
    return;
  // Устанавливаем координаты и размер порта
  rdsctrlSetViewportRect(RdsLink,Viewport,
                      rect.left,rect.top,   // координаты
                      rect.right-rect.left, // ширина
                      rect.bottom-rect.top);// высота
  // Устанавливаем положение видимой в порте области
  rdsctrlSetViewportParams(RdsLink,Viewport,-1.0,
                 GetScrollPos(HorzScroll,SB_CTL),
                 GetScrollPos(VertScroll,SB_CTL),
                 0);
  // Командуем РДС перерисовать порт вывода
  rdsctrlUpdateViewport(RdsLink,Viewport);
}
//=======================================
```

В этой функции мы получаем координаты и размер доступной для отображения области окна уже знакомой нам функцией GetAvailableRect и передаем их в РДС функцией rdsctrlSetViewportRect. Затем мы устанавливаем координаты верхнего левого угла видимой через порт области рабочего поля функцией rdsctrlSetViewportParams, получив их из полос прокрутки вызовами GetScrollPos. Масштаб изображения подсистемы мы не меняем, поэтому вместо масштабного коэффициента передаем –1. После этого мы перерисовываем порт вывода вызовом rdsctrlUpdateViewport: изображение рабочего поля подсистемы в текущем установленном для порта масштабе и положении будет нарисовано в главном окне нашей программы.

Здесь мы устанавливаем размеры порта и положение прокрутки каждый раз непосредственно перед рисованием. Можно было бы улучшить нашу программу, например, устанавливая размеры порта при изменении размеров окна, а положение прокрутки – при

любом срабатывании горизонтальной или вертикальной полосы, но мы не будем делать этого, чтобы не усложнять пример.

Фактически, для рисования подсистемы в порте вывода необходимо выполнить следующие действия:

- создать порт вывода в каком-либо окне программы и привязать к нему подсистему вызовом `rdsctrlSetViewport`;
- задать координаты и размеры порта в этом окне вызовом `rdsctrlSetViewportRect`;
- задать положение видимой через порт области и масштаб изображения вызовом `rdsctrlSetViewportParams`;
- нарисовать изображение вызовом `rdsctrlUpdateViewport`.

Порт вывода запоминает установленные параметры, поэтому не так уж важно, как эти вызовы будут распределены по разным функциям программы.

Теперь в нашей программе можно открыть какую-либо схему, и изображение ее корневой подсистемы появится в окне (рис. 129). Его можно будет прокручивать по горизонтали и вертикали, но ни изменить масштаб, ни перейти в другую подсистему пока нельзя. Кроме того, блоки подсистемы пока не реагируют на мышь и клавиатуру и не выводят всплывающие подсказки. Исправление этих недостатков начнем с добавления возможности изменения масштаба.

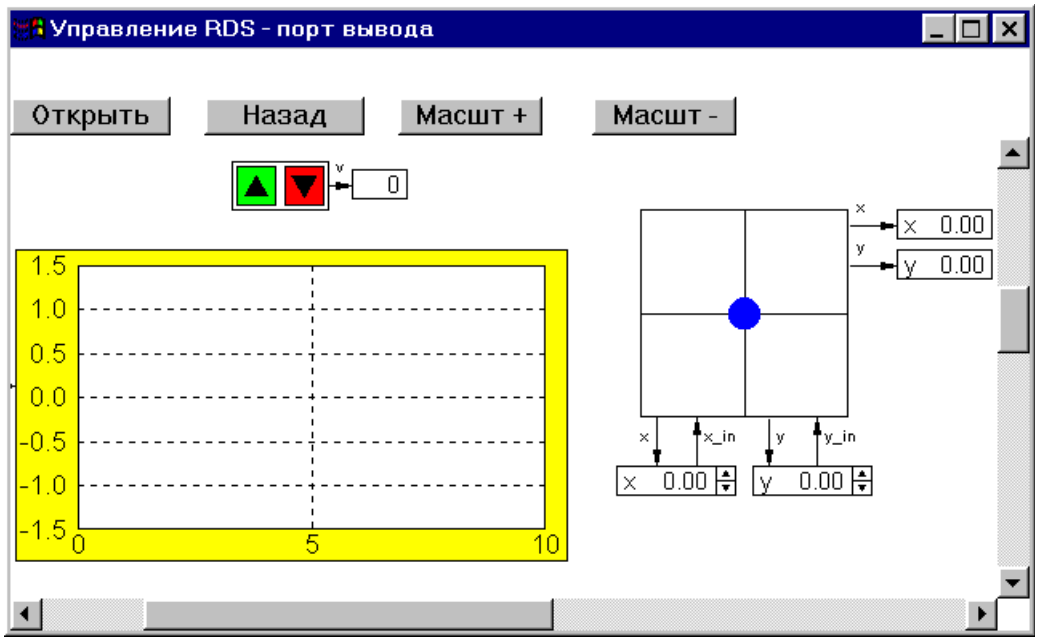


Рис. 129. Изображение подсистемы в порте вывода

Для изменения масштаба подсистемы мы предусмотрели функцию `ZoomButtonClick`, которая вызывается с разными множителями при нажатии кнопок “Масшт +” и “Масшт –”:

```
// Изменение масштаба (multiplier - множитель)
void ZoomButtonClick(double multiplier)
{ double zoom;
  RECT rect;
  int halfwidth,halfheight,cx,cy,sx,sy;

  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;
```

```
    // Получаем прямоугольник, доступный для вывода
    if(!GetAvailableRect(&rect))
      return;

    // Получаем текущий масштаб и положение прокрутки подсистемы
    rdsctrlGetViewportParams(RdsLink,Viewport,&zoom,&sx,&sy);

    // Вычисляем координаты центра видимой части в старом масштабе
    halfwidth=(rect.right-rect.left)/2,
    halfheight=(rect.bottom-rect.top)/2;
    cx=(sx+halfwidth)/zoom;
    cy=(sy+halfheight)/zoom;

    // Новый масштаб
    zoom*=multiplier;

    // Вычисление новых координат прокрутки, чтобы в новом масштабе
    // центр изображения не сдвинулся
    sx=(int)(cx*zoom)-halfwidth;
    if(sx<0) sx=0;
    sy=(int)(cy*zoom)-halfheight;
    if(sy<0) sy=0;

    // Установка нового масштаба и положения полос прокрутки
    rdsctrlSetViewportParams(RdsLink,Viewport,zoom,0,0,0);
    SetScrollBarParams();
    SetScrollPos(HorzScroll,SB_CTL,sx,TRUE);
    SetScrollPos(VertScroll,SB_CTL,sy,TRUE);

    // Обновление порта вывода
    InvalidateRect(MainWin,&rect,TRUE);
  }
  //=======================================
```

Задача этой функции – изменить текущий масштаб порта вывода в `multiplier` раз, причем так, чтобы объекты, находившиеся в центре порта, остались бы в центре и при новом масштабе. Для этого, после собственно изменения масштаба, необходимо также изменить координаты верхнего левого угла видимой в порте области, то есть изменить положение полос прокрутки.

Сначала мы получаем размеры области порта вывода вызовом функции `GetAvailableRect` – размеры порта понадобятся нам для определения его центра. Затем, вызвав `rdsctrlGetViewportParams`, мы считываем текущий масштаб порта и текущее положение левого верхнего угла видимой через него области в переменные `zoom`, `sx` и `sy` соответственно. Положение левого верхнего угла мы могли бы считать из полос прокрутки и вызовом `GetScrollPos`, но, в данном случае, удобнее получить все три значения вызовом одной функции. Теперь мы можем вычислить координаты (`cx`, `cy`), соответствующие точке рабочего поля подсистемы, которая находится в центре порта вывода в его текущем масштабе и положении. Затем мы умножаем текущий масштаб `zoom` на параметр функции `multiplier` и вычисляем новые координаты левого верхнего угла видимой области, такие, чтобы точка рабочего поля (`cx`, `cy`) оставалась в центре порта в новом масштабе. Осталось только установить новый масштаб порта вывода вызовом `rdsctrlSetViewportParams`, настроить диапазоны полос прокрутки на размер рабочего поля в новом масштабе вызовом `SetScrollBarParams` и установить их в новое положение вызовами `SetScrollPos`. После этого вызовом `InvalidateRect` мы даем Windows команду перерисовать область порта вывода при первой возможности – при рисовании функцией `DrawViewport`

содержимое подсистемы будет нарисовано согласно новому положению полос прокрутки и в новом масштабе.

Теперь наша программа увеличивает масштаб вдвое при нажатии кнопки “Масшт +” и уменьшает его вдвое при нажатии “Масшт –”. Обычно в программах ограничивают возможное увеличение и уменьшение масштаба какими-либо разумными пределами, но для упрощения примеры мы не будем этого делать. Мы также не будем выполнять увеличение масштаба в произвольной точке по щелчку мыши и выделение увеличиваемой области прямоугольной рамкой – все это легко реализуется при помощи функций RdsCtrl.dll.

§3.6.3. Смена отображаемой в порте подсистемы

В описываемый пример программы добавляются функции, позволяющие пользователю переходить от подсистемы к подсистеме в одном и том же порте вывода.

Следующее, что мы добавим в нашу программу – это возможность переходить из подсистемы в подсистему. Для этого у нас предусмотрены две функции: `SubSystemToViewport`, которая вызывается по двойному щелчку и заменяет привязанную к порту подсистему на подсистему, изображение которой находится под курсором мыши, и `BackButtonClick`, которая вызывается при нажатии кнопки “Назад” и привязывает к порту подсистему, родительскую по отношению к текущей. С помощью этих двух функций мы даем пользователю возможность перемещаться по подсистемам загруженной схемы примерно так же, как он перемещается из папки в папку в “Проводнике” Windows: двойной щелчок на подсистеме открывает ее в этом же порте вывода, нажатие кнопки “Назад” возвращает обратно в родительскую.

Начнем с функции `SubSystemToViewport` – в нее передаются координаты точки окна, в которой произошел двойной щелчок. В процедуре окна нашей программы мы вызываем эту функцию только в том случае, если функция `RdsMouseOperation` вернула `FALSE` (мы еще не делали в нашей программе реакцию на мышь, поэтому на данный момент эта функция всегда возвращает `FALSE`), то есть если двойной щелчок не обработан блоком схемы. Это в точности повторяет поведение РДС.

```
// Открыть в порте вывода подсистему под курсором мыши
void SubSystemToViewport(int x,int y)
{ TDynString blockname; // Имя подсистемы
  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;
  // Получаем имя блока в точке (x,y)
  if(!rdsctrlViewportBlockAtPos(RdsLink,Viewport,x,y,&blockname))
    return;
  // Если блока нет, ничего не делаем
  if(blockname.IsEmpty())
    return;
  // Привязываем подсистему blockname к порту вывода
  SetViewport(blockname.c_str);
  // Выводим ее полное имя в верхней части окна
  DisplayText(blockname.c_str);
}
//=======================================
```

Прежде всего нам нужно получить полное имя блока, изображение которого находится под курсором мыши, то есть в точке (`x`, `y`). Для этого мы вызываем функцию `rdsctrlViewportBlockAtPos`, передавая в нее идентификатор связи с РДС `RdsLink`, идентификатор порта вывода `Viewport`, координаты `x` и `y` и указатель на объект `blockname`, в который функция должна записать строку с полным именем блока (механизм возврата строк произвольной длины в RdsCtrl.dll был подробно рассмотрен на стр. 518). Эта

функция вернет нам в объекте либо полное имя блока, либо пустую строку – она не даст нам никакой информации, подсистема это или блок другого типа. Но нам это и не нужно: мы привязываем блок с этим именем к порту вывода вызовом `SetViewport` – если это не подсистема, РДС просто проигнорирует вызов. Таким образом, вызвав эту функцию с координатами курсора мыши мы перейдем в подсистему, находящуюся под курсором, а если такой подсистемы нет, никаких действий выполнено не будет.

Теперь напишем реакцию на нажатие кнопки “Назад”:

```
// Реакция на кнопку "Назад"
void BackButtonClick(void)
{ TDynString sysname; // Имя текущей подсистемы
  char *s;
  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;
  // Получаем имя подсистемы, привязанной к порту
  rdsctrlViewportSystem(RdsLink,Viewport,&sysname);
  if(sysname.IsEmpty()) // Строка пустая
    return;
  // Ищем последнее двоеточие в строке полного имени
  // и заменяем его на 0
  s=strrchr(sysname.c_str,':');
  if(s==NULL) // Нет двоеточия - некуда возвращаться
    return;
  *s=0;
  // Теперь из полного имени подсистемы выброшено имя
  // последнего блока, то есть мы получили имя
  // родительской подсистемы

  // Привязываем к порту родительскую подсистему
  SetViewport(sysname.c_str);
  // Выводим ее полное имя в верхней части окна
  DisplayText(sysname.c_str);
}
//=======================================
```

Задача этой функции – получить имя подсистемы, привязанной к порту в данный момент, выбросить из него последнее имя блока, после чего привязать к порту подсистему с получившимся именем. Поскольку полное имя любого блока в РДС состоит из цепочки имен подсистем на пути от корневой подсистемы до этого блока, разделенных двоеточиями (см. §1.4), выбросив из строки полного имени все после самого последнего двоеточия, мы получим имя родительской подсистемы.

Для получения полного имени текущей подсистемы порта вывода мы используем функцию `rdsctrlViewportSystem`, которая запишет это имя в строку `sysname` типа `TDynString`. Затем мы используем стандартную библиотечную функцию `strrchr` для того, чтобы найти в этой строке двоеточие, и, если оно есть, заменяем его нулевым байтом, то есть “обрезаем” строку в этой точке. Теперь в `sysname` находится имя родительской подсистемы, и мы привязываем ее к порту функцией `SetViewport`.

Внесенные в программу изменения позволяют пользователю перемещаться из подсистемы в подсистему, причем, поскольку для каждой подсистемы порт вывода запоминает положение прокрутки и масштаб, пользователь, вернувшись в какую-либо подсистему, увидит ее в том же масштабе и положении, в котором он ее оставил.

§3.6.4. Реакция на мышь и клавиатуру

В рассматриваемый пример программы добавляется реакция блоков отображаемой подсистемы на мышь и клавиатуру.

Добавим в нашу программу реакцию на мышь и клавиатуру, чтобы пользователь мог вводить данные в поля ввода, перемещать рукоятки и т.д. Фактически, наша задача – передать в РДС информацию о действиях пользователя в зоне порта вывода, заменив при этом константы Windows API на константы, используемые библиотекой RdsCtrl.dll (они описаны в “RdsCtrl.h”). Начнем с функции передачи реакции на мышь `RdsMouseOperation`: она вернет `TRUE`, если блок РДС среагировал на мышь, и `FALSE` в противном случае (при этом наша программа может выполнить какие-либо другие действия – например, перейти в другую подсистему по двойному щелчку).

```
// Реакция на мышь
BOOL RdsMouseOperation(UINT msg,int X,int Y,WPARAM keys)
{ DWORD flags=0;
  int op;
  RECT r;
  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return FALSE;
  // Получаем прямоугольник порта вывода
  if(!GetAvailableRect(&r))
    return FALSE;

  // Если курсор за пределами порта, ничего не делаем
  if(X<r.left||X>r.right||Y<r.top||Y>r.bottom)
    return FALSE;

  // В зависимости от сообщения, полученного окном,
  // устанавливаем флаги и операцию для РДС
  switch(msg)
    { case WM_LBUTTONDOWN: // Нажатие левой
        op=RDSCTRL_MOUSEOP_DOWN;
        flags=RDSCTRL_MOUSEF_LEFT;
        break;
      case WM_MBUTTONDOWN: // Нажатие средней
        op=RDSCTRL_MOUSEOP_DOWN;
        flags=RDSCTRL_MOUSEF_MIDDLE;
        break;
      case WM_RBUTTONDOWN: // Нажатие правой
        op=RDSCTRL_MOUSEOP_DOWN;
        flags=RDSCTRL_MOUSEF_RIGHT;
        break;
      case WM_LBUTTONUP: // Отпускание левой
        op=RDSCTRL_MOUSEOP_UP;
        flags=RDSCTRL_MOUSEF_LEFT;
        break;
      case WM_MBUTTONUP: // Отпускание средней
        op=RDSCTRL_MOUSEOP_UP;
        flags=RDSCTRL_MOUSEF_MIDDLE;
        break;
      case WM_RBUTTONUP: // Отпускание правой
        op=RDSCTRL_MOUSEOP_UP;
        flags=RDSCTRL_MOUSEF_RIGHT;
        break;
```

```
        case WM_LBUTTONDBLCLK: // Двойной щелчок левой
          op=RDSCTRL_MOUSEOP_DBL;
          flags=RDSCTRL_MOUSEF_LEFT;
          break;
        case WM_MOUSEMOVE: // Перемещение курсора
          op=RDSCTRL_MOUSEOP_MOVE;
          // Запоминаем флаги нажатых кнопок
          if(keys & MK_LBUTTON)
            flags|=RDSCTRL_MOUSEF_LEFT;
          if(keys & MK_MBUTTON)
            flags|=RDSCTRL_MOUSEF_MIDDLE;
          if(keys & MK_RBUTTON)
            flags|=RDSCTRL_MOUSEF_RIGHT;
          if(flags==0 && (!FreeMouseMove)) // Реакция не нужна
            return FALSE;
          break;
        default:
          return FALSE;
      }

    // Запоминаем флаги специальных клавиш клавиатуры
    if(keys & MK_CONTROL)
      flags|=RDSCTRL_MOUSEF_CTRL;
    if(keys & MK_SHIFT)
      flags|=RDSCTRL_MOUSEF_SHIFT;
    if(GetKeyState(VK_MENU)<0)
      flags|=RDSCTRL_MOUSEF_ALT;
    // Передаем в РДС
    return rdsctrlViewportMouse(RdsLink,Viewport,X,Y,op,flags);
  }
  //=======================================
```

Функция получилась довольно длинной из-за преобразования констант Windows API в константы “RdsCtrl.h”, тем не менее, устроена она очень просто. Параметрами функции являются разобранные параметры сообщения, полученного главным окном программы при каких-либо действиях пользователя мышью – эти сообщения подробно описаны в литературе по Windows API. В параметре `msg` передается тип полученного сообщения (нажата или отпущена кнопка, переместился курсор), в параметрах `X` и `Y` – координаты курсора, в параметре `keys` – флаги нажатых в данный момент служебных клавиш и кнопок. Прежде чем разбирать полученные параметры, мы проверяем, попал ли курсор в область порта вывода: если он находится за пределами порта, никаких действий совершать не нужно. Убедившись, что курсор попадает в порт вывода, мы анализируем тип сообщения `msg` и, в зависимости от него, присваиваем переменной `op` тип действия, на которое должен среагировать блок схемы (нажатие кнопки, отпускание кнопки, перемещение курсора, двойной щелчок), а переменной `flags` – флаг нажатой кнопки (левая, правая или средняя). Отдельно рассматривается перемещение курсора (сообщение Windows `WM_MOUSEMOVE`) – при этом анализируется глобальная переменная `FreeMouseMove`, которую мы устанавливаем при привязке подсистемы к порту вывода в функции `SetViewport`, и, если она равна `FALSE` (ни одному блоку подсистемы не нужна реакция на перемещение курсора без нажатия кнопок) и ни одна кнопка не нажата, мы прерываем выполнение функции.

После этого мы добавляем к переменной `flags` флаги состояния клавиш Ctrl, Shift и Alt и передаем информацию о мыши в РДС функцией `rdsctrlViewportMouse`. Результат этой функции будет и результатом и нашей функции `RdsMouseOperation`: если какой-либо блок подсистемы или сама подсистема среагирует на действия мышью, наша функция вернет `TRUE`.

Функция реакции на клавиатуру RdsKeyboardOperation устроена похожим образом:

```
// Реакция на клавиатуру
BOOL RdsKeyboardOperation(BOOL down,int keycode,DWORD flags)
{ int repeat;
  DWORD shift=0;
  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return FALSE;

  // Выделяем из данных о нажатии информацию об автоповторе
  if(down && (flags & 0x40000000)!=0)
    repeat=flags & 0xffff;
  else
    repeat=0;

  // Запоминаем состояние клавиш и кнопок мыши
  if(GetKeyState(VK_MENU)<0)
    shift|=RDSCTRL_KEYF_ALT;
  if(GetKeyState(VK_SHIFT)<0)
    shift|=RDSCTRL_KEYF_SHIFT;
  if(GetKeyState(VK_CONTROL)<0)
    shift|=RDSCTRL_KEYF_CTRL;
  if(GetKeyState(VK_LBUTTON)<0)
    shift|=RDSCTRL_KEYF_LEFT;
  if(GetKeyState(VK_RBUTTON)<0)
    shift|=RDSCTRL_KEYF_RIGHT;
  if(GetKeyState(VK_MBUTTON)<0)
    shift|=RDSCTRL_KEYF_MIDDLE;

  // Передаем информацию в РДС
  return rdsctrlViewportKeyboard(RdsLink,Viewport,
           down?RDSCTRL_KEYOP_DOWN:RDSCTRL_KEYOP_UP,
           keycode,repeat,shift);
}
//=======================================
```

В параметре down передается TRUE, если клавиша нажата, и FALSE, если отпущена, в параметре keycode – код клавиши, в параметре flags – дополнительные данные о нажатии, полученные процедурой главного окна программы при разборе сообщений WM_KEYDOWN, WM_SYSKEYDOWN, WM_KEYUP и WM_SYSKEYUP. Мы извлекаем из этих дополнительных данных информацию об автоповторе клавиши (число автоматически сгенерированных Windows нажатий при длительном удержании клавиши) и записываем ее в переменную repeat. В переменную shift мы записываем флаги нажатия клавиш Ctrl, Shift и Alt и кнопок мыши, полученные при помощи функции Windows API GetKeyState, а затем передаем всю эту информацию в РДС функцией rdsctrlViewportKeyboard.

Теперь мы транслируем все действия пользователя в РДС, и блоки текущей отображаемой подсистемы могут реагировать на щелчки, перемещения курсора, нажатия клавиш и т.п.

§3.6.5. Работа с контекстным меню блока

В рассматриваемый пример программы добавляется вывод контекстного меню блока или подсистемы, пункты которого запрашиваются у РДС.

В нашей программе мы обеспечили реакцию блоков отображаемой подсистемы на действия пользователя, однако, добавленные блоками в контекстное меню пункты

(см. §2.12.6) сейчас не отображаются – за контекстное меню отвечает наша программа, а в ней меню пока не реализовано. Нам придется сделать все вручную: получить у РДС список пунктов меню блока и связанных с ними чисел-идентификаторов, сформировать из них контекстное меню, показать его пользователю и, если он выберет какой-либо пункт, передать его идентификатор в РДС. Разумеется, мы можем добавить в меню не только пункты, полученные от блока схемы, но и свои собственные, но в этом примере мы не будем этого делать.

Для формирования контекстного меню и показа его пользователю мы предусмотрели функцию `ShowPopupMenu`, которая вызывается из процедуры главного окна в ответ на сообщение `WM_CONTEXTMENU`, и в которую передаются координаты курсора мыши на экране на момент вызова меню. Поскольку момент выбора пункта меню пользователем отстоит по времени от момента показа этого меню, мы будем записывать данные пунктов меню, полученных от РДС, в глобальный массив структур `MenuItems`, а при выборе пункта меню будем обращаться к этому массиву. Структуру и массив опишем следующим образом:

```
// Данные пункта контекстного меню блока
struct TPopupMenuData
{ TDynString BlockName; // Полное имя блока
  int MenuFunc; // Идентификатор пункта меню
  int MenuData; // Данные пункта меню
};
#define MENUITEMSMAXCOUNT 100 // Max число пунктов
TPopupMenuData MenuItems[MENUITEMSMAXCOUNT]; // Массив пунктов
int MenuItemsCount=0; // Число пунктов в массиве
//=======================================
```

Структура `TPopupMenuData` описывает данные одного пункта меню, полученные от РДС. В поле `BlockName` будет храниться полное имя блока, которому принадлежит пункт меню, в полях `MenuFunc` и `MenuData` – целые идентификаторы этого пункта. Массив `MenuItems` будет состоять из ста элементов – будем считать, что у блока не может быть больше ста пунктов контекстного меню. Фактическое число пунктов меню блока будет записываться в глобальную переменную `MenuItemsCount`.

Функция `ShowPopupMenu` будет выглядеть следующим образом:

```
// Показать контекстное меню
// (screenx,screeny) - координаты курсора
void ShowPopupMenu(int screenx,int screeny)
{ HMENU menu;
  MENUITEMINFO mi;
  RECT rect;
  POINT pt;
  int cnt,type;
  TDynString MenuBlockName;

  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;
  // Получаем прямоугольник порта вывода
  if(!GetAvailableRect(&rect))
    return;

  // Переводим экранные координаты курсора в оконные и
  // проверяем попадание в порт вывода
  pt.x=screenx;
  pt.y=screeny;
  ScreenToClient(MainWin,&pt);
```

```
if(!PtInRect(&rect,pt)) // не попали в порт
  return;

// Определяем блок, попавший под курсор
if(rdsctrlViewportBlockAtPos(RdsLink,Viewport,pt.x,pt.y,
              &MenuBlockName))
  type=RDSCTRL_MENUTYPE_BLK; // Нашли блок
else
  { // Под курсором блока нет - получаем меню подсистемы
    type=RDSCTRL_MENUTYPE_SYS;
    // Получаем имя подсистемы в порте вывода
    rdsctrlViewportSystem(RdsLink,Viewport,&MenuBlockName);
  }
// Выводим имя блока или подсистемы на индикацию
DisplayText(MenuBlockName.c_str);
// Теперь в MenuBlockName - имя блока, меню которого нужно
// получить, а в type - тип меню (меню блока или подсистемы)

// Читаем список пунктов меню блока  MenuBlockName
cnt=rdsctrlReadBlockMenuItems(RdsLink,MenuBlockName.c_str,type);
if(cnt==0) // Нет пунктов меню
  return;
if(cnt>MENUITEMSMAXCOUNT) // Слишком много пунктов
  cnt=MENUITEMSMAXCOUNT;

// Создаем меню и добавляем в него пункты
menu=CreatePopupMenu();
mi.cbSize=sizeof(mi);
mi.fMask=MIIM_DATA|MIIM_ID|MIIM_STATE|MIIM_TYPE;
MenuItemsCount=0;
for(int i=0;i<cnt;i++)
  { // Получаем указатель на структуру пункта меню i
    RDSCTRL_PMENUITEM item=rdsctrlGetMenuItemData(RdsLink,i);
    if(!item->Visible) // Невидимые пункты нам не нужны
      continue;
    // Идентификаторы наших пунктов начинаются с IDC_MENUSTART
    mi.wID=MenuItemsCount+IDC_MENUSTART;
    mi.dwTypeData=item->Text; // Название пункта
    mi.fState=0;
    if(item->Divider) // Это разделитель, а не пункт
      mi.fType=MFT_SEPARATOR;
    else
      { mi.fType=MFT_STRING;
        if(!item->Enabled) // Пункт запрещен
          mi.fState|=MFS_DISABLED;
        if(item->Checked) // Пункт с галочкой
          mi.fState|=MFS_CHECKED;
      }
    // Добавляем пункт в конец меню
    InsertMenuItem(menu,i,TRUE,&mi);
    // Запоминаем данные пункта в глобальном массиве
    MenuItems[MenuItemsCount].MenuFunc=item->MenuFunc;
    MenuItems[MenuItemsCount].MenuData=item->MenuData;
    MenuItems[MenuItemsCount].BlockName.Set(
                                item->BlockFullName);
    MenuItemsCount++;
  } // for(int i=0;...)
```

```
    // Показываем меню пользователю
    TrackPopupMenuEx(menu,0,screenx,screeny,MainWin,NULL);

    // Уничтожаем меню
    DestroyMenu(menu);
  }
  //=======================================
```

В этой функции мы, прежде всего, проверяем, попал ли курсор мыши в порт вывода. Для этого мы переводим координаты курсора на экране (`screenx`, `screeny`), переданные в функцию, в координаты внутри окна (`pt.x`, `pt.y`), и сравниваем их с границами прямоугольника порта вывода, полученные вызовом `GetAvailableRect`. Если курсор находится вне порта вывода, мы завершаем функцию и не выводим меню.

Затем мы вызываем уже знакомую нам функцию `rdsctrlViewportBlockAtPos` для определения имени блока, который находится в порте вывода по координатам (`pt.x`, `pt.y`) – имя блока записывается в строку `MenuBlockName`. Если функция вернула `TRUE`, то есть по данным координатам в порте есть блок, целой переменной `type` присваивается константа `RDSCTRL_MENUTYPE_BLK` – мы будем запрашивать у РДС контекстное меню блока. Если же она вернула `FALSE`, значит, курсор мыши находится на свободном участке рабочего поля схемы. В этом случае мы присваиваем переменной `type` константу `RDSCTRL_MENUTYPE_SYS`, чтобы получить контекстное меню окна подсистемы, а в строку `MenuBlockName` вызовом `rdsctrlViewportSystem` записываем имя подсистемы, в данный момент привязанной к порту вывода. Теперь, независимо от того, куда пришелся щелчок правой кнопкой мыши, вызвавший запрос контекстного меню, в строке `MenuBlockName` будет находиться имя блока, меню которого нужно вывести, а в переменной `type` – тип этого меню (`RDSCTRL_MENUTYPE_BLK` или `RDSCTRL_MENUTYPE_SYS`). Кроме пунктов меню блока или окна подсистемы мы можем запросить список пунктов меню, добавленных блоками в главное меню РДС (см. §2.12.7) – для этого используется константа `RDSCTRL_MENUTYPE_MAIN`. Эти пункты мы могли бы добавить, например, в главное меню нашей программы, однако, в этом примере мы не будем этого делать, чтобы не загромождать его: добавление пунктов в главное меню мало чем отличается от добавления пунктов в контекстное.

После того, как мы определились, какой тип меню какого блока нам нужен, мы считываем все данные этого меню во внутренние структуры RdsCtrl.dll функцией `rdsctrlReadBlockMenuItems`. Эта функция возвращает число считанных пунктов меню, которое мы записываем в переменную `cnt`. Если блок не добавляет в контекстное меню своих пунктов, `cnt` будет иметь нулевое значение, и мы завершаем функцию, не выводя контекстное меню. В противном случае нам нужно переписать данные пунктов меню из внутренних структур RdsCtrl.dll в наш собственный массив `MenuItems`. В данном примере мы могли бы и не делать этого, поскольку мы работаем с единственным портом вывода и вызываем `rdsctrlReadBlockMenuItems` только в одном месте нашей программы. Однако, лучше скопировать данные из внутренних структур RdsCtrl.dll как можно быстрее, поскольку эти структуры очищаются при каждом вызове `rdsctrlReadBlockMenuItems`.

Одновременно с записью данных пунктов меню в массив `MenuItems`, мы будем формировать само контекстное меню. Сначала мы создаем пустое меню `menu` вызовом функции Windows API `CreatePopupMenu`, а затем начинаем перебирать полученные из РДС пункты в цикле от 0 до `cnt`. Для получения указателя на структуру, описывающую `i`-й пункт меню, используется функция `rdsctrlGetMenuItemData`. В нее передается всего два параметра: связь с РДС `RdsLink` и номер пункта меню `i`. Функция возвращает указатель на структуру типа `RDSCTRL_MENUITEM` во внутренней памяти RdsCtrl.dll, описывающую

запрошенный пункт меню блока, который мы присваиваем переменной item. Структура RDSCTRL_MENUITEM описана в “RdsCtrl.h” следующим образом:

```
typedef struct
{ DWORD servSize;     // Размер этой структуры (для
                      // совместимости версий)
  LPSTR Text;         // Текст пункта меню
  BOOL Enabled;       // Пункт разрешен
  BOOL Visible;       // Пункт видим
  LPSTR BlockFullName;     // Имя блока-владельца пункта
  int MenuFunc,MenuData;   // Идентификаторы пункта
  BOOL Checked        // Пункт помечен
  BOOL Divider;       // Это - разделитель
  BOOL HasKey;        // Есть "горячая клавиша"
  int Key;            // Код клавиши
  DWORD KeyFlags;     // Флаги клавиши (RDS_M* è RDS_K*
                      // из RdsDef.h)
} RDSCTRL_MENUITEM;
```

Нас в этой структуре будут интересовать поля Text, Visible, Divider, Enabled и Checked, определяющие название и внешний вид пункта меню, а также BlockFullName, MenuFunc и MenuData, определяющие блок, к которому относится пункт, и целые идентификаторы этого пункта (их мы будем записывать в массив MenuItems). Для добавления пункта в меню menu мы заполняем структуру mi типа MENUITEMINFO согласно полученным через указатель item данным, а затем вызываем функцию InsertMenuItem. Поскольку мы работаем с контекстным меню, мы игнорируем все поля структуры RDSCTRL_MENUITEM, относящиеся к “горячей клавише” пункта меню (они пригодились бы нам, если бы мы добавляли пункты в главное меню нашей программы). Чтобы у всех пунктов меню были разные идентификаторы, мы делаем идентификатором пункта сумму константы IDC_MENUSTART, которой мы присвоили значение 200, и текущего числа элементов массива пунктов MenuItemsCount (оно увеличивается по мере добавления в меню пунктов). Таким образом, первый добавленный в меню пункт получит идентификатор 200, второй – 201, и т.д.

Добавив пункт в контекстное меню, мы записываем имя блока и два целых идентификатора пункта в очередной элемент массива MenuItems и увеличиваем на единицу число его элементов MenuItemsCount. Может возникнуть вопрос: зачем мы храним в массиве имя блока-владельца для каждого пункта меню? Контекстное меню всегда относится к какому-то одному блоку или подсистеме, поэтому нам достаточно было бы отвести одну глобальную переменную для имени блока, а в массиве хранить только целые идентификаторы. Однако, такой подход бы не сработал для пунктов главного меню РДС, у каждого из которых может быть свой блок-владелец. Хотя в этом примере мы и не работаем с главным меню, массив MenuItems и тип его элемента TPopupMenuData мы сделали универсальным, чтобы, при желании, главное меню можно было легко добавить.

Сформировав контекстное меню menu и заполнив массив MenuItems, мы показываем это меню по переданным в функцию экранным координатам функцией Windows API TrackPopupMenuEx, после чего уничтожаем меню функцией DestroyMenu – когда пользователь вызовет его в следующий раз, мы создадим его снова.

Теперь правый щелчок на изображении блока в порте вывода или на свободном месте подсистемы открывает контекстное меню с пунктами, созданными моделью этого блока или подсистемы (если, конечно, модель добавила в меню хотя бы один пункт). Осталось ввести в программу реакцию на выбор этих пунктов пользователем – для этого у нас предусмотрена функция PopupMenuClick, вызываемая из процедуры главного окна при получении сообщения WM_COMMAND (команда от объектов окна) с идентификатором, большим или равным IDC_MENUSTART.

```
// Выбор пункта контекстного меню
void PopupMenuClick(int id)
{ int index=id-IDC_MENUSTART; // Индекс в  MenuItems
  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;
  // Проверяем допустимость индекса пункта меню
  if(index<0 || index>=MenuItemsCount)
    return;
  // Вызываем пункт меню блока
  rdsctrlBlockMenuClick(RdsLink,
                        MenuItems[index].BlockName.c_str,
                        MenuItems[index].MenuFunc,
                        MenuItems[index].MenuData);
}
//=======================================
```

В эту функцию передается идентификатор выбранного пользователем пункта меню `id`. Поскольку в `ShowPopupMenu` мы давали пунктам последовательные идентификаторы, начиная с `IDC_MENUSTART`, для получения номера выбранного пункта, то есть, индекса `index` в массиве `MenuItems`, нам нужно вычесть `IDC_MENUSTART` из `id`. Таким образом, структура `MenuItems[index]` в поле `BlockName` содержит имя блока, которому принадлежит пункт меню, а в полях `MenuFunc` и `MenuData` – целые идентификаторы этого пункта. Эти данные мы передаем в РДС функцией `rdsctrlBlockMenuClick`, вызывающей в указанном блоке реакцию на выбор пункта меню.

§3.6.6. Вывод всплывающих подсказок

В рассматриваемый пример программы добавляется вывод всплывающих подсказок, текст и параметры которых определяются блоками отображаемой схемы.

Теперь займемся выводом всплывающей подсказки. Часть функций, отвечающих за ее вывод, уже готова: в функции `CreateControls` мы создали окно для показа подсказки (см. стр. 573), присвоив его дескриптор глобальной переменной `ToolTip`, и две области, для которых будет выводиться подсказка. Одна область связана с кнопкой “Открыть”, и для нее всегда будет выводится текст “Загрузить схему” – мы сделали ее просто для иллюстрации работы с подсказками. Вторая область, которой мы дали идентификатор `VIEWPORTTIP_ID`, пока не имеет размера – мы будем постоянно подстраивать ее размеры под текущее состояние порта вывода, а текст подсказки будем получать у РДС. На этом следует остановиться подробнее.

Число блоков, изображаемых в порте вывода, их положение и размеры зависят от выбранного пользователем масштаба и положения полос прокрутки, поэтому мы не можем заранее задать для каждого блока схемы фиксированную область всплывающей подсказки. Вместо этого мы создадим одну область подсказки на весь порт вывода и будем менять ее размеры при перемещении курсора мыши. В переменной `LastToolTipRect` типа `RECT` мы будем хранить координаты прямоугольной области последней выведенной подсказки (сразу после создания окна мы делаем этот прямоугольник пустым). Как только курсор мыши покидает область `LastToolTipRect`, мы будем расширять эту область подсказки до размеров всего порта вывода, а как только будет выведена очередная подсказка, мы уменьшим область ее действия вместе с `LastToolTipRect` до полученных из РДС размеров (обычно это размеры изображения блока, который отобразил подсказку).

Исходно и область действия подсказки, и `LastToolTipRect` пусты, поэтому, как только курсор мыши сдвинется, вызовется функция `AdjustToolTipRect`, и область действия подсказки будет расширена на весь порт вывода – таким образом, если курсор задержится над портом достаточное время, Windows запросит текст подсказки у нашей

программы, послав ее окну сообщение WM_NOTIFY с параметром TTN_NEEDTEXT. Процедура главного окна при этом вызовет еще не написанную нами функцию GetTooltip, которая должна запросить у РДС текст подсказки для данной точки порта вывода и передать его Windows для отображения. Кроме того, если в данной точке находится блок, выводящий подсказку, эта функция уменьшит зону действия подсказки до его размеров и скопирует ее в LastToolTipRect. Теперь зона подсказки ограничена изображением блока, и подсказка останется на экране до тех пор, пока либо не окончится время ее отображения, либо курсор мыши не покинет указанную область. Как только курсор выйдет за пределы изображения блока, то есть покинет прямоугольник LastToolTipRect, сработает проверка в реакции на сообщение WM_MOUSEMOVE в процедуре главного окна, и зона действия подсказки снова расширится на весь порт вывода, подготавливая таким образом программу к выводу новой подсказки. При выводе подсказки для другого блока зона действия снова сожмется до его размеров, и т.д. Это – не самый лучший и надежный способ организации вывода разных подсказок для одной обрасти окна, но для целей данного примера он вполне подходит. В реальной управляющей программе разработчик может выводить подсказки так, как ему удобно, тем более, что современные среды разработки обычно предоставляют для этого набор удобных функций.

Сейчас нам нужно написать функцию GetTooltip, которая запрашивает у РДС текст всплывающей подсказки и настраивает некоторые ее параметры:

```
// Запрос у РДС параметров всплывающей подсказки
void GetTooltip(TOOLTIPTEXT *tiptext)
{ BOOL CanShow;
  TOOLINFO ti;
  int left,top,right,bottom,reshowtimeout,hidetimeout;
  // Инициализация переданной структуры
  tiptext->hinst=NULL;
  tiptext->lpszText=NULL;
  tiptext->szText[0]=0;

  // Проверяем наличие связи с РДС и существование порта
  if(RdsCtrl==NULL || RdsLink<0 || Viewport<0)
    return;

  // Запрашиваем у РДС подсказку для точки (LastX,LastY)
  CanShow=rdsctrlVPPopupHint(RdsLink,Viewport,
                LastX,LastY,
                &HintString,
                &left,&top,&right,&bottom,
                &reshowtimeout,&hidetimeout);
  // Теперь текст подсказки записан в глобальную переменную
  // HintString. Переменная должна существовать и после
  // завершения этой функции, поэтому она сделана глобальной.

  if(CanShow && (!HintString.IsEmpty())) // Есть подсказка
    { // Заносим указатель на текст подсказки в структуру
      tiptext->lpszText=HintString.c_str;
      // Настраиваем время, через которое подсказка скроется
      SendMessage(ToolTip,TTM_SETDELAYTIME,
                (WPARAM)TTDT_AUTOPOP,(LPARAM)hidetimeout);
      // Настраиваем время повторного вывода подсказки
      SendMessage(ToolTip,TTM_SETDELAYTIME,
                (WPARAM)TTDT_RESHOW,(LPARAM)reshowtimeout);
      // Настраиваем новый прямоугольник подсказки
      ti.cbSize=sizeof(ti);
      ti.hwnd=MainWin;
```

```
            ti.uId=VIEWPORTTIP_ID;
            ti.rect.left=left;
            ti.rect.top=top;
            ti.rect.right=right;
            ti.rect.bottom=bottom;
            SendMessage(ToolTip,TTM_NEWTOOLRECT,0,(LPARAM)(&ti));
            // Запоминаем последний прямоугольник подсказки
            CopyRect(&LastToolTipRect,&(ti.rect));
          }
        }
        //=======================================
```

В эту функцию передается указатель на структуру `tiptext`, в которую она должна записать либо сам текст подсказки в поле `szText`, либо указатель на этот текст, размещенный где-то еще, в поле `lpszText`. Поскольку `szText` – это массив из восьмидесяти символов, а мы заранее не знаем, какого размера текст нам вернет РДС, мы будем писать текст подсказки в глобальную переменную `HintString` типа `TDynString`, а в поле `tiptext->lpszText` запишем указатель на внутренний динамический массив символов этой переменной. Нам нужно, чтобы текст подсказки не исчез после завершения функции `GetTooltip`, поэтому переменную `HintString` мы сделали глобальной.

Для запроса текста и параметров подсказки в конкретной точке порта вывода мы используем функцию `rdsctrlVPPopupHint`:

```
DWORD RDSCALL rdsctrlVPPopupHint(
      int Link,             // Идентификатор связи с РДС
      int VPort,            // Идентификатор порта вывода
      int x,int y,          // Точка порта вывода
      LPVOID hintstr,       // Объект для возврата текста
      int *pleft,int *ptop,      // Левый верхний и правый
      int *pright,int *pbottom,  // нижний углы области подсказки
      int *preshowtimeout,  // Интервал повторного вывода, мс
      int *phidetimeout);   // Интервал скрытия, мс
```

В параметрах `x` и `y` этой функции передаются координаты точки порта вывода, для которой запрашивается подсказка. Windows не передает координаты курсора при запросе текста подсказки, однако, при перемещении курсора мы все время запоминаем его текущие координаты в глобальных переменных `LastX` и `LastY`, поэтому в качестве координат точки запроса подсказки мы можем использовать значения этих переменных. В качестве объекта для возврата текста подсказки мы передаем указатель на глобальную переменную `HintString` (зарегистрированная нами функция возврата строк работает именно с классом `TDynString`), координаты прямоугольника подсказки мы записываем в переменные `left`, `top`, `right` и `bottom`, а интервалы повторного вывода подсказки после выхода курсора из прямоугольника (`left`, `top`) – (`right`, `bottom`) и интервал автоматического скрытия подсказки – в переменные `reshowtimeout` и `hidetimeout` соответственно (оба интервала возвращаются в миллисекундах). Функция `rdsctrlVPPopupHint` возвращает уникальный идентификатор блока, выводящего подсказку, или 0, если в данной точке порта вывода нет блока либо блок, находящийся там, подсказку не выводит. Сам идентификатор нам не нужен, нам нужно только знать, выводится подсказка или нет, поэтому результат возврата функции мы присваиваем логической переменной `CanShow`.

Если значение `CanShow` истинно, и строка подсказки `HintString` не пустая, мы передаем текст и параметры подсказки в Windows. Текст, то есть указатель на внутренний массив символов переменной `HintString`, мы записываем в структуру `tiptext` – Windows обработает этот текст, когда наша функция завершится. Временные интервалы подсказки и границы ее прямоугольника мы передаем окну подсказки `ToolTip` при помощи вызовов `SendMessage` с соответствующими параметрами (используемые для этого

структуры и константы подробно рассмотрены в литературе по Windows API). Кроме того, полученные от РДС границы прямоугольника подсказки мы копируем в глобальную переменную `LastToolTipRect`, чтобы при перемещении курсора можно было определить момент, когда он выйдет за пределы этого прямоугольника, и подготовиться к выводу новой подсказки.

Реализованный нами способ имеет один не очень очевидный недостаток: так можно выводить только однострочные подсказки, а блоки РДС часто выдают подсказки, состоящие из нескольких строк текста. Такая подсказка в нашей программе будет выглядеть как длинная строка со специальными символами в тех местах, где должны были бы быть переводы строк. Вывод многострочных подсказок выходит за рамки этого текста и не связан с организацией взаимодействия управляющей программы и РДС, поэтому мы не будем его рассматривать.

Теперь наша программа выполняет необходимый минимум функций: она может рисовать подсистему в своем окне, обеспечивает реакцию блоков на мышь и клавиатуру, вывод контекстных меню и всплывающих подсказок. В большинстве случаев для работы со схемами этого достаточно.

Глава 4. Создание модулей автоматической компиляции

Описывается принцип создания модулей автокомпиляции, которые по введенному пользователем тексту могут формировать и компилировать модель блока. Рассматривается пример такого модуля для создания простых моделей в синтаксисе языка С.

§4.1. Принцип работы модулей автокомпиляции

Рассматривается структура данных модуля автокомпиляции и его функции, описывается последовательность вызовов функции модуля при подключении и отключении моделей и выполнении компиляции.

Модули автоматической компиляции предназначены для того, чтобы облегчить создание моделей блоков пользователям, мало знакомым с программированием. Чтобы написать модель, даже если это простейшая функция, вычисляющая сумму двух входов блока и выдающая ее на выход, необходимо правильно оформить главную функцию динамически загружаемой библиотеки (DLL) и собственно функцию модели блока с учетом всех особенностей синтаксиса используемого языка программирования (описания экспортированной функции, включения всех необходимых заголовочных файлов и т.д.) Чаще всего пользователь не горит желанием изучать всю эту “лишнюю” для него информацию – он хочет просто указать с помощью какого-либо простого интерфейса, что блок имеет входы `x1` и `x2`, выход `y`, и выполняет операцию “`y=x1+x2`”. В этом ему может помочь грамотно написанный модуль автокомпиляции.

Модуль автокомпиляции нужен для того, чтобы преобразовать введенные пользователем тексты и данные, каким-либо образом описывающие работу блока, в полноценную библиотеку с экспортированной функцией модели, которая может быть загружена в РДС и подключена к указанным пользователем блокам. При этом РДС совершенно не важно, как именно пользователь описывает алгоритм работы блока и как эти алгоритмы трансформируются в готовую библиотеку – всем этим должен заниматься модуль автокомпиляции. Обычно имеет смысл дать пользователю возможность писать основные, самые важные, фрагменты модели блока на каком-либо языке программирования высокого уровня, а затем автоматически собирать из них полный исходный текст библиотеки со всеми необходимыми описаниями и вызывать внешний компилятор для ее сборки: так достигается баланс между удобством пользователя и простотой реализации модуля. Именно так устроены модули автокомпиляции, входящие в комплект РДС: они предоставляют пользователю редактор, в котором он записывает реакции блока на различные события в синтаксисе языка С (при этом он может пользоваться функциями из стандартных библиотек, функциями Windows API и сервисными функциями РДС), после чего собирают из них библиотеку с функцией модели, вызывая один из стандартных компиляторов.

Модуль автокомпиляции представляет собой экспортированную из какой-либо библиотеки функцию следующего вида:

```
extern "C" __declspec(dllexport) int RDSCALL имя_функции_модуля(
  int CallMode,                          // Режим вызова
  RDS_PCOMPMODULEDATA ModuleData,        // Данные модуля
  LPVOID ExtParam)                       // Дополнительные параметры
```

Как только в схеме появляется блок, для которого установлена автоматическая компиляция модели каким-либо модулем, РДС загружает библиотеку с функцией этого модуля и начинает вызывать эту функцию для выполнения различных действий в ответ на системные события (подключение моделей к блокам и их отключение, компиляция модели, вызов редактора и т.п.). Можно заметить, что функция модуля автокомпиляции очень похожа на функцию модели блока – как и в функцию модели, в нее передается целый идентификатор события `CallMode`, на которое должен отреагировать модуль, указатель `ModuleData` на структуру данных объекта (в данном случае объектом является модуль автокомпиляции, а не

блок схемы) и указатель `ExtParam` на дополнительные параметры, тип и структура которых зависит от конкретного события. Разумеется, события, на которые реагирует модуль автокомпиляции, отличаются от событий, на которые реагирует функция модели.

Прежде, чем модуль автокомпиляции можно будет использовать в блоках схемы, его необходимо зарегистрировать в РДС, открыв окно со списком модулей пунктом меню “Сервис | Автокомпиляция...” (рис. 130) и добавив новый модуль кнопкой “+”. В нижней части окна для добавленного модуля нужно указать имя файла библиотеки, в которой находится его функция (можно использовать условные обозначения стандартных путей, например, “DLL”), экспортированное имя функции модуля в этой библиотеке и название, под которым пользователь будет видеть этот модуль в интерфейсах РДС.

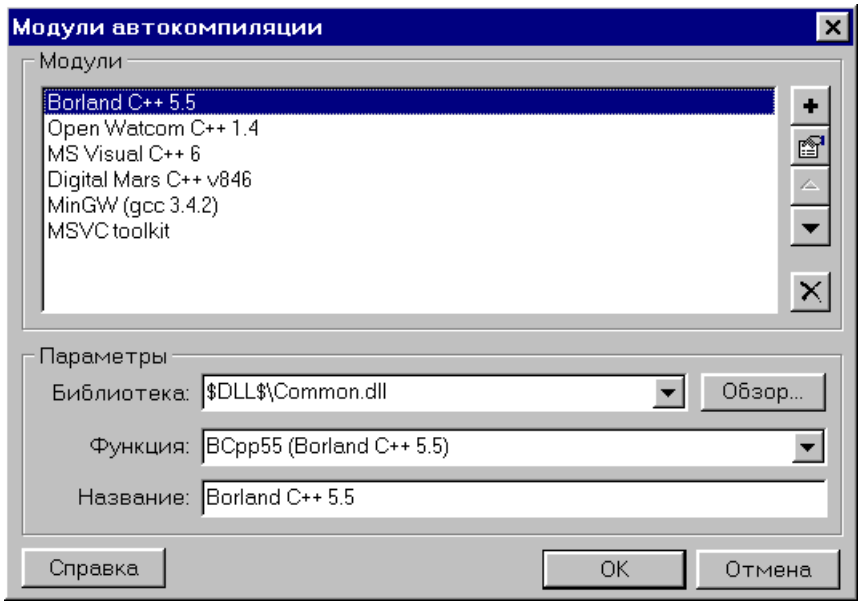


Рис. 130. Окно списка модулей автокомпиляции

В этом же окне можно настраивать модули автокомпиляции (двойным щелчком на названии модуля в списке или специальной кнопкой в правой части окна). В РДС нет собственного интерфейса для настройки модулей, его должна обеспечить функция настраиваемого модуля. Когда пользователь вызывает настройку какого-либо модуля, РДС, в свою очередь, вызывает его функцию с параметром `CallMode`, равным `RDS_COMPM_SETUP` (все константы и структуры, используемые для создания модулей автокомпиляции, описаны в файле “RdsComp.h”). Реагируя на этот вызов, функция должна самостоятельно обеспечить пользователю возможность ввода параметров, необходимых для работы модуля, например, открыв какое-либо окно с полями ввода. Если на момент настройки библиотека с модулем не загружена, РДС загружает ее перед вызовом функции и выгружает после него. При этом сразу после загрузки функция модуля вызывается с параметром `RDS_COMPM_INIT`, а перед выгрузкой – с параметром `RDS_COMPM_CLEANUP` для инициализации данных модуля и их очистки соответственно (инициализация и очистка данных модуля будут рассмотрены ниже).

В отличие от функции модели блока, функция модуля автокомпиляции должна заботиться еще и о том, как и где она будет хранить настроечные параметры модуля. Если параметры блока обычно сохраняются в файле схемы, в которой находится этот блок, с параметрами модуля автокомпиляции так поступать нельзя: они не связаны с конкретной схемой и относятся ко всему модулю в целом. Кроме того, они могут различаться на разных машинах (например, путь к исполняемому файлу используемого внешнего компилятора зависит от того, в какую папку он установлен на данной машине), поэтому при переносе схемы с машины на машину параметры модулей автокомпиляции не должны переноситься вместе с ней. Чаще всего функция хранит параметры своего модуля в каком-либо файле в

стандартной папке настроек РДС (символическое имя “INI”), и всей работой с этим файлом занимается самостоятельно.

Для того, чтобы связать блок с автоматически компилируемой моделью, пользователь должен на вкладке “Компиляция” окна параметров блока (рис. 131) выбрать модуль, который будет обслуживать модель блока, и указать имя этой модели.

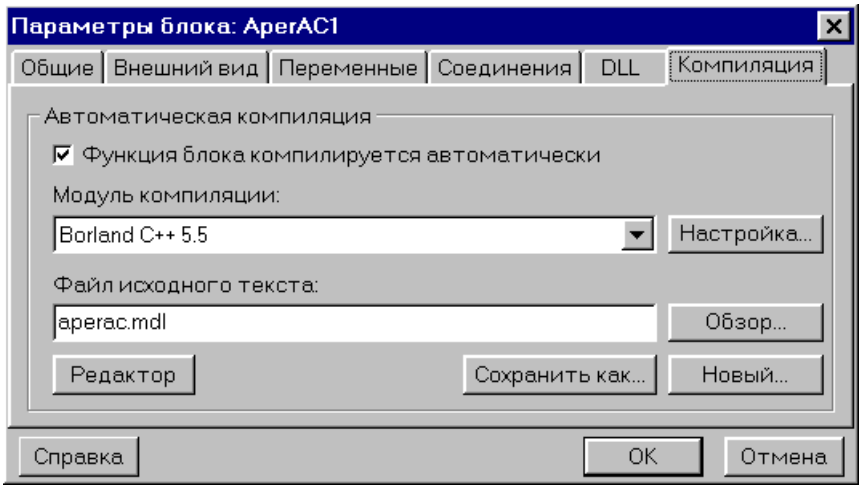


Рис. 131. Подключение автокомпилируемой модели к блоку

С точки зрения РДС, имя модели – это просто строка символов, не чувствительная к регистру (имена “Model”, “model” и “MODEL” будут считаться именами одной и той же модели). Сами модели блоков, как и параметры модулей, не сохраняются в файле схемы вместе с параметрами блока, к которому они подключены. Дело в том, что одна и та же модель может использоваться в разных схемах, и, если бы ее текст хранился в каждой из этих схем независимо, для изменения модели пришлось бы менять ее текст во всех этих схемах. Поэтому в схеме сохраняется только имя модели, имя библиотеки с обслуживающим ее модулем автокомпиляции и имя функции этого модуля. Модуль автокомпиляции должен самостоятельно организовать хранение текста модели, с которым работает пользователь, и уметь вызывать нужную модель по имени. Чаще всего каждая модель хранится в отдельном файле, и в качестве ее имени используется имя этого файла – такой подход позволяет реализовать поиск и хранение моделей очень просто. Тем не менее, разработчик модуля автокомпиляции может выбрать и другие способы хранения – например, хранить модель в таблице какой-либо базы данных, а в качестве имени модели использовать уникальный ключ (идентификатор) записи таблицы. РДС никак не ограничивает разработчика в этом вопросе.

Таким образом, хранение автоматически компилируемых моделей отдельно от схемы приводит к тому, что внесение изменений в какую-либо модель отражается на всех схемах, которые эту модель используют. При этом следует помнить, что для переноса схемы с автокомпилируемыми моделями на другую машину необходимо скопировать на нее не только файл схемы, но и все используемые этой схемой модели. Либо, если на другой машине не нужна автоматическая компиляция, отключить ее в параметрах всех блоков – при этом блоки останутся связанными с уже скомпилированными библиотеками, но исходный текст модели им будет уже не нужен. Разумеется, скомпилированные библиотеки нужно будет скопировать на другую машину вместе со схемой.

На вкладке “Компиляция” окна параметров блока имя модели может вводиться вручную или выбираться при помощи кнопок “Обзор” (для подключения к блоку существующей модели) и “Новый” (для создания новой модели). РДС не обрабатывает нажатия этих кнопок, они передаются в функцию модуля автокомпиляции, как и нажатие кнопки “Сохранить как”, которая позволяет сохранить используемую модель под другим

именем. Модуль должен сам выполнить все необходимые действия по подключению новой модели, показу необходимых для этого диалогов и т.д. При необходимости модуль может запретить эти кнопки, если он не поддерживает соответствующие функции, а также задать название поля ввода имени модели (например, выбранный на рис. 131 модуль хранит модели в файлах, поэтому в качестве названия поля ввода он установил строку “Файл исходного текста”).

Каждый модуль автокомпиляции может одновременно обслуживать произвольное число моделей, каждая из которых может быть подключена к произвольному числу блоков (рис. 132).

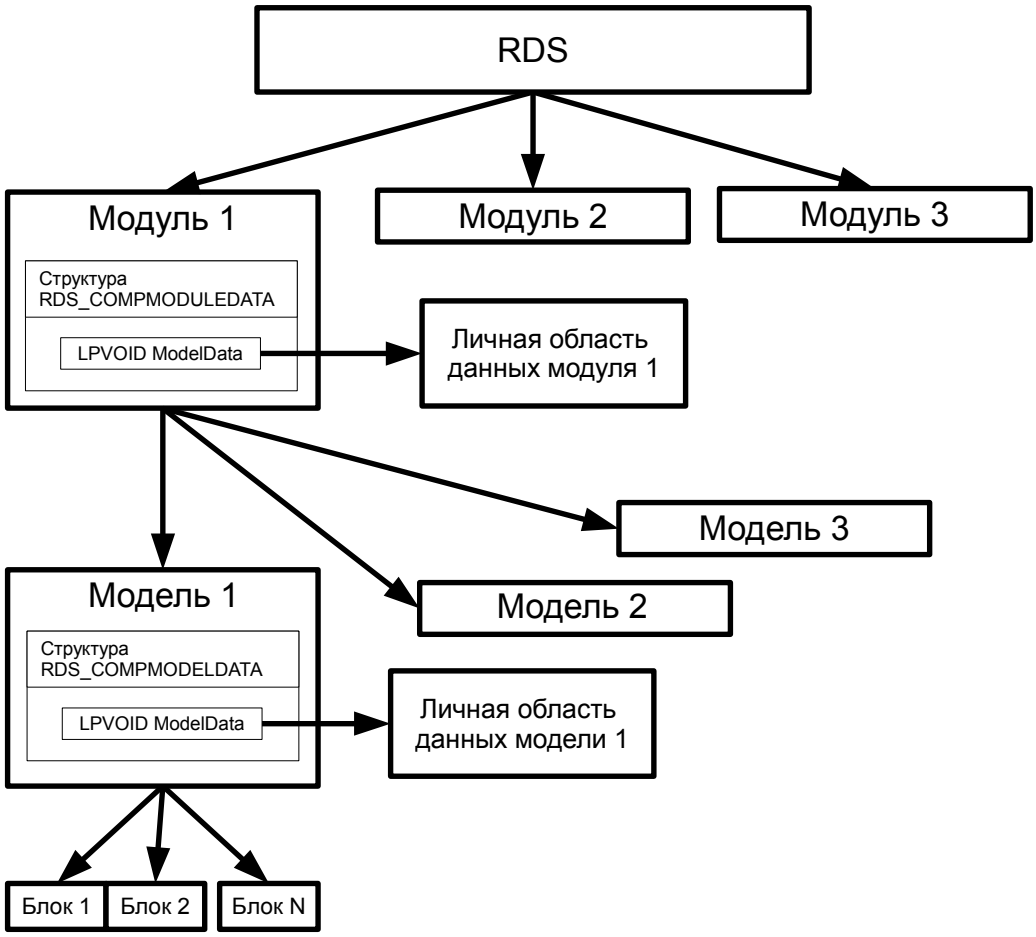


Рис. 132. Структура данных модуля автокомпиляции

При загрузке модуля автокомпиляции для него создается структура `RDS_COMPMODULEDATA`, которая хранится в памяти до момента его выгрузки, то есть до тех пор, пока последняя модель, обслуживаемая этим модулем, не будет отключена от блока. Описание этой структуры в файле “RdsComp.h” имеет следующий вид:

```
typedef struct
{ RDS_COMPHANDLE Module;    // Уникальный идентификатор модуля
  LPVOID ModuleData;        // Личная область данных модуля
  LPSTR DllFullPath;        // Полный путь к DLL модуля
  LPSTR DllFuncName;        // Имя функции DLL модуля
  int NModels;              // Число моделей, обслуживаемых модулем
} RDS_COMPMODULEDATA;
typedef RDS_COMPMODULEDATA *RDS_PCOMPMODULEDATA; // Указатель
```

Указатель на эту структуру передается во втором параметре функции модуля автокомпиляции (см. стр. 600). В поле ModuleData функция может записать указатель на личную область данных модуля, в которой она может хранить, например, его настроечные параметры. РДС не обрабатывает это поле структуры, поэтому отведением памяти под личную область и ее освобождением должна заниматься функция модуля, точно так же, как функция модели блока отводит и освобождает память под личную область данных блока (см. §2.4). Все остальные поля структуры функция модуля может только читать – они заполняются РДС автоматически.

Для каждой модели, обслуживаемой модулем, в памяти хранится структура RDS_COMPMODELDATA. Она создается РДС при первом подключении модели к какому-либо блоку и хранится в памяти до тех пор, пока в схеме остаются блоки, подключенные к этой модели. Функция модуля автокомпиляции в любой момент может получить указатель на структуру любой из своих моделей, вызвав сервисную функцию rdscompGetModelData и указав в ее параметре идентификатор модуля и номер модели (общее число моделей, обслуживаемых модулем в данный момент, всегда хранится в поле NModels структуры данных модуля RDS_COMPMODULEDATA). Структура RDS_COMPMODELDATA описана в файле “RdsComp.h” следующим образом:

```
typedef struct
{ RDS_MODELHANDLE Model;    // Уникальный идентификатор модели
  LPSTR ModelName;          // Имя модели
  LPSTR ModelNameUC;        // Имя модели в верхнем регистре
  LPVOID ModelData;         // Личная область данных модели
  int NBlocks;              // Число блоков, связанных с моделью
  RDS_COMPHANDLE Module;    // Идентификатор обслуживающего модуля
  // Параметры скомпилированной модели
  LPSTR CompDllName;        // DLL скомпилированной модели
  LPSTR CompDllFunc;        // Имя экспортированной функции
  BOOL Valid;               // Признак необходимости компиляции

  LPSTR AltModelName;       // Альтернативное имя модели
  int Tag;                  // Пользовательское поле
} RDS_COMPMODELDATA;
typedef RDS_COMPMODELDATA *RDS_PCOMPMODELDATA; // Указатель
```

Как и у модуля автокомпиляции, у модели может быть личная область данных, в которой функция модуля может хранить какие-либо необходимые для компиляции этой модели данные. Как обычно, отведением и освобождением памяти под эту область функция модуля должна заниматься самостоятельно: при первом подключении модели к блоку функция модуля вызывается с параметром RDS_COMPM_MODELINIT, в этот момент она может отвести память под личную область модели и записать указатель на нее в поле ModelData структуры RDS_COMPMODELDATA (см. рис. 132). При отключении модели от последнего блока схемы функция модуля будет вызвана с параметром RDS_COMPM_MODELCLEANUP, и должна будет освободить отведенную под личную область данных модели память.

Кроме поля ModelData функция модуля может изменять в этой структуре еще только два поля: признак необходимости компиляции Valid (его использование будет описано ниже) и целое поле Tag, которое никак не обрабатывается РДС и может использоваться разработчиком модуля по своему усмотрению. Все остальные поля либо устанавливаются РДС, либо меняются функцией модуля не непосредственно, а через вызовы специальных сервисных функций.

В поля ModelName и ModelNameUC РДС записывает указатели на строки в своей внутренней памяти, содержащие имя модели в том виде, в котором его ввел пользователь, и преобразованное в верхний регистр соответственно (как было указано выше, имя модели не

чувствительно к регистру). Функция модуля никак не может изменить эти строки, они заполняются автоматически при первом подключении модели к блоку схемы.

В целом поле `NBlocks` содержится общее число блоков схемы, к которым подключена данная модель. Это поле изменяется автоматически при подключении модели к блокам и отключении от них. Функция модуля может получить идентификатор любого из использующих данную модель блоков и описание этого блока, вызвав сервисную функцию `rdscompGetModelBlock` и передав ей идентификатор модели, номер блока и, при необходимости, указатель на структуру описания блока `RDS_BLOCKDESCRIPTION`.

В полях `CompDllName` и `CompDllFunc` содержатся указатели на строки, содержащие имя DLL, которая будет создана в результате компиляции данной модели, и имя экспортированной из нее функции блока. Эти имена устанавливаются функцией модуля перед компиляцией при помощи сервисной функции `rdscompSetModelFunction`, после компиляции РДС использует их, чтобы загрузить новую модель и подключить ее ко всем использующим ее блокам. Логическое поле `Valid` обычно используется для того, чтобы зря не компилировать модели, текст и параметры которых не изменились: при подготовке к компиляции функция модуля записывает в это поле значение `FALSE`, если требуется компиляция, и `TRUE`, если модель с момента прошлой компиляции не менялась. Если исходные тексты моделей хранятся в файлах, необходимость компиляции проще всего выяснить, сравнив время последнего изменения файла исходного текста модели с временем последнего изменения скомпилированной DLL. Если текст модели менялся позднее, чем была записана DLL, значит, он изменился, и DLL необходимо скомпилировать заново.

В поле `AltModelName` хранится указатель на строку, представляющую собой так называемое “альтернативное имя” модели. Это имя устанавливается сервисной функцией `rdscompSetAltModelName` и запоминается в файле схемы вместе с “нормальным” именем модели. Пользователь не видит альтернативное имя модели и установить его не может (при подключении модели через окно параметров блока, как на рис. 131, альтернативному имени присваивается пустая строка). Поскольку исходные тексты моделей хранятся отдельно от файлов схем, альтернативное имя является единственной возможностью записать в файл схемы какую-либо информацию, относящуюся к модели. Разработчик модуля автокомпиляции может использовать его по своему усмотрению, РДС это альтернативное имя никак не обрабатывает. Например, если тексты моделей хранятся в файлах, в альтернативном имени можно хранить полный путь к такому файлу, а в “нормальном имени” – короткий путь, возможно, с использованием символических констант.

Допустим например, что файл схемы имеет имя “scheme.rds”, файл модели – “model.mod”, и оба они находятся в папке “d:\data\files”. В этом случае в качестве имени модели целесообразно использовать текст “model.mod” – поскольку сервисные функции РДС позволяют автоматически добавлять к именам файлов без путей путь к папке, в которой находится файл загруженной схемы, схема будет работать до тех пор, пока файл схемы и файл модели находятся в одной и той же папке, куда бы они ни были скопированы. Однако, если пользователь скопирует файл схемы в другую папку, а файл модели скопировать забудет, модуль автокомпиляции “потеряет” файл модели: файл теперь находится не в папке файла схемы, и имени файла без пути для его загрузки уже недостаточно. Из этого положения можно выйти, записывая в альтернативное имя модели путь к папке, в которой находится файл модели на момент последней удачной загрузки, то есть “d:\data\files”. В этом случае модуль автокомпиляции должен быть написан так, чтобы, не обнаружив файл модели в ожидаемом месте (в данном случае – в папке файла схемы), он искал его в папке, указанной в строке альтернативного имени модели. Разумеется, разработчик может придумать и другие способы использования альтернативного имени.

Наконец, целое поле `Tag` структуры `RDS_COMPMODELDATA` может использоваться разработчиком по своему усмотрению. Функция модуля может записывать в это поле что угодно – РДС его никак не обрабатывает.

Автокомпилируемая модель может подключаться к блоку по нескольким причинам. Кроме явного включения автокомпиляции в параметрах блока (см. рис. 131), модели подключаются к блокам при загрузке схемы, при вставке блока из буфера обмена или из библиотеки блоков и т.п. Независимо от причины, при этом выполняется следующая последовательность действий:

1. Если выбранный модуль автокомпиляции еще не используется в схеме, его DLL загружается в память РДС, для него создается структура `RDS_COMPMODULEDATA`, после чего функция модуля вызывается с параметром `RDS_COMPM_INIT`. В этот момент она обычно создает личную область данных модуля и загружает в нее настроечные параметры, необходимые для его работы.
2. Функция модуля вызывается с параметром `RDS_COMPM_CANATTACHBLK`, и в нее передается имя модели и идентификатор блока, к которому будет подключаться эта модель. Структура данных модели `RDS_COMPMODELDATA` на этот момент может быть еще не создана. Реагируя на этот вызов, функция модуля должна проверить принципиальную возможность подключения данной модели к данному блоку. Например, модель, использующая статические переменные, не может подключаться к подсистемам, внешним входам/выходам и вводам шин, поскольку их структура переменных устроена не так, как у простых блоков. Если подключение модели к блоку невозможно, функция либо возвращает сообщение об ошибке, либо вызывает сервисную функцию `rdscompAttachDifferentModel`, сообщая РДС, что, хотя модель с данным именем подключить нельзя, вместо нее можно подключить модель с другим именем (например, если файл, соответствующий имени модели, отсутствует, но по альтернативному имени можно найти другой, как в примере, описанном выше).
3. Если данная модель еще не подключалась ни к одному блоку схемы, для нее создается структура `RDS_COMPMODELDATA`, функция модуля вызывается с параметром `RDS_COMPM_MODELINIT`, и в нее передается указатель на структуру данных модели. Реагируя на этот вызов, функция может создать для модели личную область данных, если это необходимо.
4. Функция модуля вызывается с параметром `RDS_COMPM_ATTACHBLOCK` и в нее передается указатель на структуру данных модели `RDS_COMPMODELDATA` и идентификатор блока, к которому она подключается.

При отключении автокомпилируемой модели от блока (опять же, причины отключения могут быть различными: выключение автокомпиляции в параметрах блока, удаление блока, завершение РДС и т.п.) выполняется следующая последовательность действий:

1. Функция модуля автокомпиляции вызывается с параметром `RDS_COMPM_DETACHBLOCK`, и в нее передается указатель на структуру данных модели `RDS_COMPMODELDATA` и идентификатор блока, от которого она сейчас будет отключена.
2. Если это был последний блок, с которым связана данная модель, функция модуля вызывается с параметром `RDS_COMPM_MODELCLEANUP`, при этом в нее передается указатель на структуру данных модели `RDS_COMPMODELDATA`. Реагируя на этот вызов, функция должна уничтожить личную область данных, если она создавалась для этой модели. После этого структура данных модели уничтожается.
3. Если данный модуль компиляции больше не используется в схеме (то есть последняя модель этого модуля отключена от последнего использовавшего ее блока), функция модуля вызывается с параметром `RDS_COMPM_CLEANUP`, после чего данные модуля уничтожаются и его DLL выгружается из памяти. Реагируя на вызов

RDS_COMPM_CLEANUP, функция должна уничтожить личную область данных модуля, если она создавалась.

Таким образом, функция модуля автокомпиляции всегда получает информацию о загрузке и выгрузке своего модуля, о создании структур данных новых моделей и уничтожении более не используемых, и о подключении моделей к новым блокам и их отключении.

РДС вызывает модуль автокомпиляции для компиляции обслуживаемых им моделей в четырех случаях: после загрузки схемы или блока, при переходе из режима редактирования в режим моделирования и при выборе пользователем пунктов меню “Система | Компилировать модели” или “Система | Перекомпилировать все модели”. В последнем случае подразумевается, что модуль должен скомпилировать даже те модели, которые не менялись с момента прошлой компиляции – это может быть полезно, например, при сбое системных часов, когда время записи скомпилированной DLL модели оказывается большим времени изменения ее текста из-за того, что время изменения текста показывается неправильно. За компиляцию моделей отвечают два последовательных вызова функции модуля: сначала функция вызывается с параметром RDS_COMPM_PREPARE для каждой модели по отдельности, а затем – для всех моделей сразу с параметром RDS_COMPM_COMPILE.

При вызове функции модуля с параметром CallMode, равным RDS_COMPM_PREPARE, функция модуля должна проверить необходимость компиляции конкретной модели, и подготовить ее для этого, если необходимо. В параметре ExtParam (см. стр. 600) при этом передается указатель на структуру RDS_COMPPREPAREDATA, содержащую два поля:

```
typedef struct
{ RDS_PCOMPMODELDATA Model;      // Указатель на структуру данных
                                 // модели
  BOOL Rebuild          // TRUE, если необходимо принудительно
                        // компилировать все модели
} RDS_COMPPREPAREDATA;
typedef RDS_COMPPREPAREDATA *RDS_PCOMPPREPAREDATA; // Указатель
```

В поле Model этой структуры содержится указатель на структуру данных модели, а в логическом поле Rebuild – TRUE, если пользователь выбрал пункт меню “Система | Перекомпилировать все модели” (то есть если модель нужно компилировать даже тогда, когда функция модуля не видит в этом необходимости) и FALSE в противном случае. Реагируя на этот вызов, функция должна установить поле Valid в структуре, указатель на которую находится в поле Model. Если данная модель должна быть скомпилирована, Valid нужно присвоить FALSE, если же компилировать не нужно, Valid присваивается TRUE.

Если в результате вызовов функции модуля с параметром RDS_COMPM_PREPARE хотя бы в одной структуре модели поле Valid получило значение FALSE, или если пользователем была выбрана принудительная компиляция всех моделей, функция вызывается с параметром RDS_COMPM_COMPILE. В параметре ExtParam при этом передается указатель на структуру RDS_COMPILEDATA:

```
typedef struct
{ // Массив указателей на структуры моделей, которые необходимо
  // компилировать
  RDS_PCOMPMODELDATA *InvalidModels;
  int IMCount;  // Размер массива (число моделей)
  BOOL Rebuild; // TRUE - принудительная компиляция
} RDS_COMPILEDATA;
typedef RDS_COMPILEDATA *RDS_PCOMPILEDATA; // Указатель
```

В поле InvalidModels находится указатель на массив указателей на структуры данных (RDS_COMPMODELDATA) моделей, которые должны быть скомпилированы, а в поле IMCount – размер этого массива. Функция модуля должна перебрать все модели в этом массиве и скомпилировать каждую из них. Перед этим вызовом РДС выгружает из памяти

все DLL этих моделей, и снова загружает их только после того, как функция модуля вернет ему управление – таким образом, DLL не будут заблокированы, и функция сможет заменить их на новые, полученные в результате компиляции. В поле `Rebuild` передается признак принудительной компиляции всех моделей, но функция модуля может его игнорировать, поскольку в любом случае в массиве `InvalidModels` будет находиться список моделей, которые нужно компилировать, независимо от того, попали они туда из-за того, что при вызове с параметром `RDS_COMPM_PREPARE` флаг `Valid` в их структурах был установлен в `FALSE`, или из-за того, что пользователь приказал компилировать все модели схемы.

Кроме описанных выше вызовов, функция модуля также вызывается для открытия редактора модели, при изменении пользователем какой-либо структуры в списке типов переменных (если эта структура используется в моделях, модуль должен скомпилировать их заново), при изменении режима РДС, при сохранении блока с автокомпилируемой моделью, для запроса разрешенности кнопок на вкладке “Компиляция” в окне параметров блока (рис. 131) и т.д. Все возможные режимы ее вызова подробно рассмотрены в приложении А.

Рассмотрев общие принципы работы модулей автоматической компиляции, перейдем к относительно простому примеру, иллюстрирующему работу такого модуля.

§4.2. Инициализация, очистка и настройка параметров модуля

Рассматривается пример модуля автоматической компиляции, функция которого реагирует пока только на самые основные вызовы: инициализацию данных, очистку данных и вызов окна настройки. Созданный модуль регистрируется в РДС.

В качестве примера создадим модуль автоматической компиляции, который даст пользователю возможность ввести текст реакции модели блока на выполнение такта моделирования (`RDS_BFM_MODEL`) в виде фрагмента программы на языке С и задать структуру статических переменных блока. Модуль мы сделаем настраиваемым, чтобы можно было использовать его вместе с одним из стандартных компиляторов, параметры командной строки которых различаются достаточно сильно (по умолчанию будем использовать бесплатный компилятор Borland C++ 5.5).

Функцию модуля компиляции можно вставить в ту же библиотеку, в которой мы размещали модели блоков, рассмотренных в предыдущих примерах, либо создать новую, начав ее исходный текст с такой же главной функции (см. стр. 34). Нам не нужно явно включать файл “RdsComp.h” – директива его включения уже содержится в файле “RdsFunc.h”, который мы используем.

Сначала опишем класс личной области данных нашего модуля автокомпиляции – в нем будут содержаться все параметры, необходимые для формирования текста DLL с моделью блока и вызова компилятора:

```
// Класс личной области данных модуля автокомпиляции
class TCAutoCompData
{ private:
    // Пути
    char *CompPath;         // к компилятору
    char *LinkPath;         // к редактору связей (link)
    char *IncludePath;      // к файлам заголовков
    char *LibPath;          // к библиотекам
    // Параметры командной строки
    char *CompParams;       // компилятора
    char *LinkParams;       // редактора связей
    // Параметры исходного текста
    char *DllMainName;      // имя главной функции DLL
    char *ModelFuncHdr;     // заголовок функции модели
    char *Exported;         // экспортированное имя функции
```

```
    // Освободить все динамические строки
    void FreeAllStrings(void)
      { rdsFree(CompPath);
        rdsFree(LinkPath);
        rdsFree(IncludePath);
        rdsFree(LibPath);
        rdsFree(CompParams);
        rdsFree(LinkParams);
        rdsFree(DllMainName);
        rdsFree(ModelFuncHdr);
        rdsFree(Exported);
      };

  public:
    // Чтение параметров модуля из INI-файла
    void ReadFromIni(void);
    // Запись параметров модуля в INI-файл
    void WriteToIni(void);
    // Настройка параметров модуля
    void Setup(void);

    // Конструктор класса
    TCAutoCompData(void)
      { CompPath=LinkPath=IncludePath=LibPath=NULL;
        CompParams=rdsDynStrCopy(
          "-I\"$INCLUDE$;$INCLUDE$\\sys\" -I\"$RDSINCLUDE$\" "
          "-O2 -Vx -Ve -X- -a8 -k- -vi "
          "-tWD -tWM -c -w-inl "
          "\"$CPPFILE$\"");
        LinkParams=rdsDynStrCopy(
          "-L\"$LIB$\" -D\"\" -aa -Tpd -x -Gn -Gi -q "
          "c0d32.obj \"$OBJFILE$\" , \"$DLLFILE$\" , , "
          "import32.lib cw32mt.lib");
        DllMainName=rdsDynStrCopy("DllEntryPoint");
        ModelFuncHdr=rdsDynStrCopy(
          "extern \"C\" __declspec(dllexport) "
          "int RDSCALL autocompModelFunc");
        Exported=rdsDynStrCopy("autocompModelFunc");
      };
    // Деструктор класса
    ~TCAutoCompData(){ FreeAllStrings(); };
};
//=========================================
```

Все параметры нашего модуля описаны как указатели на строки в секции `private` класса `TCAutoCompData`. Все эти строки будут динамическими, то есть память под них мы будем отводить различными сервисными функциями РДС, а освобождать ее – функцией `rdsFree` (см. стр. 58). В той же секции класса мы описали функцию `FreeAllStrings` для освобождения памяти, занятой всеми этими строками – из-за ее простоты текст этой функции мы разместили непосредственно внутри описания класса. В секции `public` мы описали функции для загрузки, сохранения и настройки параметров нашего модуля, а также конструктор и деструктор класса. В конструкторе мы присваиваем всем параметрам значения, подходящие для работы с компилятором Borland C++ 5.5, а в деструкторе освобождаем все строки параметров вызовом `FreeAllStrings`. Параметры нашего модуля следует рассмотреть подробнее, поскольку они играют ключевую роль в его взаимодействии с компилятором.

В строках CompPath и LinkPath будут находиться пути к исполняемым файлам компилятора и редактора связей соответственно. Компилятор будет преобразовывать файл исходного текста библиотеки с моделью блока, который сформирует наш модуль, в объектный файл (обычно с расширением ".obj"), и для этого ему потребуется путь к папке с файлами заголовков (".h"), который мы будем хранить в строке IncludePath. Редактор связей нужно запускать только после завершения работы компилятора, он должен будет преобразовать объектный файл в исполняемую библиотеку, и для этого ему потребуется путь к папке с библиотеками стандартных функций, который будет находиться в строке LibPath. Мы не можем заранее предсказать, в какой папке будет размещаться компилятор на машине пользователя, поэтому всем четырем строкам путей мы в конструкторе класса присваиваем значение NULL – пока пользователь не настроит эти пути, мы не сможем компилировать модели блоков.

И компилятору, и редактору связей в командной строке нужно передавать различные параметры, которые будут влиять на компиляцию: в частности, нужно передать пути к файлам заголовков и библиотек, указать, что компилируется именно библиотека (DLL), а не файл приложения (EXE) и т.д. Набор этих параметров зависит от конкретного используемого компилятора, у разных компиляторов они существенно отличаются, и мы должны дать пользователю возможность настроить их под свой компилятор. Параметры командной строки компилятора мы будем хранить в строке CompParams, а параметры редактора связей – в LinkParams. По умолчанию мы будем использовать компилятор Borland C++ 5.5, поэтому в конструкторе класса мы записываем в эти строки набор параметров, подходящих именно этому компилятору. Поскольку все строки параметров у нас динамические, мы присваиваем им динамические копии статических строк (символов, записанных в двойных кавычках, согласно синтаксису C), созданные при помощи сервисной функции rdsDynStrCopy.

Можно заметить, что в строках, которые мы присваиваем в конструкторе класса полям CompParams и LinkParams, содержатся символические константы "$INCLUDE$", "$RDSINCLUDE$", "$OBJFILE$" и "$DLLFILE$". Если "$RDSINCLUDE$" – это стандартное символическое имя, используемое во всех функциях РДС для указания пути к папке с входящими в комплект РДС файлам заголовков "RdsDef.h", "RdsFunc.h" и т.д., то четыре других имени в РДС не используются. Тем не менее, нам необходимо ввести их самостоятельно – мы не можем жестко включить в командную строку пути к файлам стандартных заголовков и библиотек компилятора (они настраиваются пользователем и хранятся в полях класса IncludePath и LibPath), а также имя исходного компилируемого файла, объектного файла и файла библиотеки (мы будем размещать их во временной папке, поэтому заранее мы не можем сказать, какие свободные имена удастся дать этим файлам). Перед запуском компилятора и редактора связей нам придется самостоятельно заменять эти символические имена на реальные. К счастью, в РДС есть сервисная функция, позволяющая достаточно легко произвести такую замену.

Последняя группа параметров нашего модуля обеспечивает формирование исходного текста компилируемой библиотеки так, чтобы и компилятор, и РДС могли с ней работать. В строке DllMainName будет находиться имя главной функции DLL, подходящей используемому компилятору (для Borland C++ 5.5 это "DllEntryPoint"). В строке ModelFuncHdr будет находиться заголовок функции модели блока, то есть имя этой функции вместе со всеми предшествующими ему описаниями, обеспечивающими экспорт этой функции из библиотеки (иначе РДС не сможет получить указатель на нее функцией Windows GetProcAddress). Наконец, в строке Exported будет находиться экспортированное имя функции, которое даст ей компилятор, и которое наш модуль автокомпиляции должен будет записать в параметры блока, чтобы скомпилированная функция модели к нему подключилась. В Borland C++ 5.5 экспортированное имя совпадает с

именем самой функции (то есть если в исходном тексте мы назвали функцию `autocompModelFunc`, то и ее экспортированное имя будет “autocompModelFunc”), но другие компиляторы часто добавляют к этому имени специальные символы, указывающие на тип передаваемых в функцию параметров. Например, функция `autocompModelFunc` может получить экспортированное имя “autocompModelFunc@12” или “_autocompModelFunc@12” (такой способ формирования экспортированного имени называется “name mangling” и обычно расшифровывается в описании к конкретному компилятору).

Теперь, когда мы разобрались с тем, какие параметры нужны для работы нашего модуля, необходимо написать функции для их сохранения, загрузки и настройки. Хранить параметры модуля мы будем в файле “ProgGuideAutoComp.ini” в папке “Dll\” внутри стандартной папки настроек РДС. На самом деле, было бы лучше сформировать имя INI-файла из имени нашей DLL заменой расширения на “.ini”, но мы не будем здесь этого делать, чтобы не усложнять пример (имя файла DLL можно узнать при помощи функции Windows API `GetModuleFileName`, в которую нужно передать дескриптор модуля загруженной DLL, то есть первый параметр главной функции DLL `DllEntryPoint`). Для чтения и записи параметров мы будем использовать вспомогательный объект РДС для работы с данными в формате INI-файлов, который создается функцией `rdsINICreateTextHolder`. Мы уже использовали его раньше для сохранения данных блока в текстовом виде (см. §2.8.5).

Поскольку имя файла параметров модуля нам потребуется и при загрузке, и при сохранении, имеет смысл сделать для него макроопределение:

```
// Имя INI-файла с параметрами модуля
#define TCAUTOCOMP_INI "$INI$\\Dll\\ProgGuideAutoComp.ini"
```

Объект для работы с INI-файлами, который мы будем использовать, способен сам преобразовывать символические константы в пути, поэтому в этом макросе для указания папки стандартных INI-файлов мы можем использовать строку “INI”.

Функция сохранения параметров модуля выглядит следующим образом:

```
// Запись параметров модуля в INI-файл
void TCAutoCompData::WriteToIni(void)
{ RDS_HOBJECT ini=rdsINICreateTextHolder(TRUE);

  // Создаем секцию [Paths]
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"Paths");
  // Записываем в нее пути
  rdsINIWriteString(ini,"Compiler",CompPath);
  rdsINIWriteString(ini,"Linker",LinkPath);
  rdsINIWriteString(ini,"Include",IncludePath);
  rdsINIWriteString(ini,"Lib",LibPath);

  // Создаем секцию [Params]
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"Params");
  // Записываем в нее параметры командной строки
  rdsINIWriteString(ini,"Compiler",CompParams);
  rdsINIWriteString(ini,"Linker",LinkParams);

  // Создаем секцию [Func]
  rdsSetObjectStr(ini,RDS_HINI_CREATESECTION,0,"Func");
  // Записываем в нее остальные параметры
  rdsINIWriteString(ini,"DllMain",DllMainName);
  rdsINIWriteString(ini,"ModelFunc",ModelFuncHdr);
  rdsINIWriteString(ini,"Exported",Exported);
```

```
  // Сохраняем получившееся в файл
  rdsSetObjectStr(ini,RDS_HINI_SAVEFILE,0,TCAUTOCOMP_INI);

  // Уничтожение вспомогательного объекта
  rdsDeleteObject(ini);
}
//=======================================
```

Единственное отличие этой функции от функций сохранения данных блоков, в которых мы использовали этот же вспомогательный объект, заключается в том, что здесь мы не передаем сформированный в объекте текст в РДС для сохранения в составе файла схемы, а записываем его в файл, вызывая функцию `rdsSetObjectStr` с параметром `RDS_HINI_SAVEFILE`.

Функция загрузки параметров из файла будет несколько сложнее функции сохранения, поскольку при загрузке новых значений параметров модуля нам необходимо освободить динамическую память, которую занимают старые строки параметров.

```
// Чтение из INI-файла
void TCAutoCompData::ReadFromIni(void)
{ RDS_HOBJECT ini=rdsINICreateTextHolder(TRUE);

  // Загружаем текст из файла во вспомогательный объект
  rdsSetObjectStr(ini,RDS_HINI_LOADFILE,0,TCAUTOCOMP_INI);
  // Проверяем, нет ли ошибок
  if(rdsCommandObject(ini,RDS_HINI_GETLASTERROR)) // Ошибка
    return;

  // Читаем строки из секции [Paths]
  if(rdsINIOpenSection(ini,"Paths"))
    { char *CompPath_old=CompPath,    // Старые значения
           *LinkPath_old=LinkPath,
           *IncludePath_old=IncludePath,
           *LibPath_old=LibPath;
      // Загружаем новые строки
      CompPath=rdsDynStrCopy(
           rdsINIReadString(ini,"Compiler",CompPath_old,NULL));
      LinkPath=rdsDynStrCopy(
           rdsINIReadString(ini,"Linker",LinkPath_old,NULL));
      IncludePath=rdsDynStrCopy(
           rdsINIReadString(ini,"Include",IncludePath_old,NULL));
      LibPath=rdsDynStrCopy(
           rdsINIReadString(ini,"Lib",LibPath_old,NULL));
      // Освобождаем старые строки
      rdsFree(CompPath_old);
      rdsFree(LinkPath_old);
      rdsFree(IncludePath_old);
      rdsFree(LibPath_old);
    }

  // Читаем строки из секции [Params]
  if(rdsINIOpenSection(ini,"Params"))
    { char *CompParams_old=CompParams,     // Старые значения
           *LinkParams_old=LinkParams;
      // Загружаем новые строки
      CompParams=rdsDynStrCopy(
           rdsINIReadString(ini,"Compiler",CompParams_old,NULL));
      LinkParams=rdsDynStrCopy(
           rdsINIReadString(ini,"Linker",LinkParams_old,NULL));
      // Освобождаем старые строки
      rdsFree(CompParams_old);
```

```
      rdsFree(LinkParams_old);
    }

  // Читаем строки из секции [Func]
  if(rdsINIOpenSection(ini,"Func"))
    { char *DllMainName_old=DllMainName,   // Старые значения
           *ModelFuncHdr_old=ModelFuncHdr,
           *Exported_old=Exported;
      // Загружаем новые строки
      DllMainName=rdsDynStrCopy(
           rdsINIReadString(ini,"DllMain",DllMainName_old,NULL));
      ModelFuncHdr=rdsDynStrCopy(
           rdsINIReadString(ini,"ModelFunc",
                ModelFuncHdr_old,NULL));
      Exported=rdsDynStrCopy(
           rdsINIReadString(ini,"Exported",Exported_old,NULL));
      // Освобождаем старые строки
      rdsFree(DllMainName_old);
      rdsFree(ModelFuncHdr_old);
      rdsFree(Exported_old);
    }
  // Вспомогательный объект больше не нужен
  rdsDeleteObject(ini);
}
//=======================================
```

В этой функции мы сначала загружаем текст из файла в созданный вспомогательный объект вызовом `rdsSetObjectStr` с параметром `RDS_HINI_LOADFILE`, а затем вызываем функцию `rdsCommandObject` с параметром `RDS_HINI_GETLASTERROR`, которая вернет ненулевое значение, если последняя операция с INI-файлом (то есть загрузка текста из файла) не удалась. В этом случае мы завершаем нашу функцию загрузки: файл параметров не существует или к нему нет доступа по каким-либо другим причинам, и загрузить параметры из него мы не можем.

Если же текст успешно считан во вспомогательный объект, мы начинаем читать из него параметры, установив название интересующей нас секции вызовом `rdsINIOpenSection` и получая оттуда интересующие нас строки. При этом для чтения каждой строки параметров выполняются следующие действия:

```
// Место для хранения старой строки
char *строка_old=строка;
// Создание динамической копии полученной от объекта строки
строка=rdsDynStrCopy(rdsINIReadString(ini,"имя",строка_old,...));
// Уничтожение старой строки
rdsFree(строка_old);
```

Мы не можем сначала уничтожить старую динамическую строку со значением параметра, а потом создать новую, получив ее из данных файла. Старое значение параметра необходимо нам в качестве значения по умолчанию, то есть того значения, которое вернет функция `rdsINIReadString`, если запрошенной строки в файле параметров не окажется. Поэтому для каждой строки параметров мы сохраняем старое значение во вспомогательной переменной, затем получаем указатель на строку с нужным нам названием, вызывая `rdsINIReadString` (при этом сохраненное значение передается в ее третьем параметре как значение по умолчанию), делаем динамическую копию этой строки функцией `rdsDynStrCopy` и присваиваем указатель на него полю класса, которое соответствует данному параметру. Только после этого мы освобождаем память, занятую старым значением, при помощи функции `rdsFree`.

Теперь нужно написать функцию, которая будет открывать окно настройки модуля. Технически она ничем не будет отличаться от функций настройки блоков, которых мы рассмотрели уже немало:

```
// Настройка модуля автокомпиляции
void TCAutoCompData::Setup(void)
{ RDS_HOBJECT window; // Объект-окно
  char exefilter[]=   // Фильтр для диалога выбора файла
     "Исполняемые файлы (*.exe)|*.exe\nВсе файлы|*.*";

  // Создание окна
  window=rdsFORMCreate(TRUE,-1,-1,"Параметры модуля");

  // Создание вкладки
  rdsFORMAddTab(window,0,"Пути");

  // Поле ввода выбора EXE-файла компилятора
  rdsFORMAddEdit(window,0,1,RDS_FORMCTRL_OPENDIALOG,
                 "Компилятор:",300);
  rdsSetObjectStr(window,1,RDS_FORMVAL_LIST,exefilter);
  rdsSetObjectStr(window,1,RDS_FORMVAL_VALUE,CompPath);

  // Поле ввода выбора EXE-файла редактора связей
  rdsFORMAddEdit(window,0,2,RDS_FORMCTRL_OPENDIALOG,
                 "Редактор связей:",300);
  rdsSetObjectStr(window,2,RDS_FORMVAL_LIST,exefilter);
  rdsSetObjectStr(window,2,RDS_FORMVAL_VALUE,LinkPath);

  // Выбор папки файлов заголовков
  rdsFORMAddEdit(window,0,3,RDS_FORMCTRL_DIRDIALOG,
                 "Папка заголовков:",300);
  rdsSetObjectStr(window,3,RDS_FORMVAL_VALUE,IncludePath);

  // Выбор папки файлов библиотек
  rdsFORMAddEdit(window,0,4,RDS_FORMCTRL_DIRDIALOG,
                 "Папка библиотек:",300);
  rdsSetObjectStr(window,4,RDS_FORMVAL_VALUE,LibPath);

  // Создание вкладки
  rdsFORMAddTab(window,1,"Параметры");

  // Параметры командной строки компилятора
  rdsFORMAddEdit(window,1,5,RDS_FORMCTRL_MULTILINE,
                 "Параметры компилятора:",300);
  rdsSetObjectInt(window,5,RDS_FORMVAL_MLHEIGHT,3*24);// Высота
  rdsSetObjectStr(window,5,RDS_FORMVAL_VALUE,CompParams);

  // Параметры командной строки редактора связей
  rdsFORMAddEdit(window,1,6,RDS_FORMCTRL_MULTILINE,
                 "Параметры редактора связей:",300);
  rdsSetObjectInt(window,6,RDS_FORMVAL_MLHEIGHT,3*24);// Высота
  rdsSetObjectStr(window,6,RDS_FORMVAL_VALUE,LinkParams);

  // Создание вкладки
  rdsFORMAddTab(window,2,"Описания");
```

```
    // Имя главной функции
    rdsFORMAddEdit(window,2,7,RDS_FORMCTRL_EDIT,
                   "Имя главной функции DLL:",300);
    rdsSetObjectStr(window,7,RDS_FORMVAL_VALUE,DllMainName);

    // Заголовок модели
    rdsFORMAddEdit(window,2,8,RDS_FORMCTRL_MULTILINE,
                   "Заголовок функции модели:",300);
    rdsSetObjectInt(window,8,RDS_FORMVAL_MLHEIGHT,2*24);// Высота
    rdsSetObjectStr(window,8,RDS_FORMVAL_VALUE,ModelFuncHdr);

    // Экспортированное имя
    rdsFORMAddEdit(window,2,9,RDS_FORMCTRL_EDIT,
                   "Экспортированное имя:",300);
    rdsSetObjectStr(window,9,RDS_FORMVAL_VALUE,Exported);

    // Открытие окна
    if(rdsFORMShowModalEx(window,NULL)) // Нажата OK
      { // Освобождаем старые строки параметров
        FreeAllStrings();

        // Делаем динамические копии всех строк из полей ввода
        // и присваиваем их полям класса
        CompPath=rdsDynStrCopy(
                   rdsGetObjectStr(window,1,RDS_FORMVAL_VALUE));
        LinkPath=rdsDynStrCopy(
                   rdsGetObjectStr(window,2,RDS_FORMVAL_VALUE));
        IncludePath=rdsDynStrCopy(
                   rdsGetObjectStr(window,3,RDS_FORMVAL_VALUE));
        LibPath=rdsDynStrCopy(
                   rdsGetObjectStr(window,4,RDS_FORMVAL_VALUE));
        CompParams=rdsDynStrCopy(
                   rdsGetObjectStr(window,5,RDS_FORMVAL_VALUE));
        LinkParams=rdsDynStrCopy(
                   rdsGetObjectStr(window,6,RDS_FORMVAL_VALUE));
        DllMainName=rdsDynStrCopy(
                   rdsGetObjectStr(window,7,RDS_FORMVAL_VALUE));
        ModelFuncHdr=rdsDynStrCopy(
                   rdsGetObjectStr(window,8,RDS_FORMVAL_VALUE));
        Exported=rdsDynStrCopy(
                   rdsGetObjectStr(window,9,RDS_FORMVAL_VALUE));
        // Запись изменившихся параметров в INI-файл
        WriteToIni();
      }
    // Уничтожение окна
    rdsDeleteObject(window);
  }
  //=======================================
```

Эта функция при помощи вспомогательного объекта РДС открывает окно с тремя вкладками “Пути”, “Параметры” и “Описания”, на которых расположены поля для ввода описанных выше параметров модуля. Поля на вкладке “Пути” снабжены кнопками для вызова диалогов выбора файлов и папок, чтобы пользователь мог выбрать в них местоположение исполняемых файлов компилятора и редактора связей и папок библиотек и заголовочных файлов, вместо того, чтобы вводить их вручную. Если пользователь закроет это окно нажатием кнопки “OK”, данные из полей ввода окна будут переписаны в класс личной области данных модуля (функция `Setup` является членом этого класса), после чего

параметры модуля будут записаны в INI-файл вызовом только что написанной нами функции WriteToIni.

Наконец, напишем функцию модуля автокомпиляции – именно она будет создавать объект класса TCAutoCompData и вызывать его функции. Назовем функцию нашего модуля “TestCAutoComp”:

```
// Функция модуля автокомпиляции
extern "C" __declspec(dllexport) int RDSCALL TestCAutoComp(
           int CallMode,                    // Событие
           RDS_PCOMPMODULEDATA ModuleData, // Данные модуля
           LPVOID ExtParam)      // Дополнительные параметры
{ // Приведение указателя на личную область данных
  // к правильному типу
  TCAutoCompData *data=(TCAutoCompData*)(ModuleData->ModuleData);

  switch(CallMode)
    { // Инициализация модуля
      case RDS_COMPM_INIT:
        // Создание личной области данных модуля
        ModuleData->ModuleData=data=new TCAutoCompData();
        // Чтение параметров из INI-файла
        data->ReadFromIni();
        break;

      // Очистка данных модуля
      case RDS_COMPM_CLEANUP:
        delete data;  // Удаление личной области модуля
        break;

      // Вызов окна настройки модуля
      case RDS_COMPM_SETUP:
        data->Setup();
        break;
    }
  return RDS_COMPR_DONE;
}
//=======================================
```

Можно заметить, что эта функция построена точно так же, как все рассмотренные нами ранее функции моделей блоков: внутри нее находится оператор switch, который, в зависимости от значение параметра CallMode, выполняет то или иное действие. В данном случае функция реагирует всего на три события. При инициализации модуля (RDS_COMPM_INIT) мы создаем объект класса TCAutoCompData и записываем указатель на него в поле ModuleData структуры данных модуля, переданной во втором параметре нашей функции. Этот объект будет служить модулю в качестве личной области данных. Сразу после создания мы вызываем для этого объекта функцию ReadFromIni, чтобы параметры модуля загрузились из INI-файла.

При очистке данных модуля (RDS_COMPM_CLEANUP), то есть перед выгрузкой его из памяти, мы уничтожаем ранее созданный объект. В самом начале функции мы уже привели значение поля ModuleData->ModuleData, в котором хранится указатель на объект, к типу TCAutoCompData* и присвоили его переменной data, так что во всех реакциях, кроме RDS_COMPM_INIT (на этот момент объект еще не создан), мы можем пользоваться ее значением.

Наконец, при запросе пользователем окна настройки модуля (RDS_COMPM_SETUP) мы просто вызываем написанную нами функцию Setup, которая откроет окно и, при необходимости, запишет измененные параметры в INI-файл.

Независимо от события, из-за которого вызвана наша функция, она возвращает константу `RDS_COMPR_DONE`, сигнализирующую РДС об отсутствии ошибок. В дальнейшем, когда мы добавим к нашей функции новые реакции, мы изменим это, но пока никакой обработки ошибок нам не требуется.

Скомпилировав библиотеку с нашим новым модулем автокомпиляции, мы должны зарегистрировать это модуль в РДС, чтобы пользователь смог его настраивать и подключать к блокам. Для этого следует открыть окно со списком модулей, вызвав пункт главного меню РДС “Сервис | Автокомпиляция...” и добавить в список новый модуль кнопкой “+”, после чего ввести в нижней части окна путь к файлу библиотеки с нашим модулем, экспортированное имя функции модуля и название, которое будет отображаться в списке зарегистрированных модулей, когда пользователь будет с ним работать (рис. 133).

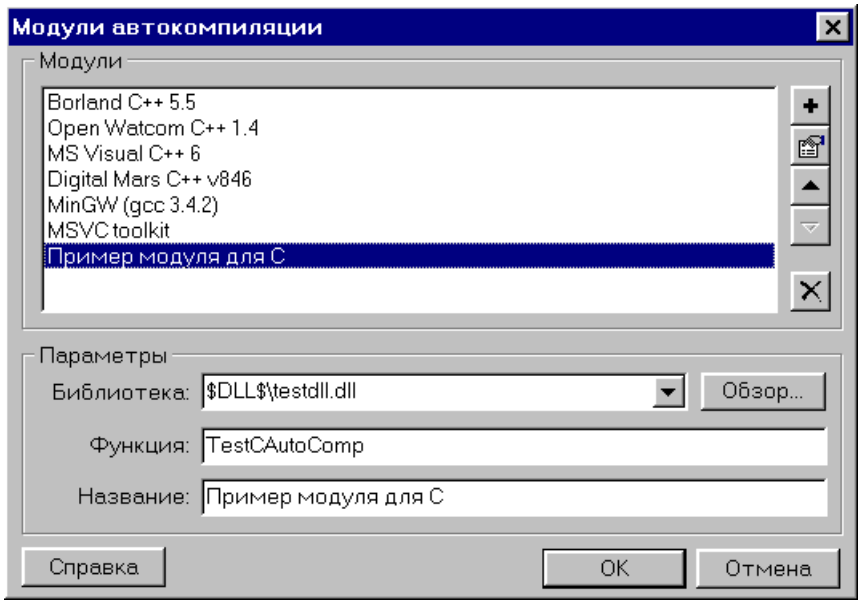


Рис. 133. Регистрация созданного модуля в РДС

Указывая имя функции модуля следует помнить о возможном искажении имен функций при экспорте – в поле “функция” окна списка модулей нужно вводить имя функции в том виде, в котором оно доступно для поиска функцией Windows API `GetProcAddress`, то есть со всеми возможными искажениями. Если в описании компилятора не удается найти информацию об этих искажениях, можно воспользоваться какой-либо программой, показывающей все экспортированные имена функций в заданном файле (например, “tdump” из комплекта Borland C++) – в списке имен можно будет легко найти свою функцию и увидеть, как изменилось ее имя. Название модуля можно сделать любым, эта строка никак не влияет на работу самого модуля. После того, как все параметры модуля введены, следует подтвердить регистрацию нового модуля нажатием “OK”.

Наш новый модуль пока мало что умеет, но его уже можно настроить на установленный на данной машине компилятор. Для этого в окне списка модулей (рис. 133) следует дважды щелкнуть на его названии или, выбрав его в списке, нажать вторую сверху кнопку рядом с этим списком. При этом модуль загрузится в память РДС (что приведет к вызову его функции с параметром `RDS_COMPM_INIT` и загрузке параметров из INI-файла, если, конечно, он существует), после чего его функция будет вызвана с параметром `RDS_COMPM_SETUP`, в результате чего она, в свою очередь, вызовет функцию `Setup` класса личной области данных модуля, которая откроет окно настройки (рис. 134).

При самом первом вызове окна настроек INI-файл параметров модуля еще не существует, поэтому все параметры будут иметь значения по умолчанию и все пути на вкладке “Пути” будут пустыми. Однако, если указать все эти пути и нажать кнопку “OK”,

функция `Setup` создаст файл и запишет в него введенные значения, после чего они будут загружаться в личную область данных модуля при каждой его инициализации.

Поскольку наш модуль в данный момент не обслуживает ни одного блока схемы (это пока невозможно, мы еще не ввели в его функцию необходимые реакции), закрытие окна настройки приведет к его немедленной выгрузке из памяти.

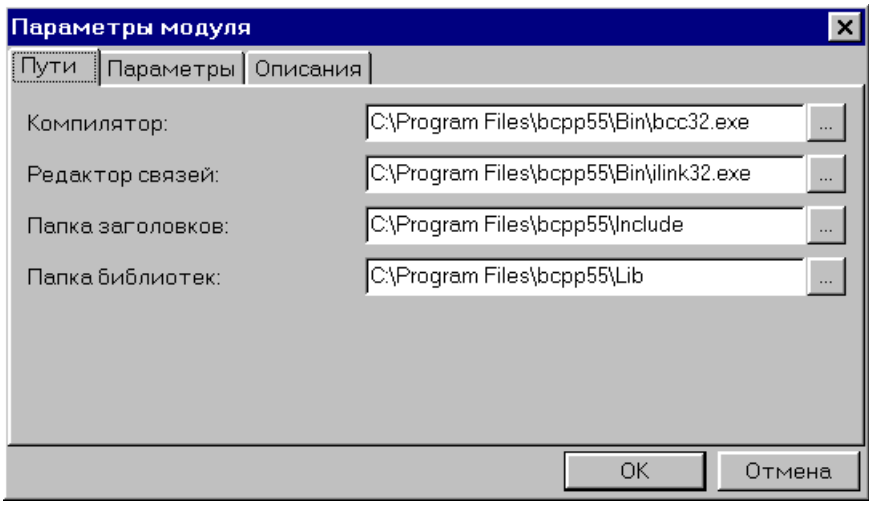


Рис. 134. Окно настройки модуля

§4.3. Подключение моделей к блокам и вызов редактора

В созданный в предыдущем параграфе модуль автоматической компиляции добавляются функции, необходимые для подключения моделей к блокам. Рассматривается простой пример текстового формата модели, в модуль добавляется редактор таких моделей.

Для того, чтобы модель, компилируемую нашим модулем, можно было подключить к блоку, необходимо ввести в функцию модуля по крайней мере две реакции:

- реакцию на запрос поддерживаемых модулем функций (`RDS_COMPM_GETOPTIONS`), в которой необходимо указать, какие из кнопок на вкладке “Компиляция” окна параметров блока (рис. 131) пользователь может нажимать, а также, при необходимости, установить заголовок поля ввода имени модели;
- реакцию на нажатие пользователем одной из упомянутых выше кнопок (`RDS_COMPM_EXECFUNCTION`), с помощью которых пользователь сможет создавать для блока новую модель или подключать к нему уже существующую.

Кроме того, имеет смысл ввести в функцию модуля проверку возможности подключения выбранной модели к конкретному блоку (`RDS_COMPM_CANATTACHBLK`). Например, если пользователь попытается подключить модель, работающую со статическими переменными, к вводу шины, нужно сообщить ему, что эта модель не сможет работать с данным типом блока. Также целесообразно добавить в функцию реакцию на вызов пользователем редактора модели (`RDS_COMPM_OPENEDITOR`) – без этого, в принципе, можно обойтись, но тогда, чтобы изменить модель, пользователю придется самостоятельно запускать какую-то внешнюю программу, что довольно неудобно.

Прежде чем вводить новые реакции в нашу функцию модуля, нужно придумать формат файла модели. В §4.2 мы решили включать в автоматически компилируемую модель структуру переменных блока и реакцию на такт моделирования. Будем хранить их в текстовом файле (с расширением “.txt”), чтобы, при желании, файл модели можно было открыть в обычном текстовом редакторе. Чтобы модуль автокомпиляции мог отличить файл модели от обычного текстового файла, наш файл модели всегда будет начинаться со строки

“$TESTCMODEL”. Если пользователь попытается подключить к блоку текстовый файл без этого слова в первой строке, функция модуля выведет ему сообщение об ошибке. Структуру статических переменных блока мы будем записывать в том же виде, в каком она сохраняется в текстовом формате схем и блоков (текст такого описания легко получить у РДС при помощи сервисных функций), предваряя это описание строкой “$VARS”. После описания переменных будет располагаться строчка со словом “$PROG” и текст на языке С, представляющий собой реакцию нашей модели на один такт моделирования, при этом в этом тексте мы дадим пользователю возможность обращаться к переменным блока по именам. Для простоты примера в моделях, компилируемых нашим модулем, мы ограничимся только простыми переменными (логическими, целыми, вещественными и т.п.) и запретим использовать сложные: структуры, матрицы, строки и переменные произвольного типа.

Таким образом, модель блока, выдающая на вещественный выход `y` сумму вещественных входов `x1` и `x2`, в нашем формате будет выглядеть так:

```
$TESTCMODEL
$VARS
struct
 begin
  signal name "Start" in run default 1
  signal name "Ready" out default 0
  double name "x1" in menu run default 0
  double name "x2" in menu run default 0
  double name "y" out menu default 0
 end
$PROG
  y=x1+x2;
```

Разумеется, в описании переменных должны присутствовать обязательные для каждого простого блока сигналы `Start` и `Ready`.

Добавим в описание класса `TCAutoCompData` новые функции-члены, которые помогут нам в реализации новых реакций модуля:

```
// Класс личной области данных модуля автокомпиляции
class TCAutoCompData
{ private:
    // (.....................)
    // (... без изменений ...)
    // (.....................)

    // Сообщение об ошибке в модели
    static void ModelErrorMsg(char *modelname,char *errortext);
  public:
    // Чтение параметров модуля из INI-файла
    void ReadFromIni(void);
    // Запись параметров модуля в INI-файл
    void WriteToIni(void);
    // Настройка параметров модуля
    void Setup(void);

    // Создать новую пустую модель
    void CreateEmptyModel(void);
    // Выбрать существующий файл модели
    void ConnectExistingModel(char *oldmodel);
    // Проверить возможность подключения модели к блоку
    int CanAttachBlock(RDS_COMPCANATTACHBLKDATA *param);
    // Открыть редактор модели
    void OpenEditor(RDS_OPENEDITORDATA *param);
```

```
    // ........................
    // ... далее без изменений ...
    // ........................
};
//=======================================
```

Кроме того, введем макроопределения для специальных слов “$TESTCMODEL”, “$VARS” и “$PROG”, которые мы решили использовать в текстовых файлах моделей:

```
// Имена секций файла модели
// Начало файла
#define TCTEXTSECTION_START     "$TESTCMODEL"
// Секция переменных
#define TCTEXTSECTION_VARS      "$VARS"
// Секция исходного текста
#define TCTEXTSECTION_PROG      "$PROG"
//=======================================
```

Статическая (то есть не имеющая доступа к данным конкретного объекта класса) функция `ModelErrorMsg` упростит нам вывод сообщений об ошибках – она будет формировать текст сообщения из имени модели и описания ошибки, после чего показывать его пользователю:

```
// Сообщение об ошибке в модели
void TCAutoCompData::ModelErrorMsg(
                char *modelname,char *errortext)
{ char *msgtext; // Здесь формируется динамический текст
  // Название модели
  msgtext=rdsDynStrCat("Модель: ",modelname,FALSE);
  rdsAddToDynStr(&msgtext,"\n",FALSE);
  // Описание ошибки
  rdsAddToDynStr(&msgtext,errortext,FALSE);
  // Показываем сообщение
  rdsMessageBox(msgtext,"Автокомпиляция",MB_OK | MB_ICONWARNING);
  // Освобождаем память, занятую динамическим текстом
  rdsFree(msgtext);
}
//=======================================
```

Работа этой функции основана на вызовах `rdsDynStrCat` и `rdsAddToDynStr`, объединяющих пару строк и уже неоднократно встречавшихся ранее в примерах. Мы не будем разбирать ее подробно.

Тела добавленных в класс функций еще не описаны, но их вызовы мы уже можем добавить в функцию модуля автокомпиляции. Измененная функция будет выглядеть следующим образом:

```
// Функция модуля автокомпиляции
extern "C" __declspec(dllexport) int RDSCALL TestCAutoComp(
           int CallMode,                    // Событие
           RDS_PCOMPMODULEDATA ModuleData,  // Данные модуля
           LPVOID ExtParam)      // Дополнительные параметры
{ // Приведение указателя на личную область данных
  // к правильному типу
  TCAutoCompData *data=(TCAutoCompData*)(ModuleData->ModuleData);
  // Вспомогательная переменная - указатель на структуру
  // функции, вызванной из окна параметров
  RDS_PCOMPEXECFUNCDATA funcdata;

  switch(CallMode)
    { // Инициализация модуля
      case RDS_COMPM_INIT:
```

```
        // Создание личной области данных модуля
        ModuleData->ModuleData=data=new TCAutoCompData();
        // Чтение параметров из INI-файла
        data->ReadFromIni();
        break;

      // Очистка данных модуля
      case RDS_COMPM_CLEANUP:
        delete data;  // Удаление личной области модуля
        break;

      // Вызов окна настройки модуля
      case RDS_COMPM_SETUP:
        data->Setup();
        break;

      // Получение поддерживаемых модулем функций
      case RDS_COMPM_GETOPTIONS:
        // Название поля ввода имени модели
        rdscompReturnModelNameLabel("Файл исходного текста:");
        // Разрешенные кнопки
        return RDS_COMPFLAG_FUNCMODELBROWSE |    // Обзор
               RDS_COMPFLAG_FUNCMODELCREATE;     // Новый

      // Выполнить функцию
      case RDS_COMPM_EXECFUNCTION:
        // Что произошло в окне параметров
        funcdata=(RDS_PCOMPEXECFUNCDATA)ExtParam;
        switch(funcdata->Function)
          { // Нажата кнопка "Новый"
            case RDS_COMPFLAG_FUNCMODELCREATE:
              data->CreateEmptyModel();
              break;
            // Нажата кнопка "Обзор"
            case RDS_COMPFLAG_FUNCMODELBROWSE:
              data->ConnectExistingModel(funcdata->ModelName);
              break;
          }
        break;

      // Проверка возможности присоединения модели к блоку
      case RDS_COMPM_CANATTACHBLK:
        return data->CanAttachBlock(
                     (RDS_PCOMPCANATTACHBLKDATA)ExtParam);

      // Открыть окно редактора
      case RDS_COMPM_OPENEDITOR:
        data->OpenEditor((RDS_POPENEDITORDATA)ExtParam);
        break;
    }
  return RDS_COMPR_DONE;
}
//=======================================
```

В функцию добавлены четыре новых реакции, которые были перечислены в начале этого параграфа. Когда пользователь выберет наш модуль автокомпиляции в выпадающем списке в окне параметров блока (см. рис. 131), функция `TestCAutoComp` будет вызвана с

параметром CallMode, равным RDS_COMPM_GETOPTIONS. Реагируя на этот вызов, мы устанавливаем название поля ввода для имени модели функцией rdscompReturnModelNameLabel и возвращаем целое число, составленное из битовых флагов разрешенных кнопок. В данном случае мы возвращаем всего два флага: RDS_COMPFLAG_FUNCMODELBROWSE (разрешена кнопка "Обзор") и RDS_COMPFLAG_FUNCMODELCREATE (разрешена кнопка "Новый"). Таким образом, на вкладке "Компиляция" окна параметров блока будет запрещена кнопка "Сохранить как" и ввод имени модели вручную: пользователь сможет только либо создать для блока новую пустую модель, либо подключить к нему уже существующую.

Если пользователь нажмет на одну из этих кнопок, функция модуля будет вызвана с CallMode, равным RDS_COMPM_EXECFUNCTION, при этом в параметре ExtParam будет передан указатель на структуру RDS_COMPEXECFUNCDATA, описывающую нажатую кнопку:

```
typedef struct
{ int Function;       // Функция (RDS_COMPFLAG_FUNC*)
  LPSTR ModelName;    // Содержимое строки имени модели
  // Следующие параметры могут быть не определены
  // для некоторых функций
  RDS_HOBJECT BlockVars;   // Переменные блока (только для
                           // RDS_COMPFLAG_FUNCMODELCREATE)
  int BlockType;           // Тип блока
} RDS_COMPEXECFUNCDATA;
typedef RDS_COMPEXECFUNCDATA *RDS_PCOMPEXECFUNCDATA;// Указатель
```

Нас в этой структуре будет интересовать только поле Function, в котором передается идентификатор нажатой пользователем кнопки (значение этого поля совпадает с одним из флагов, возвращенных нами в реакции на RDS_COMPM_GETOPTIONS), и поле ModelName, указывающее на текущее установленное в параметрах блока имя модели.

Если поле Function равно константе RDS_COMPFLAG_FUNCMODELCREATE, значит, пользователь нажал на кнопку "Новый" – в этом случае нам необходимо создать новый файл модели, запросив предварительно у пользователя имя файла для него, и присоединить эту модель к блоку. Этим будет заниматься функция CreateEmptyModel. Если же Function равно RDS_COMPFLAG_FUNCMODELBROWSE, пользователь нажал на кнопку "Обзор", и нам нужно дать ему возможность выбрать модель для блока из находящихся на диске файлов. Этим будет заниматься функция ConnectExistingModel, в нее мы передадим текущее имя модели funcdata->ModelName, чтобы она использовала путь из этого имени в качестве начальной папки для диалога открытия. Все остальные кнопки на вкладке "Компиляция" у нас запрещены, поэтому и реакция на их нажатие нам не потребуется.

Перед присоединением автоматически компилируемой модели к блоку функция модуля будет вызвана с параметром RDS_COMPM_CANATTACHBLK, и в параметре ExtParam будет передан указатель на структуру RDS_COMPCANATTACHBLKDATA:

```
typedef struct
{ LPSTR ModelName;    // Имя модели
  LPSTR ModelNameUC;  // Имя модели в верхнем регистре
  LPSTR AltModelName; // Альтернативное имя модели
  RDS_BHANDLE Block;  // Идентификатор подключаемого блока

  int AttachReason;   // Причина подключения модели (RDS_COMP_AR_*)

  BOOL ChangeModel;   // Команда на подключение к блоку другой
                      // модели
} RDS_COMPCANATTACHBLKDATA;
// Указатель
typedef RDS_COMPCANATTACHBLKDATA *RDS_PCOMPCANATTACHBLKDATA;
```

Этот вызов производится перед тем, как для модели будет создана структура параметров и личная область данных (см. рис. 132), поэтому функция модуля должна принять решение о возможности подключения модели к блоку, зная только идентификатор блока `Block`, имя модели `ModelName` (в поле `ModelNameUC` находится то же самое имя, но преобразованное в верхний регистр) и альтернативное имя модели `AltModelName`, если оно есть. Реагируя на этот вызов, функция должна вернуть константу `RDS_COMPR_DONE`, если данную модель можно подключить к данному блоку, `RDS_COMPR_ERROR`, если подключение невозможно и нужно сообщить об этом пользователю, или `RDS_COMPR_ERRORNOMSG`, если подключение невозможно, но сообщать пользователю об этом не нужно (например, если функция уже вывела сообщение об ошибке). Кроме того, функция может записать в поле `ChangeModel` значение `TRUE` и установить новое имя модели вызовом `rdscompAttachDifferentModel`, если вместо запрошенной нужно подключить к блоку какую-то другую модель. В нашем случае возможность подключения модели к блоку будет проверять функция нашего класса `CanAttachBlock`, которая сама сообщит пользователю об ошибке и вернет нужную константу. Внутри нее мы просто будем проверять тип блока: наши модели могут быть подключены только к простым блокам, поэтому, если блок в поле `Block` структуры `RDS_COMPCANATTACHBLKDATA` будет иметь другой тип, функция выведет сообщение об ошибке.

Наконец, если пользователь скомандует открыть окно редактора подключенной к блоку автокомпилируемой модели, функция модуля будет вызвана с параметром `RDS_COMPM_OPENEDITOR`, и в `ExtParam` будет находиться указатель на структуру `RDS_OPENEDITORDATA`, в полях которой содержится указатель на структуру данных модели, для которой нужно вызвать редактор, и идентификатор блока, с которым в данный момент работает пользователь (в подавляющем большинстве случаев эта информация не нужна):

```
typedef struct
{ RDS_PCOMPMODELDATA Model;      // Данные модели
  RDS_BHANDLE Block;             // Идентификатор блока
} RDS_OPENEDITORDATA;
typedef RDS_OPENEDITORDATA *RDS_POPENEDITORDATA; // Указатель
```

Наша функция модуля просто передает указатель на эту структуру в функцию класса личной области данных модуля `OpenEditor`, которая и будет заниматься окном редактирования модели.

Теперь мы можем написать все те новые функции-члены класса, которые вызываются из функции модуля. Начнем с самой простой из них – функции проверки возможности подключения модели к блоку:

```
// Проверка возможности назначения модели блоку
int TCAutoCompData::CanAttachBlock(
                RDS_COMPCANATTACHBLKDATA *param)
{ RDS_BLOCKDESCRIPTION blockdescr;
  BOOL ok;

  // Получаем описание блока, к которому подключается модель
  blockdescr.servSize=sizeof(blockdescr);
  rdsGetBlockDescription(param->Block,&blockdescr);
  // Блок должен быть простого типа, иначе наша модель со
  // статическими переменными не сможет сним работать
  ok=(blockdescr.BlockType==RDS_BTSIMPLEBLOCK);
  // Если модель подключается вручную, выводим сообщение
  if(param->AttachReason==RDS_COMP_AR_MANUALSET && (!ok))
    { ModelErrorMsg(param->ModelName,
          "Модель может подключаться только к простому блоку");
```

```
      return RDS_COMPR_ERRORNOMSG; // Без сообщения пользователю
    }
  return ok?RDS_COMPR_DONE:RDS_COMPR_ERROR;
}
//=======================================
```

Для того, чтобы узнать тип блока, к которому подключается модель (его идентификатор находится в поле `Block` структуры `RDS_COMPCANATTACHBLKDATA`, указатель на которую передан в параметре функции `param`), мы должны заполнить структуру его описания `blockdescr` функцией `rdsGetBlockDescription`. Если это не простой блок, то есть если поле `BlockType` структуры описания блока не равно `RDS_BTSIMPLEBLOCK`, модель не сможет с ним работать – переменной `ok` будет присвоено значение `FALSE`. В этом случае наша функция должна вернуть РДС одну из двух констант, сигнализирующих об ошибке: `RDS_COMPR_ERROR` или `RDS_COMPR_ERRORNOMSG`. Если модель вручную подключается к блоку пользователем, мы можем сами вывести ему сообщение об ошибке, указав ее конкретную причину. При ручном подключении модели поле `AttachReason` структуры, переданной в параметрах функции, будет содержать константу `RDS_COMP_AR_MANUALSET` – в этом случае мы показываем пользователю сообщение об ошибке при помощи написанной нами ранее функции `ModelErrorMsg` (кроме самого текста сообщения в нее передается имя модели, взятое из поля `param->ModelName`) и возвращаем константу `RDS_COMPR_ERRORNOMSG`, поскольку сообщение уже выведено и РДС уже не нужно этого делать. При подключении модели к блоку в процессе загрузки блока или схемы выводить сообщение пользователю в отдельном окне не следует, поскольку загрузка приостановится, пока он не нажмет “OK” в модальном окне сообщения. Тем не менее, предупредить его об ошибке надо, поэтому в этом случае мы возвращаем константу `RDS_COMPR_ERROR`: РДС добавит эту ошибку в общий список ошибок и покажет их пользователю по окончании загрузки. Это сообщение будет не таким конкретным, как то, которое мы выводим вручную, тем не менее, оно укажет пользователю на блок, в котором возникла ошибка.

Прежде чем заниматься функциями, которые будут работать с текстовым файлом модели, напишем несколько вспомогательных функций, которые облегчат нам работу. Очевидно, нам потребуется функция для записи в файл строки текста:

```
// Записать строку текста в файл
BOOL WriteString(HANDLE file,char *text)
{ DWORD res,size;
  size=strlen(text);
  if(!WriteFile(file,text,size,&res,NULL))
    return FALSE;
  return (res==size);
}
//=======================================
```

В эту функцию передается дескриптор файла, который уже должен быть открыт для записи, и указатель на строку, которую нужно в него записать. Функция самостоятельно определит длину этой строки, запишет ее, и вернет логическое значение, указывающее на успешность записи.

Кроме нее, нам нужна будет функция, которая загрузит в память текстовый файл с заданным именем. Причем нужно написать ее так, чтобы, при желании, можно было загрузить только часть файла, чтобы проверить, начинается ли он с “$TESTCMODEL” (то есть является ли он файлом модели в нашем формате).

```
// Загрузка текстового файла в память
// filename - имя файла, maxread - сколько читать или 0
// для чтения всего файла
char *ReadTextFile(char *filename,DWORD maxread)
{ HANDLE f;
  DWORD size,actread;
```

```
  char *fullpath,*buffer;
  BOOL ok=TRUE;

  // Получаем полный путь к файлу
  fullpath=rdsGetFullFilePath(filename,NULL,NULL);
  if(fullpath==NULL) // Нет пути - ошибка
    return NULL;

  // Открываем файл для чтения
  f=CreateFile(fullpath,GENERIC_READ,0,NULL,OPEN_EXISTING,0,NULL);
  rdsFree(fullpath); // Имя файла больше не нужно
  if(f==INVALID_HANDLE_VALUE) // Ошибка открытия
    return NULL;

  // Определяем размер файла
  size=GetFileSize(f,NULL);
  if(size==0xFFFFFFFF) // Ошибка или слишком большой
    { CloseHandle(f);
      return NULL;
    }

  // Если есть ограничение, читаем только часть файла
  if(maxread!=0 && maxread<size)
    size=maxread;

  // Отводим память для загрузки файла
  buffer=(char*)rdsAllocate(size+1);
  if(buffer==NULL) // Не удалось отвести
    { CloseHandle(f);
      return NULL;
    }

  // Считываем файл в память, если он не пустой
  if(size)
    { if(ReadFile(f,buffer,size,&actread,NULL))
       ok=(actread==size);
      else
       ok=FALSE;
    }
  // Закрываем файл
  CloseHandle(f);
  if(!ok) // Ошибка чтения
    { rdsFree(buffer);
      return NULL;
    }
  // Дописываем после конца считанного текста нулевой байт
  buffer[size]=0;
  return buffer;
}
//=======================================
```

В эту функцию передается имя файла `filename` (возможно, с символическими константами, обозначающими стандартные пути) и максимальный размер считываемого текста `maxread` (чтобы считать весь файл, нужно передать в этом параметре 0). Функция возвращает указатель на динамически отведенную функцией `rdsAllocate` строку, в которой находится текст, загруженный из файла, либо `NULL` при ошибке.

Мы не будем подробно разбирать эту функцию – в основном, она состоит из различных вызовов Windows API для работы с файлами. Следует только обратить внимание

на то, что перед открытием файла переданное в параметре `filename` имя преобразуется в динамическую строку с полным путем к файлу `fullpath`, которая после использования освобождается вызовом `rdsFree`. Таким образом, если в эту функцию будет передано имя файла без пути, она автоматически добавит к нему путь к загруженной в данный момент схеме, а если в имени файла будут присутствовать символические обозначения стандартных путей РДС (“DLL”, “INI” и т.п.), они будут заменены на сами эти пути.

Теперь мы можем написать функцию `CreateEmptyModel`, которая вызывается при нажатии пользователем кнопки “Новый” на вкладке “Компиляция” окна параметров блока (рис. 131):

```
// Создать новый пустой файл модели
void TCAutoCompData::CreateEmptyModel(void)
{ // Текст пустой модели
  char *text=TCTEXTSECTION_START "\r\n"
             TCTEXTSECTION_VARS  "\r\n"
             "struct\r\nbegin\r\n"
             " signal name \"Start\" in run default 1\r\n"
             " signal name \"Ready\" out default 0\r\n"
             "end\r\n"
             TCTEXTSECTION_PROG  "\r\n";
  char *relpath,*fullpath;
  HANDLE file;
  BOOL ok;

  // Вызываем диалог сохранения
  relpath=rdsCallFileDialog("",
          RDS_CFD_SAVE|RDS_CFD_OVERWRITEPROMPT,
          "Текстовые файлы (*.txt)|*.txt\nВсе файлы|*.*",
          "txt",
          "Новая модель");
  if(relpath==NULL) // Пользователь нажал "Отмена"
    return;

  // Преобразуем относительный путь в полный
  fullpath=rdsGetFullFilePath(relpath,NULL,NULL);

  // Открываем файл для записи
  file=CreateFile(fullpath,GENERIC_WRITE,0,NULL,
                  CREATE_ALWAYS,0,NULL);
  rdsFree(fullpath); // Полный путь больше не нужен
  if(file==INVALID_HANDLE_VALUE) // Ошибка открытия
    ok=FALSE;
  else
    { ok=WriteString(file,text);
      CloseHandle(file);
    }
  if(!ok) // Ошибка
    { rdsMessageBox("Невозможно создать файл модели",
                    "Автокомпиляция",MB_OK | MB_ICONERROR);
      return;
    }

  // Пустой файл модели записан - устанавливаем его имя
  // в качестве имени модели блока
  rdscompReturnModelName(relpath);
```

```
    // Освобождаем динамическую строку
    rdsFree(relpath);
  }
  //=======================================
```

В самом начале функции мы объявляем указатель `text` на текст, который мы будем использовать в качестве пустой модели блока. Секция переменных этой модели "$VARS" содержит только два обязательных для простого блока сигнала, а секция исходного текста "$PROG" пуста. Затем мы вызываем сервисную функцию `rdsCallFileDialog`, которая откроет диалог выбора файла (в данном случае – диалог сохранения) и вернет динамическую строку с именем файла, выбранного пользователем:

```
LPSTR RDSCALL rdsCallFileDialog(
      LPSTR initialfile,    // Исходное имя файла
      DWORD flags,          // Флаги
      LPSTR filter,         // Фильтры файлов
      LPSTR defext,         // Расширение по умолчанию
      LPSTR title);         // Название диалога
```

В первом параметре этой функции передается имя файла, который должен быть выбран в диалоге в момент его открытия. В данном случае мы передаем пустую строку – файл пока не выбран. Во втором параметре (`flags`) указываются битовые флаги, определяющие внешний вид и поведение диалога. Мы передаем два флага: `RDS_CFD_SAVE` (нам нужен диалог сохранения) и `RDS_CFD_OVERWRITEPROMPT` (при выборе уже существующего файла предупреждать пользователя о том, что он будет перезаписан). В третьем параметре передается строка шаблонов имен файлов, которые будут показаны в диалоге – мы уже встречались с такими строками в полях ввода для выбора файлов в окнах настройки блоков (см. стр. 384). Здесь нас интересуют текстовые файлы, поэтому в этой строке мы записали два шаблона: текстовые файлы (*.txt) и все файлы (*.*). В двух последних параметрах передаются расширение выбранного файла по умолчанию (оно будет автоматически добавлено к имени файла, если пользователь не укажет расширение) и название диалога, которое будет видеть пользователь.

Если пользователь выйдет из диалога, не выбрав файл, функция `rdsCallFileDialog` вернет значение `NULL`, в противном случае она преобразует имя выбранного пользователем файла в динамическую строку, заменив в ней стандартные пути на символические константы и убрав из нее путь, если он совпадает с путем к загруженной в данный момент схеме, сформировав таким образом относительный путь к выбранному пользователем файлу. Этот относительный путь мы записываем в переменную `relpath` (он нам еще понадобится), после чего вызовом `rdsGetFullFilePath` преобразуем его в полный путь, который записываем в переменную `fullpath`. Может возникнуть вопрос: зачем мы получаем из функции `rdsCallFileDialog` относительный путь, если потом мы все равно преобразуем его в полный? Можно было бы передать в `rdsCallFileDialog` флаг `RDS_CFD_ABSPATH` – в этом случае она сразу вернула бы полный путь. Однако, нам нужны будут оба этих пути: относительный мы используем в качестве имени автокомпилируемой модели, которое будет храниться в параметрах блока (так нам не потребуется корректировать имена моделей при переносе схемы в другую папку или на другую машину, если схема и модели будут находится в одной папке), а абсолютный – для непосредственной работы с файлом.

Далее вызовом Windows API `CreateFile` мы открываем файл, путь к которому находится в переменной `fullpath`, для записи и записываем его дескриптор в переменную `file` (сразу после этого строку `fullpath` мы освобождаем – она больше не нужна). Если при открытии файла не возникло никаких ошибок, мы записываем в него текст пустой модели `text` при помощи написанной нами ранее функции `WriteString`, после чего

закрываем файл функцией Windows API `CloseHandle`. Если в процессе записи возникли ошибки, мы выводим об этом сообщение пользователю.

Теперь, когда новый файл модели записан, нужно установить его имя `relpath` в качестве имени модели данного блока. Для этого используется функция `rdscompReturnModelName` – РДС реагирует на ее вызов только в реакциях на действия пользователя на вкладке “Компиляция” окна параметров блока, то есть только при вызове функции модуля с параметром `RDS_COMPM_EXECFUNCTION`. Наша функция `CreateEmptyModel` вызывается именно из такой реакции, поэтому вызов `rdscompReturnModelName(relpath)` выполнится правильно: текст `relpath` будет записан в поле ввода имени модели в окне параметров блока.

Функция подключения к блоку уже существующей модели `ConnectExistingModel` будет немногим сложнее: в ней нам нужно открыть диалог выбора файла, убедиться, что выбранный файл начинается с текста “$TESTCMODEL”, а затем установить его имя в качестве имени модели блока:

```
// Выбрать файл модели
void TCAutoCompData::ConnectExistingModel(char *oldmodel)
{ char *relpath,*buf;
  BOOL ok=FALSE;
  // Длина текста "$TESTCMODEL"
  int prefixlen=strlen(TCTEXTSECTION_START);

  // Вызываем диалог открытия файла
  relpath=rdsCallFileDialog(oldmodel,
           RDS_CFD_OPEN|RDS_CFD_MUSTEXIST,
           "Текстовые файлы (*.txt)|*.txt\nВсе файлы|*.*",
           "txt",
           "Файл модели");
  if(relpath==NULL) // Пользователь нажал "Отмена"
    return;

  // Читаем начало файла: является ли он нашей моделью?
  buf=ReadTextFile(relpath,prefixlen);
  if(buf==NULL) // Ошибка чтения
    ModelErrorMsg(relpath,"Ошибка чтения файла");
  else if(strcmp(buf,TCTEXTSECTION_START)) // Плохой префикс
    ModelErrorMsg(relpath,"Файл не является моделью блока");
  else
    ok=TRUE;
  rdsFree(buf); // Считанный текст больше не нужен

  if(ok) // Делаем выбранный файл именем модели блока
    rdscompReturnModelName(relpath);
  rdsFree(relpath);
}
//=========================================
```

В этой функции мы тоже вызываем `rdsCallFileDialog` для открытия диалога выбора файла, но здесь мы передаем исходное имя файла `oldmodel` (диалог откроется в папке этого файла) и другие флаги: `RDS_CFD_OPEN` (нам нужен диалог открытия) и `RDS_CFD_MUSTEXIST` (выбранный файл должен существовать). Имя файла `relpath`, возвращенное функцией `rdsCallFileDialog`, мы передаем в написанную нами ранее функцию `ReadTextFile`, которая считает в память начало этого файла (эту функцию мы написали так, чтобы она могла работать с относительными путями к файлам). Размер считываемой части файла мы ограничиваем длиной строки `TCTEXTSECTION_START` (“$TESTCMODEL”). Если данные из файла прочитаны успешно, мы сравниваем их с

“$TESTCMODEL” и при несовпадении сообщаем пользователю, что указанный файл не является моделью блока в нашем формате. Если же начало файла совпало со строкой “$TESTCMODEL”, мы делаем имя этого файла именем модели блока вызовом `rdscompReturnModelName`.

Теперь модели, обслуживаемые нашим модулем, могут присоединяться к блокам. В нашем примере нам не нужна личная область данных модели, поэтому функция модуля не реагирует на вызовы `RDS_COMPM_MODELINIT` и `RDS_COMPM_MODELCLEANUP`. Если бы нам нужно было хранить в памяти какие-то данные для каждой модели, в реакции на `RDS_COMPM_MODELINIT` мы отводили бы под них память, а в реакции на `RDS_COMPM_MODELCLEANUP` – освобождали бы ее.

Осталось добавить в наш модуль редактор моделей, то есть написать функцию `OpenEditor`. Но прежде мы напишем еще две вспомогательных функции, которые понадобятся нам для разбора текстового файла модели. Первая из них будет искать в тексте заданное ключевое слово, которое должно обязательно находиться в начале строки (мы будем использовать ее для поиска в тексте файла модели слов “$VARS” и “$PROG”) и возвращать указатель на него:

```
// Найти ключевое слово в начале строки текста
char *FindKeywordAtLineStart(char *text,char *word)
{ char *s=text; // Устанавливаем s на начало текста
  if(text==NULL) return NULL;
  // Ищем в цикле
  for(;;)
    { s=strstr(s,word); // Ищем word начиная с s
      if(s==NULL) // Не найдено
        return NULL;
      if(s==text) // Найдено в начале текста - годится
        return s;
      // Найдено в середине текста - перед s должен находиться
      // перевод строки
      if(s[-1]=='\r' || s[-1]=='\n') // Есть перевод строки
        return s;
      // Перед s - другой символ. Продолжаем поиск
    }
}
//========================================
```

Работа этой функции достаточно проста и основана на стандартной библиотечной функции `strstr`, которая ищет заданную подстроку в строке. Мы не будем подробно на ней останавливаться.

Вторая необходимая нам вспомогательная функция будет разбивать загруженный в память текст файла модели на описание переменных (текст после строки “$VARS”) и исходный текст реакции на такт моделирования (текст после строки “$PROG”):

```
// Разбить текст модели на описание переменных и текст программы
BOOL ProcessModelText(
          char *text,     // текст модели в памяти
          char **pVars,   // возвращаемое начало переменных
          char **pProg)   // возвращаемое начало программы
{ char *s,*s1;

  if(text==NULL || pVars==NULL || pProg==NULL)
    return FALSE;

  // Ищем "$VARS" в начале строки
  s=FindKeywordAtLineStart(text,TCTEXTSECTION_VARS);
```

```
  if(s==NULL) // Нет такой строки
    return FALSE;
  // Найдено - записываем в pVars начало текста после этого
  // слова с пропуском всех пустых строк
  s+=strlen(TCTEXTSECTION_VARS); // Пропускаем название секции
  s+=strspn(s,"\r\n"); // Пропускаем пустые строки после названия
  *pVars=s;

  // Ищем "$PROG" в начале строки оставшегося текста
  s1=FindKeywordAtLineStart(s,TCTEXTSECTION_PROG);
  if(s1==NULL) // Нет такой строки
    return FALSE;
  // Записываем в pProg начало текста после этого слова
  // с пропуском всех пустых строк
  s=s1+strlen(TCTEXTSECTION_PROG); // Пропускаем название
  s+=strspn(s,"\r\n"); // Пропускаем пустые строки после названия
  *pProg=s;
  // Для завершения предыдущей секции записываем нулевой байт
  // вместо '$' в "$PROG"
  *s1=0;
  return TRUE;
}
//=========================================
```

Эта функция возвращает `TRUE`, если в переданном ей тексте удалось найти оба описания, и `FALSE`, если хотя бы одно из них отсутствует. Она использует другую нашу вспомогательную функцию – `FindKeywordAtLineStart`. Сначала мы ищем в переданном в функцию тексте `text` ключевое слово `TCTEXTSECTION_VARS` (“$VARS”). Если оно отсутствует, функция возвращает `FALSE`: текст модели имеет неправильный формат. В противном случае в переменную `s` записывается указатель на начало текста после слова “$VARS” и всех пустых строк, которые могут за ним следовать. Для пропуска самого ключевого слова мы просто добавляем к `s` длину этого слова, а для пропуска всех переводов строк и возвратов каретки после него используем библиотечную функцию `strspn`. После этого найденное начало описания переменных мы записываем в переменную, на которую указывает параметр функции `pVars`.

Затем мы точно так же ищем в оставшейся части текста ключевое слово `TCTEXTSECTION_PROG` (“$PROG”) и записываем в переменную, на которую указывает параметр функции `pProg`, указатель на начало текста после этого ключевого слова и всех пустых строк, которые могут за ним следовать. Кроме того, знак “$” в слове “$PROG” мы заменяем на нулевой байт, чтобы описание переменных, расположенное перед этим словом, завершалось нулем, и с ним можно было работать как с отдельной строкой.

Окно редактора модели мы будем открывать с помощью того же самого вспомогательного объекта РДС, который мы использовали в окнах настройки блоков и для настройки модуля автокомпиляции. Редактор модели должен позволять вводить структуру переменных блока и текст реакции на такт моделирования, поэтому мы сделаем в нем кнопку для вызова редактора переменных РДС (мы уже делали так в §2.16.1 на стр. 483) и многострочное поле ввода для текста реакции. Для реакции на нажатие кнопки нам потребуется функция обратного вызова объекта-окна (см. §2.7.3), поэтому кроме самой функции `OpenEditor` нам придется написать еще и эту функцию обратного вызова. Пока напишем только ее прототип:

```
// Прототип функции обратного вызова окна редактора
void RDSCALL TCAutoCompData_EditorCallback(
          RDS_HOBJECT window,              // Объект-окно
          RDS_PFORMSERVFUNCDATA data);     // Данные
```

Функция OpenEditor, в которую мы передаем указатель на структуру RDS_OPENEDITORDATA, полученный от РДС, будет выглядеть следующим образом:

```
// Открыть редактор модели
void TCAutoCompData::OpenEditor(RDS_OPENEDITORDATA *param)
{ char *modelpath,*modeltext;
  char *vars,*prog;
  RDS_HOBJECT win;
  BOOL ok;

  // Полный путь к файлу модели
  modelpath=rdsGetFullFilePath(param->Model->ModelName,NULL,NULL);

  // Читаем весь файл модели в память
  modeltext=ReadTextFile(modelpath,0);
  if(modeltext==NULL) // Ошибка чтения
    { ModelErrorMsg(param->Model->ModelName,
                    "Ошибка чтения файла модели");
      return;
    }

  // Разбиваем загруженный текст на описание переменных и программу
  if(!ProcessModelText(modeltext,&vars,&prog))
    { rdsFree(modeltext);
      ModelErrorMsg(param->Model->ModelName,
                    "В файле нет необходимых секций");
      return;
    }

  // Создаем объект-окно редактора
  win=rdsFORMCreate(FALSE,-1,-1,param->Model->ModelName);

  // Создаем в нем невизуальное поле и записываем в
  // него описание переменных
  rdsFORMAddEdit(win,0,1000,RDS_FORMCTRL_NONVISUAL,NULL,0);
  rdsSetObjectStr(win,1000,RDS_FORMVAL_VALUE,vars);

  // Создаем кнопку для вызова редактора переменных
  rdsFORMAddEdit(win,0,1,RDS_FORMCTRL_BUTTON|RDS_FORMFLAG_LINE,
                 "Переменные:",150);
  rdsSetObjectStr(win,1,RDS_FORMVAL_VALUE,"Изменить...");

  // Создаем многострочное поле ввода для текста программы
  rdsFORMAddEdit(win,0,2,RDS_FORMCTRL_MULTILINE,
                 "Такт моделирования:",600);
  rdsSetObjectStr(win,2,RDS_FORMVAL_VALUE,prog);
  rdsSetObjectInt(win,2,RDS_FORMVAL_MLHEIGHT,5*24);// Высота
  rdsSetObjectInt(win,2,RDS_FORMVAL_MLRETURNS,1); // Enter

  // Загруженный текст модели больше не нужен - мы все
  // переписали в поля окна
  rdsFree(modeltext);

  // Открываем окно (с функцией обратного вызова)
  ok=rdsFORMShowModalServ(win,TCAutoCompData_EditorCallback);
  if(ok)
    { // Нажата кнопка "OK" - записываем текст модели в файл
      HANDLE file;
```

```
      file=CreateFile(modelpath,GENERIC_WRITE,0,NULL,
                      CREATE_ALWAYS,0,NULL);
      if(file==INVALID_HANDLE_VALUE) // Ошибка
        ok=FALSE;
      else // Записываем в файл
        { // Заголовок и секция переменных
          ok=WriteString(file,TCTEXTSECTION_START "\r\n"
                              TCTEXTSECTION_VARS  "\r\n");
          // Описание переменных
          ok=ok && WriteString(file,
                      rdsGetObjectStr(win,1000,RDS_FORMVAL_VALUE));
          // Секция текста программы
          ok=ok && WriteString(file,TCTEXTSECTION_PROG "\r\n");
          // Текст программы
          ok=ok && WriteString(file,
                      rdsGetObjectStr(win,2,RDS_FORMVAL_VALUE));
          CloseHandle(file);
        }
      if(!ok)
        ModelErrorMsg(param->Model->ModelName,"Ошибка записи");
    }

  // Уничтожаем окно
  rdsDeleteObject(win);
  // Освобождаем строку с именем файла модели
  rdsFree(modelpath);
}
//=======================================
```

В параметре `param` этой функции передается указатель на структуру `RDS_OPENEDITORDATA`, в поле `Model` которой содержится, в свою очередь, указатель на структуру данных модели `RDS_COMPMODELDATA` (см. стр. 604). Таким образом, для получения имени модели блока внутри функции `OpenEditor` нам нужно обратиться к `param->Model->ModelName`. Имена моделей у нас представляют собой имена файлов с относительными или символическими путями, поэтому для работы с файлом модели мы преобразуем ее имя в полный путь к файлу сервисной функцией `rdsGetFullFilePath` и записываем его в переменную `modelpath` (эту динамическую строку нам нужно будет потом освободить вызовом `rdsFree`). Затем мы загружаем весь текст файла модели в динамическую строку `modeltext` при помощи написанной нами ранее функции `ReadTextFile`. Функция `ReadTextFile` может работать и с относительными путями, поэтому в ее параметре можно передать как полный путь к файлу модели `modelpath`, так и относительный `param->Model->ModelName`. Полный путь нам обязательно потребуется позже, когда мы будем записывать внесенные пользователем изменения обратно в файл модели: там мы будем пользоваться функциями Windows API.

Если текст модели считан в память без ошибок (значение `modeltext` не равно `NULL`), мы выделяем из нее текст описания переменных `vars` и текст реакции на такт моделирования `prog` вызовом функции `ProcessModelText`. Если эта функция вернет `TRUE`, `vars` и `prog` будут указывать на соответствующие блоки текста, каждый из которых является независимой строкой, то есть завершается нулевым байтом (текст реакции `prog` располагается в конце файла модели, поэтому нулевой байт в его конце обеспечивает функция загрузки текста `ReadTextFile`, а нулевой байт в конце описания переменных `vars` вставляет функция `ProcessModelText`).

Разобрав текст модели на две части, мы создаем объект-окно `win` и добавляем в него три поля: невидимое поле `RDS_FORMCTRL_NONVISUAL` с идентификатором 1000, в которое

мы записываем текст описания переменных vars (оно нужно нам только для того, чтобы иметь доступ к этому тексту из функции TCAutoCompData_EditorCallback, которая будет вызываться при нажатии на кнопку в нашем окне), кнопку RDS_FORMCTRL_BUTTON с идентификатором 1 для вызова редактора переменных, и многострочное поле ввода RDS_FORMCTRL_MULTILINE с идентификатором 2, в которое мы записываем текст реакции модели на такт моделирования prog. В многострочном поле мы разрешаем вставку перевода строки клавишей Enter, вызвав для него функцию rdsSetObjectInt с параметрами RDS_FORMVAL_MLRETURNS и 1 – без этого вызова нажатие клавиши Enter приводило бы к автоматическому нажатию кнопки окна по умолчанию, то есть “OK”. Хотя в этом случае пользователь все равно мог вставить перевод строки в поле, нажав Ctrl+Enter, использование одной клавиши Enter будет для него более привычным. После создания полей ввода мы открываем получившееся модальное окно функцией rdsFORMShowModalServ.

Если пользователь закрыл окно кнопкой “OK” (rdsFORMShowModalServ вернула TRUE), мы должны записать изменения, внесенные пользователем, обратно в файл модели. Для этого мы открываем файл modelpath для записи функцией Windows API CreateFile и записываем туда последовательно заголовки файла и секции переменных (“$TESTCMODEL” и “$VARS” на отдельных строках), текст описания переменных из поля 1000 (редактированием этого текста занимается еще не написанная нами функция обратного вызова TCAutoCompData_EditorCallback), заголовок секции исходного текста (“$PROG”) и сам текст реакции на такт моделирования из поля ввода 2. Если при записи возникли ошибки, мы сообщаем об этом пользователю. В конце функции OpenEditor мы уничтожаем все динамически созданные объекты.

Теперь мы должны написать функцию TCAutoCompData_EditorCallback, которая будет вызывать редактор переменных, описание которых находится в невидимом поле нашего окна с идентификатором 1000. Мы уже вызывали редактор переменных таким образом, поэтому эта функция будет очень похожа на функцию TNetSendRcvData_Setup_Check (см. стр. 488), отличаясь от нее только в деталях – даже идентификатор невидимого поля, в котором хранится текст описания переменных, у них совпадает:

```
// Функция обратного вызова окна редактора модели
void RDSCALL TCAutoCompData_EditorCallback(
           RDS_HOBJECT window,RDS_PFORMSERVFUNCDATA data)
{ RDS_HOBJECT dv=NULL;
  RDS_VARDESCRIPTION vdescr;
  char *varstr;

  switch(data->Event)
    { case RDS_FORMSERVEVENT_CLICK: // Нажата кнопка
        // Создаем объект для редактирования переменных
        dv=rdsVSCreateEditor();
        // Заполняем объект описанием переменных (поле 1000)
        if(!rdsVSCreateByDescr(dv,
                rdsGetObjectStr(window,1000,RDS_FORMVAL_VALUE)))
          break;
        // Вызываем для объекта редактор переменных
        if(!rdsVSExecuteEditor(dv,TRUE,RDS_HVAR_FALLPLAIN,
                          0,"Переменные блока"))
          break;
        // Пользователь нажал "OK" – получаем текст описания
        // измененных переменных
        vdescr.servSize=sizeof(vdescr);
        if(!rdsVSGetVarDescription(dv,-1,&vdescr))
          break;
```

```
            varstr=rdsCreateVarDescriptionString(vdescr.Var,
                                                 TRUE,0,NULL);
            if(varstr)
              { // Заносим обратно в поле 1000
                rdsSetObjectStr(win,1000,RDS_FORMVAL_VALUE,vars);
                rdsFree(varstr);
              }
            break;
        }
      // Уничтожаем объект-редактор, если он был создан
      rdsDeleteObject(dv);
    }
    //=======================================
```

В этой функции мы проверяем событие (`data->Event`), из-за которого она вызвана окном редактора модели, и, если это нажатие кнопки (`RDS_FORMSERVEVENT_CLICK`), начинаем подготовку к вызову редактора переменных. Мы не проверяем, какая именно кнопка нажата – в нашем окне кнопка всего одна. Сначала мы создаем вспомогательный объект-редактор `dv` функцией `rdsVSCreateEditor`, после чего загружаем в него текстовое описание структуры переменных, находящееся в невидимом поле 1000 при помощи функции `rdsVSCreateByDescr` (идентификатор объекта-окна, из которого вызвана функция `TCAutoCompData_EditorCallback` и которому принадлежит это поле, передается в параметре `window`). Теперь можно открыть редактор переменных функцией `rdsVSExecuteEditor`. В наших моделях мы разрешаем пользователю использовать только простые переменные, поэтому при вызове этой функции мы указываем флаг `RDS_HVAR_FALLPLAIN` (все флаги и параметры этой функции приведены в приложении А). Если пользователь нажал “OK”, функция `rdsVSExecuteEditor` вернет `TRUE` – в этом случае мы получим идентификатор структуры, находящейся в объекте `dv`, при помощи функции `rdsVSGetVarDescription`, после чего сформируем динамическую строку с ее описанием функцией `rdsCreateVarDescriptionString` и запишем ее обратно в поле окна с идентификатором 1000. Теперь в поле 1000 будет находиться описание структуры переменных блока, отредактированное пользователем.

Наш модуль автокомпиляции еще не умеет делать то, для чего он предназначен – компилировать модели блоков. Тем не менее, некоторые его функции уже можно проверить. Поскольку наш модуль уже зарегистрирован в РДС (мы сделали это в §4.2), мы можем создать для какого-нибудь блока новую модель и ввести в нее переменные и текст реакции на такт моделирования. Для этого следует создать новый блок, в окне его параметров выбрать вкладку “Компиляция” (рис. 135), включить флаг “Функция блока компилируется автоматически” и выбрать в выпадающем списке модулей название, которое мы дали нашему модулю при регистрации.

Поле ввода имени модели и кнопка “Сохранить как” при этом станут запрещенными: при вызове нашего модуля с параметром `RDS_COMPM_GETOPTIONS` мы разрешили пользователю использовать только кнопки “Обзор” и “Новый”. Нажатие на кнопку “Новый” вызовет диалог сохранения файла, и, при вводе в нем какого-либо имени, будет записан пустой файл модели. Если у нас уже есть какой-нибудь файл модели (например, набранный в любом текстовом редакторе текст со стр. 619), можно подключить его к блоку нажатием кнопки “Обзор”. В любом из этих случаев после выбора имени файла оно должно появиться в поле ввода имени модели. Если после этого закрыть окно параметров блока кнопкой “OK”, РДС автоматически откроет созданный нами редактор модели (рис. 136).

В нижней части окна можно вводить текст реакции блока на такт моделирования в синтаксисе языка С, в верхней находится кнопка для вызова редактора переменных. Если нажать на нее, откроется обычное окно редактора переменных блока (рис. 137).

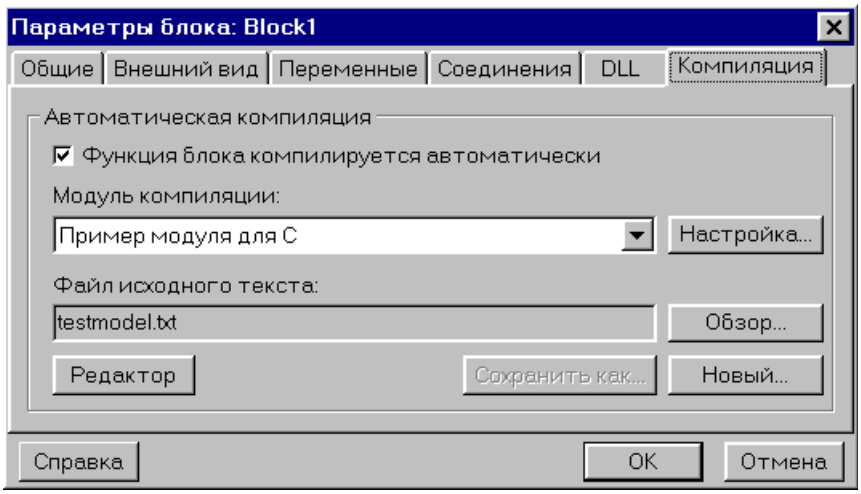


Рис. 135. Подключение модели к блоку через созданный модуль автокомпиляции

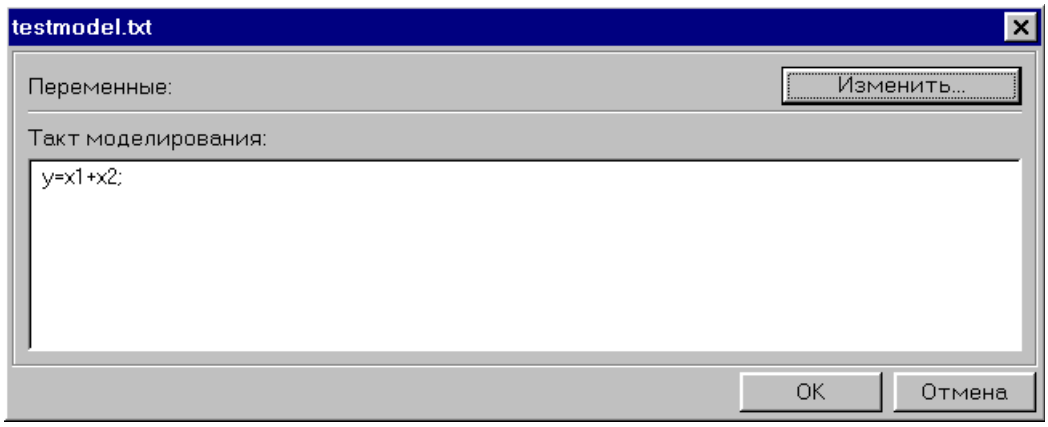


Рис. 136. Редактор модели

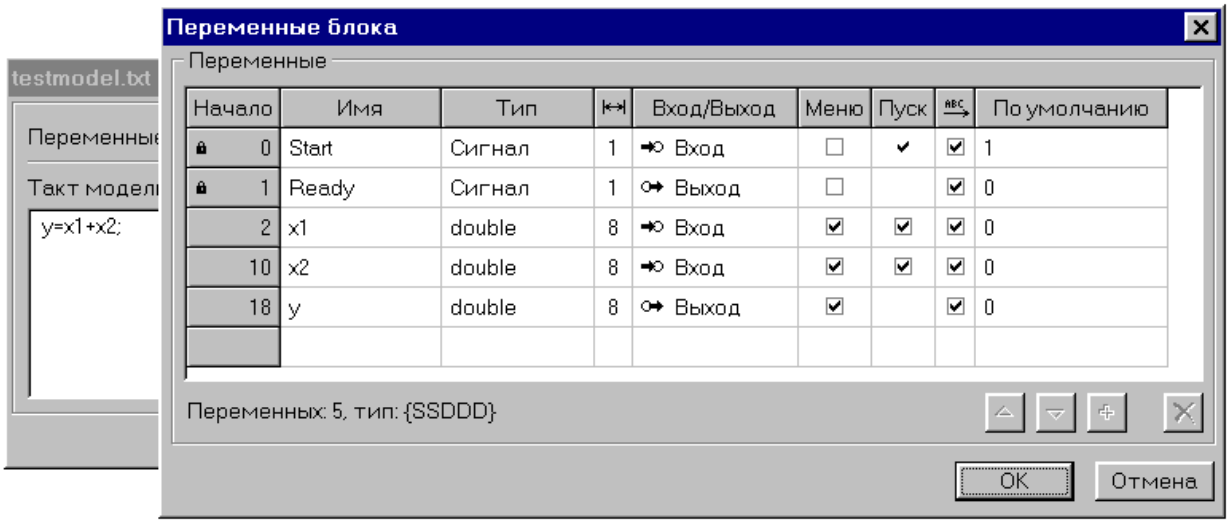


Рис. 137. Редактор переменных, вызванный из редактора модели

При закрытии окна редактора модели все внесенные в структуру переменных и в текст реакции изменения должны записаться обратно в файл – это можно увидеть, открыв этот файл в каком-либо текстовом редакторе.

§4.4. Компиляция моделей

Описывается способ формирования исходного текста модели блока на языке С и вызова для этого текста внешнего компилятора.

Для того, чтобы модуль автокомпиляции мог преобразовывать введенные пользователем модели в исполняемые файлы библиотек (DLL), нам нужно добавить в его функцию реакции на два вызова: `RDS_COMPM_PREPARE` и `RDS_COMPM_COMPILE`. В реакции на `RDS_COMPM_PREPARE` модуль должен проверить, нужно ли компилировать указанную в параметрах вызова модель, и подготовить ее к компиляции, если это необходимо (и, кроме того, записать в параметры блока имя файла библиотеки, которая будет создана в результате компиляции, и имя экспортированной из нее функции блока). В реакции на `RDS_COMPM_COMPILE` модуль должен будет скомпилировать либо все модели, которые он сам пометил как требующие компиляции в реакции на `RDS_COMPM_PREPARE`, либо вообще все свои модели по требованию пользователя, который выбрал пункт главного меню РДС “Система | Перекомпилировать все модели”. В процессе компиляции должны будут быть созданы файлы DLL, имена которых занесены в блоки в реакции на `RDS_COMPM_PREPARE`, и всем блокам, связанным с каждой скомпилированной моделью, должна быть присвоена структура переменных этой модели.

Кроме этих двух реакций нам нужно будет добавить еще и реакцию на присоединение модели к блоку `RDS_COMPM_ATTACHBLOCK`. Дело в том, что, даже если модель не нужно компилировать заново, при присоединении ее к блоку структура переменных этого блока должна быть приведена в соответствие со структурой, описанной в файле модели. Если этого не сделать, скомпилированная модель не сможет работать с этим блоком до тех пор, пока пользователь не изменит файл модели и модулю не придется опять компилировать ее.

Нам нужно решить, где мы будем размещать DLL, полученные в результате компиляции наших моделей, и какие имена мы будем им давать. Поскольку тексты автоматически компилируемых моделей мы храним в отдельных файлах, причем каждую модель – в отдельном файле, имеет смысл помещать файл скомпилированной DLL в одну папку с файлом модели и давать ему имя, отличающееся от имени модели только расширением. Таким образом, из файла “testmodel.txt” получится библиотека “testmodel.dll”, находящаяся в той же папке. Библиотеки имеет смысл размещать в одной папке с моделями еще и потому, что, раз пользователь смог сохранить файл модели в этой папке, запись в эту папку ему разрешена, и мы сможем переписать туда же скомпилированный файл DLL.

Прежде чем мы включим реакцию на два упомянутых выше вызова в наш модуль, напишем несколько вспомогательных функций. Для того, чтобы проверять, нужно ли компилировать модель, мы будем сравнивать время последней записи в файл модели с временем последней записи в файл DLL, получающийся в результате ее компиляции. Если запись в файл модели производилась позже записи в DLL, значит, модель изменена уже после последней компиляции, и ее необходимо компилировать снова. Таким образом, нам понадобится функция, определяющая время последней записи заданного файла:

```
// Получить время последней записи в файл
BOOL GetFileLastWrite(char *filename,FILETIME *pLastWrite)
{ HANDLE file;
  BOOL ok=TRUE;

  // Открываем файл для чтения
  file=CreateFile(filename,GENERIC_READ,FILE_SHARE_READ,
          NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
```

```
  if(file==INVALID_HANDLE_VALUE) // К файлу нет доступа
    return FALSE;

  if(pLastWrite)
    { // Файл открыт - получаем время изменения
      ok=GetFileTime(file,NULL,NULL,pLastWrite);
    }
  CloseHandle(file);
  return ok;
}
//=======================================
```

В эту функцию передается имя файла `filename`, время изменения которого нужно получить, и указатель `pLastWrite` на структуру Windows `FILETIME`, в которую функция должна записать это время. Функция возвращает `TRUE`, если ей удалось считать время, и `FALSE` в случае возникновения ошибок (например, если файла не существует). Фактически, функция состоит из трех вызовов функций Windows API: `CreateFile` для открытия файла (время изменения можно получить только у открытого файла), `GetFileTime` для получения времени его изменения, и `CloseHandle` для его закрытия. Мы не будем рассматривать ее более подробно.

На базе этой функции мы сделаем еще одну, которая будет проверять, нужно ли компилировать модель. В нее будут передаваться имена файлов модели и DLL с полными путями, а она будет сравнивать их времена изменения.

```
// Проверить необходимость компиляции DLL
BOOL CheckDllTime(char *modelfile,char *dllfile)
{ FILETIME modeltime,dlltime;
  // Получаем время изменения DLL
  if(!GetFileLastWrite(dllfile,&dlltime))
    return FALSE; // Файла DLL, вероятно, нет

  // Получаем время изменения файла модели
  if(!GetFileLastWrite(modelfile,&modeltime))
    return FALSE; // Нет доступа к файлу модели - ошибка

  // Если modeltime<=dlltime, можно не компилировать
  return CompareFileTime(&modeltime,&dlltime)<=0;
}
//=======================================
```

Эта функция будет возвращать `TRUE`, если компиляция не требуется, и `FALSE`, если ее нужно провести. В параметре `modelfile` передается имя файла модели, а в `dllfile` – имя файла DLL, оба имени должны содержать полные пути. Времена изменения этих файлов записываются в переменные `modeltime` и `dlltime` соответственно. Если время изменения DLL определить не удалось, вероятнее всего, файл DLL просто еще не существует, то есть модель еще ни разу не компилировалась – в этом случае функция возвращает `FALSE`: компиляция необходима. Неудача при определении времени изменения файла модели сигнализирует о какой-то ошибке, причины которой не ясны (например, файл модели может быть открыт другой программой с исключительным доступом). В этом случае функция тоже возвращает `FALSE`: лучше попытаться скомпилировать модель лишний раз, чем игнорировать ее изменение, которое не удалось определить из-за ошибки. Наконец, если оба времени изменения определены, они сравниваются друг с другом функцией Windows API `CompareFileTime`. Если время изменения модели `modeltime` окажется меньшим времени изменения DLL `dlltime`, функция вернет `TRUE`, поскольку в это случае компиляция не нужна. В противном случае функция вернет `FALSE`: модель изменена позднее DLL, и DLL устарела.

Нам также потребуется функция, которая запустит заданный EXE-файл с заданными параметрами командной строки и дождется его завершения – мы используем ее для запуска компилятора и редактора связей. Причем, поскольку в настройках блока в параметрах командной строки мы решили использовать символические константы “$INCLUDE$”, “$OBJFILE$” и “$DLLFILE$”, не поддерживаемые РДС, нам либо нужно заменять эти константы на их реальные значения перед передачей командной строки в нашу функцию, либо делать это внутри нее. Последнее удобней, поскольку нам нужно вызывать два разных исполняемых файла (компилятор и редактор связей) с разными параметрами командной строки, и если мы будем заменять в этих параметрах константы на их значения вне функции запуска, нам придется повторить один и тот же фрагмент программы два раза. Таким образом, лучше производить замену внутри функции, для чего мы будем, кроме имени исполняемого файла и строки параметров, передавать в нее два массива: массив имен констант и массив их значений. Возвращать эта функция будет логическое значение: `TRUE` при успешном запуске и `FALSE` при ошибке.

```
// Запуск программы и ожидание ее завершения
BOOL RunAndWait(
           char *path,           // Имя EXE-файла
           char *parameters,     // Параметры командной строки
           char **search,        // Массив имен констант
           char **replace)       // Массив значений констант
{ char *cmd,*commandline,*tempdir;
  STARTUPINFO startup; // Структура параметров запуска процесса
  PROCESS_INFORMATION info; // Структура описания процесса
  BOOL ok=TRUE;

  // Формирование полной командной строки
  cmd=rdsDynStrCat("\"",path,FALSE);
  rdsAddToDynStr(&cmd,"\" ",FALSE);
  rdsAddToDynStr(&cmd,parameters,FALSE);
  rdsAddToDynStr(&cmd," ",FALSE); // Пробел (на всякий случай)
  // В cmd теперь - команда запуска, символические константы
  // в ней еще не обработаны

  // Заменяем все константы на их значения
  commandline=rdsStringReplace(cmd,search,replace,
                          -1,RDS_SRF_STDPATHS);
  rdsFree(cmd); // cmd больше не нужна

  // В commandline - полная командная строка запуска программы
  if(commandline==NULL)
    return FALSE;

  // Получение имени временной папки РДС без '\'
  tempdir=rdsTransformFileName(
               rdsGetSystemPath(RDS_GSPTEMPPATH),
               RDS_TFN_EXCLUDEPATHBS,NULL,NULL);

  // Заполнение структуры STARTUPINFO
  ZeroMemory(&startup,sizeof(STARTUPINFO));
  startup.cb=sizeof(STARTUPINFO);

  // Запуск процесса
  if(!CreateProcess(NULL,commandline,NULL,NULL,FALSE,
           0,NULL,tempdir,&startup,&info))
    ok=FALSE; // Запустить не получилось
```

```
  if(ok)
    { // Ждем завершения процесса
      WaitForSingleObject(info.hProcess,INFINITE);
      // Процесс завершен - закрываем полученные дескрипторы
      CloseHandle(info.hThread);
      CloseHandle(info.hProcess);
    }

  // Освобождаем все динамические строки
  rdsFree(commandline);
  rdsFree(tempdir);
  return ok;
}
//=======================================
```

В параметре `path` в эту функцию передается полный путь к запускаемому EXE-файлу, в параметре `parameters` – строка его параметров (объединение этих двух строк даст полную командную строку для запуска программы). В параметре `search` передается указатель на массив символических констант, которые нужно заменить в командной строке (он должен завершаться значением `NULL`), а в параметре `replace` – массив значений этих констант. Два последних параметра подобраны так, чтобы их можно было без изменений передать в сервисную функцию РДС `rdsStringReplace`, которая занимается поиском и заменой в строках. Эта функция описана следующим образом:

```
LPSTR RDSCALL rdsStringReplace(
      LPSTR string,   // исходная строка
      LPSTR *search,  // массив строк для поиска
      LPSTR *replace, // массив строк для замены
      int count,      // размер массива search или -1
      DWORD flags);   // флаги
```

Она ищет в строке `string` все вхождения элементов массива `search`, заменяет их на элементы массива `replace` с тем же индексом, и возвращает динамически сформированную строку, получившуюся в результате этих замен (как обычно, ее нужно освобождать функцией `rdsFree`). В параметре `count` передается число элементов в массиве `search` или –1, если массив завершается константой `NULL`: в этом случае функция самостоятельно определит его размер. Параметр `flags` может быть произвольной комбинацией битовых флагов `RDS_SRF_STDPATHS` (заменять в строке стандартные символические константы путей РДС на их значения) и `RDS_SRF_IGNORECASE` (не учитывать регистр символов при поиске). Естественно, для нормальной работы этой функции размер массива `replace` должен быть равен размеру массива `search`, или может быть на единицу меньше, если последний элемент `search` равен `NULL`. Если, например, выполнить следующую программу:

```
char *names[]={"_PI_VAL_","$PI$",NULL};
char *values[]={"3.1415926","Пи"};
char *result=rdsStringReplace(
      "Значение $PI$ равно _PI_VAL_",
      names,values,-1,0);
rdsMessageBox(result,"Результат",MB_OK);
rdsFree(result);
```

то функция `rdsMessageBox` выведет сообщение “Значение Пи равно 3.1415926”.

Вернемся к функции `RunAndWait`. Сначала из параметров `path` и `parameters` нам нужно сформировать команду запуска программы `path` с параметрами командной строки `parameters`. Для этого мы функциями `rdsDynStrCat` и `rdsAddToDynStr` (они уже неоднократно рассматривались ранее) формируем динамическую строку `cmd` вида <*двойная кавычка*>+`path`+<*двойная кавычка и пробел*>+`parameters`+<*пробел*>. Поскольку в пути к EXE-файлу запускаемой программы могут содержаться пробелы, мы заключаем его в

двойные кавычки – так принято в Windows. Путь к исполняемому файлу и строку параметров мы разделяем пробелом, и в конец строки добавляем еще один пробел (практика показывает, что без этого пробела некоторые программы работают неправильно). Затем получившуюся строку мы обрабатываем функцией rdsStringReplace, чтобы заменить в ней все символические константы (как стандартные для РДС, так и переданные нами в массиве search) на их значения. В результате мы получаем динамическую строку commandline. После ее получения мы освобождаем строку cmd – она больше не нужна.

Затем мы формируем еще одну динамическую строку, содержащую путь к временной папке РДС без завершающего этот путь символа '\'. Для этого мы вызываем функцию rdsGetSystemPath с параметром RDS_GSPTEMPPATH (она вернет указатель на строку во внутренней памяти РДС, содержащую путь к временной папке), а затем обрабатываем возвращенное ей значение функцией rdsTransformFileName с параметром RDS_TFN_EXCLUDEPATHBS. Эта функция предназначена для обработки имен файлов и путей и всегда возвращает динамическую строку, так что нужно будет не забыть освободить ее в конце. Параметр RDS_TFN_EXCLUDEPATHBS указывает ей на необходимость удалить в конце переданного ей имени символ '\', если, конечно, он там присутствует. Мы могли бы удалить его и не пользуясь сервисными функциями РДС, но так текст функции получается короче.

После этого мы подготавливаем структуру параметров запуска STARTUPINFO, необходимую для запуска процесса (вся подготовка заключается в занесении в поле cb размера этой структуры и обнулении всех остальных ее полей – параметры запуска нам не нужны) и вызываем функцию Windows API CreateProcess, передавая ей командную строку для запуска commandline и путь к папке tempdir (эта папка станет текущей папкой процесса). CreateProcess запустит программу, указанную в commandline (если это возможно, иначе она немедленно вернет FALSE), запишет в структуру info типа PROCESS_INFORMATION, указатель на которую передан в ее последнем параметре, дескрипторы запущенного процесса и его потока, и вернет нам управление. Нам нужно дождаться окончания работы запущенной программы, поэтому мы вызываем функцию Windows API WaitForSingleObject, передавая ей дескриптор запущенного процесса info.hProcess и константу INFINITE (бесконечность) в качестве времени ожидания. Эта функция вернет управление только после завершения указанного в ее параметре процесса.

Дождавшись завершения запущенной программы, мы закрываем дескрипторы процесса и потока, возвращенные нам функцией CreateProcess (мы обязаны это сделать), освобождаем созданные динамические строки и возвращаем логическую переменную ok (если программу запустить не удалось, она будет иметь значение FALSE).

Теперь все вспомогательные функции готовы, и мы можем ввести в функцию модуля новые реакции. Для этого добавим в класс TCAutoCompData три новых функции-члена:

```
// Класс личной области данных модуля автокомпиляции
class TCAutoCompData
{ private:
    // ......................
    // ... без изменений ...
    // ......................

    // Сформировать в файле исходный текст программы
    BOOL WriteSourceCode(HANDLE file,char *name,
                RDS_HOBJECT varset,char *prog);
    // Загрузить и скомпилировать одну модель
    void LoadAndProcessModel(RDS_COMPMODELDATA *data,
                int fileset,BOOL varsonly);
```

```
    // Сообщение об ошибке в модели
    static void ModelErrorMsg(char *modelname,char *errortext);
  public:
    // Подготовиться к компиляции модели
    void PrepareToCompileModel(RDS_COMPPREPAREDATA *param);
    // Компилировать модели
    void CompileModels(RDS_COMPILEDATA *param);
    // Связать блок с моделью
    void AttachBlock(RDS_COMPBLOCKOPDATA *param);

    // ..........................
    // ... далее без изменений ...
    // ..........................
};
//=======================================
```

Функции `PrepareToCompileModel` и `CompileModels` будут вызываться из функции модуля при ее вызове с параметрами `RDS_COMPM_PREPARE` и `RDS_COMPM_COMPILE` соответственно. Из функции `CompileModels` мы будем вызывать для каждой компилируемой модели функцию `LoadAndProcessModel`, кроме того, формирование исходного текста компилируемой библиотеки мы для удобства вынесем в функцию `WriteSourceCode`. При присоединении модели к каждому блоку (в реакции на `RDS_COMPM_ATTACHBLOCK`) мы будем вызывать функцию `AttachBlock` – в ней мы присвоим блоку структуру переменных модели, если это необходимо.

Внутрь оператора `switch` в функции модуля нужно добавить следующие метки `case`:

```
    // Подготовка к компиляции
    case RDS_COMPM_PREPARE:
      data->PrepareToCompileModel(
               (RDS_PCOMPPREPAREDATA)ExtParam);
      break;

    // Компиляция моделей
    case RDS_COMPM_COMPILE:
      data->CompileModels((RDS_PCOMPILEDATA)ExtParam);
      break;

    // Присоединение модели к блоку
    case RDS_COMPM_ATTACHBLOCK:
      data->AttachBlock((RDS_PCOMPBLOCKOPDATA)ExtParam);
      break;
```

Функция `PrepareToCompileModel` должна проверить, требуется ли компиляция модели, указанной в ее параметре, и передать в блок имя DLL и имя функции блока, которые получатся в результате компиляции:

```
// Подготовиться к компиляции модели
void TCAutoCompData::PrepareToCompileModel(
                RDS_COMPPREPAREDATA *param)
{ char *modelpath,*dllpath;

  // Выясняем, нужно ли перекомпилировать DLL
  if(param->Rebuild) // Принудительно компилировать все
    param->Model->Valid=FALSE;
  else // Компилировать при необходимости
    { // Время последней записи DLL должно быть не меньше
      // времени записи текста модели
```

```
        // Полный путь к файлу модели
        modelpath=rdsGetFullFilePath(
                  param->Model->ModelName,NULL,NULL);
        // Полный путь к файлу DLL
        dllpath=rdsTransformFileName(
                  modelpath,RDS_TFN_CHANGEEXT,".dll",NULL);
        // Сравниваем времена последней записи
        param->Model->Valid=CheckDllTime(modelpath,dllpath);
        // Освобождаем строки
        rdsFree(modelpath);
        rdsFree(dllpath);
      }

    // Получаем имя DLL с сокращенным путем для записи
    // в параметры блока
    dllpath=rdsTransformFileName(
            param->Model->ModelName,RDS_TFN_CHANGEEXT,".dll",NULL);

    // Записываем в параметры блоков библиотеку и функцию в ней,
    // которые получатся в результате компиляции
    rdscompSetModelFunction(param->Model->Model,
                  dllpath,Exported);
    rdsFree(dllpath);
  }
  //=======================================
```

В эту функцию передается указатель `param` на структуру `RDS_COMPPREPAREDATA` (см. стр. 607) с двумя полями: указателем на структуру данных модели `Model` и признаком принудительной компиляции всех моделей `Rebuild`. Функция должна установить `Model->Valid` в `TRUE`, если данной модели не требуется компиляция, и в `FALSE`, если компиляция требуется.

Если поле `param->Rebuild` истинно (пользователь приказал заново скомпилировать все модели), `param->Model->Valid` устанавливается в `FALSE` – модель нужно компилировать. В противном случае мы должны сравнить времена последнего изменения файлов модели и DLL – если модель изменена позднее, DLL нужно компилировать заново. Для этого мы сначала преобразуем имя модели `param->Model->ModelName`, которое является относительным путем к ее файлу, в полный путь `modelpath` функцией `rdsGetFullFilePath`. Затем, заменяя расширение у файла модели на ".dll" вызовом функции `rdsTransformFileName` с параметром `RDS_TFN_CHANGEEXT`, мы получаем имя файла DLL с полным путем `dllpath` (и `modelpath`, и `dllpath` – динамически строки, поэтому их нужно будет освободить функцией `rdsFree`). Получив оба пути, мы передаем их в написанную ранее `CheckDllTime`, которая сравнит времена изменения этих файлов и вернет `FALSE`, если модель изменена позднее библиотеки. Возвращенное ей значение мы присваиваем `param->Model->Valid`.

Независимо от того, требуется модели компиляция или нет, в конце функции мы настраиваем блок на работу с нужной библиотекой и функцией. Если пользователь, например, подключает к блоку новую автоматически компилируемую модель, которой в данный момент не требуется компиляция (файл модели старше файла DLL), в параметрах блока на данный момент еще не установлены библиотека и функция модели, и модуль автокомпиляции все равно должен сообщить блоку, с какой библиотекой и какой функцией он теперь будет работать. Для этого мы преобразуем имя модели в имя файла DLL `dllpath` заменой расширения при помощи вызова `rdsTransformFileName`. В данном случае мы берем имя файла модели не с полным путем, а с относительным

(param->Model->ModelName), чтобы получившееся имя библиотеки тоже содержало относительные пути. Таким образом, например, если файл модели находится в одной папке со схемой, в которой используется эта модель, в параметрах блока имя модели не будет содержать пути вообще, и имя файла DLL блока, полученное из имени модели, тоже не будет содержать пути. Именно это нам и нужно: при переносе всей папки со схемой, моделями и полученными из них файлами DLL, схема не утратит работоспособности, поскольку все пути в параметрах блоков – относительные.

После того, как мы получили имя файла DLL, в котором будет находиться экспортированная функция модели этого блока, мы настраиваем все блоки, использующие данную автоматически компилируемую модель, на работу с этой библиотекой и этой функцией вызовом сервисной функции rdscompSetModelFunction, в первом параметре которой передается идентификатор данной модели (param->Model->Model), во втором – сформированное нами имя библиотеки с относительными путями (dllpath), в третьем – имя экспортированной функции (поле Exported класса личной области нашего модуля, которое мы сделали одним из настраиваемых параметров). После этого мы освобождаем динамическую строку dllpath и завершаем функцию PrepareToCompileModel.

Функция CompileModels, в которой мы будем компилировать модели, в структуре данных которых поле Valid имеет значение FALSE, будет достаточно простой, поскольку собственно компиляцию модели мы вынесли в отдельную функцию LoadAndProcessModel. Здесь же мы просто будем перебирать в цикле все модели:

```
// Компилировать все модели
void TCAutoCompData::CompileModels(RDS_COMPILEDATA *param)
{
  // Проверяем необходимые параметры
  if(CompPath==NULL || LinkPath==NULL ||
     IncludePath==NULL || LibPath==NULL)
    return; // Компиляция невозможна

  // В цикле перебираем все модели из param->InvalidModels
  for(int i=0;i<param->IMCount;i++)
    { int tempset;
      // Создаем новый набор временных файлов
      tempset=rdsTMPCreateFileSet();
      // Компилируем модель
      LoadAndProcessModel(param->InvalidModels[i],tempset,FALSE);
      // Удаляем все созданные в процессе временные файлы
      rdsTMPDeleteFileSet(tempset);
    }
}
//=======================================
```

В параметре param этой функции передается указатель на структуру RDS_COMPILEDATA (см. стр. 607). В ней нас будут интересовать два поля: массив указателей на структуры данных моделей InvalidModels и его размер IMCount. При вызове модуля для компиляции РДС включает в этот массив только те модели, которые требуют компиляции, поэтому нам не нужно вручную проверять поле Valid в их структурах данных. Таким образом, нам просто нужно вызвать LoadAndProcessModel для каждой модели, указатель на структуру данных (RDS_COMPMODELDATA, см. стр. 604) которой находится в массиве param->InvalidModels. Однако, для некоторого облегчения написания функции компиляции модели LoadAndProcessModel, перед ее вызовом мы создаем набор временных файлов сервисной функцией РДС rdsTMPCreateFileSet, а после него удаляем все временные файлы этого набора функцией rdsTMPDeleteFileSet, при этом идентификатор созданного набора мы передаем в функцию LoadAndProcessModel,

чтобы она могла с ним работать. Остановимся на этих функциях для работы с временными файлами подробнее.

В процессе компиляции нам придется создать на диске некоторое количество дополнительных файлов: для работы компилятора нужно сформировать в каком-либо файле исходный текст библиотеки на языке С, компилятор преобразует его в объектный файл и т.д. После завершения компиляции все эти файлы желательно удалить, чтобы они зря не занимали место на диске (лучше не оставлять после себя "мусор"). Значит, нам нужно запоминать где-то имена всех созданных нами файлов, а потом, когда эти файлы уже не нужны, удалять их. В РДС есть специальные сервисные функции, которые позволяют автоматизировать этот процесс – ими мы и будем пользоваться. Эти функции, кроме того, позволяют решить проблему подбора имен для временных файлов. Конечно, мы можем заложить в нашу программу жестко заданные имена файлов: например, исходный текст библиотеки, который будет обрабатываться компилятором, мы можем всегда формировать в файле "source.cpp" в папке временных файлов РДС – кажется, что там он никому не будет мешать. Однако, представим себе, что на одной машине запущено две копии РДС, и им обеим одновременно пришлось заняться компиляцией своих моделей. Обе копии попытаются записать в папку файл с одним и тем же именем, что приведет к конфликту. К счастью, мы можем предоставить выбор имени для файла сервисной функции РДС – она подберет такое имя, которое не совпадает ни с одним файлом в заданной папке, причем она сразу создаст файл с этим именем нулевого размера, чтобы ни одна другая программа не смогла "захватить" это имя и помешать нам использовать этот файл. Нам останется только открыть его для записи и сформировать в нем исходный текст библиотеки.

Все временные файлы, с которыми работает РДС, объединены в наборы, каждый из которых имеет целый идентификатор. Прежде чем начать работать с временными файлами, нужно вызвать функцию `rdsTMPCreateFileSet`, которая вернет уникальный, нигде в данный момент не используемый, идентификатор набора. Этот идентификатор используется в вызовах всех остальных функций работы с временными файлами.

Для того, чтобы подобрать имя для временного файла и создать файл нулевого размера, "заняв" таким образом это имя, вызывается функция `rdsTMPCreateEmptyFile`:

```
LPSTR RDSCALL rdsTMPCreateEmptyFile(
      int setid,       // Идентификатор набора временных файлов
      LPSTR name);     // Желаемое имя с путем
```

В параметре `setid` передается идентификатор набора, к которому будет относиться создаваемый файл, а в параметре `name` – его желаемое имя с указанием пути (можно использовать как полные пути, так и относительные, с символическими константами РДС). Если файл с именем `name` в данный момент не существует, функция оставит это имя без изменений, если же такой файл уже есть, она изменит его имя, сохранив путь и расширение и сделав это имя уникальным в данной папке. Подобрав имя, функция создаст пустой файл, запомнит его имя во внутренней памяти РДС и вернет указатель на это запомненное имя с полным путем. Поскольку функция не создает динамических строк, результат ее возврата не нужно освобождать в вызвавшей программе.

Если имя для файла подбирать не нужно, можно просто добавить заданное имя к списку временных файлов набора функцией `rdsTMPRememberFileName`. Она может быть полезна в тех случаях, когда временные файлы создаются какой-либо другой программой (например, компилятором), но мы знаем их имена и хотим удалить их по окончании работы. В эту функцию тоже можно передать имя с относительным путем, она возвращает указатель на запомненное во внутренней памяти РДС имя с полным путем.

По окончании работы с временными файлами можно удалить их все одним вызовом функции `rdsTMPDeleteFileSet`, в которую передается идентификатор удаляемого набора. Зная полное имя временного файла, можно удалить только этот файл и с диска, и из набора функцией `rdsTMPDeleteFile`, однако, следует помнить, что даже после удаления

всех файлов набора вызовами rdsTMPDeleteFile, пустой набор все равно останется в памяти, и его нужно будет удалить вызовом rdsTMPDeleteFileSet. Также следует учитывать, что при завершении РДС или загрузке новой схемы все временные файлы всех наборов удаляются автоматически, поэтому нет способа не удалять файл, объявленный временным при помощи функций rdsTMPCreateEmptyFile или rdsTMPRememberFileName.

Возвращаясь к функции CompileModels, можно видеть, что, вызывая в цикле функцию LoadAndProcessModel, мы передаем ей не только указатель на структуру данных компилируемой модели param->InvalidModels[i] и идентификатор созданного для нее набора временных файлов tempset, но еще и значение FALSE в третьем параметре. Этот третий параметр функции LoadAndProcessModel мы будем использовать для ограничения ее работы: если он равен FALSE, функция должна будет скомпилировать модель и присвоить всем блокам, обслуживаемым моделью, ее структуру переменных, если же параметр будет равен TRUE, функция должна будет только присвоить блокам структуру переменных, не компилируя модель. Это ограничение позволит нам использовать одну и ту же функцию и для компиляции модели в реакции RDS_COMPM_COMPILE, и для установки структуры переменных блока в реакции RDS_COMPM_ATTACHBLOCK.

Прежде чем заняться функцией LoadAndProcessModel, напишем сначала функцию формирования исходного текста компилируемой библиотеки WriteSourceCode. Будем считать, что файл, в который нужно записать исходный текст, уже открыт на запись, а файл модели уже считан и разобран на структуру переменных блока и текст реакции на такт моделирования. Эта функция будет довольно объемной из-за большого размера исходного текста, который она формирует.

```
// Сформировать в файле исходный текст программы
BOOL TCAutoCompData::WriteSourceCode(
      HANDLE file,          // дескриптор открытого файла
      char *name,           // имя модели (для сообщения)
      RDS_HOBJECT varset,   // набор переменных
      char *prog)           // исходный текст реакции
{ BOOL ok;
  RDS_VARDESCRIPTION vdescr;
  char *undef=NULL;

  // Начальные описания и тип главной функции DLL
  ok=WriteString(file,
    "#include <windows.h>\r\n"
    "#include <stdlib.h>\r\n"
    "#include <math.h>\r\n"
    "#include <RdsDef.h>\r\n"
    "#define RDS_SERV_FUNC_BODY GetServiceFunc\r\n"
    "#include <RdsFunc.h>\r\n"
    "double _HugeDouble;\r\n\r\n" // значение ошибки
    "int WINAPI ");
  // Имя главной функции DLL (из настроек модуля)
  ok=ok && WriteString(file,DllMainName);
  // Тело главной функции DLL
  ok=ok && WriteString(file,
    "(HINSTANCE /*hinst*/,unsigned long reason,"
    "void */*lpReserved*/)\r\n"
    "{ if(reason==DLL_PROCESS_ATTACH)\r\n"
    "    { if(!RDS_SERV_FUNC_BODY())\r\n"
    "        { MessageBox(NULL,\"Нет сервисных функций РДС\",\"");
```

```
// Имя модели в сообщении об ошибке
ok=ok && WriteString(file,name);
// Продолжение главной функции DLL
ok=ok && WriteString(file,"\",MB_OK | MB_ICONERROR);\r\n"
  "          return 0;\r\n"
  "        }\r\n"
  "      else\r\n"
  "        rdsGetHugeDouble(&_HugeDouble);\r\n"
  "    }\r\n"
  "  return 1;\r\n"
  "}\r\n\r\n");

// Заголовок функции блока (из настроек)
ok=ok && WriteString(file,ModelFuncHdr);
// Продолжение функции блока и проверка типов переменных
ok=ok && WriteString(file,
  "(int CallMode,RDS_PBLOCKDATA BlockData,LPVOID ExtParam)\r\n"
  "{ switch(CallMode)\r\n"
  "    { case RDS_BFM_VARCHECK:\r\n"
  "        if(strcmp((char*)ExtParam,\"");
// Строка типа переменных блока (из varset)
ok=ok && WriteString(file,
  rdsGetObjectStr(varset,RDS_HVAR_GETTYPESTRING,0));
// Завершение проверки типа переменных
ok=ok && WriteString(file,
  "\")) return RDS_BFR_BADVARSMSG;\r\n"
  "        break;\r\n");

// Такт моделирования
ok=ok && WriteString(file,
  "      case RDS_BFM_MODEL:\r\n");

// Макрос для начала дерева переменных
ok=ok && WriteString(file,"#define _pVarDataStart "
        " ((BYTE*)(BlockData->VarData))\r\n");
// Макросы для переменных блока
vdescr.servSize=sizeof(vdescr);
if(rdsVSGetVarDescription(varset,-1,&vdescr))
  { // В vdescr теперь - описание всей структуры переменных блока
    char *type,*soff;
    int offset=0;
    int n=vdescr.StructFields; // Число полей

    // Записываем макрос для каждой переменной
    for(int i=0;i<n;i++)
      if(rdsVSGetVarDescription(varset,i,&vdescr))
        { // Получили описание i-й переменной блока
          switch(vdescr.Type) // Тип переменной
            { case RDS_VARTYPE_SIGNAL:
              case RDS_VARTYPE_LOGICAL:
              case RDS_VARTYPE_CHAR:
                type="char"; break;
              case RDS_VARTYPE_SHORT:
                type="short int"; break;
              case RDS_VARTYPE_INT:
                type="int"; break;
              case RDS_VARTYPE_FLOAT:
                type="float"; break;
```

```
                    case RDS_VARTYPE_DOUBLE:
                      type="double"; break;
                    default: // Тип не поддерживается
                      // Смещение к следующей переменной
                      offset+=vdescr.DataSize;
                      continue;
                  }
                if(i==0 && vdescr.Type!=RDS_VARTYPE_SIGNAL)
                  break;  // Первая переменная - не сигнал
                          // Продолжать не имеет смысла
                ok=ok && WriteString(file,"#define ");
                // Имя переменной
                ok=ok && WriteString(file,vdescr.Name);
                ok=ok && WriteString(file," (*((");
                ok=ok && WriteString(file,type);
                ok=ok && WriteString(file,"*)(_pVarDataStart+");
                // Смещение к переменной (offset) в виде строки
                soff=rdsItoA(offset,10,0);
                ok=ok && WriteString(file,soff);
                rdsFree(soff);
                ok=ok && WriteString(file,")))\r\n");
                // Смещение к следующей переменной
                offset+=vdescr.DataSize;
                // Добавление к undef директивы отмены этого макроса
                rdsAddToDynStr(&undef,"#undef ",FALSE);
                rdsAddToDynStr(&undef,vdescr.Name,FALSE);
                rdsAddToDynStr(&undef,"\r\n",FALSE);
              } // rdsVSGetVarDescription(varset,i,...))
        } // if(rdsVSGetVarDescription(...)

      // Вставляем текст пользователя
      ok=ok && WriteString(file,"{\r\n");
      ok=ok && WriteString(file,prog);
      ok=ok && WriteString(file,"\r\n}\r\n");

      // Отмена макросов переменных
      if(undef!=NULL)
        { ok=ok && WriteString(file,undef);
          rdsFree(undef);
        }
      // Отмена макроса начала дерева переменных
      ok=ok && WriteString(file,"#undef _pVarDataStart\r\n");

      // Завершение функции модели
      ok=ok && WriteString(file,
        "        break;\r\n"
        "    } // switch\r\n"
        "  return RDS_BFR_DONE;\r\n"
        "}\r\n"
        );

      return ok;
    }
    //=======================================
```

В параметре `file` этой функции передается дескриптор открытого для записи файла, в который мы будем писать исходный текст, в параметре `name` – указатель на строку имени модели, которая нужна будет нам для вывода сообщения об ошибке (пользователь должен

знать, в какой модели произошла ошибка), в параметре varset – идентификатор вспомогательного объекта РДС, содержащего считанную из модели структуру переменных, и, наконец, в параметре prog – указатель на текст реакции на такт моделирования.

Прежде всего, нам нужно записать директивы включения заголовочных файлов, необходимых для компиляции модели, и главную функцию DLL – то есть, все то, что мы до сих пор писали вручную в примерах моделей блоков. Для этого мы вызываем функцию WriteString, передавая ей текст, содержащий все указанные строки программы, разделенные возвратом каретки и переводом строки (“\r\n”), вплоть до имени главной функции DLL. Имя главной функции мы сделали параметром модуля, поэтому в качестве него мы записываем в файл поле DllMainName нашего класса. Затем мы записываем в файл параметры и тело главной функции (они не зависят от параметров модуля), пока не дойдем до вывода сообщения о невозможности получения доступа к сервисным функциям РДС. Заголовком этого сообщения должно быть имя модели, поэтому его мы записываем в файл из параметра name. Потом мы снова записываем большой фрагмент текста, не зависящий от параметров модуля и модели, включающий последний параметр функции MessageBox, чтение значения, используемое математическими функциями для индикации ошибки, сервисной функцией rdsGetHugeDouble, и завершение главной функции DLL. Заголовок функции модели блока мы опять берем из настроек модуля: он находится в поле ModelFuncHdr. Затем опять следует фиксированный фрагмент текста, до тех пор, пока мы не дойдем до формирования реакции модели на вызов проверки типов переменных RDS_BFM_VARCHECK: строку типа переменных, с которой нужно сравнить полученную формируемой нами функцией модели строку, мы должны получить из набора переменных модели varset. Для этого мы вызываем функцию получения строки объекта rdsGetObjectStr, передавая ей идентификатор объекта varset и константу RDS_HVAR_GETTYPESTRING (функция вернет указатель на строку во внутренней памяти объекта, поэтому освобождать эту строку нам не придется). Полученную строку мы записываем в файл, за ней снова следует фиксированный фрагмент текста: завершение реакции RDS_BFM_VARCHECK и начало реакции на такт моделирования RDS_BFM_MODEL. Теперь нам нужно записать текст модели, переданный в параметре prog, но сначала нужно сформировать макросы для доступа к статическим переменным (см. стр. 42), чтобы в тексте реакции можно было использовать имена этих переменных.

Перед записью макроса для самой первой переменной мы должны, как обычно, ввести вспомогательное макроопределение для указателя на начало дерева переменных блока, приведенного к какому-либо однобайтовому типу (во предыдущих примерах мы почти всегда называли этот макрос “pStart”). Конечно, можно было бы обойтись и без него, выполняя это приведение внутри каждого макроса переменной, но при этом макроопределение становится длинным и сложным для понимания (слишком много скобок и приведений типов друг за другом). В данном случае понятность макроса нам не важна – формируемый нами текст не предназначен для чтения человеком, однако, поскольку раньше мы использовали такой вспомогательный макрос, будем использовать его и сейчас, чтобы записываемый функцией WriteSourceCode текст было проще сопоставить с примерами моделей, которыми мы занимались раньше. Мы записываем в файл следующую строчку:

```
#define _pVarDataStart ((BYTE*)(BlockData->VarData))
```

Теперь мы сможем обращаться к любой переменной блока, отсчитывая от pVarDataStart заданное число байтов.

В отличие от строки типа, макросы для переменных мы не можем получить из объекта varset одним вызовом. Придется перебрать все его переменные, формируя макрос для каждой из них вручную. Для этого нам понадобится структура описания переменной vdescr типа RDS_VARDESCRIPTION, в поле servSize которой нужно занести ее

собственный размер, чтобы сервисные функции РДС смогли проверить, ту ли структуру мы просим заполнить.

Чтобы перебрать все переменные объекта varset, нам, прежде всего, нужно узнать их число и получить идентификатор содержащейся в этом объекте структуры, полями которой и являются эти переменные. Для этого мы заполняем vdescr описанием внутренней структуры переменных объекта, вызывая сервисную функцию rdsVSGetVarDescription и передавая ей идентификатор объекта varset, –1 в качестве номера переменной (так мы получим не описание одной из переменных объекта, а описание всей его структуры переменных) и указатель на заполняемую структуру &vdescr. Если функция вернет TRUE, в поле vdescr.StructFields будет находиться общее число переменных в объекте varset. Мы переписываем его во вспомогательную переменную n и используем ее как условие завершения цикла, в котором мы перебираем переменные объекта. Поскольку в этом цикле мы будем получать описание каждой переменной через ту же самую структуру vdescr, мы не можем использовать поле vdescr.StructFields в качестве условия завершения цикла – после первого же вызова rdsVSGetVarDescription в цикле в этом поле окажется другое значение. Именно поэтому перед циклом мы переписываем его в n. Для того, чтобы вставлять в макросы смещение к переменной от начала дерева, мы вводим вспомогательную переменную offset и присваиваем ей нулевое значение – в цикле мы будем каждый раз добавлять к ней размер очередной переменной.

В цикле мы получаем описание i-й переменной объекта, и, в зависимости от ее типа vdescr.Type, записываем в type указатель на строку, в которой находится описание типа этой переменной в синтаксисе языка C. Для типа RDS_VARTYPE_DOUBLE ('D') это будет строка “double”, для RDS_VARTYPE_INT ('I') – “int”, и т.д. Если переменная не относится ни к одному из простых типов (в этом случае мы попадем на метку default оператора switch), мы добавляем к offset размер этой переменной и переходим к началу цикла – наши модели не поддерживают такие переменные, и макросы для них не нужны.

Если счетчик цикла i равен нулю, значит, сейчас мы будем записывать в файл макрос для самой первой переменной блока. Эта переменная обязательно должна быть сигналом, поскольку наши модели предназначены для работы только с простыми блоками, структура переменных которых всегда начинается с двух сигналов (см. стр. 20). Если тип первой переменной vdescr.Type отличается от RDS_VARTYPE_SIGNAL, мы выходим из цикла записи макросов оператором break – модель не предназначена для работы с такой структурой переменных, и продолжать не имеет смысла.

Далее в теле цикла мы формируем макроопределение для переменной i. Для этого мы записываем в файл конструкцию вида

```
#define vdescr.Name (*((type*)(_pVarDataStart+offset)))
```

Здесь vdescr.Name, type и offset – значения соответствующих переменных нашей функции (для записи в файл целое число offset предварительно переводится в строку функцией rdsItoA, эта строка затем освобождается вызовом rdsFree). После этого мы добавляем к offset размер текущей переменной vdescr.DataSize (таким образом offset становится смещением к следующей переменной). Далее мы добавляем к динамической строке undef (в начале функции мы присвоили ей NULL) директиву для отмены только что записанного макроса. Когда мы запишем в файл текст реакции на такт моделирования и нам понадобится отменить все введенные макросы, нам не нужно будет снова перебирать все переменные объекта varset – нам достаточно будет просто записать в файл строку undef.

Записав в файл макросы для всех простых переменных объекта varset, мы вставляем в него текст реакции на такт моделирования из параметра prog, окружив его фигурными скобками (так мы даем пользователю возможность описывать свои

вспомогательные переменные в тексте реакции), после чего записываем строку отмены макросов undef, если она не пустая, и освобождаем занятую ей память. Затем мы записываем в файл директиву отмены макроса _pVarDataStart, дописываем завершающие строки функции модели, включая оператор return, и возвращаем значение логической переменной ok, в которую все это время записывали результат вызова функции WriteString. Таким образом, если при записи текста в файл произойдет ошибка, функция WriteSourceCode вернет FALSE.

Если обработать этой функцией модель со стр. 619 (для определенности будем считать, что она находится в файле “testmodel.txt” – имя модели передается в функцию WriteSourceCode и используется в сообщении об ошибке), то, при параметрах модуля по умолчанию получится следующий текст программы (фрагменты текста, зависящие от модели и параметров модуля автокомпиляции, выделены жирным шрифтом):

```
#include <windows.h>
#include <stdlib.h>
#include <math.h>
#include <RdsDef.h>
#define RDS_SERV_FUNC_BODY GetServiceFunc
#include <RdsFunc.h>
double _HugeDouble;

int WINAPI DllEntryPoint(HINSTANCE /*hinst*/,
                unsigned long reason,void */*lpReserved*/)
{ if(reason==DLL_PROCESS_ATTACH)
    { if(!RDS_SERV_FUNC_BODY())
        { MessageBox(NULL,"Нет сервисных функций РДС",
                "testmodel.txt",MB_OK | MB_ICONERROR);
          return 0;
        }
      else
        rdsGetHugeDouble(&_HugeDouble);
    }
  return 1;
}

extern "C" __declspec(dllexport) int RDSCALL autocompModelFunc(
           int CallMode,RDS_PBLOCKDATA BlockData,LPVOID ExtParam)
{ switch(CallMode)
    { case RDS_BFM_VARCHECK:
        if(strcmp((char*)ExtParam,"{SSDDD}"))
                    return RDS_BFR_BADVARSMSG;
        break;
      case RDS_BFM_MODEL:
#define _pVarDataStart  ((BYTE*)(BlockData->VarData))
#define Start (*((char*)(_pVarDataStart+0)))
#define Ready (*((char*)(_pVarDataStart+1)))
#define x1 (*((double*)(_pVarDataStart+2)))
#define x2 (*((double*)(_pVarDataStart+10)))
#define y (*((double*)(_pVarDataStart+18)))
{
y=x1+x2;
}
#undef Start
#undef Ready
#undef x1
#undef x2
```

```
#undef y
#undef _pVarDataStart
        break;
    } // switch
  return RDS_BFR_DONE;
}
```

Этот текст мало чем отличается от исходных текстов простых моделей, которые мы до сих пор писали вручную.

При работе с компилятором и редактором связей, которые будут преобразовывать сформированный функцией `WriteSourceCode` текст программы в работоспособную библиотеку, нам нужно будет проверять, удалось ли компилятору создать объектный файл, а редактору связей – файл DLL. Для проверки наличия этих файлов и получения их размера мы напишем вспомогательную функцию `ReadFileSize`, которая, получив имя файла с полным путем, вернет младшие четыре байта его размера (файлы, размер которых не умещается в четырехбайтовое беззнаковое целое, нас не интересуют):

```
// Получить размер файла по его имени
DWORD ReadFileSize(char *filename)
{ HANDLE f;
  DWORD size;
  // Открываем файл на чтение
  f=CreateFile(filename,GENERIC_READ,0,NULL,OPEN_EXISTING,0,NULL);
  if(f==INVALID_HANDLE_VALUE) // Ошибка - нет файла
    return 0;
  // Получаем размер файла
  size=GetFileSize(f,NULL);
  // Закрываем файл
  CloseHandle(f);
  return size;
}
//=========================================
```

Теперь мы можем, наконец, написать функцию `LoadAndProcessModel`, которая будет загружать файл модели, вызывать `WriteSourceCode`, после чего запускать компилятор с редактором связей.

```
// Загрузить и скомпилировать одну модель
void TCAutoCompData::LoadAndProcessModel(
      RDS_COMPMODELDATA *data,   // структура данных модели
      int fileset,               // набор временных файлов
      BOOL varsonly)             // только установка переменных
{ char *modeltext,*dllpath,*aux;
  char *vars,*prog;
  char *st_srcfile,*st_dllfile,*st_objfile;
  RDS_HOBJECT varset;
  RDS_BLOCKDESCRIPTION bdescr;
  BOOL ok=TRUE;
  // Массивы для автоматической замены строк
  char *search[]={"$INCLUDE$",  // 0
                  "$LIB$",      // 1
                  "$CPPFILE$",  // 2
                  "$OBJFILE$",  // 3
                  "$DLLFILE$",  // 4
                  NULL};
  char *replace[sizeof(search)/sizeof(char*)];

  bdescr.servSize=sizeof(bdescr);
```

```
// Читаем весь файл модели в память
modeltext=ReadTextFile(data->ModelName,0);
if(modeltext==NULL) // Ошибка чтения
  { ModelErrorMsg(data->ModelName,"Ошибка чтения файла модели");
    return;
  }

// Файл модели считан в modeltext - разбиваем на секции
// vars и prog
if(!ProcessModelText(modeltext,&vars,&prog))
  { rdsFree(modeltext);
    ModelErrorMsg(data->ModelName,
              "В файле нет необходимых секций");
    return;
  }

// Создаем объект для работы с переменными
varset=rdsVSCreateEditor();

// Записываем в объект получившийся текст описания переменных
if(!rdsVSCreateByDescr(varset,vars))
  { rdsFree(modeltext);
    rdsDeleteObject(varset);
    ModelErrorMsg(data->ModelName,
              "Ошибка в секции описания переменных");
    return;
  }

// Копируем переменные во все блоки, к которым подключена модель
for(int i=0;i<data->NBlocks;i++)
  { // Берем очередной блок в списке блоков модели
    RDS_BHANDLE block=
              rdscompGetModelBlock(data->Model,i,&bdescr);
    if(block==NULL) continue;
    if(bdescr.BlockType!=RDS_BTSIMPLEBLOCK)
      continue; // Только для простых блоков
    // Записываем переменные в блок
    rdsVSApplyToBlock(varset,block,NULL);
  }

if(varsonly)
  { // Нужно только установить переменные, компилировать не нужно
    rdsFree(modeltext);
    rdsDeleteObject(varset);
    return;
  }

// Формирование исходного текста компилируемой библиотеки
st_srcfile=rdsTMPCreateEmptyFile(fileset,"$TEMP$\\model.cpp");
// В st_srcfile - указатель на имя созданного временного
// файла (строка имени находится во внутренней памяти РДС)
if(st_srcfile)
  { HANDLE file=CreateFile(st_srcfile,GENERIC_WRITE,
                           0,NULL,CREATE_ALWAYS,0,NULL);
    if(file==INVALID_HANDLE_VALUE) // Ошибка
      ok=FALSE;
```

```
    else
      { // Записываем в файл исходный текст программы
        ok=WriteSourceCode(file,data->ModelName,varset,prog);
        CloseHandle(file);
      }
  }
if(!ok)
  ModelErrorMsg(data->ModelName,
              "Невозможно создать файл исходного текста");

// Текст модели и список переменных больше не нужны
rdsFree(modeltext);
rdsDeleteObject(varset);

// Создание временных файлов .obj и .dll
st_objfile=rdsTMPCreateEmptyFile(fileset,"$TEMP$\\model.obj");
st_dllfile=rdsTMPCreateEmptyFile(fileset,"$TEMP$\\model.dll");

// Занесение различных путей в список замены replace
replace[0]=IncludePath;  // "$INCLUDE$"
replace[1]=LibPath;      // "$LIB$"
replace[2]=st_srcfile;   // "$CPPFILE$"
replace[3]=st_objfile;   // "$OBJFILE$"
replace[4]=st_dllfile;   // "$DLLFILE$"

// Запуск компилятора
if(ok)
  { if(RunAndWait(CompPath,CompParams,search,replace))
      { // Проверяем наличие .obj
        if(ReadFileSize(st_objfile)==0) // Нулевого размера
          { ok=FALSE;
            ModelErrorMsg(data->ModelName,
                          "Объектный файл не создан - "
                          "в тексте модели есть ошибки");
          }
      }
    else // Ошибка запуска
      { ok=FALSE;
        ModelErrorMsg(data->ModelName,
                    "Ошибка запуска компилятора");
      }
  }

// Запуск редактора связей
if(ok)
  { if(RunAndWait(LinkPath,LinkParams,search,replace))
      { // Проверяем наличие .dll
        if(ReadFileSize(st_dllfile)==0) // Нулевого размера
          { ok=FALSE;
            ModelErrorMsg(data->ModelName,
                    "Файл DLL не создан - "
                    "в модели есть ошибки");
          }
      }
    else // Ошибка запуска
      { ok=FALSE;
        ModelErrorMsg(data->ModelName,
                    "Ошибка запуска редактора связей");
```

```
        }
      }

    // Удаляем файлы tds и lib, созданные редактором связей
    aux=rdsTransformFileName(st_dllfile,
                  RDS_TFN_CHANGEEXT,".tds",NULL);
    DeleteFile(aux); rdsFree(aux);
    aux=rdsTransformFileName(st_dllfile,
                  RDS_TFN_CHANGEEXT,".lib",NULL);
    DeleteFile(aux); rdsFree(aux);

    // Полный путь к файлу DLL (куда ссылается блок)
    dllpath=rdsGetFullFilePath(data->CompDllName,NULL,NULL);
    // Копируем созданный во временной директории файл DLL
    // в папку модели
    if(ok)
      { ok=CopyFile(st_dllfile,dllpath,FALSE);
        if(!ok)
          ModelErrorMsg(data->ModelName,
                  "Ошибка копирования полученного файла DLL");
      }

    // Освобождаем строку пути к DLL
    rdsFree(dllpath);
  }
  //=======================================
```

В эту функцию передается указатель на структуру данных модели `data` (см. стр. 604), идентификатор набора временных файлов `fileset` (набор мы создали в функции `CompileModels` перед вызовом `LoadAndProcessModel`) и логический параметр `varsonly`, принимающий значение `TRUE`, если нам нужно только установить структуру переменных в блоках, использующих эту модель. В самом начале функции мы описываем массивы строк `search` и `replace`, которые мы будем использовать для замены символических констант, введенных нами в командных строках компилятора и редактора связи, на их реальные значения. Эти массивы будут передаваться в написанную нами вспомогательную функцию `RunAndWait` (стр. 638), в которой они используются в вызове `rdsStringReplace`. Массив `search` мы сразу инициализируем указателями на имена используемых нами констант, завершая его значением `NULL`, а массив `replace` пока оставляем пустым – указатели на значения констант мы запишем в него позже. Число элементов в массиве `replace` мы делаем равным числу элементов массива `search`, вычисляя его как размер `search` в байтах (`sizeof(search)`), деленный на размер одного элемента (`sizeof(char*)`).

Сначала мы считываем в память файл модели, имя которой находится в параметре `data->ModelName`, функцией `ReadTextFile`, и записываем возвращенный ей указатель на динамически отведенную область памяти с загруженным текстом вы переменную `modeltext`. Мы написали эту функцию так, чтобы в нее можно было передавать имя файла с относительными путями и символическими константами, каким является имя модели в нашем модуле автокомпиляции, поэтому нам не нужно предварительно обрабатывать эти имя функцией `rdsGetFullFilePath`. Считанный текст мы разбиваем на секцию описания переменных `vars` и секцию реакции на такт моделирования `prog`, создаем объект для работы с переменными `varset` и формируем в нем набор переменных, указанных в секции `vars`. Теперь текст модели разобран – можно приступать к установке переменных блоков, формированию текста программы на языке С и его компиляции.

Сначала мы должны скопировать в каждый блок, к которому присоединена данная модель, переменные из объекта varset. Блоки, обслуживаемые моделью, мы перебираем в цикле for со счетчиком i, изменяющимся от 0 до data->NBlocks (в этом поле структуры данных модели всегда хранится общее число блоков с этой моделью). В цикле мы получаем идентификатор i-го блока модели функцией rdscompGetModelBlock, которая попутно заполняет структуру описания этого блока bdescr, и, если этот блок – простой (его тип RDS_BTSIMPLEBLOCK), копируем в него структуру переменных объекта varset функцией rdsVSApplyToBlock.

После того, как всем блокам присвоена структура переменных модели, мы проверяем, не равен ли TRUE параметр varsonly. Если это так, функция LoadAndProcessModel вызвана только для установки переменных блоков, и записывать текст программы DLL и вызывать компилятор не нужно. В этом случае мы уничтожаем все созданные динамические строки и объекты и немедленно завершаем функцию.

Теперь нам нужно сформировать исходный текст DLL с функцией модели блока согласно разобранному файлу модели и параметрам модуля автокомпиляции. После компиляции этот файл нужно будет удалить, поэтому мы воспользуемся встроенным в РДС механизмом работы с временными файлами (он был описан на стр. 644) – создадим пустой временный файл вызовом функции rdsTMPCreateEmptyFile, передав ей идентификатор набора временных файлов из параметра fileset и имя создаваемого файла “$TEMP$\model.cpp”. Эта функция преобразует константу “$TEMP$” в путь к папке временных файлов РДС и, если там нет файла с именем “model.cpp”, создаст пустой файл с этим именем. Если такой файл там уже есть, функция автоматически изменит имя файла так, чтобы оно было уникальным. В любом случае, функция вернет указатель на строку с полным путем к созданному файлу, которую мы запишем в переменную st_srcfile. Эта строка находится во внутренней памяти РДС и нам не нужно освобождать ее вызовом rdsFree. Созданный таким образом файл будет автоматически удален при удалении набора fileset (мы делаем это в функции CompileModels после вызова LoadAndProcessModel) или при завершении РДС.

Создав пустой временный файл, мы открываем его для записи вызовом функции Windows API CreateFile и формируем в нем исходный текст программы DLL вспомогательной функцией WriteSourceCode. После этого мы можем освободить память, занимаемую загруженным текстом модели modeltext и удалить объект varset, содержащий набор переменных модели – они нам больше не потребуются.

Теперь мы имеем файл с текстом модели блока на языке C, который нам нужно скомпилировать. Для этого нам предстоит вызвать компилятор, который преобразует исходный текст в объектный файл, а затем – редактор связей, который преобразует объектный файл в исполняемый файл DLL и создаст еще пару вспомогательных файлов. Эти файлы нам тоже нужно будет удалить после компиляции, кроме того, для объектного и исполняемого файлов нужно подобрать уникальные имена, отсутствующие в данный момент в папке временных файлов РДС. Мы делаем два вызова функции rdsTMPCreateEmptyFile: один – для объектного файла, для которого мы предлагаем имя “$TEMP$\model.obj”, другой – для исполняемого (“$TEMP$\model.dll”). Теперь st_objfile указывает на полный путь к объектному файлу, а st_dllfile – на полный путь к DLL, которую должен создать редактор связей.

На данный момент мы уже знаем значения всех символических констант, которые мы разрешаем пользователю указывать в командной строке, поэтому мы заносим их в массив replace. В нулевой и первый элементы массива записываются указатели на пути к папкам заголовков и библиотек компилятора из параметров модуля IncludePath и LibPath, а в три оставшихся элемента – указатели на пути к созданным нами файлу исходного текста,

объектному файлу и исполняемому файлу DLL (st_srcfile, st_objfile и st_dllfile соответственно).

Для запуска компилятора мы вызываем вспомогательную функцию RunAndWait, передавая ей путь к исполняемому файлу компилятора из параметра модуля CompPath, его командную строку с символическими константами CompParams, а также массивы для поиска и замены символических констант search и replace. Если RunAndWait вернула TRUE, значит, компилятор был успешно запущен и уже завершил работу (функция RunAndWait ждет завершения процесса) – при этом мы проверяем размер объектного файла st_objfile, который должен быть создан в результате компиляции. Если этот файл отсутствует или имеет нулевой размер, значит, компилятору не удалось его создать. При правильных настройках модуля это будет означать наличие ошибок в тексте модели, о чем мы сообщаем пользователю. Если объектный файл имеет ненулевой размер, мы точно так же вызываем редактор связей, после чего проверяем наличие и размер исполняемого файла st_dllfile. Если редактор связей успешно создал этот файл, нам нужно скопировать его в ту же папку, в которой находится файл модели, и дать ему имя, отличающееся от имени файла модели только расширением ".dll". Но сначала мы удаляем вспомогательные файлы, имеющие то же имя, что и файл DLL, но с расширениями ".tds" и ".lib". Для этого мы заменяем в st_dllfile расширение вызовом rdsTransformFileName с параметром RDS_TFN_CHANGEEXT, удаляем файл с получившимся именем и освобождаем строку имени вызовом rdsFree. Можно было бы добавить имена этих файлов в набор fileset и положиться на механизм удаления временных файлов, но нам все равно пришлось бы вызывать rdsTransformFileName для получения имени файла (имена этих файлов жестко определяются редактором связей и не должны подбираться по уникальности), поэтому лучше сразу удалить их вызовом DeleteFile.

Файл DLL, созданный в результате компиляции, имеет имя st_dllfile. Относительный путь к файлу DLL блока, установленный нами в реакции модуля на RDS_COMPM_PREPARE, находится в поле CompDllName структуры данных модели data. Мы преобразуем этот относительный путь в полный путь dllpath вызовом rdsGetFullFilePath, после чего функцией Windows API копируем st_dllfile в dllpath и освобождаем динамическую строку dllpath. Теперь библиотека с моделью блока сформирована и помещена в нужную папку с нужным именем, и РДС подключит ее к блоку, как только модуль автокомпиляции вернет управление.

Нам осталось написать функцию AttachBlock, которая вызывается из реакции RDS_COMPM_ATTACHBLOCK при подключении модели к блоку. В ней мы должны, при необходимости, скопировать в блок переменные модели, иначе, если мы присоединяем к новому блоку модель, не требующую в данный момент компиляции, блок останется со старой структурой переменных.

```
// Связать блок с моделью
void TCAutoCompData::AttachBlock(RDS_COMPBLOCKOPDATA *param)
{
  // Если модель устанавливается пользователем вручную,
  // нужно поменять у блока структуру переменных
  if(param->AttachReason==RDS_COMP_AR_MANUALSET)
    LoadAndProcessModel(param->Model,-1,TRUE);
}
//=======================================
```

Здесь мы поступаем просто: если модель подключена к блоку пользователем вручную (параметр param->AttachReason имеет значение RDS_COMP_AR_MANUALSET), мы вызываем функцию LoadAndProcessModel для модели param->Model со значением –1 вместо идентификатора набора временных файлов и TRUE в последнем параметре varsonly. При таком сочетании параметров функция загрузит файл указанной модели,

скопирует во все обслуживаемые ей блоки структуру переменных, описанную в этом файле, после чего завершится без вызова компилятора. Конечно, нам нужно переписать структуру переменных только в один блок, идентификатор которого передается в `param->Block`, но, чтобы не усложнять `LoadAndProcessModel`, мы обработаем все блоки. Это не приведет к сколько-нибудь существенным задержкам.

Может возникнуть вопрос: почему мы копируем переменные модели в блок только при ручном подключении? Дело в том, что во всех остальных случаях подключения модели к блоку – при загрузке блока из файла, вставке из буфера обмена и т.п. – структура переменных блока уже должна быть правильной, ведь на момент сохранения блока или его копирования в буфер модель уже была подключена, а значит, она уже установила в блоке правильные переменные. Можно придумать ситуацию, в которой это будет не так: например, если мы сохраним схему с блоком, к которому подключена модель, а потом заменим файл модели и соответствующий ей скомпилированный файл DLL на другие, то на момент загрузки схемы переменные блока не будут соответствовать подключаемой модели, причем модель не будет требовать компиляции. Однако, ничего страшного при этом не произойдет: формируемые нами модели содержат стандартную проверку типов переменных `RDS_BFM_VARCHECK`, и при неправильной структуре переменных модель просто откажется работать, пока пользователь не скомпилирует ее заново.

Если бы мы вызывали `LoadAndProcessModel` при каждом подключении модели к блоку, это привело бы к существенному замедлению загрузки схемы, потому что при этом каждый раз загружался бы файл модели. Выбранный нами вариант позволяет добиться некоторого компромисса между удобством и надежностью без сильного усложнения модуля автокомпиляции. Для того, чтобы структура переменных блока всегда соответствовала подключаемой модели, можно было бы создать для модели личную область данных и загружать туда описание переменных из файла модели при ее подключении к самому первому блоку, а для последующих подключений использовать уже загруженные данные (тогда файл загружался бы только один раз), но это существенно усложнило бы модуль, поэтому в данном примере мы не будем этого делать.

Теперь мы имеем работоспособный настраиваемый модуль автоматической компиляции, позволяющий задавать структуру переменных блока и его реакцию на такт моделирования. При внесении изменений в модель этот модуль будет автоматически вызывать указанный в его параметрах компилятор и заменять DLL блоков на новую. Компилятор будет вызываться в отдельном окне консоли, как на рис. 138 (для лучшей читаемости рисунка окно консоли изображено белым, хотя на самом деле оно будет черным). После внесения изменений в модель компилятор будет вызываться при переходе в режим моделирования или расчета, при загрузке схемы, использующей измененные модели, или по команде пользователя.

На самом деле, созданный нами модуль подходит только для создания очень простых моделей. В нем не хватает многих возможностей – в частности, работы с матрицами, строками и динамическими переменными. Входящие в комплект РДС модули для работы с основными компиляторами языка C++ поддерживают динамические переменные, сохранение и загрузку параметров блоков, окна настройки, реакцию на функции и т.п. Все это можно добавить и в наш модуль за счет усложнения пользовательского интерфейса редактора модели и функции формирования текста компилируемой DLL, но, поскольку созданный нами модуль предназначен только для иллюстрации основных приемов создания модулей автоматической компиляции, мы не будем этого делать.

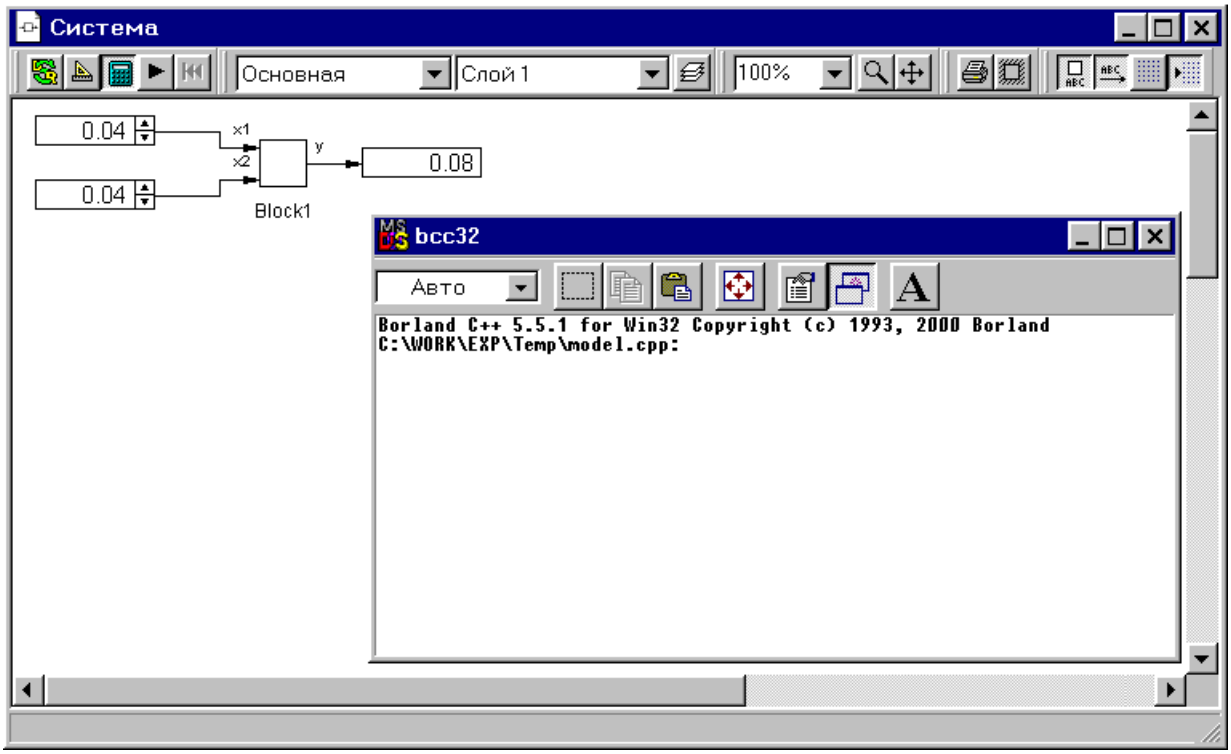


Рис. 138. Окно консоли вызванного компилятора

Список литературы

1. Дорри М.Х., Климачев С.Н. Машинный синтез структур систем автоматического управления. – Учебное пособие: Московский институт радиотехники, электроники и автоматики. 1978 – 116 с.
2. Dorri M.H., Ivanov M.A., Klimachev S.N., Pylaev A.M. EXPRESS-RADIUS package for Control System Simulation and Design/ – Recent advances in computer-aided control system engineering. M. Jamshidi and C.J. Herget (Editors), ELSEVIER SCIENCE PUBLISHERS B.V., (ISBN: 0-444-89255-9 (Vol. 9)) pp. 417–429, 1992.
3. Дорри М.Х., Рощин А.А. Инструментальная программно-алгоритмическая система для разработки исследовательских комплексов // Мехатроника, автоматизация, управление. № 12, 2008. С. 12 – 17.
4. Dorri M H., Roshchin A.A. Multicomputer Research Desks for Simulation and Development of Control Systems. IFAC, CD "PREPRINTS of the 17th IFAC World Congress July 6-11,2008, Seoul, Korea", p. 15244-15249
5. Дорри М.Х., Рощин А.А. Инструментальный программный комплекс РДС (Расчет Динамических Систем) – средство моделирования и разработки алгоритмов управления // Проблемы управления. 2009. № 4. С.52–58.
6. Дорри М.Х., Рощин А.А. Методические указания по выполнению лабораторных работ для студентов, обучающихся по специальности 220201. – М.: МГИРЭА(ТУ), 2006 – 70 с.
7. http://www.youtube.com/watch?v=KNPlaXGszAc
8. Charles Petzold. Programming Windows, Fifth Edition (Microsoft Programming Series). Microsoft Press, 1998
9. Харт, Джонсон М. Системное программирование в среде Windows, 3-е издание.: пер. с англ. - М.: Издательский дом "Вильямс", 2005 - 592 с. : ил. – Парал.тит.англ.
10. Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, OpenGL Architecture Review Board. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2. Addison-Wesley Professional, 2005.
11. Порев В.Н. Компьютерная графика. – СПб.: БХВ-Петербург, 2002 – 432 с.: ил.
12. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ = Introduction to Algorithms. – М.: «Вильямс», 2009 – 1290 с.:ил.
13. Бахвалов Н.С. Численные методы (анализ, алгебра, обыкновенные дифференциальные уравнения). – М.: Наука. Гл. ред. физ.-мат. лит., 1973
14. Дьяконов В.П. Справочник по алгоритмам и программам на языке бейсик для персональных ЭВМ: Справочник. – М.: Наука. Гл. ред. физ.-мат. лит., 1987. – 240 с.
15. Забудский Н.А. Внешняя баллистика – СПб.: Типография Императорской Академии Наук, 1895 – 578 с. :табл., ил.
16. Вентцель Д.А., Шапиро Я.М. Внешняя баллистика: Учебник для втузов : в 2 ч. Ч. 1. – М.; Л. : Оборонгиз, 1939.
17. Гребенюк Г.Г., Лубков Н.В., Никишов С.М. Информационные аспекты управления муниципальным хозяйством. М.: ЛЕНАНД, 2011. – 320 с.

Алфавитный указатель