

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

М.Х.Дорри, А.А.Рощин

**Расчет Динамических Систем (РДС)
Справочное руководство
Часть II: разработка собственных автокомпилируемых блоков**

Москва 2017

Во второй части справочного руководства рассматривается использование встроенных модулей автоматической компиляции моделей блоков. Эти модули облегчают создание нестандартных моделей. Пользователь записывает основной алгоритм работы модели в виде фрагмента программы на языке C++, а модуль автоматической компиляции при помощи внешнего компилятора создает на основе этого фрагмента исполняемый файл библиотеки, который подключается к схеме.

Оглавление

Глава 3. Использование стандартных модулей автокомпиляции.....	6
§3.1. Принцип действия стандартного модуля автокомпиляции.....	6
§3.2. Установка стандартных модулей автокомпиляции и компиляторов.....	10
§3.2.1. Установка и подключение компилятора Borland C++ 5.5.....	10
§3.2.2. Установка и подключение компилятора Open Watcom C++.....	12
§3.2.3. Установка и подключение компилятора Digital Mars C++.....	13
§3.2.4. Установка и подключение компилятора MinGW GCC.....	15
§3.2.5. Установка и подключение компилятора Microsoft Visual C++ 2003 toolkit.....	16
§3.2.6. Подключение модуля автоматического поиска модулей.....	18
§3.3. Создание нового блока с автокомпилируемой моделью.....	20
§3.4. Окно параметров блока с автокомпилируемой моделью.....	27
§3.5. Копирование блоков и схем с автокомпилируемыми моделями и совместное использование моделей.....	28
§3.6. Окно редактора модели.....	32
§3.6.1. Элементы и меню окна редактора модели.....	32
§3.6.2. Статические переменные блока.....	38
§3.6.3. Динамические переменные блока.....	42
§3.6.4. Описания программы и реакции блока на события.....	46
§3.6.5. Функции блока.....	57
§3.6.6. Настроечные параметры блока.....	60
§3.6.7. Параметры модели.....	69
§3.6.8. Установка параметров блоков с автокомпилируемой моделью.....	76
§3.7. Принципы создания автокомпилируемых моделей блоков.....	80
§3.7.1. Устройство формируемой модулем программы.....	80
§3.7.2. Работа со статическими переменными блока.....	92
§3.7.2.1. Модели с простыми статическими переменными.....	92
§3.7.2.2. Модели с матрицами.....	99
§3.7.2.3. Модели с массивами.....	103
§3.7.2.4. Модели со структурами.....	107
§3.7.2.5. Модели со строками.....	112
§3.7.2.6. Использование сигналов.....	117
§3.7.2.7. Использование входов со связанными сигналами.....	123
§3.7.2.8. Использование выходов с управляющими переменными.....	126
§3.7.3. Работа с динамическими переменными.....	129
§3.7.3.1. Подключение к динамической переменной.....	129
§3.7.3.2. Создание динамических переменных.....	136
§3.7.3.3. Динамические переменные сложных типов.....	141
§3.7.4. Моделирование длящихся во времени процессов.....	145
§3.7.4.1. Общие принципы численного моделирования непрерывных процессов.....	145
§3.7.4.2. Система дифференциальных уравнений и задание начальных условий.....	152
§3.7.4.3. Важность правильного выбора шага расчета.....	159
§3.7.4.4. Создание динамического блока по шаблону.....	168
§3.7.5. Блоки, программно рисующие свое изображение.....	173
§3.7.6. Блоки с настраиваемыми пользователем параметрами.....	194
§3.7.7. Задание пользователем имен динамических переменных.....	203
§3.7.8. Программное управление динамическими переменными.....	206
§3.7.9. Всплывающие подсказки.....	211
§3.7.10. Пометки на блоках.....	223
§3.7.11. Реакция блока на мышшь.....	226

§3.7.12. Добавление пунктов в контекстное и главное меню.....	238
§3.7.13. Вызов функций блоков.....	245
§3.7.13.1. Общие принципы работы с функциями блока.....	245
§3.7.13.2. Вызов функции у всех блоков подсистемы.....	250
§3.7.13.3. Вызов функции у одного блока.....	258
§3.7.13.4. Регистрация и поиск исполнителя функции.....	268
§3.7.13.5. Объекты функций в автокомпилируемых моделях.....	279
§3.8. Краткий перечень вводимых в модель описаний и реакций на события.....	282
§3.8.1. Дополнительные описания, вводимые в модель.....	282
§3.8.2. Создание и уничтожение блока.....	286
§3.8.2.1. Инициализация блока.....	286
§3.8.2.2. Очистка данных блока.....	287
§3.8.2.3. Добавление блока пользователем.....	287
§3.8.2.4. Удаление блока пользователем.....	288
§3.8.2.5. Перед выгрузкой схемы.....	289
§3.8.3. Моделирование и переключение режимов.....	290
§3.8.3.1. Выполнение такта расчета.....	290
§3.8.3.2. Запуск расчета.....	291
§3.8.3.3. Остановка расчета.....	291
§3.8.3.4. Сброс расчета.....	292
§3.8.3.5. Переход в режим редактирования.....	292
§3.8.3.6. Переход в режим моделирования.....	293
§3.8.3.7. Изменение динамической переменной.....	293
§3.8.4. Реакции блока на мышь и клавиатуру.....	295
§3.8.4.1. Нажатие кнопки мыши.....	295
§3.8.4.2. Отпускание кнопки мыши.....	297
§3.8.4.3. Двойной щелчок мыши.....	297
§3.8.4.4. Перемещение курсора мыши.....	298
§3.8.4.5. Нажатие клавиши.....	298
§3.8.4.6. Отпускание клавиши.....	300
§3.8.5. Вызов функции блока.....	301
§3.8.6. Загрузка и запись данных блока и всей схемы.....	303
§3.8.6.1. Загрузка данных блока.....	303
§3.8.6.2. Запись данных блока.....	304
§3.8.6.3. Перед сохранением схемы.....	305
§3.8.6.4. После сохранения схемы.....	305
§3.8.6.5. После загрузки схемы.....	306
§3.8.7. Загрузка и запись мгновенного состояния блока.....	306
§3.8.7.1. Загрузка состояния блока.....	306
§3.8.7.2. Запись состояния блока.....	307
§3.8.8. Реакции окна подсистемы.....	308
§3.8.8.1. Действия с окном подсистемы.....	308
§3.8.8.2. Нажатие кнопки мыши (в окне подсистемы).....	309
§3.8.8.3. Отпускание кнопки мыши (в окне подсистемы).....	309
§3.8.8.4. Двойной щелчок (в окне подсистемы).....	310
§3.8.8.5. Перемещение курсора (в окне подсистемы).....	310
§3.8.8.6. Нажатие клавиши (в окне подсистемы).....	310
§3.8.8.7. Отпускание клавиши (в окне подсистемы).....	311
§3.8.9. Внешний вид блока.....	312
§3.8.9.1. Размер блока изменен.....	312
§3.8.9.2. Проверка изменения размера блока.....	313

§3.8.9.3. Блок перемещен.....	314
§3.8.9.4. Рисование блока.....	315
§3.8.9.5. Дополнительное рисование блока.....	315
§3.8.10. Обмен данными по сети.....	316
§3.8.10.1. Получены данные по сети.....	316
§3.8.10.2. Сетевое соединение установлено.....	317
§3.8.10.3. Сетевое соединение разорвано.....	318
§3.8.10.4. Данные приняты сервером.....	319
§3.8.10.5. Ошибка сети.....	319
§3.8.11. Прочие реакции.....	321
§3.8.11.1. Переименование блока.....	321
§3.8.11.2. Вызов настройки.....	321
§3.8.11.3. Сообщение от управляющей программы.....	322
§3.8.11.4. Срабатывание таймера.....	323
§3.8.11.5. Обновление окон блока.....	324
§3.8.11.6. Всплывающая подсказка.....	324
§3.8.11.7. Вызов контекстного меню.....	325
§3.8.11.8. Выбор пункта меню.....	326
§3.8.11.9. После создания статических переменных.....	327
§3.8.11.10. Прочие события.....	327
§3.9. Настройки стандартного модуля автокомпиляции.....	328
§3.9.1. Общие настройки модуля.....	328
§3.9.2. Добавление и изменение шаблонов моделей.....	331
§3.9.3. Подключение универсальных модулей и настройка путей.....	333
§3.9.4. Символические имена параметров в настройках.....	337
§3.9.5. Настройка переменных окружения компилятора.....	341
§3.9.6. Запуск компилятора и редактора связей.....	342
§3.9.7. Разбор ошибок компиляции.....	345
§3.9.8. Общие описания в программе.....	350
§3.9.9. Параметры формирования исходного текста.....	351
§3.9.10. Настройка обработки исключений и ошибок.....	356
Список литературы.....	360
Алфавитный указатель.....	361

Глава 3. Использование стандартных модулей автокомпиляции

В этой главе описывается создание пользовательских моделей блоков при помощи входящих в состав РДС модулей автоматической компиляции. Пользователь записывает основные действия, выполняемые блоком, в виде программ на языке C++, а модуль, при помощи внешнего компилятора, собирает из этих программ полноценную модель блока.

§3.1. Принцип действия стандартного модуля автокомпиляции

Рассматриваются общие принципы работы входящих в состав РДС модулей автоматической компиляции, приводится список поддерживаемых компиляторов.

В РДС модули автоматической компиляции призваны облегчать написание моделей блоков – эти модули работают посредниками между пользователем и одним из установленных в системе компиляторов какого-либо языка высокого уровня. Обычно пользователь пишет только самые необходимые фрагменты программы модели, а затем модуль автокомпиляции формирует из них полный текст программы и передает его компилятору, создающему исполняемый файл динамической библиотеки (DLL). После этого функция модели из этой библиотеки автоматически подключается к одному или нескольким блокам схемы. Такая схема работы позволяет пользователю сосредоточиться на алгоритме работы блока, не вдаваясь в тонкости правильного оформления исходного текста программы и описаний, необходимых для создания динамических библиотек Windows. Стандартные модули автокомпиляции РДС поддерживают только язык C++, и, при их использовании, все фрагменты программ пользователь должен писать на этом языке. Для написания простейших моделей достаточно знать, как записываются математические выражения и операторы присваивания. Если же пользователь захочет расширить возможности своих моделей, ему будет полезно изучить сам язык C++ [4] и его стандартные библиотеки.

Модули автокомпиляции не являются неотъемлемой частью РДС – они не встроены в главную программу “rds.exe”, а, как и модели блоков, находятся во внешних динамически подключаемых библиотеках. В состав РДС по умолчанию входит несколько модулей, рассчитанных на работу с компиляторами языка C++: часть модулей настроена на работу с конкретными компиляторами разных производителей, часть – универсальные, предназначенные для подключения произвольного компилятора. Все эти модули имеют один и тот же интерфейс пользователя, позволяющий достаточно гибко настраивать взаимодействие с компилятором. Фактически, модули для конкретных компиляторов отличаются от универсальных модулей только набором параметров по умолчанию. В этом параграфе будут рассмотрены только самые общие принципы работы этих стандартных модулей, более подробно их настройка рассматривается в §3.9 (стр. 328). Модули для других языков программирования в состав РДС не входят, при необходимости, они могут быть созданы сторонними разработчиками (правила написания таких модулей подробно рассмотрены в главе 4 руководства программиста [1]).

Все стандартные модули автокомпиляции находятся в библиотеке “Common.dll” в папке DLL РДС. Для подключения какого-либо из них к РДС в стандартном окне подключения модулей автокомпиляции (см. §2.19.1 части I, а также §3.2 на стр. 10 здесь) необходимо указать имя этой библиотеки с префиксом “\$DLL\$” и имя функции, обслуживающей конкретный модуль, из следующей таблицы:

<i>Имя функции модуля</i>	<i>Компилятор, для которого предназначен модуль</i>
BCpp55	Borland C++ 5.5 (бесплатный, http://www.embarcadero.com/products/cbuilder/free-compiler) или Borland C++ Builder 6 (коммерческий)
OpenWatcomCpp	Open Watcom C++ (бесплатный, http://www.openwatcom.org)

<i>Имя функции модуля</i>	<i>Компилятор, для которого предназначен модуль</i>
DigitalMars	Digital Mars C++ (бесплатный, http://www.digitalmars.com)
Gcc_MinGW	MinGW GCC (бесплатный, http://www.mingw.org)
MSVCTK2003	Microsoft Visual C++ 2003 toolkit (бесплатный)
MSVCpp6	Microsoft Visual C++ 6 (коммерческий)
UserComp1, UserComp2, UserComp3	Универсальные модули, которые пользователь должен настроить самостоятельно

Принимая решение об установке и подключении к РДС того или иного компилятора, следует, по возможности, выбирать тот, для которого имя функции модуля приведено в таблице на первых строчках. Проще всего установить и подключить к РДС компиляторы Borland и Watcom, сложнее всего – MinGW и Microsoft Visual C++ 2003 (последний в настоящее время недоступен для загрузки на официальном сайте Microsoft). Модули с именами “UserComp1”, “UserComp2” и “UserComp3” не рассчитаны на какой-либо стандартный компилятор, это универсальные модули, в настройках которых необходимо указать пути ко всем исполняемым файлам компилятора, папкам библиотек и заголовков, параметры командной строки и т.п. Это требует определенного опыта в работе с компиляторами, управляемыми через командную строку, поэтому, при наличии у пользователя какого-либо из перечисленных выше стандартных компиляторов, рекомендуется использовать специально предназначенный для него модуль. Подключение компиляторов к стандартным модулям будет рассмотрено далее в §3.2 (стр. 10), настройка универсальных – в §3.9 (стр. 328).

Кроме перечисленных выше, в библиотеке “Common.dll” находится еще один дополнительный модуль автокомпиляции с именем функции “SearchComp”. Этот модуль не рассчитан на какой-либо конкретный компилятор и не может сам компилировать модели. Вместо этого он пытается найти среди модулей из приведенной выше таблицы тот, который пользователь уже настроил, и обращается к нему. Это возможно, поскольку все стандартные модули автокомпиляции совместимы: у них одинаковый формат моделей и одинаковый способ формирования текста программ. Основное назначение этого дополнительного модуля – передача другому пользователю схемы с автокомпилируемыми блоками. Если, например, передать схему, в которой для компиляции используется модуль для Borland, другому пользователю, у которого настроен только модуль для Watcom, этому другому пользователю придется заменять используемый модуль во всех блоках схемы. Если же заранее переключить все блоки на использование модуля “SearchComp”, у каждого пользователя будет использоваться тот модуль, который он успешно настроил и использует сам. Обращения к модулю “SearchComp” несколько задерживают загрузку схемы, поэтому, если все пользователи работают с одним и тем же компилятором, лучше не использовать этот модуль.

Независимо от используемого компилятора, все модули работают по одному принципу. В режиме редактирования пользователь может связать с каким-либо блоком схемы автоматически компилируемую модель, описание которой хранится на диске в отдельном файле, как правило, с расширением “.mdl”. Модуль автокомпиляции позволяет пользователю вводить и изменять эту модель в специальном окне редактора (рис. 301), открываемом при двойном щелчке на блоке или из контекстного меню этого блока. Одна и та же модель может быть связана с несколькими блоками схемы или даже с разными блоками в разных схемах, при этом ее изменение одновременно отражается на работе всех использующих ее блоков во всех схемах. В окне редактора пользователь не только вводит фрагменты программы, которые будут выполняться для данного блока при наступлении

различных системных событий, но и задает список статических и динамических переменных блока, набор его настроечных параметров, внешний вид окна настройки и т.п.

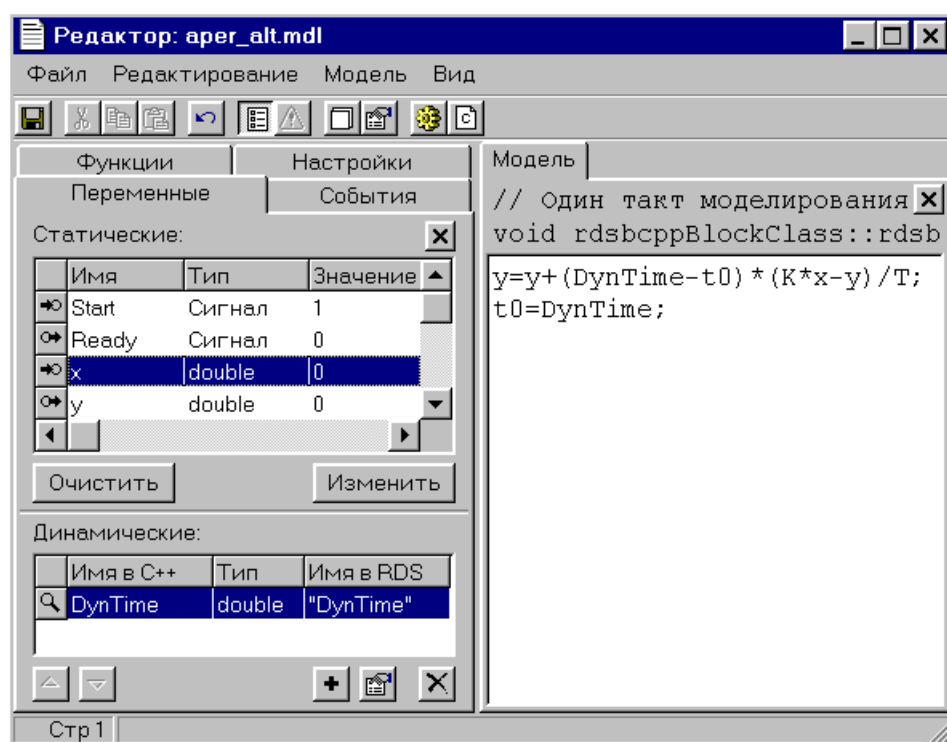


Рис. 301. Типичный внешний вид окна редактора модели

При переходе РДС из режима редактирования в режим моделирования или расчета модуль автокомпиляции проверяет наличие изменений в модели. Если изменения есть, то из введенных пользователем фрагментов программы модуль составляет полный исходный текст для динамически подключаемой библиотеки с моделью блока, записывает этот текст в файл на диске и запускает компилятор, указанный в настройках модуля. В РДС в одной динамической библиотеке может содержаться несколько моделей блоков, но, для упрощения взаимодействия, все стандартные модули автокомпиляции формируют библиотеки по принципу “одна модель – одна библиотека”. Таким образом, если в схеме, например, используется три разных автокомпилируемых модели (не важно, к скольким блокам они подключены), модуль сформирует три отдельных исходных текста на языке C++ и три раза запустит компилятор для формирования из них исполняемых файлов DLL. Если компиляция прошла без ошибок, полученные модели, то есть функции, экспортированные из созданных библиотек, будут подключены к блокам и начнут работу. Если же компилятор не смог создать исполняемые файлы из-за ошибок во введенных пользователем фрагментах, модуль автокомпиляции разберет список ошибок, полученный от компилятора, и предъявит его пользователю. Таков общий принцип работы модуля автокомпиляции. Разумеется, в описанном процессе есть множество особенностей, которые могут быть настроены пользователем. Например, можно указать, какие дополнительные описания помещаются в автоматически формируемый текст программы, как модель должна перехватывать возникающие в программе исключения и т.п. – все это будет рассмотрено далее в §3.9 (стр. 328).

Выше уже было упомянуто, что файлы моделей хранятся на диске не внутри файла самой схемы, а в отдельных файлах, ссылки на которые находятся в файле схемы вместе с другими параметрами блоков, что позволяет использовать одни и те же модели в разных схемах. Однако, это приводит к тому, что для переноса схемы с такими моделями на другую машину или в другую папку на этой же машине часто недостаточно скопировать только файл

схемы (“.rds”) – кроме него требуются еще и файлы используемых в схеме моделей (“.mdl”). Чтобы упростить перенос схем, файлы моделей обычно сохраняют либо в ту же папку, в которой находится использующая их схема или несколько схем, либо в папку стандартных моделей, которая указывается в настройках РДС (см. §2.18 части I). Если сохранить файл модели в ту же папку, что и схему, его имя будет запомнено в параметрах блока без пути. При этом всю папку со схемой целиком можно будет перемещать, в том числе и с машины на машину, без потери работоспособности – модуль автокомпиляции при отсутствии полного пути к файлу модели всегда ищет его в папке со схемой. Это удобно для специализированных моделей, используемых только в одной схеме или группе схем (такие схемы при этом лучше всего разместить в одной папке). Если же создаваемая модель будет достаточно универсальной, чтобы ее имело смысл использовать в разных схемах, не связанных между собой, ее лучше записать в стандартную папку моделей. В этом случае в имени файла модели, которое будет запомнено в схеме, вместо конкретного пути будет подставлено символическое обозначение “\$MODEL\$”. При загрузке этой схемы модуль автокомпиляции будет искать указанный файл модели в стандартной папке, где бы она ни находилась, и схему тоже можно будет перемещать без потери ее работоспособности. Разумеется, при перемещении схемы на другую машину следует скопировать файл модели в папку стандартных моделей на этой машине.

Технически модуль автокомпиляции позволяет сохранить файл модели в любую папку, в том числе и не относящуюся к РДС или к конкретной схеме, в которую входит блок с этой моделью. Однако, это не рекомендуется из-за возможных проблем при переносе схемы: если пользователь решит передать кому-либо созданную им схему и все ее модели, эти модели на другой машине придется размещать в папках с точно такими же именами. Например, если на машине пользователя модель находилась в папке “e:\documents\user\models”, и этот полный путь был запомнен в какой-либо схеме, загрузка такой схемы на другой машине будет требовать наличия файла модели именно в этой папке, что может создать проблемы, особенно, если на этой машине нет диска “e:”. Следует также учитывать, что, если сохранять файл модели в одну из стандартных папок РДС, модуль автокомпиляции всегда будет заменять путь к ней на соответствующее символическое обозначение – это касается не только стандартной папки моделей, вместо пути к которой подставляется “\$MODEL\$”, но и папки DLL (“\$DLL\$”), папки настроек (“\$INI\$”) и т.п. Полный список символических обозначений стандартных папок приводится в §A.5.4.9 приложения к руководству программиста [2].

Создаваемый при помощи стандартного модуля автокомпиляции файл динамической библиотеки с моделью блока всегда располагается в одной папке с файлом этой модели и получает то же имя, что и этот файл, только с расширением “.dll”. Если, например, файл модели с именем “whatever.mdl” находится в папке стандартных моделей РДС, то исполняемый файл DLL, созданный на его основе, тоже будет находиться в папке стандартных моделей и получит имя “whatever.dll”. Если файл модели находится в одной папке со схемой, скомпилированный файл тоже будет находиться в папке со схемой, и т.п. По времени изменения скомпилированного файла модуль проверяет необходимость компиляции модели: если файл модели имеет более позднее время изменения, чем соответствующий ему файл DLL, значит, после последней компиляции в модель были внесены изменения, и компиляцию следует выполнить заново. Если системные часы по какой-либо причине работают неверно (например, их время сбросилось из-за разряда батарейки), эта проверка может не сработать, и, несмотря на наличие в модели изменений, модуль не будет ее компилировать. В этом случае можно выбрать в главном меню РДС пункт “система | перекомпилировать все модели”, который заставит все активные в данный момент модули автокомпиляции принудительно скомпилировать все модели, используемые в загруженной схеме. Можно также принудительно скомпилировать конкретную модель, открыв ее редактор (см. стр. 36). Для удобства работы рекомендуется восстановить работу

системных часов – в противном случае после каждого изменения моделей придется вручную принудительно их компилировать.

Следует учитывать, что описанные принципы работы модулей (хранение моделей в отдельных файлах, редактирование их текстов в отдельных окнах и т.п.) относятся только к стандартным модулям автокомпиляции, входящим в состав РДС. Модули, созданные сторонними разработчиками, могут работать по-другому.

Далее будут рассмотрены процедуры установки и подключения конкретных компиляторов и настройка предназначенных для них модулей.

§3.2. Установка стандартных модулей автокомпиляции и компиляторов

Описываются действия, необходимые для установки некоторых стандартных компиляторов языка C++, а также настройка модулей автокомпиляции на работу с ними.

§3.2.1. Установка и подключение компилятора Borland C++ 5.5

Описывается установка бесплатного компилятора Borland C++ 5.5 и подключение модуля автокомпиляции, который с ним работает. Этот модуль также может использоваться для работы с коммерческим компилятором Borland C++ Builder 6.

Модуль автоматической компиляции “BCpp55” из библиотеки “Common.dll” предназначен для работы с бесплатно распространяемым компилятором Borland C++ 5.5, а также с коммерческой средой разработки Borland C++ Builder 6, поскольку параметры компилятора у них одинаковые.

Borland C++ 5.5 можно загрузить с web-сайта производителя по адресу “<http://www.embarcadero.com/products/cbuilder/free-compiler>”. По состоянию на январь 2013 года для этого нужно заполнить на сайте форму, указав в ней имя пользователя и адрес электронной почты, на которую будет выслана ссылка для загрузки компилятора. Получив эту ссылку и загрузив файл “freecommandlinetools.exe”, необходимо запустить этот файл и установить компилятор, следуя инструкциям. Программа установки запросит имя папки, в которую будет установлен компилятор – после выбора папки путь к ней необходимо записать или запомнить. Этот путь нужно будет потом ввести в настройках модуля автокомпиляции. Перед завершением программа установки предложит добавить в систему путь к папке исполняемых файлов компилятора, однако, для работы в РДС это не обязательно: все необходимые пути явно указываются в настройках модуля автокомпиляции, и системные файлы можно не трогать.

После того, как компилятор установлен, необходимо подключить модуль для работы с ним, если он еще не подключен, и указать в настройках модуля путь к папке установки компилятора. Для этого следует запустить РДС и открыть окно установки модулей автокомпиляции пунктом главного меню “сервис | автокомпиляция”. Если модуль уже установлен, его нужно выбрать в списке в верхней части окна (рис. 302), если же нет, нужно в этом окне нажать кнопку “+”, выбрать в выпадающем списке “библиотека” файл “\$DLL\$Common.dll” (можно ввести это имя вручную), в поле ввода “функция” ввести “BCpp55” (первые две буквы – в верхнем регистре, третья и четвертая – в нижнем), а в поле “название” – название модуля, например, “Borland C++ 5.5” (после ввода имени функции название модуля должно появиться в поле само). В результате окно примет вид, подобный изображенному на рис. 302.

Теперь можно открыть окно настройки модуля, либо дважды щелкнув на строчке этого модуля в списке, либо выбрав модуль в списке и нажав вторую сверху кнопку справа от него (к этой кнопке выводится всплывающая подсказка “настройка модуля”). В окне настройки на вкладке “компилятор” (рис. 303) следует в поле ввода “папка Borland C++” указать путь к папке установки компилятора, введя его вручную или выбрав в стандартном диалоге, вызываемом кнопкой “обзор”.

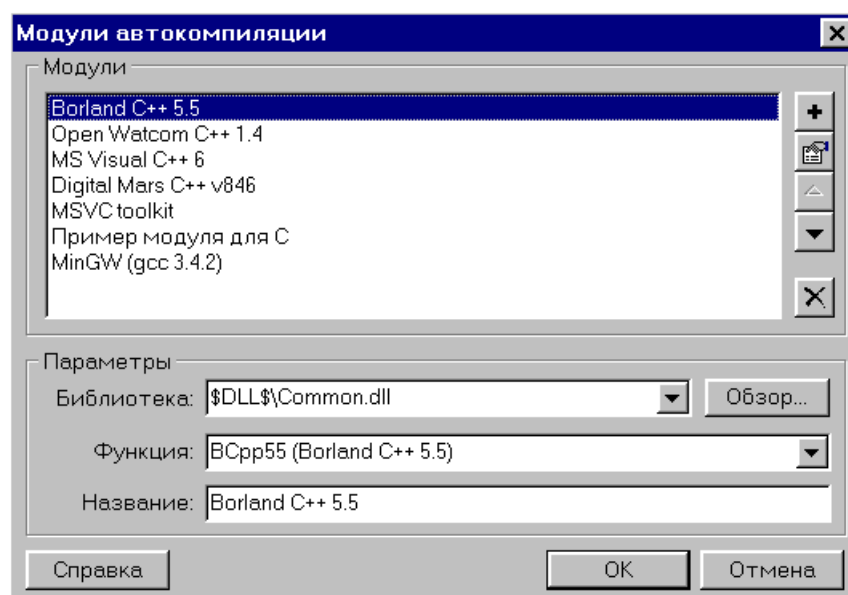


Рис. 302. Параметры модуля для Borland C++ 5.5 в окне установки модулей автокомпиляции

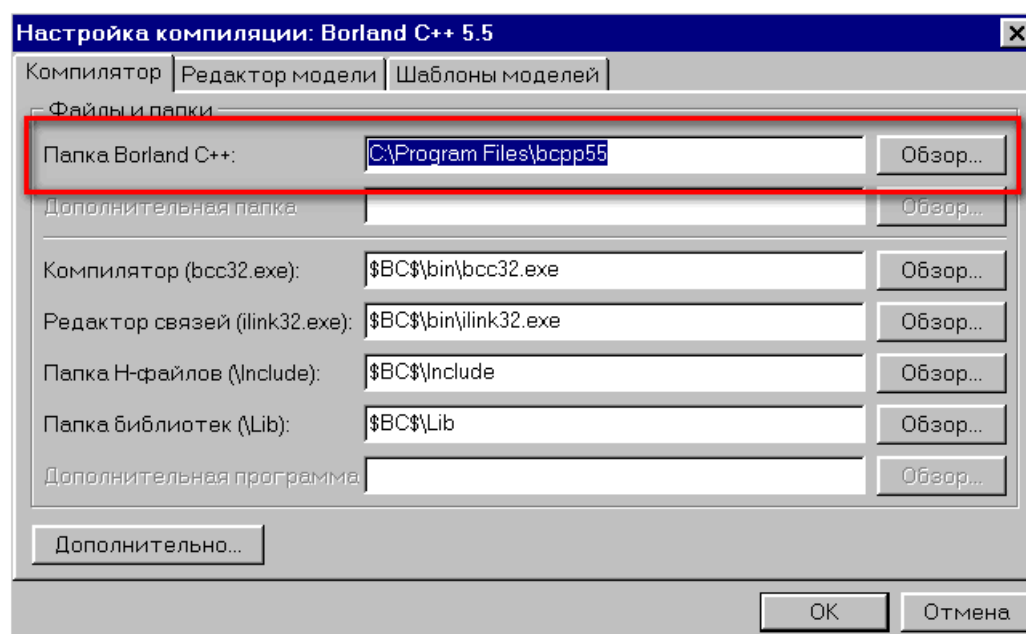


Рис. 303. Указание папки установки компилятора Borland C++ 5.5

После ввода пути к папке установки остальные поля вкладки автоматически заполняются путями по умолчанию для данного компилятора, причем все пути будут заданы относительно указанной папки установки компилятора, вместо которой будет использовано символическое обозначение "\$BC\$". Теперь следует закрыть сначала окно настройки, а затем окно установки модулей автокомпиляции кнопкой "OK" (кнопка "отмена" не сохранит введенные изменения). На этом первичная установка и настройка компилятора и модуля завершена, никаких других настроек для создания автокомпилируемых моделей не требуется. Окно настройки модуля позволяет задать множество других параметров, отвечающих за работу с компилятором и за включаемые в формируемую модель дополнительные возможности, но их можно не изменять – значения по умолчанию,

автоматически сформированные при подключении модуля, подходят для работы с данным компилятором. Настройка дополнительных параметров модуля автокомпиляции рассматривается в §3.9 (стр. 328).

Теперь при редактировании схемы можно создавать автокомпилируемые модели, привязывая их к модулю для Borland C++ 5.5.

§3.2.2. Установка и подключение компилятора Open Watcom C++

Описывается установка бесплатного компилятора Open Watcom C++ и подключение модуля автокомпиляции, который с ним работает.

Модуль автоматической компиляции “OpenWatcomCpp” из библиотеки “Common.dll” предназначен для работы с бесплатно распространяемым компилятором Open Watcom C++, который можно загрузить с сайта производителя по адресу “<http://www.openwatcom.org>”. Загрузив исполняемый файл программы установки компилятора, следует запустить его и установить компилятор, следуя инструкциям программы. Если программа предложит внести изменения в системные переменные окружения, от этого можно отказаться – для работы данного компилятора с РДС это не требуется.

После того, как компилятор установлен, необходимо подключить модуль для работы с ним, если он еще не подключен, и указать в настройках модуля путь к папке установки компилятора. Для этого следует запустить РДС и открыть окно установки модулей автокомпиляции пунктом главного меню “сервис | автокомпиляция”. Если модуль уже установлен, его нужно выбрать в списке в верхней части окна (рис. 304), если же нет, нужно в этом окне нажать кнопку “+”, выбрать в выпадающем списке “библиотека” файл “\$DLL\$Common.dll” (можно ввести это имя вручную), в поле ввода “функция” ввести “OpenWatcomCpp” (имя функции чувствительно к регистру символов), а в поле “название” – название модуля в том виде, в котором его будет видеть пользователь (после ввода имени функции название модуля должно появиться в поле само).

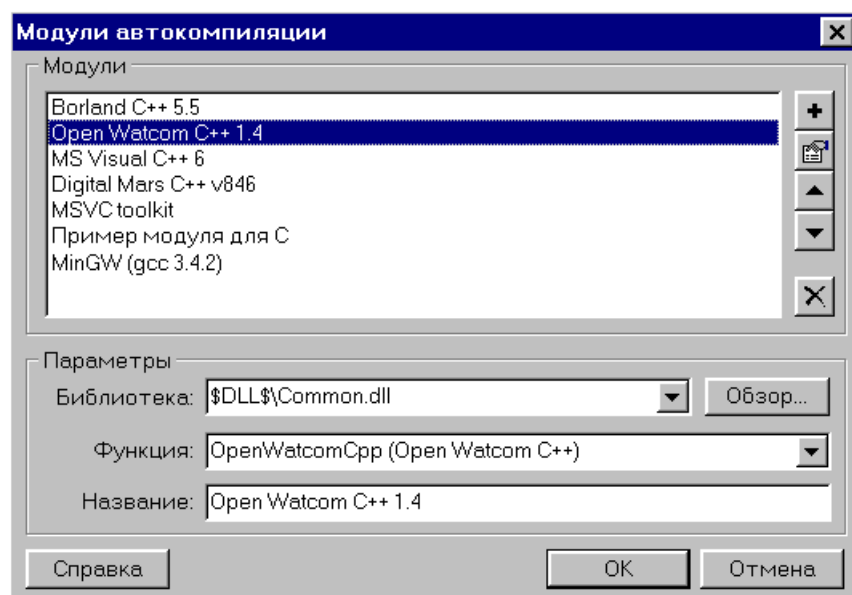


Рис. 304. Параметры модуля для Open Watcom C++ в окне установки модулей автокомпиляции

Затем необходимо открыть окно настройки модуля, либо дважды щелкнув на строчке этого модуля в списке, либо выбрав модуль в списке и нажав вторую сверху кнопку справа от него (к этой кнопке выводится всплывающая подсказка “настройка модуля”), после чего в поле ввода “папка Open Watcom C++” следует указать путь к папке установки компилятора,

введя его вручную или выбрав в стандартном диалоге, вызываемом кнопкой “обзор” (рис. 305). Все остальные поля вкладки при этом автоматически заполнятся путями по умолчанию для данного компилятора. Все пути будут заданы относительно указанной папки установки, вместо которой будет использовано символическое обозначение “\$BC\$”. Эти пути, а также все остальные параметры модуля автокомпиляции, подходят для работы с данным компилятором, и как-либо изменять их не нужно (настройка всех дополнительных параметров модуля автокомпиляции рассматривается в §3.9 на стр. 328).

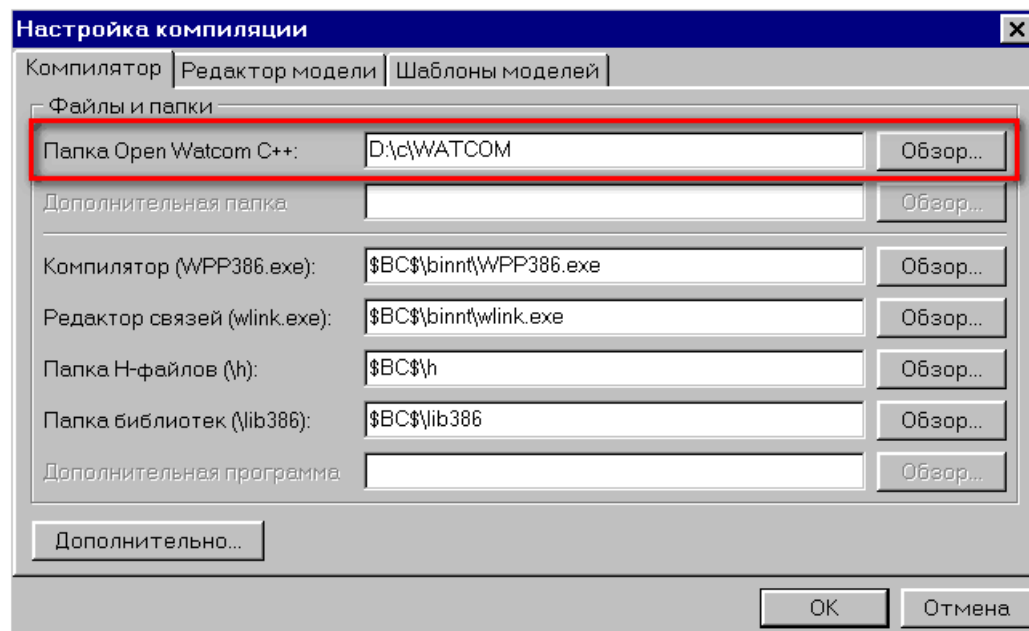


Рис. 305. Указание папки установки компилятора Open Watcom C++

Далее следует закрыть сначала окно настройки, а затем окно установки модулей автокомпиляции кнопкой “ОК” (кнопка “отмена” не сохранит введенные изменения). Теперь при редактировании схемы можно создавать автокомпилируемые модели, привязывая их к модулю для Open Watcom C++.

§3.2.3. Установка и подключение компилятора Digital Mars C++

Описывается установка бесплатного компилятора Digital Mars C++ и подключение модуля автокомпиляции, который с ним работает.

Модуль автоматической компиляции “DigitalMars” из библиотеки “Common.dll” предназначен для работы с бесплатно распространяемым компилятором Digital Mars C++, который можно загрузить с сайта производителя по адресу “<http://www.digitalmars.com>”. С сайта можно загрузить архив в формате ZIP (как правило, он называется “dmXXXc.zip”, где вместо символов XXX указан номер последней версии компилятора). Этот архив необходимо распаковать вручную в любую папку на диске – путь к этой папке необходимо запомнить или записать, поскольку его нужно будет ввести при настройке параметров модуля автокомпиляции. Никаких изменений в системные переменные окружения вносить не требуется.

После того, как архив с компилятором распакован, необходимо подключить модуль для работы с ним, если он еще не подключен, и указать в настройках модуля путь к папке компилятора. Для этого следует запустить РДС и открыть окно установки модулей автокомпиляции пунктом главного меню “сервис | автокомпиляция”. Если модуль уже установлен, его нужно выбрать в списке в верхней части окна (рис. 306), если же нет, нужно в этом окне нажать кнопку “+”, выбрать в выпадающем списке “библиотека” файл

“\$DLL\$Common.dll” (можно ввести это имя вручную), в поле ввода “функция” ввести “DigitalMars” (имя функции чувствительно к регистру символов), а в поле “название” – название модуля в том виде, в котором его будет видеть пользователь (после ввода имени функции название модуля должно появиться в поле само).

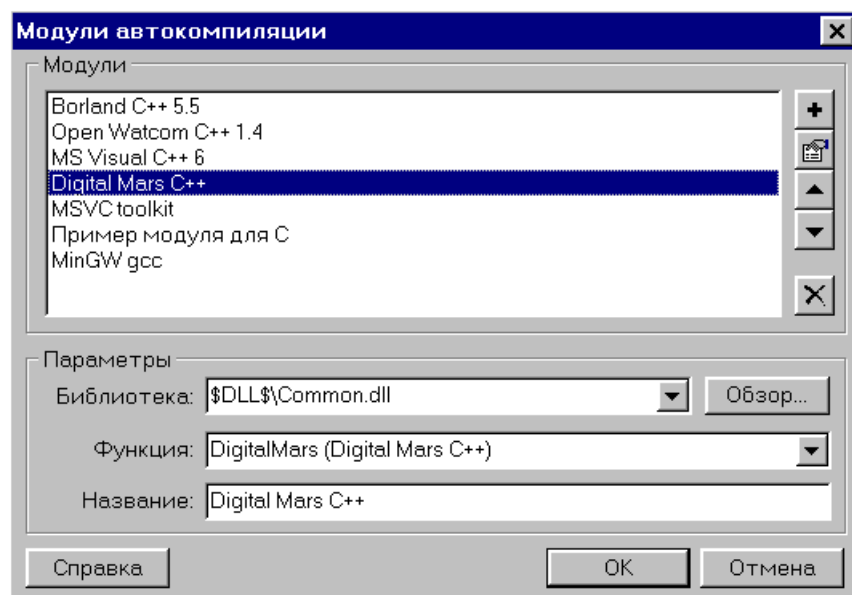


Рис. 306. Параметры модуля для Digital Mars C++ в окне установки модулей автокомпиляции

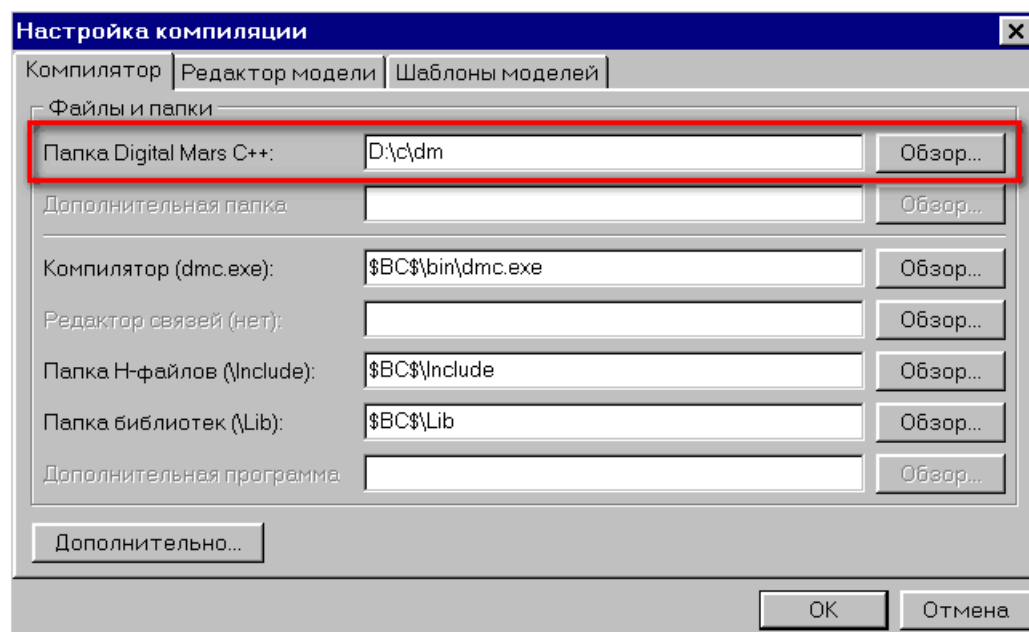


Рис. 307. Указание папки установки компилятора Digital Mars C++

Затем необходимо открыть окно настройки модуля, либо дважды щелкнув на строчке этого модуля в списке, либо выбрав модуль в списке и нажав вторую сверху кнопку справа от него (к этой кнопке выводится всплывающая подсказка “настройка модуля”), после чего в поле ввода “папка Digital Mars C++” следует указать путь к папке установки компилятора, введя его вручную или выбрав в стандартном диалоге, вызываемом кнопкой “обзор” (рис. 307). Все остальные поля вкладки при этом автоматически заполнятся путями по

умолчанию для данного компилятора, причем все пути будут заданы относительно указанной папки установки, вместо которой будет использовано символическое обозначение “\$BC\$”. Эти пути, а также все остальные параметры модуля автокомпиляции, подходят для работы с данным компилятором, и как-либо изменять их не нужно (настройка всех дополнительных параметров модуля автокомпиляции рассматривается в §3.9 на стр. 328).

Следует обратить внимание на то, что, в отличие от других описанных здесь компиляторов, у Digital Mars C++ нет отдельного редактора связей – его функции выполняет программа компилятора “dmc.exe”, поэтому путь к редактору связей в настройках этого модуля не задается, и соответствующее поле ввода остается пустым.

Теперь при редактировании схемы можно создавать автокомпилируемые модели, привязывая их к модулю для Digital Mars C++.

§3.2.4. Установка и подключение компилятора MinGW GCC

Описывается установка бесплатного компилятора MinGW GCC и подключение модуля автокомпиляции, который с ним работает.

Модуль автоматической компиляции “Gcc_MinGW” из библиотеки “Common.dll” предназначен для работы с бесплатно распространяемым компилятором GCC из набора MinGW, который можно загрузить с сайта по адресу “<http://www.mingw.org>”. С этого сайта необходимо загрузить программу установки “mingw-get-inst” и запустить ее. В ней следует выбрать установку компилятора C++ (остальные компиляторы для работы с РДС не требуются, их можно устанавливать по желанию) и указать для его установки какую-либо папку, **путь к которой не содержит пробелов**. После установки компилятора необходимо добавить в переменные окружения операционной системы путь к его папке “bin”, иначе компилятор не сможет работать. Например, если сам компилятор установлен в папку “d:\mingw”, то в системный файл “autoexec.bat” следует добавить команду “path d:\mingw\bin”.

После того, как компилятор установлен, необходимо подключить модуль для работы с ним, если он еще не подключен, и указать в настройках модуля путь к папке компилятора. Для этого следует запустить РДС и открыть окно установки модулей автокомпиляции пунктом главного меню “сервис | автокомпиляция”. Если модуль уже установлен, его нужно выбрать в списке в верхней части окна (рис. 308), если же нет, нужно в этом окне нажать кнопку “+”, выбрать в выпадающем списке “библиотека” файл “\$DLL\$Common.dll” (можно ввести это имя вручную), в поле ввода “функция” ввести “Gcc_MinGW” (имя функции чувствительно к регистру символов), а в поле “название” – название модуля в том виде, в котором его будет видеть пользователь (после ввода имени функции название модуля должно появиться в поле само).

Затем следует открыть окно настройки модуля, либо дважды щелкнув на строчке этого модуля в списке, либо выбрав модуль в списке и нажав вторую сверху кнопку справа от него (к этой кнопке выводится всплывающая подсказка “настройка модуля”), после чего в поле ввода “папка MinGW gcc” следует указать путь к папке установки компилятора, введя его вручную или выбрав в стандартном диалоге, вызываемом кнопкой “обзор” (рис. 309). При этом поля путей к компилятору и редактору связей заполнятся автоматически (в них вместо папки установки компилятора будет использовано символическое обозначение “\$BC\$”), а поля путей к файлам заголовков и библиотек останутся пустыми – в данном модуле автокомпиляции эти два пути не задаются. Другие параметры модуля изменять не требуется (настройка всех дополнительных параметров модуля автокомпиляции рассматривается в §3.9 на стр. 328).

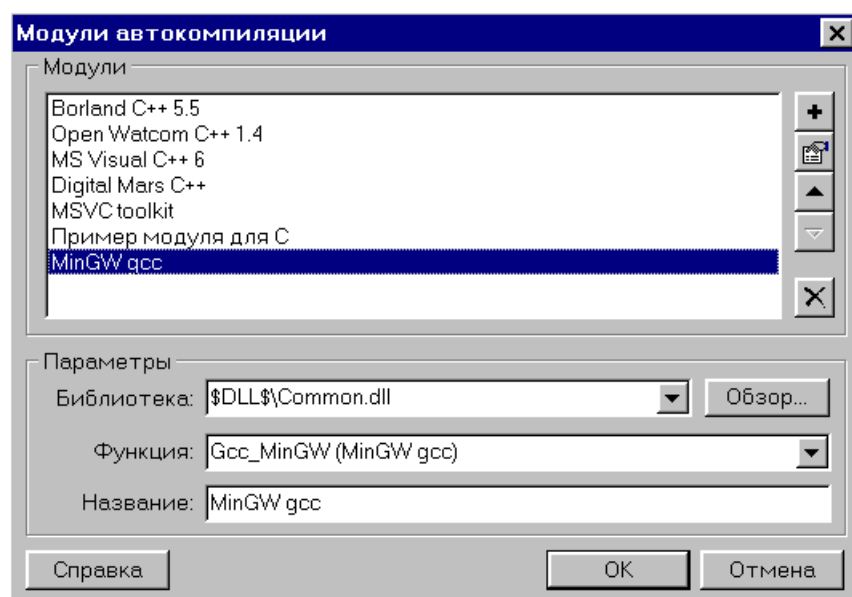


Рис. 308. Параметры модуля для MinGW GCC в окне установки модулей автокомпиляции

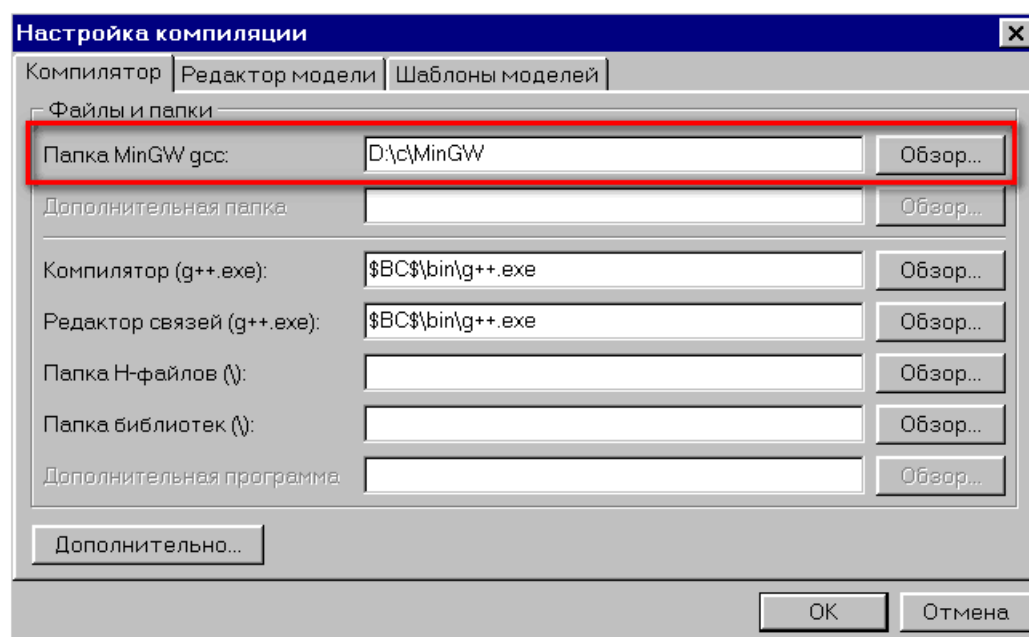


Рис. 309. Указание папки установки компилятора MinGW GCC

Теперь при редактировании схемы можно создавать автокомпилируемые модели, привязывая их к модулю для MinGW GCC.

§3.2.5. Установка и подключение компилятора Microsoft Visual C++ 2003 toolkit

Описывается установка бесплатного компилятора Microsoft Visual C++ 2003 и подключение модуля автокомпиляции, который с ним работает. Также описывается настройка модуля для работы с коммерческим компилятором Microsoft Visual C++ 6.

Модуль автоматической компиляции “MSVCTK2003” из библиотеки “Common.dll” предназначен для работы с бесплатно распространявшимся компилятором Microsoft Visual

C++ 2003. Ранее этот компилятор можно было загрузить с официального сайта Microsoft, однако, на данный момент ссылка на него на сайте отсутствует. Тем не менее, пользователи, у которых этот компилятор остался еще с тех пор, могут использовать его для работы с РДС.

Для установки Microsoft Visual C++ 2003 необходима как программа установки самого компилятора “VCToolkitSetup.exe”, так и Microsoft Platform SDK (может называться по-разному, например, “PSDK-x86.exe”). Необходимо запустить программу установки компилятора и следовать всем ее инструкциям, а затем, когда она завершится, запустить программу установки Platform SDK и выбрать в ней установку компонентов, необходимых для компиляции тридцатидвухбитных программ для платформы Windows (Win32). Все необходимые изменения в системные переменные окружения программы установки внесут самостоятельно.

После того, как компилятор и Platform SDK установлены, необходимо подключить модуль для работы с ними, если он еще не подключен, и указать в настройках модуля путь к папкам установки компилятора и Platform SDK. Для этого следует запустить РДС и открыть окно установки модулей автокомпиляции пунктом главного меню “сервис | автокомпиляция”. Если модуль уже установлен, его нужно выбрать в списке в верхней части окна (рис. 310), если же нет, нужно в этом окне нажать кнопку “+”, выбрать в выпадающем списке “библиотека” файл “\$DLL\$\Common.dll” (можно ввести это имя вручную), в поле ввода “функция” ввести “MSVCTK2003” (все символы в верхнем регистре), а в поле “название” – название модуля в том виде, в котором его будет видеть пользователь (после ввода имени функции название модуля должно появиться в поле само).

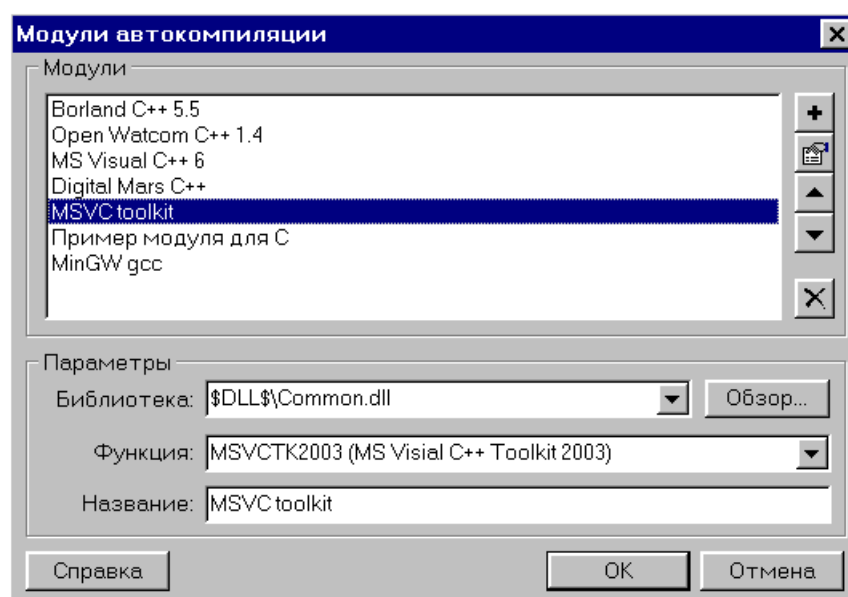


Рис. 310. Параметры модуля для Microsoft Visual C++ 2003 toolkit в окне установки модулей автокомпиляции

Затем необходимо открыть окно настройки модуля, либо дважды щелкнув на строчке этого модуля в списке, либо выбрав его в этом списке и нажав вторую сверху кнопку справа от него (к этой кнопке выводится всплывающая подсказка “настройка модуля”), после чего в поле ввода “папка MS Visual C++” следует указать путь к папке установки компилятора, введя его вручную или выбрав в стандартном диалоге, вызываемом кнопкой “обзор”, а в поле ввода “папка PSDK” точно так же указать путь к папке установки Platform SDK (рис. 311). Все остальные поля вкладки при этом автоматически заполняются путями по умолчанию для Microsoft Visual C++, и все пути будут заданы относительно указанной папки установки, вместо которой будет использовано символическое обозначение “\$BC\$”. Другие параметры

модуля изменять не требуется (настройка всех дополнительных параметров модуля автокомпиляции рассматривается в §3.9 на стр. 328). Теперь при редактировании схемы можно создавать автокомпилируемые модели, привязывая их к модулю для Microsoft Visual C++ 2003.

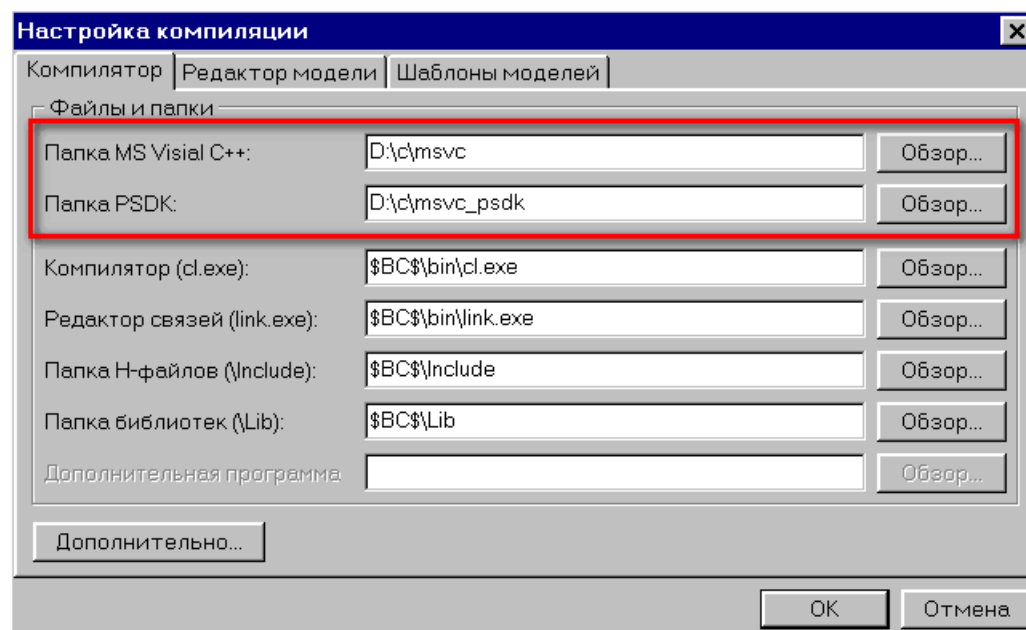


Рис. 311. Указание папки установки компилятора и PSDK для Microsoft Visual C++ 2003 toolkit

Кроме бесплатного Microsoft Visual C++ 2003 toolkit, для работы с РДС можно использовать близкий к нему по параметрам коммерческий компилятор Microsoft Visual C++ 6. Функция модуля автокомпиляции для работы с ним называется “MSVCpp6” и, как и все остальные, находится в библиотеке “Common.dll”. Настраивается этот модуль точно так же, как и модуль для Microsoft Visual C++ 2003, в его параметрах тоже необходимо задать пути к самому компилятору и к папке Platform SDK (в данном случае Platform SDK устанавливается вместе с Visual C++ 6).

§3.2.6. Подключение модуля автоматического поиска модулей

Описывается подключение дополнительного модуля автокомпиляции, который не компилирует модели сам, а обращается к одному из других, уже подключенных и настроенных, модулей.

Дополнительный модуль автоматической компиляции “SearchComp” из библиотеки “Common.dll” не предназначен для работы с каким-либо конкретным компилятором. При подключении к блоку он перебирает все остальные модули автокомпиляции из “Common.dll” и ищет среди них подключенный и настроенный, который и будет использоваться для работы с моделями и их компиляции. Например, если в РДС настроен только модуль “BCpp55” для работы с Borland C++ 5.5 (см. §3.2.1 на стр. 10), подключение к какому-либо блоку автокомпилируемой модели с указанием модуля “SearchComp” приведет к тому, что эта модель будет компилироваться именно при помощи Borland C++ 5.5.

Это может оказаться полезным в тех случаях, когда схему с автокомпилируемыми моделями необходимо передать другому пользователю. Чтобы схема сохранила работоспособность, необходимо, чтобы у другого пользователя был подключен тот же самый модуль автокомпиляции и установлен тот же самый компилятор, которые использовались при создании схемы. Можно, конечно, отключить в параметрах всех блоков автоматическую компиляцию моделей и передать схему вместе с уже скомпилированными DLL этих моделей.

Однако, если необходимо сохранить возможность автоматической компиляции, другому пользователю придется либо устанавливать у себя конкретный, использованный при создании схемы, компилятор, либо переключать в схеме все автокомпилируемые блоки на использование другого, уже установленного, модуля при помощи функции пакетной обработки, описанной в §2.15.4 части I. Чтобы не заставлять пользователя, которому передается схема, совершать эти действия, можно подключить ко всем автокомпилируемым блокам модуль “SearchComp”. При этом на каждой машине этот модуль будет вызывать для редактирования и компиляции модели тот модуль, который на ней настроен. Если на двух машинах используются разные модули автокомпиляции из библиотеки “Common.dll”, схема с подключенным модулем “SearchComp” будет работать на обеих без каких-либо дополнительных настроек.

Перебор модулей автокомпиляции в поисках настроенного выполняется в том же порядке, в котором модули перечислены в таблице на стр. 6: сначала будет проверен модуль для Borland C++, затем – для Open Watcom C++, и т.д. Модуль считается подключенным и настроенным, если в его настройках правильно указаны пути к исполняемым файлам компилятора и редактора связей (если, конечно, последний представляет собой отдельную программу), и эти исполняемые файлы существуют. Настройка путей к компилятору подробно описывается в §3.2.1 – §3.2.5 и в §3.9.3 (стр. 333). Проверяются только стандартные модули, то есть те, которые перечислены в §3.1. Модули, созданные сторонними разработчиками, могут иметь другой формат файлов моделей и, вероятнее всего, не смогут работать с моделями, созданными одним из стандартных модулей. Кроме того, дополнительный модуль находится в библиотеке “Common.dll” и знает только о стандартных модулях автокомпиляции, находящихся в той же библиотеке, поэтому и перебирает в поисках настроенного только их.

Поскольку дополнительный модуль автокомпиляции “SearchComp” не выполняет компиляцию самостоятельно, а обращается к другим модулям, собственных настроек он не имеет. Для того, чтобы его можно было использовать, достаточно просто подключить его к РДС. Если он еще не подключен, следует запустить РДС и открыть окно установки модулей автокомпиляции пунктом главного меню “сервис | автокомпиляция”. Если в этом окне среди прочих модулей нет модуля с функцией “SearchComp” и библиотекой “Common.dll”, необходимо нажать кнопку “+”, выбрать в выпадающем списке “библиотека” файл “\$DLL\$Common.dll” (можно ввести это имя вручную), в поле ввода “функция” ввести “SearchComp” (буквы “S” и “C” – в верхнем регистре), а в поле “название” – какое-либо понятное пользователю название для модуля, например, “поиск среди настроенных” (после ввода имени функции название модуля должно появиться в поле само). В результате окно примет вид, подобный изображенному на рис. 312.

При попытке настроить этот модуль (например, при двойном щелчке на названии модуля в списке) будет выдаться сообщение о том, что он только замещает один из стандартных, с указанием, какой именно из стандартных модулей будет использован для компиляции (рис. 313).

Теперь при редактировании схемы можно создавать автокомпилируемые модели, привязывая их к модулю “SearchComp” (“поиск среди настроенных”). Вместо него каждый раз будет вызываться тот из стандартных модулей, компилятор для которого установлен и настроен.

Следует иметь в виду, что при поиске настроенных модулей автокомпиляции модуль “SearchComp” игнорирует альтернативные наборы параметров модулей. Имя такого набора вводится в окне подключения модулей автокомпиляции после имени функции через дробь: например, ввод в качестве имени функции “BCpp55” подключает модуль автокомпиляции для Borland C++ с основным набором параметров, а “BCpp55/new” – тот же модуль, но с альтернативным набором параметров “new” (подробнее об этом – в §2.19.1 части I). В поиске участвуют только основные наборы параметров, поэтому если пользователь для какого-то

модуля ввел только альтернативный набор, не введя основной, такой модуль не будет считаться правильно настроенным и не будет найден.

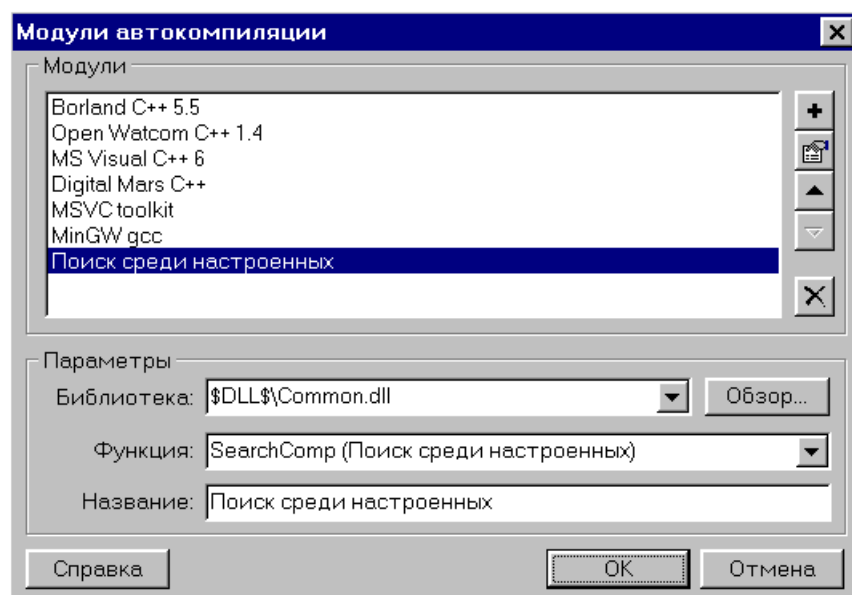


Рис. 312. Параметры дополнительного модуля поиска в окне установки модулей автокомпиляции

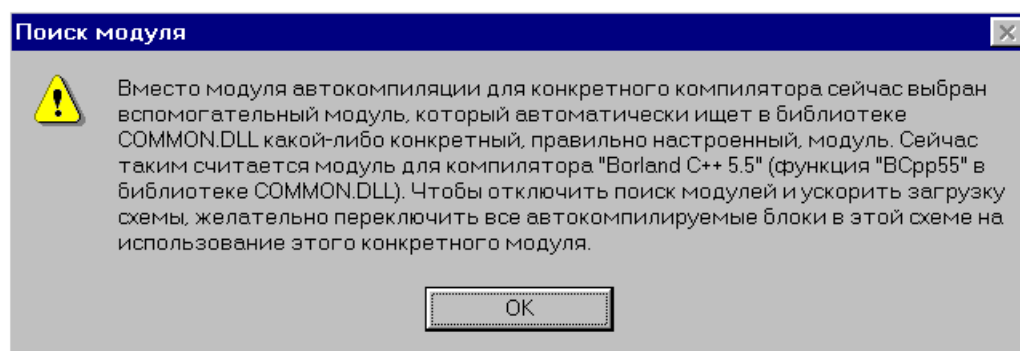


Рис. 313. Сообщение, выдаваемое при вызове настройки дополнительного модуля поиска

Следует также учитывать, что поиск настроенного модуля может немного замедлить загрузку схемы, поэтому, если схему не предполагается передавать другим пользователям, лучше использовать в ней не модуль поиска, а конкретный модуль автокомпиляции для установленного компилятора.

§3.3. Создание нового блока с автокомпилируемой моделью

Описываются действия, необходимые для создания нового блока с автокомпилируемой моделью. Поведение такого блока будет определяться программами, написанными пользователем.

Как правило, автокомпилируемые модели подключаются к новым, только что созданным, блокам, после чего настройка различных параметров такого блока производится уже не напрямую через РДС, а через редактор модели. Пользователь может, как и раньше, использовать окно параметров блока (см. §2.9.1 части I) для изменения любых параметров блока с подключенной моделью, за исключением списка статических переменных, который должен соответствовать модели и может быть задан только в ее редакторе, и подключенной к блоку функции модели, за которую теперь отвечает модуль автокомпиляции. Однако, если

одна и та же модель подключена к нескольким блокам, значительно удобнее задавать их параметры одновременно, через функцию групповой установки в редакторе модели (см. §3.6.8 на стр. 76).

Рассмотрим сначала самый распространенный случай: создание нового блока с новой автокомпилируемой моделью. Прежде всего следует создать в какой-либо подсистеме только что созданной или загруженной с диска схемы (см. §2.2 и §2.4 части I) новый простой блок и открыть окно его параметров. Действия, необходимые для этого, были подробно описаны в §2.9.1, поэтому здесь приведем их коротко. Находясь в режиме редактирования (см. §1.3), следует:

- нажать на свободном месте окна подсистемы правую кнопку мыши;
- выбрать в открывшемся контекстном меню пункт “создать | новый блок” (рис. 314 слева);
- нажать на созданном блоке (он будет выглядеть как белый квадрат с черной рамкой) правую кнопку мыши;
- выбрать в открывшемся контекстном меню пункт “параметры” (рис. 314 справа).

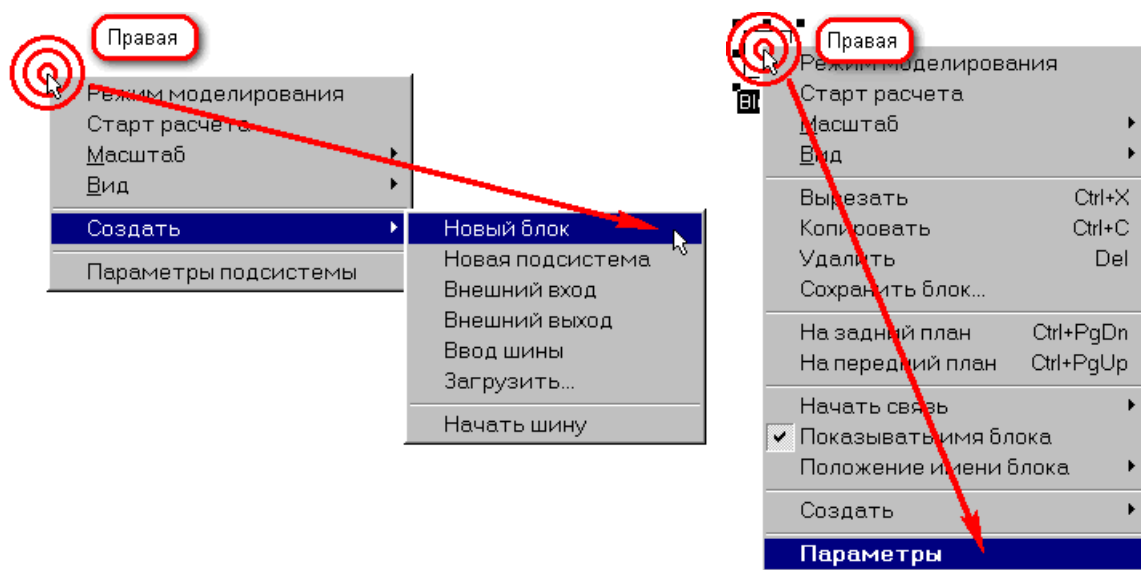


Рис. 314. Создание простого блока (слева) и вызов окна его параметров (справа)

В результате откроется окно параметров блока, в котором следует выбрать вкладку “компиляция” (рис. 315).

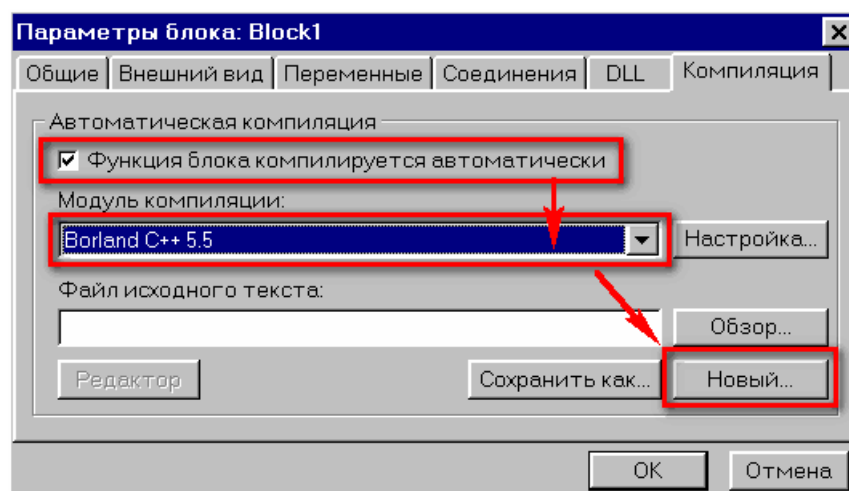


Рис. 315. Создание для блока новой автокомпилируемой модели

На этой вкладке нужно установить флажок “функция блока компилируется автоматически” и выбрать в выпадающем списке “модуль автокомпиляции” название подключенного модуля, который будет отвечать за компиляцию модели (см. §3.2 на стр. 10). Здесь и далее в примерах будет использоваться модуль для Borland C++ 5.5, но можно выбрать любой из установленных и настроенных стандартных модулей.

Поле “файл исходного текста” на вкладке пока пустое – для создания нового файла модели следует нажать кнопку “новый”. Обычно сразу после ее нажатия открывается окно “новая модель”, в котором можно создать пустую модель (рис. 316) или выбрать шаблон для создаваемой модели (рис. 317). Однако, если в РДС отсутствуют какие-либо шаблоны моделей, это окно будет пропущено и сразу откроется диалог сохранения файла (см. рис. 318 ниже).

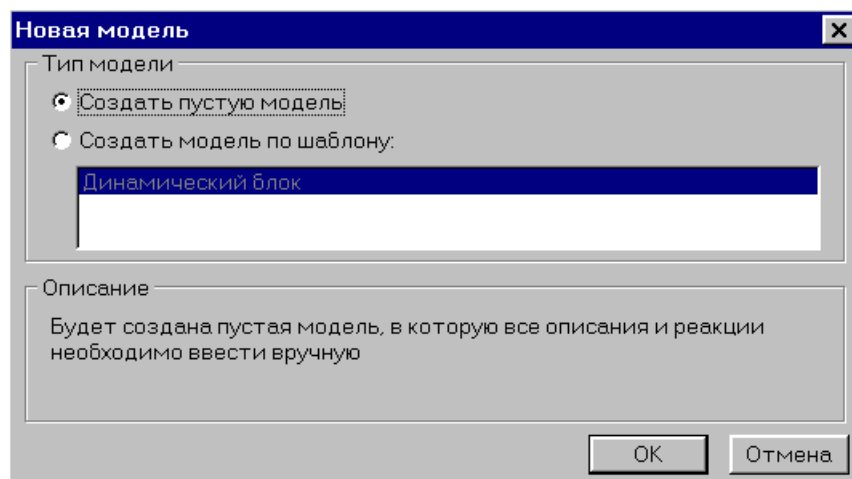


Рис. 316. Выбор создания пустой автокомпилируемой модели

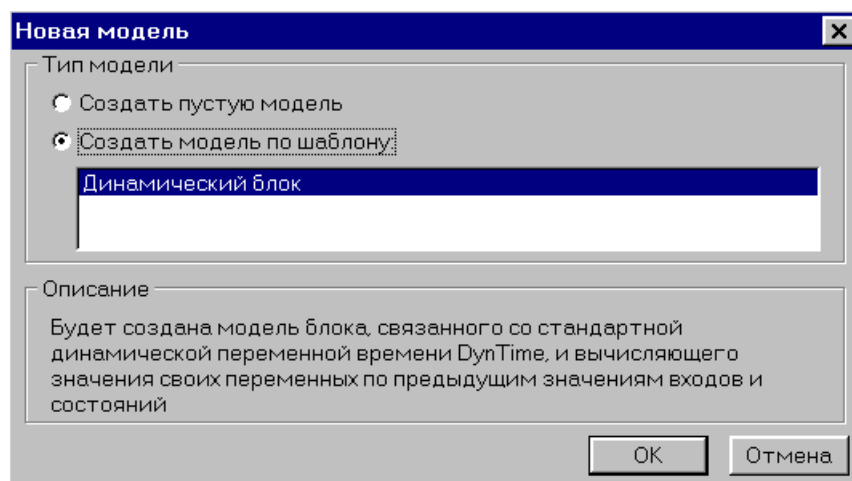


Рис. 317. Выбор создания модели по шаблону

В окне “новая модель” можно установить либо флажок “создать пустую модель” для создания модели, в которой не будет никаких реакций на события и переменных по умолчанию, либо флажок “создать модель по шаблону”. В последнем случае становится активным список имеющихся в РДС шаблонов моделей (их может создавать пользователь, см. §3.9.2 на стр. 331). При выборе шаблона в списке нижняя часть окна отображает краткое описание этого шаблона (рис. 317). Модели, созданные по шаблону, ничем принципиально не отличаются от моделей, созданных самостоятельно – в шаблоне просто уже заранее описаны некоторые реакции и переменные и добавлены пустые функции, так что

пользователю остается только заполнить их. Все это можно сделать и вручную, создав пустую модель.

Установив один из флажков в окне и, при необходимости, выбрав шаблон для создаваемой модели из списка, следует закрыть окно кнопкой “ОК”.

После закрытия окна “новая модель” (или, при отсутствии в РДС шаблонов моделей, сразу после нажатия кнопки “новый” на вкладке “компиляция” окна параметров блока, см. рис. 315) появится стандартный диалог сохранения, в котором нужно ввести имя файла для создаваемой модели и указать папку, в которую он будет записан (рис. 318). Этот файл будет связан с блоком до тех пор, пока автоматическая компиляция модели не будет отключена или пока сам блок не будет удален, поэтому к выбору размещения файла модели следует подойти внимательно.

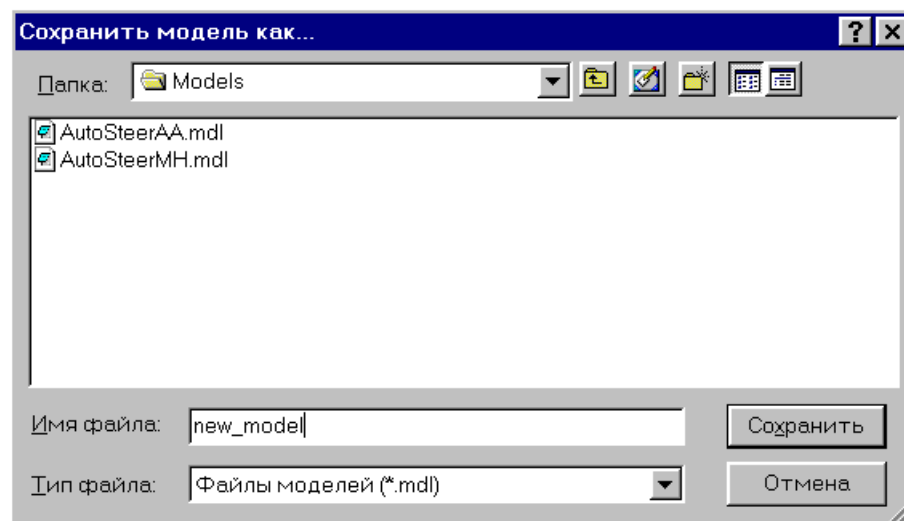


Рис. 318. Диалог сохранения файла модели

По умолчанию пользователю предлагается сохранить файл модели либо в ту же папку, в которой находится загруженная в РДС в данный момент схема, либо, если схема была создана только что и еще ни разу не сохранялась, в папку стандартных моделей, которая указывается в настройках РДС (рекомендации по размещению файлов моделей приведены на стр. 8). После нажатия в диалоге сохранения кнопки “сохранить” (см. рис. 318 выше) будет создан указанный файл модели, и его путь появится в поле ввода “файл исходного текста” (рис. 319).

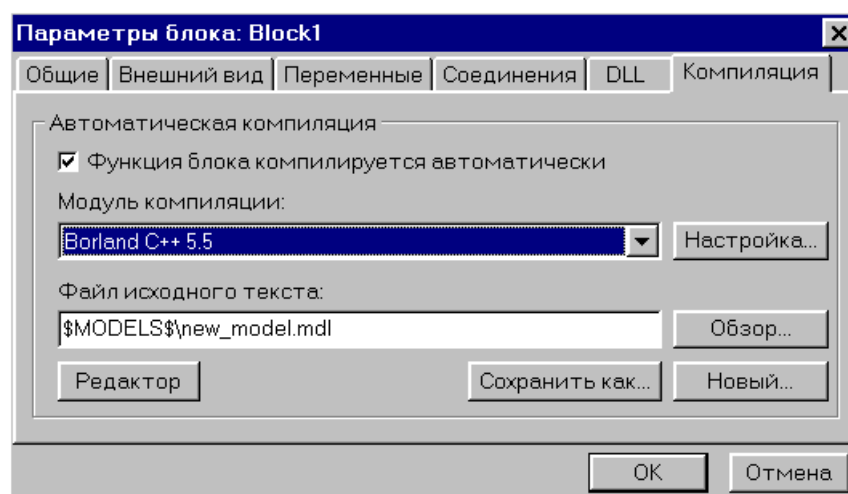


Рис. 319. Имя файла модели в окне параметров блока

Если в диалоге не было указано расширение файла, он автоматически получает стандартное для всех модулей автокомпиляции расширение “.mdl”. Если файл был сохранен в ту же папку, что и схема, имя файла появится в поле ввода без пути, если же он был сохранен в одну из стандартных папок РДС, путь к этой папке будет заменен на ее символическое обозначение (текст “\$MODELS\$” на рис. 319 указывает на стандартную папку моделей).

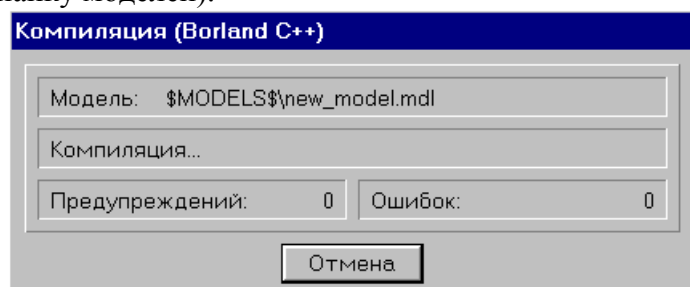


Рис. 320. Окно компиляции

Теперь можно закрыть окно параметров блока кнопкой “OK”. В этот момент модуль автокомпиляции соберет на основе только что сохраненного файла модели исполняемый файл DLL. На экране при этом на некоторое время появится окно компиляции (рис. 320) и, возможно, мелькнет черное окно консоли, в котором работает компилятор. Созданный исполняемый файл будет

иметь то же имя, что и файл модели, только с расширением “.dll”, и будет помещен в одну с ним папку. Например, для файла модели “\$MODELS\$new_model.mdl” с рис. 319 исполняемый файл DLL будет называться “new_model.dll” и находиться он будет в стандартной папке моделей РДС. Этот файл будет немедленно подключен к блоку, хотя это и не будет заметно: созданная модель – пустая, и, поэтому, никак не проявляет себя при работе РДС. Сразу после этого откроется окно редактора модели – пустое (рис. 321), если было выбрано создание пустой модели, или с частично заполненными списками переменных и несколькими вкладками для ввода реакций модели на события, если модель была создана по шаблону. Элементы и меню этого окна будут описаны в §3.6 (стр. 32).

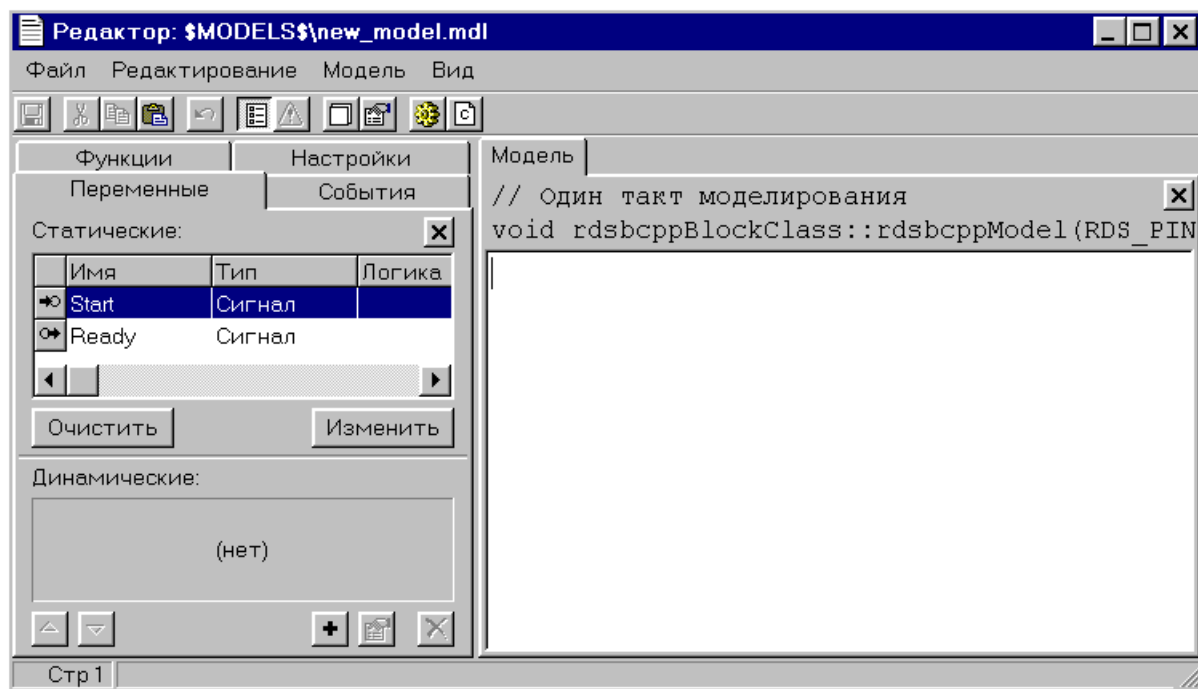


Рис. 321. Окно редактора пустой модели

Окно редактора модели – не модальное, оно может быть скрыто другими окнами РДС. Его всегда можно вызвать на передний план нажатием соответствующей ему кнопки на панели списка открытых окон или выбором его пункта в меню “окна” (см. §2.1 части I). Если окно редактора закрыто, в режиме редактирования его можно всегда снова открыть пунктом “редактор модели” контекстного меню блока. Кроме того, двойной щелчок на блоке с

автокомпилируемой моделью по умолчанию открывает окно редактора, а не окно параметров, как у всех прочих блоков.

В качестве простейшего примера, не вдаваясь пока в подробности организации моделей и их реакции на системные события, создадим модель блока-сумматора, который будет выдавать на вещественный выход “y” сумму вещественных входов “x1” и “x2”. Для этого в окне редактора, изображенном на рис. 321, нужно задать список переменных блока и заполнить текст на вкладке “модель” в правой части окна.

Список статических переменных блока, к которым относятся его входы и выходы, находится в верхней части вкладки “переменные” на левой панели окна. В данный момент там содержатся только сигналы “Start” и “Ready”, обязательные для каждого простого блока в РДС (см. §1.4). Нам нужно добавить к ним три новых переменных – два входа и один выход. Для этого нужно нажать кнопку “изменить” под списком (рис. 322), которая откроет стандартное окно редактирования переменных блока, описанное ранее в §2.9.2. Это окно нужно заполнить согласно рис. 323.

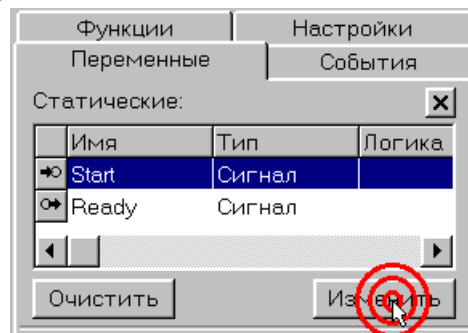


Рис. 322. Вызов редактора статических переменных блока

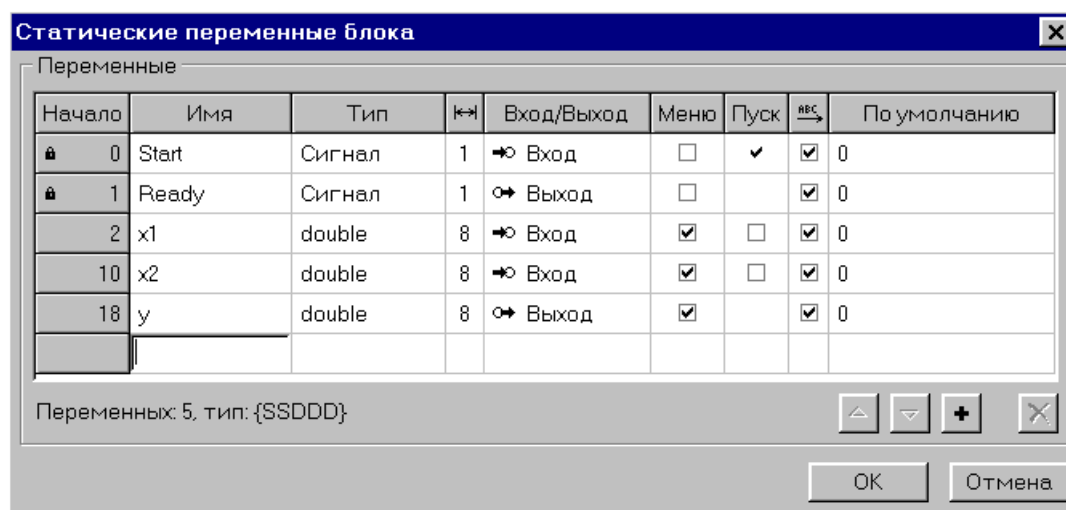


Рис. 323. Окно редактора переменных для простейшего сумматора

У переменных “Start” и “Ready” следует убрать флажки в колонке “меню” (мы не будем использовать эти переменные, и они не нужны нам в меню подключения связей). Нужно также добавить в список три переменных вещественного типа “double”, две из которых, “x1” и “x2”, будут входами, а третья, “y” – выходом. Флажки в колонке “пуск” можно не устанавливать: поскольку при создании у всех простых блоков (а, значит, и у нашего) автоматически устанавливается флажок “запуск каждый такт” (см. §2.9.1), наша модель будет запускаться в каждом такте расчета, и нам не обязательно привязывать ее запуск к срабатыванию входных связей. Это не очень хорошо для быстродействия схемы – наша модель будет работать “вхолостую” даже тогда, когда входы не изменились – но для простейшего примера это подходит.

Заполнив окно редактора переменных, нужно закрыть его кнопкой “ОК”, после чего ввести в текстовое поле на вкладке “модель” в правой части окна редактора фрагмент программы на языке C, который будет выполнять суммирование. В данном случае это будет оператор присваивания “y=x1+x2;” (рис. 324). Он должен заканчиваться точкой с запятой – это требование синтаксиса языка C. На этом создание модели завершается – можно

сохранить ее, выбрав в окне редактора пункт “файл | сохранить” (окно редактора имеет собственную строку меню, отличающуюся от меню РДС, см. §3.6.1 на стр. 32).

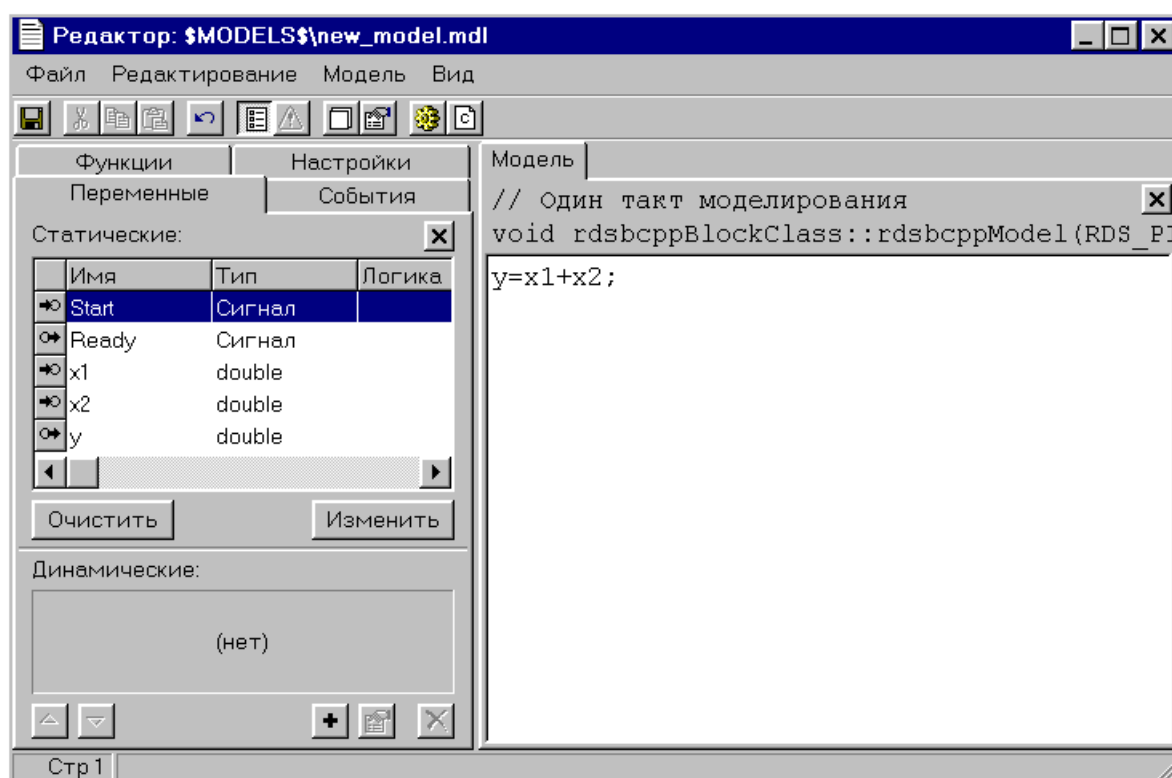


Рис. 324. Окно редактора модели простейшего сумматора

Если теперь переключиться в окно подсистемы, в которой находится созданный блок (не важно – закрыв окно редактора модели или просто щелкнув по окну подсистемы, оставив редактор на заднем плане), и попытаться присоединить что-нибудь к его входам или выходу, обнаружится, что добавленные нами в редакторе модели переменные не появились в блоке, и у него нет ни входов “x1” и “x2”, ни выхода “y”. Дело в том, что все изменения, сделанные в редакторе модели, переписываются в блоки только после компиляции этой модели. Поэтому измененную модель нужно сначала скомпилировать – к тому же, это проверит ее на ошибки. Скомпилировать модель можно несколькими способами: можно выбрать пункт главного меню РДС “система | компилировать модели” или нажать Ctrl+F9, можно просто переключиться в режим моделирования и обратно в режим редактирования, но лучше всего снова открыть редактор модели двойным щелчком на блоке и выбрать в его меню пункт “модель | компилировать” или нажать кнопку с желтой шестеренкой. Так мы скомпилируем только нашу модель и сразу увидим в нижней части окна список ошибок компиляции, если они есть.

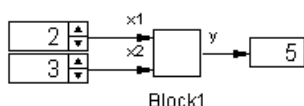


Рис. 325. Тестирование сумматора

Если строчка оператора присваивания была введена верно, компилятор установлен и модуль автокомпиляции настроен на него (см. §3.2 на стр. 10), ошибок не будет. Теперь можно закрыть окно редактора модели, подключить к входам блока поля ввода, а к выходу – индикатор (см. §2.7), затем запустить расчет и убедиться, что блок выполняет свою функцию – значение на его выходе должно быть равно сумме значений на входах (рис. 325).

Мы только что создали простейшую автокомпилируемую модель алгебраического блока. Стандартные модули автокомпиляции позволяют создавать и более сложные модели, реагирующие на изменение динамических переменных, выполняющие заданные действия по таймеру, имеющие окна настройки и т.д. В реакциях на события, вводимых в редакторе

модели, можно использовать не только все операторы языков C/C++, но и функции стандартных библиотек и специализированные функции РДС, подробно описанные в приложении к руководству программиста. Конечно, создание модели блока при помощи модуля автокомпиляции не дает той же гибкости и того же богатства возможностей, что и написание программы модели в полноценной среде разработки со встроенным отладчиком, однако, для большей части моделей, необходимых пользователю, возможностей редактора модели вполне хватает. Далее, в §3.6 (стр. 32) будет подробно рассмотрена работа с этим редактором и добавление в модель реакций на самые важные системные события. Пользователям, желающим полностью использовать возможности модуля автокомпиляции, рекомендуется внимательно прочесть руководство программиста [1] – описанные в нем функции и приемы программирования применимы и к автокомпилируемым моделям.

§3.4. Окно параметров блока с автокомпилируемой моделью

Рассматривается вкладка “компиляция” окна параметров блока и ее функции. На этой вкладке к блоку можно подключить новую модель или переключить его на работу с другой, ранее созданной, моделью.

В §3.3 выше было описано создание автокомпилируемой модели блока через кнопку “новый” на вкладке “компиляция” окна параметров этого блока (см. рис. 319 на стр. 23). Кроме этой кнопки, на вкладке присутствует несколько других, вызывающих модуль автокомпиляции для различных действий с моделью, а также поле ввода “файл исходного текста”, в которое можно вводить имя файла вручную.

Кнопка “обзор” позволяет подключить к блоку один из уже существующих файлов модели. Ее нажатие вызывает диалог открытия файла, в котором следует выбрать файл модели, который будет подключен к этому блоку. После выбора файла его имя появляется в поле ввода “файл исходного текста”. Как и при создании новой модели, если ее файл находится в той же папке, что и схема, в поле ввода появится имя файла без пути. Если же файл находится в одной из стандартных папок РДС, путь к этой папке будет заменен на ее символическое обозначение (“\$MODELSS\$”, “\$INIS\$” и т.п., полный список символических обозначений стандартных папок приведен в §A.5.4.9 приложения к руководству программиста [2]). Таким образом, чтобы подключить к блоку уже существующую модель, нужно:

- открыть окно параметров блока;
- включить на вкладке “компиляция” флажок “функция блока компилируется автоматически”;
- выбрать в выпадающем списке “модуль автокомпиляции” название модуля, который будет отвечать за компиляцию модели (см. §3.2 на стр. 10);
- нажать кнопку “обзор” и выбрать подключаемый файл модели (имя выбранного файла появится в поле ввода);
- закрыть окно параметров кнопкой “ОК”.

Файл модели будет подключен к блоку в момент закрытия окна параметров кнопкой “ОК”, при этом прежний список статических переменных блока будет заменен на новый, загруженный из файла модели. Сразу после подключения автоматически откроется окно редактора модели. Следует помнить, что если один и тот же файл модели подключен к нескольким блокам, изменения в модели (текста ее программы и прочих параметров, например, списка переменных) будут распространяться на все блоки с этой моделью – как в текущей схеме, так и во всех остальных. В текущей загруженной схеме изменения в модели записываются в блоки, связанные с ней, в момент компиляции, в других схемах – в момент загрузки схемы. Часто пользователь хочет, взяв какую-либо уже существующую модель за основу, создать новую, похожую на нее модель. При этом он должен понимать, что если он создаст новый блок, подключит его к старой модели и начнет ее редактировать, внесенные изменения повлияют не только на новый блок, но и на старый. Если необходимо создать

копию модели для ее дальнейшего изменения, проще всего скопировать блок со старой моделью в буфер обмена и вставить его копию в ту же самую схему: при этом модуль автокомпиляции спросит у пользователя, нужно ли копировать модель и как назвать созданную копию (см. стр. 28).

Вместо выбора существующего файла модели кнопкой “обзор” или создания нового кнопкой “новый”, можно вручную ввести какое-либо имя файла модели в поле ввода “файл исходного текста” (разумеется, перед этим нужно установить флажок автоматической компиляции и выбрать модуль). Если после этого закрыть окно параметров кнопкой “ОК”, к блоку будет либо подключен указанный файл модели, если он существует, либо будет создан и подключен новый пустой файл модели, если файла с таким именем нет.

Кнопка “сохранить как”, активная только при уже подключенной к блоку автокомпилируемой модели, создает копию файла этой модели (имя запрашивается у пользователя) и подключает эту копию к блоку. Прежний файл модели при этом остается подключенным ко всем блокам, кроме данного. Так можно отделить один блок от группы блоков, использующих одну и ту же модель, и вносить в его новую модель изменения, не затрагивающие остальные блоки. Действие этой кнопки отличается от действия пункта меню “файл | сохранить как” в редакторе модели (см. стр. 34) – последний подключает к новой модели не один конкретный блок, а все блоки загруженной схемы, использовавшие прежнюю модель.

Кнопка “редактор” тоже активна только при наличии подключенной модели, она открывает окно редактора модели, оставляя модальное окно параметров блока на переднем плане. Поскольку при подключении новой модели к блоку окно редактора открывается автоматически, а при редактировании схемы его можно вызвать из контекстного меню блока (или, если у блока нет окна настройки, просто двойным щелчком), эта кнопка используется редко.

Следует учитывать, что подключение к блоку автокомпилируемой модели блокирует в окне его параметров кнопку “изменить” на вкладке “переменные”, которая вызывает редактор переменных блока, а также поля ввода имени файла библиотеки DLL и функции в ней на вкладке “DLL”. Редактирование переменных для таких блоков производится через редактор модели одновременно для всех блоков с данной моделью (см. §3.6.2 на стр. 38), а подключением DLL к блоку занимается модуль автокомпиляции, а не пользователь. Если отключить на вкладке “компиляция” флажок “функция блока компилируется автоматически”, эти поля и кнопки снова станут активными, но блок останется привязанным к скомпилированному при последнем изменении модели файлу DLL.

§3.5. Копирование блоков и схем с автокомпилируемыми моделями и совместное использование моделей

Описываются особенности копирования блоков с автокомпилируемыми моделями в пределах одной схемы и между схемами. Рассматривается совместное использование одной модели несколькими блоками и схемами, а также связанные с этим возможные проблемы и способы их решения.

Хранение автокомпилируемых моделей блоков не внутри файла схемы, а в отдельных файлах дает возможность использовать одну и ту же модель в разных блоках разных схем. В файле схемы запоминается не сам текст модели, а только имя файла модели и путь к нему (подробнее об этом см. на стр. 8). При этом необходимо следить за тем, чтобы у РДС был доступ к файлам моделей: произвольное их перемещение и переименование, а также перемещение самого файла схемы может привести к неработоспособности этой схемы. Если, например, файл схемы был перемещен в другую папку средствами файлового менеджера Windows, при загрузке этой схемы РДС может не найти связанные с ней файлы моделей. Эту ошибку легко исправить, вручную скопировав файлы моделей в новую папку или изменив пути к ним в окнах параметров всех затронутых блоков (см. рис. 319 на стр. 23, список всех

используемых схемой автокомпилируемых моделей можно узнать в окне информации о схеме, описанном в §2.17 части I). Кроме того, в некоторых случаях РДС может автоматически скопировать файл модели в новую папку – об этом пользователю выдаются соответствующие сообщения, в которых он может согласиться на такое копирование или отказаться от него. Ниже эти сообщения будут рассмотрены подробно.

Если автокомпилируемая модель находится в одной папке с использующей ее схемой, в файле схемы запоминается только имя файла модели без пути – это позволяет перемещать всю папку со схемой и ее моделями без потери работоспособности. Однако, полный путь к файлу модели тоже запоминается скрыто от пользователя (для этого используется так называемое “альтернативное” имя модели, к которому имеют доступ только программисты, создающие модули автокомпиляции – этот параметр описан в §4.1 руководства программиста [1]). Запомненный полный путь в некоторых случаях позволяет модулю автокомпиляции находить “потерянные” модели. Если, например, переместить или скопировать файл схемы, в одной папке с которой находились файлы ее автокомпилируемых моделей, а сами файлы моделей не скопировать, при загрузке схемы модулю не удастся найти эти модели в новой папке. Потерпев неудачу, он попытается найти их по запомненному для каждой модели полному пути. Если файлы модели все еще находятся на старом месте, пользователю будет выдано сообщение, в котором ему будет предложено либо привязать схему к старым моделям, либо сделать для них копию в новой папке (рис. 326).

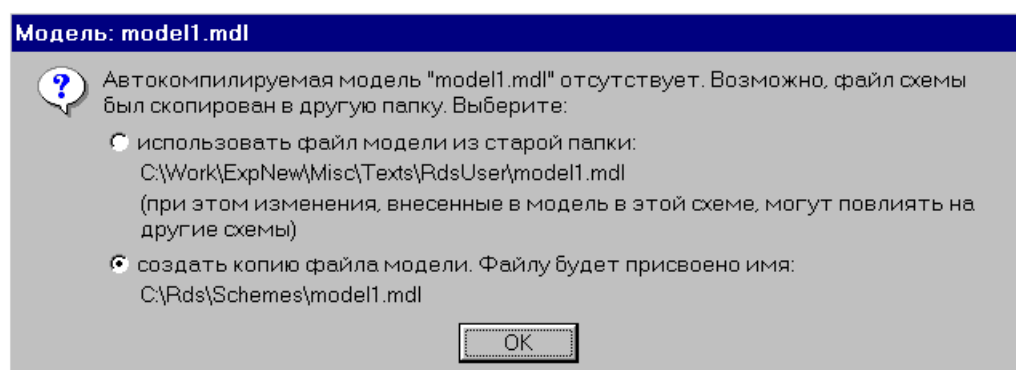


Рис. 326. Запрос, выдающийся пользователю при загрузке схемы, не нашедшей свой файл модели

Пользователь может установить один из двух взаимоисключающих флажков:

- “Использовать файл модели из старой папки”. При установке этого флажка в схеме будет запомнен полный путь к старому файлу модели, оставшемуся на прежнем месте. Схема будет оставаться привязанной к этому файлу при всех ее последующих перемещениях. Если в этот файл будут внесены изменения при работе с какой-либо другой схемой (например, со старой копией этой схемы), эти изменения распространятся и на данную схему. Выбирать этот вариант нужно с осторожностью, привязка схемы к файлу модели в нестандартной папке по абсолютному пути обычно не рекомендуется: с большой вероятностью такой файл модели находится в папке с другой схемой, и, если кто-нибудь переместит эту папку целиком вместе со всеми ее файлами, данная схема уже не сможет найти модель (при этом другая, перемещенная, схема сохранит работоспособность).
- “Создать копию файла модели”. При установке этого флажка файл модели из старой папки будет скопирован в папку, в которой находится данная схема – фактически, выполняется то, что должен был сделать пользователь, чтобы копируемая или перемещаемая в новую папку схема сохранила работоспособность. Выбор этого варианта наиболее предпочтителен: раз файл модели находится в нестандартной папке, он, вероятнее всего, не универсален, а относится к какой-то конкретной схеме, находящейся в этой же папке. Скопировав эту схему в другую папку, нужно скопировать и относящиеся к

ней файлы моделей, при этом старая и новая схемы будут работать независимо, и изменения их моделей не будут влиять друг на друга. Новое имя, которое файл модели получит после копирования, отображается под флажком. Путь к файлу будет совпадать с путем к схеме, с которой он будет связан (он копируется в ее папку), а имя, по возможности, останется прежним. В параметрах блока имя файла модели будет запомнено без пути, поскольку он находится в одной папке со схемой. После копирования файла модели папку схемы со всеми внутренними файлами можно будет перемещать без потери работоспособности.

При копировании блока с автокомпилируемой моделью через буфер обмена также возможно несколько вариантов: пользователь должен решить, будет ли копия блока связана с тем же самым файлом модели (у двух блоков будет общая модель), или необходимо сделать для нее копию модели, чтобы модели исходного и скопированного блока можно было изменять независимо. Об этом пользователю выдается соответствующий запрос (рис. 327).

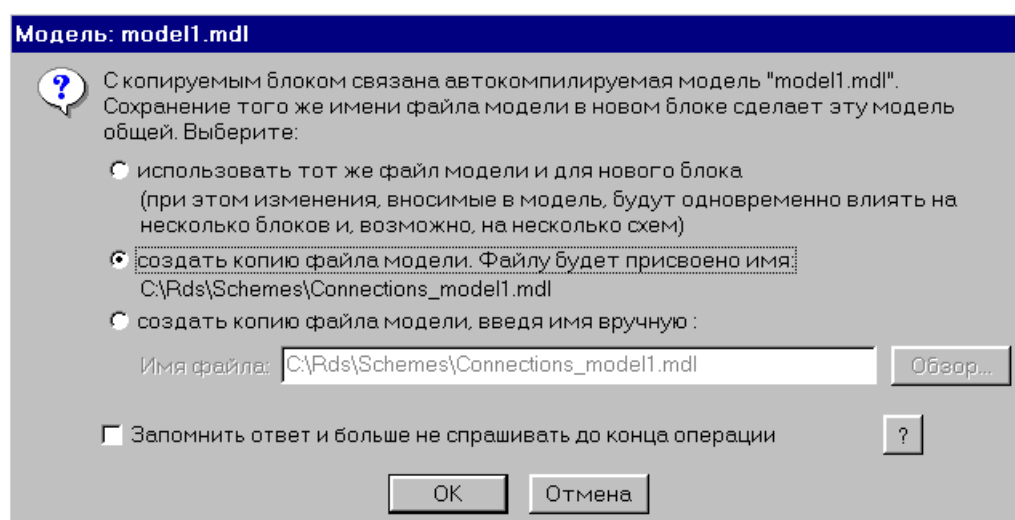


Рис. 327. Запрос, выдающийся пользователю при вставке блока с автокомпилируемой моделью из буфера обмена

Пользователь должен установить один из трех взаимоисключающих флажков:

- “Использовать тот же файл модели для нового блока”. При выборе этого флажка копия блока будет связана с тем же файлом модели, что и исходный блок. Модель станет общей для этих двух блоков, и изменения в ней будут влиять на оба сразу.
- “Создать копию файла модели”. При установке этого флажка вместе с копией блока будет сделана копия файла модели – старый блок останется привязанным к старому файлу, новый будет привязан к этой копии. Модели блоков при этом можно будет изменять независимо, они не будут влиять друг на друга. Модель будет скопирована в папку данной схемы. Имя для копии файла модели выбирается автоматически, оно выводится под флажком. Модуль автокомпиляции пытается сохранить имя файла модели при копировании, если же это невозможно (например, копия будет помещена в ту же папку, что и оригинал, и не может иметь с ним одинаковое имя), к старому имени модели спереди добавляется имя схемы, а сзади, возможно, число для обеспечения уникальности имени.
- “Создать копию файла модели, введя имя вручную”. Этот флажок действует аналогично предыдущему, за исключением того, что имя файла для копии модели пользователь может выбрать вручную, введя его в поле ввода под флажком или нажав кнопку “обзор”.

В нижней части окна запроса находится дополнительный флажок “запомнить ответ”, при установке которого выбранный пользователем вариант будет запомнен и, в дальнейшем, будет автоматически применяться ко всем копируемым блокам без выдачи запроса.

Запомнены могут быть только первый и второй варианты – третий требует ввода имени копии модели вручную, поэтому не может быть применен автоматически без запроса пользователю. Выбранный вариант запоминается только до конца текущей операции, т. е. вставки группы блоков из буфера обмена.

Если пользователь закроет окно запроса не кнопкой “ОК”, а кнопкой “Отмена”, то, независимо от установленного флажка, копирование блока из буфера обмена будет отменено.

Следует учитывать, что при некоторых настройках совместного использования модели (см. стр. 75) запрос при копировании выдаваться не будет, и файл модели будет копироваться автоматически.

Похожий запрос выдается пользователю в тех случаях, когда он сохраняет схему в другой файл при помощи пункта главного меню РДС “файл | сохранить как”, если эта схема использует автокомпилируемые модели, находящиеся в одной с ней папке. Пользователь может либо оставить схему связанной со старыми файлами моделей, либо сделать их копии (рис. 328).

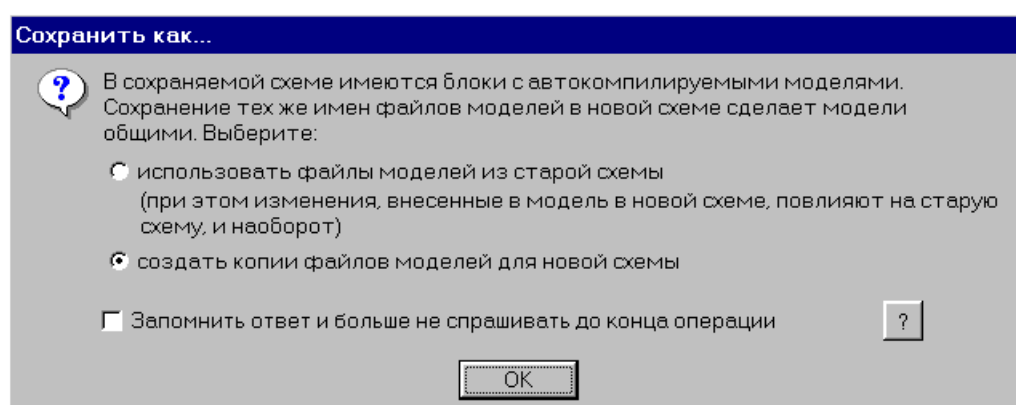


Рис. 328. Запрос, выдающийся пользователю при сохранении схемы с автокомпилируемыми моделями под другим именем

Пользователь должен установить один из двух взаимоисключающих флажков:

- “Использовать файлы моделей из старой схемы” – схема остается связанной со старыми файлами моделей. Этот вариант рекомендуется использовать только в том случае, если схема сохраняется под другим именем в ту же папку, что и старая. В этом случае в обеих схемах ссылки на файлы моделей будут запомнены без путей, и всю папку с этими схемами и их моделями можно будет перемещать с места на место без потери работоспособности. Модели у обеих схем будут общими, и изменения в них будут влиять сразу на обе схемы.
- “Создать копии файлов моделей” – в папке новой схемы создаются копии всех файлов моделей, связанных со старой и находившихся в одной с ней папке. Имена этим копиям присваиваются автоматически, пользователь не может ввести их вручную (тем не менее, он может позже открыть окна параметров блоков новой схемы и изменить там имена файлов моделей вручную кнопкой “сохранить как”, см. §3.4 на стр. 27). Выбор этого варианта рекомендуется при сохранении копии схемы в другую папку, чтобы скопированная схема стала полностью независимой от старой и не имела с ней общих моделей.

Дополнительный флажок “запомнить ответ” запоминает выбранный пользователем вариант и автоматически применяет его до следующего сохранения схемы.

При работе с моделями, которые используются в нескольких схемах одновременно, следует помнить, что для компиляции измененной модели модулю необходим монопольный доступ к файлу DLL, создаваемому в результате компиляции – в противном случае он не

сможет заменить этот файл на новый. Такое может случиться, если две схемы, использующие одну и ту же модель, открыты в двух одновременно запущенных копиях РДС. Файл DLL при этом будет загружен в память обоих приложений и заблокирован от изменений, что приведет к невозможности компиляции модели. Решить эту проблему можно временно закрыв все одновременно работающие копии РДС кроме той, из которой вызван редактор модели. После внесения в модель изменений и успешной компиляции можно будет снова открыть схемы в нескольких копиях РДС.

§3.6. Окно редактора модели

Рассматривается окно редактора модели и действия, которые в нем можно выполнить.

§3.6.1. Элементы и меню окна редактора модели

Описываются меню, кнопки и панели окна редактора модели. Перечисляются все пункты меню редактора с их краткими описаниями.

Окно редактора модели (рис. 329) – не модальное, то есть, будучи открытым, оно не блокирует работу других окон РДС. Пользователь может свободно переключаться между несколькими открытыми окнами редактора, окнами подсистем и другими окнами РДС, щелкая по ним мышью, выбирая их названия в пункте главного меню “окна”, или нажимая соответствующие этим окнам кнопки на панели открытых окон (см. §2.1 части I).

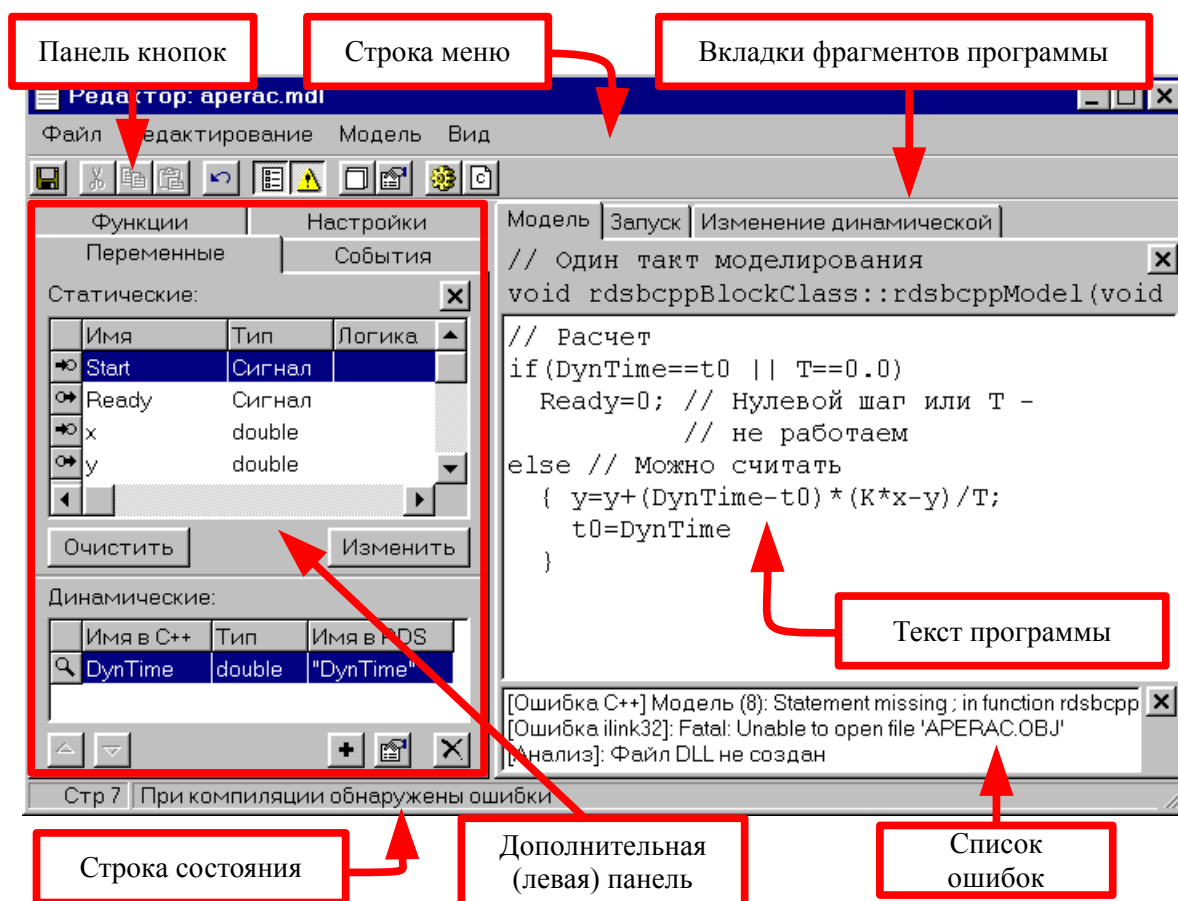


Рис. 329. Элементы окна редактора модели

В отличие от большинства остальных окон РДС, окно редактора модели имеет свою собственную строку меню, пункты которого относятся к конкретному файлу модели, открытому в данном окне редактора. Часть функций меню продублирована кнопками на

панели, расположенной непосредственно под этим меню. В нижней части окна редактора находится строка состояния, в которой отображается положение текстового курсора на вкладке фрагмента программы и сообщение о результатах последней компиляции.

Основная площадь окна, как правило, разделена по вертикали на две неравные части. Слева находится отключаемая дополнительная панель, с четырьмя вкладками, на которых задаются:

- статические и динамические переменные блока (вкладка “переменные”, см. §3.6.2 на стр. 38 и §3.6.3 на стр. 42);
- события, на которые реагирует модель (вкладка “события”, см. §3.6.4 на стр. 46);
- функции, которые вызывает или на вызов которых отвечает блок (вкладка “функции”, см. §3.6.5 на стр. 57);
- параметры блока, сохраняемые вместе с ним в схеме, и внешний вид окна настройки этих параметров (вкладка “настройки”, см. §3.6.6 на стр. 60).

Ширину панели можно изменять, перетаскивая разделитель, находящийся справа от нее, влево и вправо левой кнопкой мыши. Вкладки этой панели будут подробно рассмотрены ниже в соответствующих параграфах.

Справа находится панель с вкладками, каждая из которых соответствует тому или иному фрагменту программы модели: реакции на событие, реакции на вызов функции, добавляемому в программу описанию и т.п. Эти вкладки можно открывать и закрывать по желанию пользователя: открываются они при помощи вкладки “события” дополнительной панели (см. стр. 46), а закрываются кнопкой с крестиком в правой верхней части самой вкладки. На каждой вкладке находится большое многострочное поле ввода, в которое пользователь записывает фрагмент программы, выполняющийся при наступлении соответствующего события. Например, на рис. 329 открыта вкладка “модель”, соответствующая циклическому вызову блока в режиме расчета (это самая часто используемая реакция блока, определяющая его поведение в схеме), и на ней введен текст программы, которая будет вызываться в каждом такте. В верхней части вкладки находится краткий комментарий, описывающий назначение события, которому эта вкладка соответствует. Комментарий записан в формате языка C и, в некоторых случаях, под ним указывается заголовок функции, которую модуль автокомпиляции создаст для данного события. Например, на вкладке “модель” на рис. 329 функция будет называться “void rdsbcppBlockClass::rdsbcppModel(void)” (то есть функция “rdsbcppModel” класса “rdsbcppBlockClass”, не принимающая параметров и не возвращающая значений). Текст заголовка функции можно выделить и скопировать в буфер обмена. Все эти описания полезны для программистов, создающих сложные модели – например, зная, какие параметры принимает функция, можно использовать их внутри текста фрагмента программы. Все функции, создаваемые для всех возможных событий, описаны в §3.8 (стр. 282). В большинстве случаев, обычные пользователи, имеющие дело с простыми моделями, могут не обращать внимания на описание функции в заголовке вкладки.

Если при последней компиляции в программе модели были обнаружены ошибки, список этих ошибок выводится в правой части окна под вкладками фрагментов программы. Высоту списка можно изменять, перетаскивая его верхнюю границу левой кнопкой мыши. Двойной щелчок на строке в этом списке открывает вкладку с текстом программы, в котором обнаружена ошибка, и устанавливает курсор на строчку с этой ошибкой. На рис. 329 в программе была специально допущена ошибка (отсутствует точка с запятой в конце оператора присваивания “t0=DynTime” в предпоследней строчке). Можно видеть, что в списке ошибок при этом появилась не одна, а сразу три строчки – так часто бывает при работе с C/C++, поскольку одна ошибка может проявиться несколько раз. В приведенном на рисунке примере первая строчка (“statement missing ; in function...”) описывает допущенную ошибку, то есть отсутствие в конце оператора точки с запятой, и при двойном щелчке на ней,

курсор установится на строчку “t0=DynTime” на вкладке “модель”. Вторая строчка в списке – это сообщение редактора связей о том, что он не может открыть объектный файл: из-за ошибки в тексте фрагмента программы компилятор не смог сформировать этот файл, так что это – следствие все той же ошибки. Наконец, третья строчка добавлена самим модулем автокомпиляции и сообщает об отсутствии файла DLL: поскольку редактор связей не нашел объектный файл, ему не из чего было создать исполняемый файл библиотеки. Таким образом, если в списке ошибок содержится несколько строчек, искать ошибку нужно начиная с самой первой из них – после ее исправления остальные ошибки могут исчезнуть сами.

Рассмотрим пункты меню окна редактора модели.

- Подменю “файл”:
 - ♦ Пункт “открыть файл” (клавиша Ctrl+O) открывает произвольный текстовый файл и помещает его на вкладку в правой части панели (рис. 330). Обычно так открывают файлы, каким-либо образом связанные с моделью – например, файлы заголовков, включенные в модель командой “#include” препроцессора языка C. В заголовке вкладки отображается имя файла, в верхней ее части – имя с полным путем. Файл на вкладке можно просматривать и редактировать, для его сохранения служит кнопка в левой верхней части вкладки. Если имя файла упоминается где-либо в тексте модели, его также можно открыть сочетанием клавиш Ctrl+Enter, если текстовый курсор находится внутри имени файла. Например, если в тексте модели есть строчка “#include <RdsFunc.h>”, файл “RdsFunc.h” можно открыть, щелкнув на его имени (это поместит текстовый курсор в точку щелчка) и нажав Ctrl+Enter.

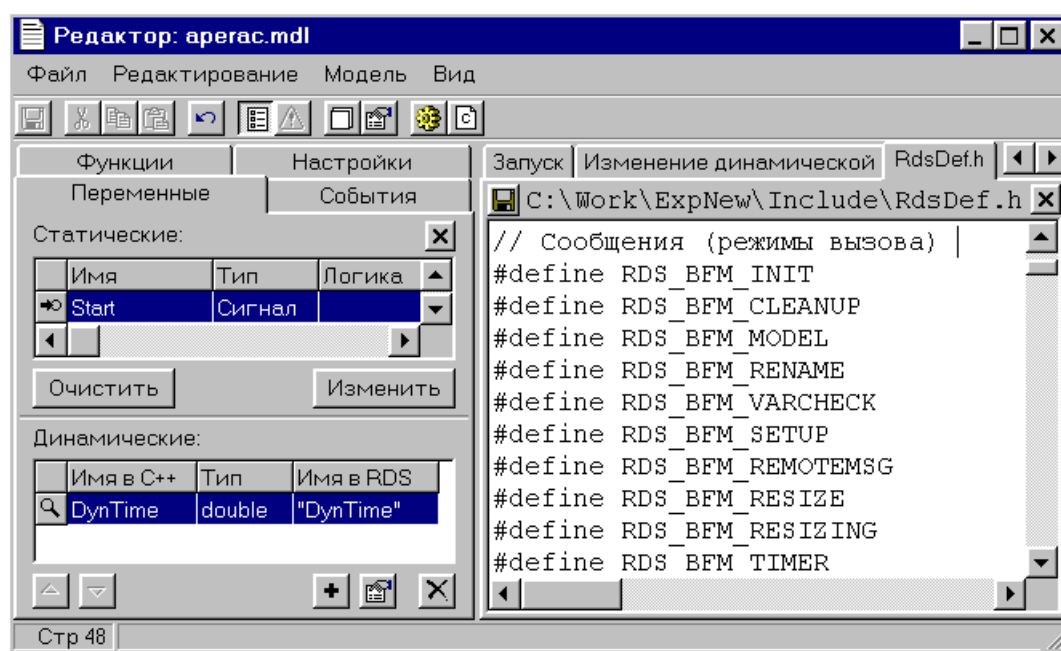


Рис. 330. Вкладка открытого файла в редакторе модели

- ♦ Подменю “модели” содержит список всех автокомпилируемых моделей, которые обслуживаются этим же модулем. Выбор в этом списке имени модели откроет ее окно редактора. Текущая открытая модель помечена в этом списке галочкой.
- ♦ Пункт “сохранить” (клавиша Ctrl+S) сохраняет изменения в файле модели, открытом в окне редактора.
- ♦ Пункт “сохранить как” записывает открытую модель в файл с другим именем и привязывает все блоки, использующие данную модель, к новому файлу. Действие этого пункта отличается от действия кнопки “сохранить как” в окне параметров блока (см. стр. 28) – та кнопка подключает к модели, сохраненной под другим именем, только

- один конкретный блок, а этот пункт меню – все блоки загруженной схемы, использовавшие прежнюю модель.
- ♦ Пункт “вернуться к сохраненному” отменяет все изменения в модели, сделанные с момента последнего ее сохранения.
 - ♦ Пункт “сохранить текст C++” позволяет записать в отдельный, выбранный пользователем, файл текст программы на языке C++, формируемый модулем для передачи компилятору. При желании, пользователь может проанализировать этот текст, чтобы понять, как именно модуль вставляет в текст программы написанные пользователем фрагменты, и какие описания добавляются в текст автоматически. Этот текст можно также просмотреть на отдельной вкладке окна, выбрав пункт меню “модель | показать текст C++”.
 - ♦ Пункт “сохранить как шаблон” записывает текст открытой в редакторе модели в качестве одного из шаблонов для создания новых моделей (имя файла задается в появляющемся диалоге сохранения). Шаблоны моделей должны быть записаны в папке “Common” внутри стандартной папки шаблонов РДС (см. §2.18 части I) – в открывающемся при выборе данного пункта меню диалоге эта папка выбрана по умолчанию. Хотя диалог и позволяет сохранить шаблон модели в другую папку, модуль автокомпиляции в этом случае его не увидит, и он не появится в списке шаблонов при создании модели (см. рис. 317 на стр. 22). При создании модели по шаблону вместо создания нового пустого файла модели в качестве этого нового файла будет скопирован выбранный файл шаблона, и пользователь сможет не создавать модель “с нуля”, а изменять или дописывать уже созданные фрагменты.
 - Подменю “редактирование”:
 - ♦ Пункт “отменить” (клавиша Ctrl+Z) позволяет отменить одну последнюю операцию редактирования в тексте программы на текущей открытой вкладке. Этот пункт никак не связан с одноименным пунктом главного меню РДС с тем же сочетанием клавиш (см. §2.1). Общая отмена операций в РДС позволяет последовательно отменить несколько действий пользователя, отмена в окне редактора модели – только одно, самое последнее, и только редактирование исходного текста. Изменение списка переменных, настроечных параметров и т.п. отменено быть не может (можно просто закрыть окно редактора, не сохранив модель – это отменит все изменения, внесенные в нее с момента последнего сохранения).
 - ♦ Пункт “вырезать” (клавиша Ctrl+X) помещает выделенный на вкладке текст фрагмента программы в буфер обмена, одновременно удаляя его с вкладки.
 - ♦ Пункт “копировать” (клавиша Ctrl+C) помещает в буфер обмена либо выделенный текст на вкладке фрагмента программы, если активно поле редактирования текста, либо имя переменной, если на левой панели окна выделена какая-либо переменная.
 - ♦ Пункт “вставить” (клавиша Ctrl+V) вставляет в позиции текстового курсора на вкладке фрагмента программы текст из буфера обмена.
 - ♦ Пункт “удалить” (клавиша Delete) стирает выделенный на вкладке текст программы. Этим пунктом нельзя удалить переменную или параметр на левой панели, для этого на панели есть специальные кнопки.
 - ♦ Пункт “найти” (клавиша Ctrl+F) позволяет найти среди всех введенных фрагментов программы заданный текст. При его выборе открывается окно (рис. 331), в котором вводится текст для поиска и устанавливаются флажки, управляющие этим поиском. На панели “параметры” можно включить или выключить поиск с учетом регистра символов или по целым словам, на панели “где искать” можно выбрать поиск по всем фрагментам программы или только по тем, вкладки которых в данный момент открыты (открытие вкладок фрагментов, отвечающих за различные описания и реакции на события, описано в §3.6.4 на стр. 46). Вкладка с найденным текстом становится

текущей и этот текст выделяется. Если выбран поиск по всем страницам (вкладкам), и вкладка с найденным текстом закрыта, она откроется автоматически.

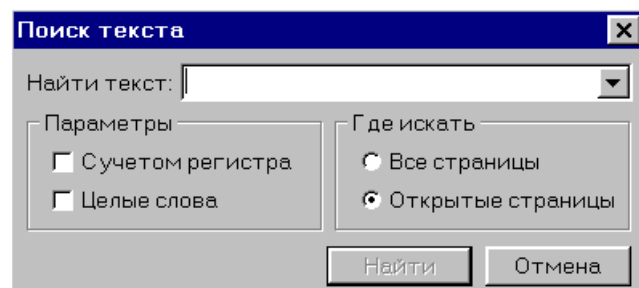


Рис. 331. Окно поиска текста

- ♦ Пункт “найти и заменить” (клавиша Ctrl+R) ищет среди всех введенных фрагментов программы один заданный текст и заменяет его на другой. При его выборе открывается окно (рис. 332) в котором задаются оба фрагмента текста и устанавливаются флажки управления поиском. Флажки на панелях “параметры” и “где искать” действуют аналогично пункту меню “поиск”, а флажки “подтверждение замены” и “заменить все вхождения” управляют производимой заменой фрагментов: включение первого будет требовать от пользователя подтверждения замены, включение второго выполнит не одну замену найденного фрагмента, а будет заменять все найденные.

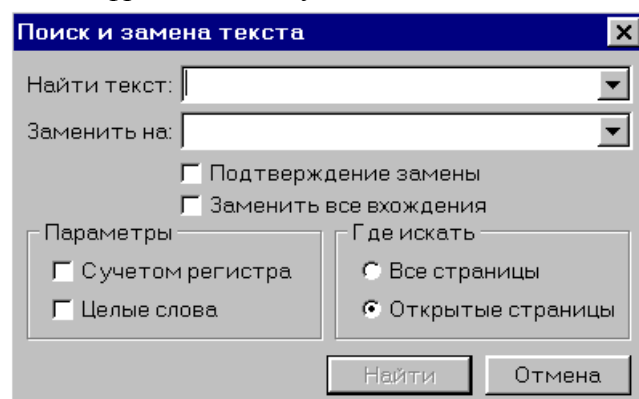













Рис. 332. Окно поиска и замены текста

- ♦ Пункт “найти далее” (клавиша F3) повторяет последний выполненный поиск или замену.
- Подменю “модель”:
 - ♦ Пункт “компилировать” (клавиша Alt+F9) компилирует модель, если с момента последней компиляции в нее были внесены какие-либо изменения (если изменений не было, об этом выдается сообщение). После компиляции новая функция модели в полученном файле DLL подключается ко всем связанным с этой моделью блокам, и блоки получают статические переменные, заданные в модели. Если одновременно запущено несколько копий РДС, в которые загружены схемы, использующие одну и ту же модель, компиляция завершится с ошибкой: модуль не сможет заменить старый файл DLL на новый, только что скомпилированный, поскольку старый файл будет заблокирован другими схемами.
 - ♦ Пункт “принудительно компилировать” компилирует модель, даже если время изменения файла DLL этой модели позднее, чем время последнего изменения самой модели (то есть после последней компиляции изменений в модели не было). Этот пункт меню может быть полезен при отказе системных часов, из-за которого времени изменения файлов определяются неверно. Также он может пригодиться при изменениях настроек модуля автокомпиляции: эти изменения не влияют на файл

модели, но, тем не менее, требуют ее повторной компиляции, поскольку меняется сам способ формирования программы из файла модели.

- ◆ Пункт “показать текст C++” открывает в окне дополнительную вкладку, на которой будет показан формируемый модулем исходный текст программы на языке C++, передаваемый компилятору для создания исполняемого файла DLL. Анализ этого текста может помочь пользователям, желающим изучить принципы написания моделей блоков без использования модуля автокомпиляции.
- ◆ Пункт “показать отчет компилятора” запускает компиляцию модели, после чего открывает в окне дополнительную вкладку, на которой отображаются командные строки, переданные компилятору и редактору связей, их параметры, а также все сообщения, выданные этими программами в процессе работы. Этот текст бывает полезен при настройке модуля автокомпиляции на нестандартный компилятор, в нем будет присутствовать полный список всех выданных компилятором и редактором связей ошибок. В большинстве случаев ошибки разбираются модулем автокомпиляции автоматически и показываются пользователю в нижней части окна редактора (см. рис. 329 на стр. 32), однако, для этого модуль должен быть правильно настроен на разбор возвращаемых компилятором сообщений. В процессе этой настройки (она описана в §3.9.7 на стр. 345) необходимо видеть эти сообщения, чтобы заложить в параметры модуля их структуру. Кроме того, в этом тексте можно увидеть, правильно ли передаются компилятору параметры, введенные в настройках модуля.
- ◆ Пункт “следующая страница” (клавиша Ctrl+PgDn) делает активной следующую по счету открытую вкладку редактора.
- ◆ Пункт “предыдущая страница” (клавиша Ctrl+PgUp) делает активной предыдущую по счету открытую вкладку редактора.
- ◆ Пункт “закрыть страницу” (клавиша Ctrl+F4) закрывает текущую активную вкладку. Закреть вкладку можно также кнопкой с крестиком в правой верхней ее части.
- ◆ Пункт “настройка модуля автокомпиляции” открывает окно настройки модуля, который обслуживает открытую в редакторе модель. Это то же самое окно, которое открывается по двойному щелчку на названии этого модуля в списке всех модулей, установленных в РДС (см. рис. 302 на стр. 11). Подробно настройка модуля будет описана в §3.9 (стр. 328).
- ◆ Пункт “установка параметров блоков” открывает окно групповой установки, описанное в §2.15.3, для всех блоков, к которым подключена эта модель. Это окно позволяет одновременно установить различные параметры блоков с этой моделью, подробнее работа с ним описана в §3.6.8 (стр. 76).
- ◆ Пункт “параметры модели” открывает окно настройки модели, в котором можно включить или выключить различные автоматически добавляемые в модель возможности, задать список связанных с этой моделью дополнительных файлов и ввести текст описания модели. Настройка модели описана в §3.6.7 (стр. 69).
- Подменю “вид”:
 - ◆ Пункт “переменные и события” включает и выключает отображение дополнительной панели редактора, расположенной в левой части окна (см. рис. 329 на стр. 32). На этой панели задаются переменные, параметры и функции блока, включаются реакции на события и т.п. Убрав эту панель, можно увеличить площадь области текста программы (когда потребуется, например, изменить переменные блока, ее можно будет снова включить).
 - ◆ Пункт “список ошибок” включает и выключает отображение списка ошибок последней компиляции в нижней части окна (см. рис. 329). Этот список также можно закрыть кнопкой с крестиком в его правой верхней части.

Непосредственно под меню находится панель с кнопками, дублирующими основные его пункты:

Кнопка	Выполняемое действие и пункт меню	Клавиши
	Сохранение изменений в модели (пункт меню “файл сохранить”)	Ctrl+S
	Переместить выделенный текст в буфер обмена (пункт меню “редактирование вырезать”)	Ctrl+X
	Копировать выделенный текст или имя переменной в буфер обмена (пункт меню “редактирование копировать”)	Ctrl+C
	Вставить текст из буфера обмена (пункт меню “редактирование вставить”)	Ctrl+V
	Отмена последнего изменения на текущей вкладке редактора (пункт меню “редактирование отменить”)	Ctrl+Z
	Включает/выключает боковую панель редактора (пункт меню “вид переменные и события”)	нет
	Включает/выключает список ошибок компиляции (пункт меню “вид список ошибок”)	нет
	Вызывает групповую установку параметров блоков, связанных с этой моделью (пункт меню “модель установка параметров блоков”)	нет
	Открывает окно параметров модели (пункт меню “модель параметры модели”)	нет
	Запускает компиляцию модели, если в нее были внесены изменения (пункт меню “модель компилировать”)	Alt+F9
	Открывает на отдельной вкладке текст программы, автоматически формируемой модулем, который передается компилятору при каждой компиляции модели (пункт меню “модель показать текст C++”)	нет

Окно редактора модели может быть закрыто в любой момент – для компиляции оно не требуется. При попытке закрыть окно редактора, не сохранив изменения в модели, пользователь получит сообщение об этом. Если при компиляции модели, редактор которой закрыт, возникнут ошибки, окно редактора откроется автоматически.

§3.6.2. Статические переменные блока

Описывается добавление в модель блока статических переменных, которые могут служить входами и выходами этого блока. Именно через статические переменные блоки получают данные по связям.

Статические и динамические переменные блока задаются на вкладке “переменные” дополнительной (левой) панели окна редактора модели. Если эта панель отключена, ее можно снова включить пунктом меню “вид | переменные и события”.

Вкладка (рис. 333) разделена по вертикали на две части: в верхней вводятся статические переменные, в нижней – динамические (границу раздела можно перетаскивать вверх и вниз левой кнопкой мыши). В этом параграфе рассматриваются только статические переменные, динамические описываются в §3.6.3 (стр. 42).

Статические переменные (см. §1.4 части I) используются как входы и выходы блока, а также для хранения промежуточных значений. Статическими эти переменные называются потому, что они создаются перед подключением модели к блоку и, хотя их значения и изменяются в процессе работы, их структура остается постоянной. Технически РДС

позволяет моделям блоков изменять структуру и типы своих статических переменных в процессе работы при помощи специальных вызовов, описанных в §2.16.1 руководства программиста [1], но эта возможность используется редко и не поддерживается стандартными модулями автокомпиляции. Таким образом, в автокомпилируемой модели набор статических переменных с их именами и типами остается постоянным в течение всего времени жизни блока.

Каждая статическая переменная имеет роль (вход, выход или внутренняя переменная), определенный тип, уникальное в блоке имя, значение по умолчанию и, возможно, комментарий, выводящийся вместе с ее именем в меню подключения связи к блоку (см. §2.7.1). Кроме того, с входами и выходами могут быть связаны дополнительные переменные, разрешающие или запрещающие работу выхода (см. §3.7.2.8 на стр. 126) и получающие сигнал о срабатывании входа (см. §3.7.2.7 на стр. 123).

Входы и выходы – основные роли статических переменных блока. Большинство блоков, как стандартных, так и пользовательских, занимаются тем, что считывают полученные по связям от других блоков значения со своих входов и вычисляют по ним значения своих выходов, которые точно так же, по связям, передаются в другие блоки. Простейший пример модели блока, выдающего на выход сумму двух своих входов, уже был приведен в §3.3 (стр. 20). Если у блока нет входов или выходов, к нему невозможно подключить связи, и он может обмениваться информацией с другими только через вызовы функций (см. §3.6.5 на стр. 57 и §3.7.13 на стр. 245) или динамические переменные (см. §3.6.3 на стр. 42 и §3.7.3 на стр. 129) – такие блоки тоже встречаются, но гораздо реже. Внутренние статические переменные используются, в основном, для хранения промежуточных значений.

Возможные типы переменных блока, поддерживаемые РДС, подробно описаны в §1.4 части I. Все они, за исключением произвольного, то есть программно изменяемого, типа, поддерживаются стандартным модулем автокомпиляции. Таким образом, статическая переменная в автокомпилируемом блоке может иметь один из следующих типов:

- Целые типы – `char`, `short` и `int` – предназначены для работы с целыми числами. В автокомпилируемых моделях лучше всего применять тип `int`: из всех трех целых типов у него самая большая разрядность (32 бита), и в переменной такого типа может храниться число в диапазоне $-2147483648 \dots 2147483647$. Типы `short` и `char` добавлены в РДС для совместимости и используются крайне редко.
- Вещественные типы – `float` и `double` – позволяют работать с вещественными числами. В автокомпилируемых моделях имеет смысл использовать тип `double`: у него больше разрядность и диапазон значений – модуль числа типа `double` может находиться в диапазоне $2.23 \times 10^{-308} \dots 1.79 \times 10^{308}$. Кроме того, все стандартные математические блоки в РДС используют именно тип `double`, и для него предусмотрено специальное значение, обозначающее ошибку выполнения математической операции. В автокомпилируемых моделях это значение хранится в глобальной переменной `rdsbcppHugeDouble` (пример ее использования приведен в §3.7.2.1 на стр. 95).
- Логический тип – переменная такого типа может принимать значения 0 (“ложь”) и 1 (“истина”).

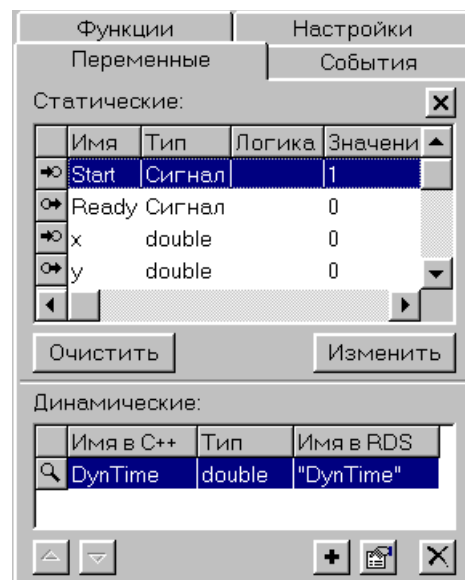


Рис. 333. Вкладка “переменные” дополнительной панели

- Сигнальный тип – как и логический, может иметь только значения 0 и 1, но передача сигналов по связям принципиально отличается: значение сигнального выхода передается на сигнальный вход только в том случае, если оно равно единице. Таким образом, появившаяся на сигнальном входе блока единица никогда не будет заменена нулем по связи и останется там до тех пор, пока сам блок не сотрет значение своего собственного входа – это полезно для регистрации наступления какого-либо события, обозначаемого сигнальной связью. Особенности передачи сигналов подробно рассмотрены в §1.4 части I и в руководстве программиста. Пример автокомпилируемой модели, использующей сигнальные переменные, приведен в §3.7.2.6 на стр. 117.
- Массив – набор пронумерованных переменных одинакового типа. Элементы массива нумеруются начиная с нуля, для обращения к конкретному элементу его индекс, как принято в языке C, записывается в квадратных скобках: i -й элемент массива A записывается как “A[i]”. Тип элемента массива может быть любым, за исключением другого массива (тем не менее, можно создавать массивы матриц, см. ниже). Размер массива не фиксирован, он может изменяться в процессе работы блока. По умолчанию проверка допустимости индекса не производится, и если попытаться обратиться к элементу за пределами текущего размера массива, работа модели будет аварийно завершена. В параметрах модели можно включить такую проверку – это может замедлить работу модели блока, но зато, при ошибке, модель выдаст более понятное диагностическое сообщение (см. стр. 71). Пример работы с массивами в модели блока приведен в §3.7.2.3 (стр. 103).
- Матрица – двумерная таблица переменных одинакового типа. Элемент матрицы определяется индексом строки и индексом столбца, начинающимися с нуля. Обращения к конкретному элементу матрицы в интерфейсе РДС (например, при присоединении к нему связи, см. §2.7.3 части I) и в программе отличаются: в РДС индексы перечисляются в квадратных скобках через запятую (“M[2, 7]” – элемент матрицы M в строке с индексом 2 и столбце с индексом 7), а в программе индексы пишутся в отдельных квадратных скобках по правилам языка C (“M[r][c]” – элемент матрицы M в строке с индексом r и столбце с индексом c). Тип элемента матрицы может быть любым, в том числе и другой матрицей (в РДС вложенность матриц ограничена пятью). Размер матрицы, как и у массива, не фиксирован, он может изменяться в процессе работы блока. По умолчанию обращение к элементу матрицы за пределами ее текущего размера не отслеживается и может прервать работу модели. В параметрах модели можно включить такую проверку ценой некоторого уменьшения скорости работы (см. стр. 71). Пример работы с матрицами в модели блока приведен в §3.7.2.2 (стр. 99).
- Строка – последовательность символов в кодировке Windows CP1251, завершающаяся нулевым байтом. Многобайтовые символы Unicode не поддерживаются.
- Структура – жестко заданный набор полей, каждое из которых имеет свое имя и тип. Всей структуре целиком присваивается имя типа, под которым она будет известна пользователю. Набор полей структуры с именами и типами и имя ее собственного типа задаются через окно списка структур, вызываемое пунктом главного меню РДС “система | структуры” (см. §2.14 части I). Стандартные блоки РДС используют единственную структуру с именем типа “Complex” и двумя вещественными полями “Re” и “Im”, предназначенную для работы с комплексными числами. Пользователь может добавлять и редактировать свои структуры, но следует иметь в виду, что редактирование состава полей структуры, уже используемой в модели какого-либо блока, может привести к неработоспособности этого блока. К автокомпилируемым моделям это относится в меньшей степени: после редактирования структуры следует просто скомпилировать модель заново, и, если в тексте программы нет обращений к удаленным полям структуры,

блок восстановит работоспособность. Пример работы со структурами в модели блока приведен в §3.7.2.4 (стр. 107).

Первые две переменные простого блока всегда представляют собой сигнальный вход и сигнальный выход, пользователь не может ни удалить их, ни изменить их роль и тип – их можно только переименовать. Сигнальный вход (по умолчанию он называется “Start”) разрешает срабатывание модели блока, если в параметрах блока не установлен флажок “запуск каждый такт”. Сигнальный выход (по умолчанию – “Ready”) сообщает подключенным к нему блокам об успешном срабатывании данного блока. Подробнее эти фиксированные сигналы описаны в §1.4 и в примерах на стр. 120 и 135.

Список статических переменных блока отображается на верхней части вкладки “переменные” в окне редактора модели (см. рис. 333 на стр. 39). В списке – четыре колонки:

- колонка без названия – роль переменной (вход, выход или внутренняя), изображаемая такими же картинками, как и в меню присоединения связей: “круг и стрелка от него” – выход, “круг и стрелка к нему” – вход, отсутствие картинки – внутренняя;
- “имя” – имя переменной;
- “тип” – название или условное обозначение типа переменной;
- “логика” – имя вспомогательной логической, сигнальной или целой переменной, подключенной к данной (см. примеры в §3.7.2.7 на стр. 123 и §3.7.2.8 на стр. 126);
- “значение” – значение переменной по умолчанию, которое она получает при сбросе расчета.

Колонки в списке можно переставлять и менять их ширину, перетаскивая левой кнопкой мыши их заголовки или границы заголовков. Нажатием Ctrl+C выделенную в списке переменную можно скопировать в буфер обмена для последующей вставки в текст программы. Редактировать переменные непосредственно в списке нельзя, для этого следует нажимать кнопку “изменить” под списком, которая вызывает стандартное окно редактора переменных РДС, подробно описанное в §2.9.2 части I. Кнопка “очистить” полностью очищает список статических переменных блока – при пустом списке для блоков с данной моделью не будет устанавливаться структура переменных. Разумеется, упомянутые выше сигнальные переменные “Start” и “Ready” у простых блоков все равно будут присутствовать, просто они будут недоступны для программы модели.

Для каждой статической переменной в формируемый модулем автокомпиляции текст программы автоматически добавляется одноименный объект, принадлежащий одному из специальных классов доступа к статическим переменным. При этом внутри реакций на события, которые вводятся в правой части окна редактора, к переменным можно обращаться просто по имени, используя их в стандартных выражениях языка С. Если, например, в модели заданы вещественные статические переменные “x”, “y” и “K”, в тексте программы можно писать “y=K*x;”. К полям структур можно обращаться, как и принято в С, отделяя их точкой от имени переменной структуры, индексы массивов и матриц указываются в квадратных скобках (примеры моделей, использующих различные типы переменных, приведены в §3.7.2 на стр. 92). При этом следует помнить, что, с точки зрения компилятора, все переменные блока – это объекты некоторых специальных классов С++, для которых определены операторы присваивания, операторы приведения типов и т.п. и за которыми скрыты “настоящие” переменные блока. Это не имеет значения, если переменные используются в математических выражениях: в приведенном выше примере “y=K*x;” для объектов K и x будет автоматически вызван оператор приведения к вещественному типу, а для объекта y – оператор присваивания этому объекту вещественного числа. Однако, если попытаться взять указатель на y, это будет не указатель на вещественное число, а указатель на объект некоторого класса, что следует учитывать при написании моделей (подробнее об этом – на стр. 87).

§3.6.3. Динамические переменные блока

Описывается добавление в модель блока динамических переменных, с помощью которых блоки могут обмениваться информацией без явного проведения связей между ними.

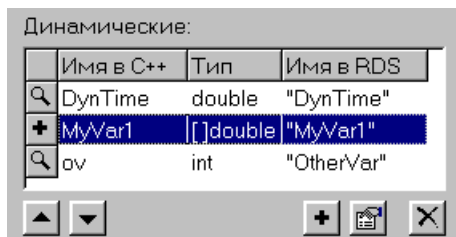


Рис. 334. Список динамических переменных

В нижней части вкладки “переменные” дополнительной панели окна редактора модели, непосредственно под уже описанным списком статических переменных, задается список динамических переменных блока (рис. 334). Динамические переменные (см. §1.5 части I) создаются моделями блоков скрыто от пользователя для обмена информацией друг с другом. Чаще всего такие переменные применяются в тех случаях, когда множеству блоков схемы необходимо одно и то же значение – передача его по связям ко всем

блокам во всех подсистемах загромождала бы схему. Типичный пример – текущее значение времени, которое нужно для работы всем динамическим блокам. Один из стандартных блоков РДС – планировщик расчета – создает вещественную динамическую переменную с именем “DynTime” и постоянно записывает в нее значение времени, изменяющееся с заданной скоростью и шагом, а все остальные блоки находят эту переменную и считывают ее значение. В РДС встроен специальный механизм подписки на динамическую переменную, который позволяет блоку найти переменную с заданным именем и типом – кратко он описан в §1.5 части I, подробнее – в §2.6 руководства программиста [1]. Примеры моделей блоков, создающих динамические переменные и подписывающихся на них, приведены в §3.7.3 (стр. 129), сейчас будет описан только интерфейс, позволяющий добавлять в модель такие переменные.






Список динамических переменных состоит из четырех колонок:

- Безымянная колонка – содержит знак “+”, если модель блока создает эту переменную, изображение лупы, если модель будет обращаться к переменной, созданной другим блоком, или пустой белый квадрат, если вся работа с динамической переменной будет выполняться программистом вручную, путем вызова функций-членов создаваемого для нее объекта (см. §3.7.8 на стр. 206).
- “Имя в C++” – имя объекта, создаваемого в модели для доступа к данной динамической переменной. Именно это имя будет использоваться в текстах фрагментов программы, вводимых в редакторе модели, внутри операторов присваивания или получения значения переменной. Оно может отличаться от имени переменной в РДС, указываемого отдельно.
- “Тип” – описание типа переменной. Динамические переменные в РДС имеют те же самые типы, что и статические (см. стр. 39).
- “Имя в RDS” – имя динамической переменной, используемое в РДС. Под этим именем переменная будет известна в РДС, оно будет использоваться для поиска и создания переменной. Имя либо жестко задается в модели (“фиксированное имя”), либо считывается из настроечного параметра блока (см. §3.7.7 на стр. 203). При жестком задании имени все блоки с данной моделью будут связаны с одной и той же динамической переменной, имена таких фиксированных переменных в списке отображаются в кавычках. Если же имя переменной хранится в параметре блока, его можно сделать разным в разных блоках с одной и той же моделью. Для динамических переменных, вся работа с которыми выполняется программистами вручную, имя не указывается.

Чаще всего, чтобы избежать путаницы, имя переменной в C++ делают таким же, как и имя в РДС, если оно фиксировано. Однако, это не всегда возможно: требования к синтаксису динамических переменных в РДС мягче требований к идентификаторам C++, и некоторые имена не могут быть использованы в программе без изменений. Формально имя переменной в C++ может быть никак не связано с ее именем в РДС – например, объект для доступа к

стандартной динамической переменной времени “DynTime” можно назвать просто “t”, и, в этом случае, во всех фрагментах программы, вводимых в редакторе модели, для обращения к значению времени можно будет использовать объект t.

Непосредственно под списком находятся кнопки, позволяющие добавлять, удалять и изменять переменные:

<i>Кнопка</i>	<i>Действие</i>
	Переместить выбранную переменную на одну позицию вверх по списку.
	Переместить выбранную переменную на одну позицию вниз по списку.
	Добавить новую переменную (открывает отдельное окно, см. рис. 335).
	Изменить выбранную переменную (открывает отдельное окно).
	Удалить выбранную переменную.

Порядок переменных в списке нужен только для их визуального упорядочения на панели редактора модели. Он никак отражается на работе модели, все динамические переменные равноправны.

При добавлении новой переменной или изменении уже существующей ее параметры отображаются в отдельном окне (рис. 335). В верхней части окна можно установить либо флажок “стандартная переменная” и выбрать одну из стандартных динамических переменных в выпадающем списке (параметры стандартных переменных жестко заданы), либо флажок “произвольная переменная”, если все параметры переменной указываются вручную.

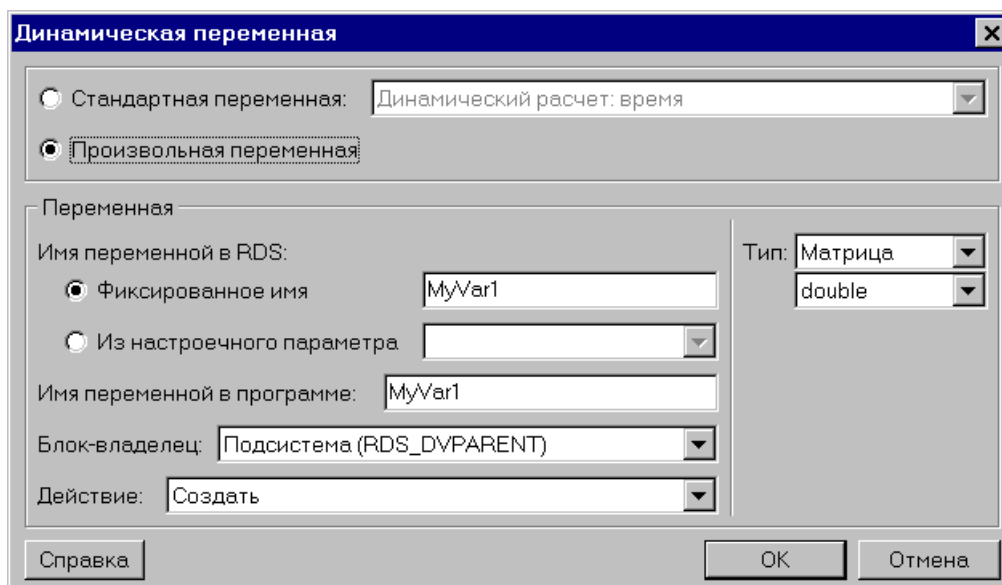


Рис. 335. Окно параметров динамической переменной

На данный момент в модуле автокомпиляции предусмотрена единственная стандартная переменная: текущее время динамического расчета “DynTime” типа double. При ее выборе все поля окна автоматически заполняются запомненными для этой переменной параметрами – достаточно нажать кнопку “OK”, чтобы связь с этой стандартной динамической переменной была добавлена в модель блока. Изменить при этом можно будет только имя переменной в программе, все остальные поля ввода будут заблокированы. Список стандартных динамических переменных во внутреннем формате РДС хранится в файле “dynvars.lst” в папке “Common” внутри стандартной папки настроек РДС (путь к

стандартной папке настроек можно узнать, вызвав пункт главного меню “сервис | настройки RDS”, см. §2.18), формат этого файла здесь не рассматривается.

Если в верхней части окна установлен флажок “произвольная переменная”, все параметры динамической переменной следует ввести вручную. Должны быть заполнены следующие поля:

- “Имя переменной в RDS” – имя динамической переменной. Все остальные блоки будут получать к ней доступ по этому имени. Если блок подключается к переменной, созданной другим блоком, имя этой переменной необходимо узнать у создателя модели последнего. Если же блок сам создает переменную, для нее желательно выбрать уникальное имя, чтобы переменная с таким же именем не использовалась другими разработчиками моделей. Имена динамических переменных чувствительны к регистру (“Var”, “VAR” и “var” будут считаться разными переменными) и не должны содержать знака доллара, точек, запятых и скобок. Для большей совместимости моделей лучше всего использовать в них только латинские буквы, цифры и знак подчеркивания. Имя переменной в РДС вводится только в том случае, если в поле “действие” (см. ниже) будет указано одно из стандартных действий. В окне можно выбрать один из двух вариантов указания имени динамической переменной:
 - ◆ “Фиксированное имя” – имя жестко задается в программе модели, все блоки с этой моделью будут связаны с одной и той же переменной. Само имя вводится в поле ввода справа от названия этого варианта.
 - ◆ “Из настроечного параметра” – указывается имя настроечного параметра блока, из которого будет считываться имя динамической переменной. Имя параметра либо вводится вручную в поле справа от названия варианта, либо выбирается в этом поле из списка уже имеющихся у блока параметров. Параметр для хранения имени динамической переменной обязательно должен иметь тип “rdsbcppString”, используемый в автокомпилируемых моделях для работы со строками. Если введенный в поле параметр отсутствует в блоке, при закрытии окна кнопкой “ОК” будет предложено создать его вместе с полем для его ввода в окне настроек (см. §3.6.6 на стр. 60 и §3.7.7 на стр. 203).
- “Имя переменной в программе” – имя, которое будет использоваться в тексте программы для обращения к данной переменной. Это имя должно удовлетворять стандартным требованиям к идентификаторам в языке С, то есть состоять только из букв, цифр и знака подчеркивания и не начинаться с цифры. По умолчанию в качестве имени переменной в программе подставляется копия имени переменной в РДС, в которой все недопустимые для языка С символы заменены на подчеркивания. Пользователь может изменить это имя так, как ему удобно, оно используется только внутри модели и недоступно РДС и другим блокам.
- “Тип” – тип переменной, которую нужно создать или найти. Тип выбирается из выпадающего списка. Если выбрать в списке “массив” или “матрица”, под ним появится дополнительный список для выбора типа элемента массива или матрицы (на рис. 335, например, переменная имеет тип “матрица double”).
- “Блок-владелец” – блок, внутри которого будет искаться или создаваться динамическая переменная. Можно выбрать один из трех вариантов: “этот блок”, “подсистема” и “система” (в выпадающем списке после названия каждого варианта в скобках приведена стандартная константа, которой этот вариант обозначается в сервисных функциях РДС). Если выбрать вариант “этот блок”, при подключении к переменной или ее создании модель будет обращаться к своему собственному блоку (этот вариант практически не применяется, поскольку при этом другие блоки, вероятнее всего, не будут иметь доступа к переменной). Вариант “подсистема” указывает модели обращаться для создания или поиска переменной к родительской подсистеме данного блока (к переменной будут иметь доступ все блоки этой подсистемы и вложенных в нее). Наконец, вариант “система”

указывает модели обращаться для создания или поиска переменной к корневой подсистеме схемы (к переменной будут иметь доступ все блоки схемы). Если в поле “действие”, описанном ниже, выбирается вариант “вручную”, блок-владелец не указывается: он будет задан программистом вручную при работе с объектом переменной внутри программы.

- “Действие” – что именно модель должна сделать с переменной. Можно выбрать один из четырех вариантов: “создать”, “подписаться”, “найти и подписаться” и “все действия – вручную”. Если выбран вариант “создать”, переменная будет создана в блоке, указанном в поле “блок-владелец”, если, конечно, в нем еще нет переменной с этим же именем. Если выбран вариант “подписаться”, модель попытается получить доступ к переменной с заданными именем и типом в блоке из поля “блок-владелец”. При этом, если такой переменной в этом блоке нет, дальнейший поиск производиться не будет. Если выбран вариант “найти и подписаться”, модель, как и в предыдущем варианте, сначала попытается получить доступ к заданной переменной в блоке из поля “блок-владелец”. Если в нем переменная не будет найдена, модель будет искать ее в родительской подсистеме этого блока. Если ее нет и там, поиск будет осуществляться в ее родительской подсистеме, и т.д. до корневой подсистемы схемы. Наконец, если выбран вариант “все действия – вручную”, модель не будет создавать переменную или подписываться на нее (поля ввода “имя переменной в RDS” и “блок-владелец” при этом будут заблокированы). Все действия с переменной в этом случае должны выполняться программистом в реакциях на различные события. Чаще всего этот вариант используется в тех случаях, когда имя динамической переменной формируется программно и его нельзя жестко указать в редакторе модели (пример модели с программным заданием имени переменной приведен в §3.7.8 на стр. 206).

Фактически, действия, выполняемые моделью для динамической переменной, определяются сочетанием полей “блок-владелец” и “действие”. Их можно проиллюстрировать следующей таблицей:

		Блок-владелец		
		Этот блок	Подсистема	Система
Действие	Создать	Переменная будет создана в этом блоке, другие блоки не будут иметь к ней доступа	Переменная будет создана в родительской подсистеме этого блока, ее смогут найти блоки этой же подсистемы и всех вложенных	Переменная будет создана в корневой подсистеме схемы, ее смогут найти все блоки
	Подписаться	Модель получит доступ к переменной в своем собственном блоке, только если она там есть	Модель получит доступ к переменной в родительской подсистеме своего блока, только если она там есть	Модель получит доступ к переменной в корневой подсистеме схемы, только если она там есть
	Найти и подписаться	Модель будет искать переменную начиная с собственного блока вверх по иерархии подсистем	Модель будет искать переменную начиная с родительской подсистемы собственного блока вверх по иерархии подсистем	
	Все действия – вручную	Модель не выполняет никаких действий для доступа к переменной или ее создания, в программе для переменной просто создается объект, с которым должен работать сам программист		

Следует помнить, что, в отличие от статических переменных, которые создаются вместе с блоком и, с точки зрения модели, существуют всегда, динамические переменные могут появляться и исчезать в процессе работы. Добавление динамической переменной в редактор модели блока не гарантирует ее существование – блок может не найти переменную с указанными именем и типом в заданной подсистеме. Даже если модели указано создать переменную, РДС может не дать ей сделать это, если переменная с таким именем уже существует в указанной в параметрах подсистеме. Обращение к отсутствующей переменной в программе модели вызовет критическую ошибку, поэтому по умолчанию модуль автокомпиляции автоматически блокирует все вызовы модели, если хотя бы одна из внесенных в список динамических переменных отсутствует. Эту блокировку можно отключить в параметрах модели (см. §3.6.7 на стр. 69), но при этом, перед обращением к переменной, необходимо проверять ее существование – пример этого приведен на стр. 133. Переменные, для которых не задано никаких действий, из автоматической проверки существования исключаются, независимо от включения блокировки в параметрах модели.

Нажатие кнопки “ОК” занесет новые параметры переменной в список. Нажатием Ctrl+C выделенную в списке переменную можно скопировать в буфер обмена для последующей вставки в текст программы.

§3.6.4. Описания программы и реакции блока на события

Описывается ввод фрагментов программ на языке C++, которые определяют поведение блока при наступлении различных системных событий.

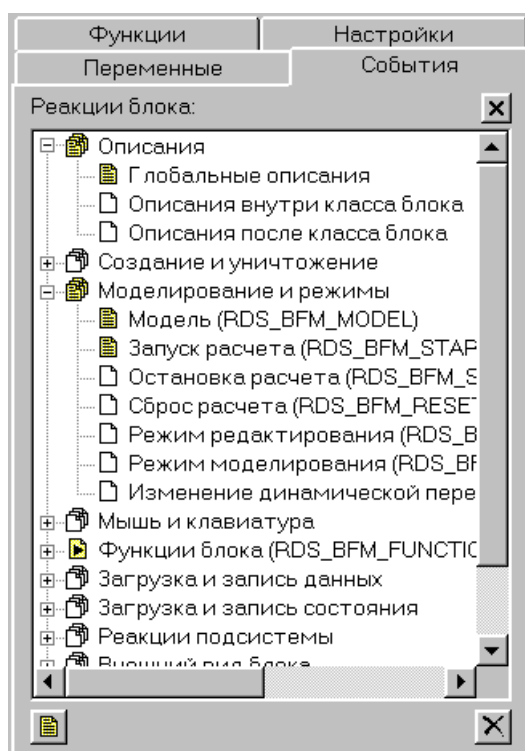


Рис. 336. Список фрагментов программы модели

Программа модели блока, формируемая модулем автокомпиляции, может состоять из большого числа вводимых пользователем фрагментов – как реакций блока на различные события, так и описаний, которые пользователь считает нужным включить в модель. Эти фрагменты добавляются и удаляются на вкладке “события” дополнительной панели окна редактора модели (рис. 336). Разбиение полной программы модели блока на отдельные фрагменты, связанные с конкретными событиями, позволяет переложить ответственность за правильную сборку полной программы на модуль автокомпиляции – пользователю достаточно указать, что именно должен делать блок при наступлении того или иного события, а модуль самостоятельно оформит этот текст в виде функции и включит в программу ее вызов в нужном месте.

Список фрагментов программы на этой вкладке организован в виде дерева, в котором события и описания объединены в группы по смыслу. Каждую группу можно раскрыть нажатием на значок “+” слева от ее названия.

Перед названием каждого отдельного фрагмента в группе отображается иконка желтого цвета, если для этого фрагмента введен текст программы, и белого, если текста нет, то есть реакция на данное событие или данное описание отсутствует в модели. Иконка рядом с названием группы имеет желтый цвет, если хотя бы один фрагмент внутри нее содержит текст программы – так можно видеть заполненные события и описания даже если их группы

не раскрыты. На рис. 336, например, видно, что пользователем введены тексты фрагментов программы для глобальных описаний модели, для выполнения одного такта расчета (пункт “модель” в группе “моделирование и режимы”), для реакции на запуск расчета и для каких-то реакций блока на вызов функций (группа “функции блока” не раскрыта, поэтому не видно, для каких именно). Рядом с названием каждой реакции блока на событие в скобках отображается имя константы, соответствующей этому событию в описаниях РДС. Зная имя этой константы, можно найти описание события в приложении к руководству программиста [2] и изучить возможности и примеры реакции на него. Модуль автокомпиляции позволяет включить в программу реакции не на все возможные события, но для большинства моделей будет достаточно тех, что есть. Список всех реакций и описаний, которые можно ввести в автокомпилируемую модель, приведен в §3.8 (стр. 282).

Не каждому введенному фрагменту программы модели соответствует открытая вкладка в правой части окна редактора, некоторые вкладки могут быть закрыты. При закрытии вкладки текст соответствующей реакции не исчезает, он просто больше не показывается пользователю. Для того, чтобы снова открыть закрытую вкладку с фрагментом программы, следует либо дважды щелкнуть на названии фрагмента в списке, либо выделить его в списке левым щелчком мыши и нажать кнопку под списком слева. Для того, чтобы полностью удалить фрагмент программы из модели, следует выделить его в списке и нажать кнопку под списком справа (редактор запросит подтверждение удаления текста). Пустые тексты фрагментов при сохранении модели удаляются автоматически. Если для модели не введено ни одного фрагмента программы, при открытии окна редактора автоматически появляется вкладка для реакции на один такт расчета, поскольку это самое часто используемое в моделях событие (см. рис. 321 на стр. 24).

Чтобы понять, как именно введенные пользователем фрагменты вставляются в формируемый текст полной программы модели, необходимо представлять себе общую структуру этого текста. Этот текст содержит (см. также §3.8.1 на стр. 282):

- команды включения необходимых для работы файлов заголовков (“windows.h”, “math.h” и т.п.);
- описания глобальных переменных и служебные функции, автоматически формируемые модулем согласно параметрам модели;
- автоматически формируемую главную функцию DLL, которую должна иметь каждая динамически подключаемая библиотека в Windows;
- глобальные описания пользователя, если таковые имеются;
- автоматически формируемое описание класса блока, содержащее все переменные и функции, необходимые для его работы, включая пользовательские описания полей и функций класса, если они есть;
- дополнительные пользовательские описания, если таковые имеются;
- функцию модели блока, автоматически формируемую согласно правилам РДС, содержащую оператор `switch` с большим количеством меток `case` (по одной метке на каждое событие), из которого вызываются функции реакции на события;
- введенные пользователем реакции на события, оформленные как функции-члены класса блока.

В целом, формируемый модулем автокомпиляции текст программы устроен примерно так же, как и примеры моделей блоков в руководстве программиста [1], за исключением того, что там для доступа к переменным блока используются макросы, а в автокомпилируемых моделях – специальные классы, и все данные блока тоже всегда оформляются как класс. Именно поэтому модуль автокомпиляции может работать только с компиляторами языка C++, а компиляторы “чистого” C для этого не подходят. Пользователь, создающий автокомпилируемые модели, может и не обращать особого внимания на структуру формируемого текста, но знание этой структуры может помочь в поиске ошибок. Если место какого-либо фрагмента в общем тексте программы покажется непонятным,

всегда можно вызвать пункт меню окна редактора “модель | показать текст C++” и найти свой фрагмент в общем тексте.

Примеры моделей с реакциями на различные события будут рассмотрены далее в §3.7 (стр. 80), здесь же мы кратко перечислим доступные в редакторе модели фрагменты с их характеристиками. Все упомянутые ниже константы и структуры подробно описаны в руководстве программиста и приложениях к нему.

- Группа “описания” – описания и функции, не связанные непосредственно с реакциями блока на события (см. также §3.8.1 на стр. 282):
 - ◆ “Глобальные описания” – описания глобальных переменных и пользовательских функций, а также команды включения нестандартных файлов заголовков, вставляемые перед описанием генерируемого для блока класса. Внутри функций в этих описаниях нельзя ссылаться на переменные блока: эти описания не являются частью класса блока и не имеют доступа к его полям. Чаще всего здесь размещают глобальные функции общего назначения.
 - ◆ “Описания внутри класса блока” – фрагмент, размещаемый в самом конце секции `public` класса блока. Здесь обычно описывают дополнительные пользовательские параметры, которые хранятся вместе с блоком, а также дополнительные функции, которым необходим доступ к статическим или динамическим переменным блока. Следует помнить, что, в отличие от настроечных параметров блока (см. §3.6.6 на стр. 60), для описанных здесь параметров модуль автокомпиляции не формирует команды загрузки и сохранения, поэтому их значения теряются при выгрузке схемы из памяти и ее последующей загрузке. Если дополнительные данные, хранимые в блоке, должны сохраняться вместе со схемой, необходимо либо переместить их в настроечные параметры, либо загружать и сохранять их вручную в реакциях на соответствующие события.
 - ◆ “Описания после класса блока” – фрагмент, размещаемый сразу после класса блока. Здесь обычно описывают функции, которым необходим доступ к классу блока, или тела функций-членов класса блока, описания которых сделаны в фрагменте “описания внутри класса блока”. Можно разместить здесь и описания глобальных функций и переменных.
- Группа “создание и уничтожение” – реакции на события, связанные с появлением и исчезновением блоков:
 - ◆ “Инициализация блока” – реакция на событие `RDS_BFM_INIT`, возникающее в момент подключения модели к блоку: при загрузке схемы с этим блоком, при добавлении блока в схему из библиотеки, при подключении модели к блоку в окне параметров и т.п. (см. стр. 286). Обычно в реакции на это событие присваивают начальные значения каким-либо дополнительным параметрам, описанным в классе блока, а также отводят дополнительную память, если она нужна модели. В автокомпилируемых моделях пользовательская реакция на это событие используется крайне редко, поскольку инициализация всех необходимых для работы блока объектов добавляется в программу модели автоматически, без участия пользователя. Следует помнить, что в реакции на событие инициализации **нельзя** обращаться к статическим переменным блока – это событие возникает до проверки допустимости структуры переменных, поэтому переменные блока на момент реакции могут не существовать.
 - ◆ “Очистка блока” – реакция на событие `RDS_BFM_CLEANUP`, возникающее перед отключением модели от блока: при выгрузке схемы из памяти, при стирании блока, при замене одной модели на другую и т.п. (см. стр. 287). Обычно здесь освобождают дополнительную память, отведенную при инициализации блока. Как и реакция на событие инициализации, пользовательская реакция на очистку данных блока в автокомпилируемых моделях практически не используется – все необходимые действия добавляются в программу автоматически. Следует помнить, что в реакции на

событие очистки **нельзя** обращаться к статическим переменным блока – это событие возникает, в том числе, и для блоков с недопустимой структурой переменных.

- ◆ “Добавление блока пользователем” – реакция на событие `RDS_BFM_MANUALINSERT`, возникающее при добавлении блока в схему пользователем при помощи вставки из буфера обмена, загрузки из отдельного файла, добавления из библиотеки или с панели блоков. При загрузке блока в память в составе схемы это событие не возникает. Здесь можно, например, запросить у пользователя какие-либо параметры нового блока. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_MANUALINSERTDATA`, содержащую причину добавления блока в схему (см. стр. 287).
- ◆ “Удаление блока пользователем” – реакция на событие `RDS_BFM_MANUALDELETE`, возникающее перед тем, как блок или содержащая его подсистема будут удалены из схемы пользователем. При выгрузке всей схемы из памяти это событие не возникает. Здесь можно, например, предупредить пользователя о том, что данный блок важен для работы всей схемы. Отменить удаление в реакции на это событие нельзя – блок будет удален независимо от выполненных в фрагменте программы действий. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_MANUALDELETEDATA`, содержащую описание действий пользователя, приведших к удалению блока (см. стр. 288).
- ◆ “Перед выгрузкой системы” – реакция на событие `RDS_BFM_UNLOADSYSTEM`, возникающее перед тем, как вся загруженная в данный момент схема будет удалена из памяти из-за загрузки другой схемы, создания новой или завершения РДС. Здесь можно, например, стереть временные файлы, созданные моделью в процессе работы.
- Группа “моделирование и режимы” – реакции на события, связанные с работой модели в режиме расчета и с переключением режимов РДС:
 - ◆ “Модель” – реакция на событие `RDS_BFM_MODEL`, которое возникает у всех простых блоков в каждом такте расчета. В режиме расчета все модели простых блоков, запуск которых разрешен, циклически вызываются для реакции на это событие – именно так устроен расчет в РДС. Это – самое часто используемое событие в автокомпилируемых моделях. Фактически, большинство таких моделей состоит только из реакции на это событие, в которой значения выходов блока вычисляются по значениям его входов (см. пример в §3.3 на стр. 20). Следует помнить, что реакция будет вызвана только в том случае, если запуск модели разрешен, то есть в параметрах блока включен расчет каждый такт или его первая сигнальная переменная имеет ненулевое значение (см. §1.3 части I).
 - ◆ “Запуск расчета” – реакция на событие `RDS_BFM_STARTCALC`, которое возникает при переходе РДС в режим расчета. Следует учитывать, что это событие возникает при любом запуске расчета, в том числе и при его продолжении после остановки. В этой реакции обычно выполняют какие-либо вычисления, которые не нужно выполнять постоянно на каждом такте. Например, здесь можно вычислить значения каких-либо сложных функций от настроечных параметров – настроечные параметры не могут быть изменены пользователем без выхода из режима расчета, поэтому их значения в течение всего расчета можно считать неизменными. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_STARTSTOPDATA` (см. стр. 291), содержащую параметры запуска расчета, в том числе и признак того, что расчет запущен в первый раз после сброса – этот признак можно использовать для инициализации блока.
 - ◆ “Остановка расчета” – реакция на событие `RDS_BFM_STOPCALC`, которое возникает при выходе РДС из режима расчета. Здесь можно, например, сообщить пользователю о возникших в процессе расчета ошибках, если это необходимо.

- ◆ “Сброс расчета” – реакция на событие `RDS_BFM_RESETCALC`, которое возникает сразу после сброса расчета во всей схеме или ее отдельной подсистеме (последнее возможно только в результате вызова специальных функций РДС моделями блоков – например, блоком оптимизации). Здесь обычно сбрасывают в исходные значения различные параметры блока, значения которых меняются в процессе расчета. Сбрасывать значения статических переменных не нужно, при сбросе им автоматически присваиваются заданные для них в редакторе значения по умолчанию.
- ◆ “Режим редактирования” – реакция на событие `RDS_BFM_EDITMODE`, которое возникает при переходе РДС в режим редактирования и сразу после загрузки схемы (если режим редактирования не запрещен внешним приложением, после загрузки новой схемы РДС переходит в этот режим автоматически). В реакции на это событие можно взводить какие-либо флаги, меняющие поведение блока. Например, блоки, которые сами рисуют свои изображения, могут по-разному вести себя в режимах редактирования, моделирования и расчета. Для отслеживания текущего режима можно использовать реакции на события смены режимов.
- ◆ “Режим моделирования” – реакция на событие `RDS_BFM_CALCMODE`, которое возникает при переходе РДС в режим моделирования. Это событие, как и событие перехода в режим редактирования, можно использовать для отслеживания текущего режима.
- ◆ “Изменение динамической переменной” – реакция на событие `RDS_BFM_DYNVARCHANGE`, которое возникает при создании и удалении динамической переменной, на которую подписался блок, а также в результате уведомления каким-либо другим блоком всех остальных об изменении значения этой переменной. Следует помнить, что это событие не возникает автоматически при записи в динамическую переменную нового значения – записавший значение блок должен явно уведомить все остальные блоки, использующие эту переменную, о том, что ее значение изменилось. В автокомпилируемых моделях это делается вызовом у переменной функции-члена `NotifySubscribers`: например, после изменения значения переменной `DynVar1` необходимо сделать вызов `“DynVar1.NotifySubscribers();”`, чтобы у всех блоков, подписанных на эту переменную (то есть блоков, запросивших к ней доступ), возникло событие `RDS_BFM_DYNVARCHANGE`. В функцию, формируемую для реакции на это событие, передается специальный идентификатор типа `RDS_PDYNVARLINK` (см. стр. 293), указывающий на изменившуюся переменную. Примеры моделей, реагирующих на изменение динамической переменной и уведомляющих другие блоки о таком изменении, рассмотрены в §3.7.3 (стр. 129).
- Группа “мышь и клавиатура” – реакции на действия пользователя:
 - ◆ “Нажатие кнопки мыши” – реакция на событие `RDS_BFM_MOUSEDOWN`, которое возникает в режимах моделирования и расчета при нажатии пользователем кнопки мыши в пределах изображения блока, находящегося на видимом активном слое, если в параметрах этого блока разрешена реакция на мышшь. В этой реакции можно не только определить факт нажатия кнопки, но и узнать, в какой части изображения блока в этот момент находился курсор. Пример модели, реагирующей на нажатие кнопки мыши, приведен в §3.7.11 (стр. 226). В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_MOUSEDATA`, содержащую идентификатор нажатой кнопки, координаты курсора мыши и т.п. (см. стр. 295).
 - ◆ “Отпускание кнопки мыши” – реакция на событие `RDS_BFM_MOUSEUP`, которое возникает в режимах моделирования и расчета при отпускании ранее нажатой пользователем кнопки мыши над изображением блока. Это событие – парное к предыдущему. Чаще всего оно используется при создании блоков, изображающих кнопки: при нажатии кнопки мыши блок изображает кнопку в нажатом состоянии до

тех пор, пока не получит информацию об отпускании кнопки. В функцию, формируемую для реакции на это событие, передается указатель на структуру RDS_MOUSEDATA, содержащую идентификатор кнопки, координаты курсора мыши и т.п.

- ◆ “Двойной щелчок мыши” – реакция на событие RDS_BFM_MOUSEDBLCLICK, которое возникает в режимах моделирования и расчета при двойном щелчке на изображении блока, находящегося на видимом активном слое, если в параметрах этого блока разрешена реакция на мышь. Следует учитывать, что из-за природы двойного щелчка перед реакцией на это событие блок сначала среагирует на нажатие и отпускание кнопки. Как и для остальных событий, связанных с мышью, в функцию реакции передается указатель на структуру RDS_MOUSEDATA.
- ◆ “Перемещение мыши” – реакция на событие RDS_BFM_MOUSEMOVE, которое возникает в режимах моделирования и расчета при перемещении курсора мыши над изображением блока, находящегося на видимом активном слое, если в параметрах этого блока разрешена реакция на мышь. В параметрах блока можно разрешить реакцию на это событие только при нажатых кнопках мыши или независимо от состояния кнопок (последнее используется реже). Эта реакция чаще всего используется для создания блоков, имитирующих различные рукоятки, которые пользователь может двигать, меняя значения переменных. В функцию реакции на это событие тоже передается указатель на структуру RDS_MOUSEDATA.
- ◆ “Нажатие клавиши” – реакция на событие RDS_BFM_KEYDOWN, которое возникает в режимах моделирования и расчета при нажатии какой-либо клавиши на клавиатуре, если окно подсистемы с этим блоком внутри находится на переднем плане и в параметрах блока разрешена реакция на клавиатуру. В реакции на это событие можно выполнять какие-либо действия по нажатиям разных клавиш, создавая таким образом “виртуальные пульты управления”. В функцию, формируемую для реакции на это событие, передается указатель на структуру RDS_KEYDATA, содержащую код клавиши, флаги и т.п. (см. стр. 298).
- ◆ “Отпускание клавиши” – реакция на событие RDS_BFM_KEYUP, которое возникает в режимах моделирования и расчета при отпускании ранее нажатой клавиши на клавиатуре, если окно подсистемы с этим блоком внутри находится на переднем плане и в параметрах блока разрешена реакция на клавиатуру. Это событие – парное к предыдущему. В функцию, формируемую для реакции на это событие, тоже передается указатель на структуру RDS_KEYDATA.
- Группа “функции блока” объединяет реакции на вызовы функций блока, внесенных в список функций на вкладке “функции” (см. §3.6.5 на стр. 57). В этой группе для каждой внесенной в список функции можно ввести текст программы реакции. Если в редакторе модели не добавлено ни одной функции блока, группа будет пустой. Примеры моделей, использующих функции блоков, приведены в §3.7.13 (стр. 245).
- Группа “загрузка и запись данных” – события, связанные с загрузкой и сохранением данных блока и всей схемы:
 - ◆ “Загрузка данных блока” – реакция на событие RDS_BFM_LOADTXT, которое возникает при загрузке параметров блока в момент загрузки схемы или при чтении данных этого блока из файла в библиотеке или из буфера обмена при вставке его в схему. В функцию реакции на это событие передается строка текста, сформированная этим же блоком при сохранении, которую модель должна разобрать, получив из нее все необходимые значения и присвоив их нужным параметрам. Эта реакция редко используется в автокомпилируемых моделях, поскольку загрузка значений настроечных параметров блока (см. §3.6.6 на стр. 60) производится автоматически, и никаких дополнительных реакций для этого не требуется. Создавать эту реакцию нужно только в том случае,

если у блока есть какие-то свои, нестандартные, параметры (например, оформленные как поля класса блока), которые необходимо сохранять и загружать вручную. Кроме того, ее можно использовать для выполнения каких-либо действий сразу после загрузки настроечных параметров блока – пример этого приведен в §3.7.7 (стр. 203).

- ◆ “Запись данных блока” – реакция на событие `RDS_BFM_SAVETXT`, которое возникает при сохранении параметров блока в момент сохранения всей схемы или при записи данных этого блока в отдельный файл или в буфер обмена. Это событие – парное к предыдущему. Для блоков, в которых нет нестандартных параметров, реакция на него не требуется – сохранение настроечных параметров добавляется в программу модели автоматически.
- ◆ “Перед сохранением схемы” – реакция на событие `RDS_BFM_BEFORESAVE`, возникающее у всех блоков непосредственно перед сохранением схемы в файл. Эта реакция используется крайне редко, в основном – в блоках, вносящих временные изменения в схему, чтобы они могли их отменить перед сохранением.
- ◆ “После сохранения схемы” – реакция на событие `RDS_BFM_AFTERSAVE`, возникающее у всех блоков сразу после сохранения схемы в файл. Используется крайне редко.
- ◆ “После загрузки схемы” – реакция на событие `RDS_BFM_AFTERLOAD`, возникающее у всех блоков сразу после загрузки схемы из файла. Обычно используется для инициализации каких-либо журналов событий (пример такой модели приведен на стр. 274).
- Группа “загрузка и запись состояния блока” – события, связанные с запоминанием состояния блока и его восстановлением по команде:
 - ◆ “Загрузка состояния блока” – реакция на событие `RDS_BFM_LOADSTATE`, возникающее у блока, если его текущее состояние восстанавливается по команде от модели какого-либо другого блока. Сохранение и загрузка состояния одного или нескольких блоков используется в РДС для управления расчетом: вызовом сервисной функции `rdsSaveSystemState` можно сохранить в памяти текущее состояние одного или нескольких блоков, а затем, вызвав `rdsLoadSystemState`, вернуть эти блоки в запомненное состояние (подробнее это описано в §2.14.3 руководства программиста). Реагируя на событие загрузки состояния, модель должна загрузить значения всех тех параметров, которые она сохранила в реакции на событие `RDS_BFM_SAVESTATE`. Для загрузки должна использоваться сервисная функция `rdsReadBlockData`. Следует учитывать, что текущие значения статических переменных блока восстанавливать не нужно – они сохраняются и восстанавливаются автоматически.
 - ◆ “Запись состояния блока” – реакция на событие `RDS_BFM_SAVESTATE`, возникающее у блока, если его текущее состояние сохраняется по команде от модели какого-либо другого блока. Это событие – парное к предыдущему. Реагируя на него, модель должна сохранить значения всех изменяющихся со временем параметров блока, чтобы потом, в реакции на событие `RDS_BFM_LOADSTATE`, восстановить их. Для сохранения должна использоваться сервисная функция `rdsWriteBlockData`.
- Группа “реакции подсистемы” объединяет реакции на различные действия пользователя с окном подсистемы:
 - ◆ “Действия с окном подсистемы” – реакция на событие `RDS_BFM_WINDOWOPERATION`, возникающее у подсистемы, если ее окно открывается или закрывается, а также у блоков других типов, если открывается или закрывается окно их родительской подсистемы. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_WINOPERATIONDATA`, содержащую текущий режим РДС, дескриптор окна и т.п. (см. стр. 308).

- ♦ “Нажатие кнопки мыши”, “отпускание кнопки мыши” и др. – реакции на события, аналогичные событиям из группы “мышь и клавиатура”, но вызываемые не для блоков, а для подсистем при действиях мышью в свободном от блоков месте рабочего поля их окон. Если, например, ни один из блоков не среагировал на нажатие кнопки мыши, и для подсистемы разрешена реакция на мышшь в свободных областях ее окна, будет вызвана реакция модели этой подсистемы. Поскольку автокомпилируемые модели подсистем на практике не применяются, события из этой группы используются крайне редко.
- Группа “внешний вид блока” – события, связанные с изображением блока и его положением на рабочем поле:
 - ♦ “Размер блока изменен” – реакция на событие `RDS_BFM_RESIZE`, возникающее у блоков, внешний вид которых рисуется программно, сразу после изменения размеров такого блока пользователем. В этой реакции можно отменить изменение размеров или скорректировать заданные пользователем размеры – например, изменять ширину блока синхронно с высотой, чтобы пропорции блока оставались неизменными. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_RESIZEDATA`, содержащую новые размеры блока и маркер выделения, который пользователь перетаскивал, чтобы изменить размеры (см. стр. 312).
 - ♦ “Проверка изменения размера” – реакция на событие `RDS_BFM_RESIZING`, возникающее у блоков, внешний вид которых рисуется программно, в процессе изменения размера такого блока перетаскиванием одного из восьми маркеров выделения блока, то есть при каждом движении курсора мыши. Чаще всего эта реакция используется для создания визуальной обратной связи при изменении размеров: в ней можно корректировать размер инверсного прямоугольника, которым обозначается новый размер блока в процессе перетаскивания маркеров. В функцию, формируемую для реакции на это событие, тоже передается указатель на структуру `RDS_RESIZEDATA`. Пример модели, реагирующей на это событие, приведен на стр. 183.
 - ♦ “Блок перемещен” – реакция на событие `RDS_BFM_MOVED`, возникающее у блока сразу после перемещения его пользователем или программным вызовом из модели другого блока. Реакция на это событие используется достаточно редко – чаще всего, для автоматического перемещения каких-либо блоков вместе с данным. Модель блока не может отменить перемещение блока в этой реакции. В функцию, формируемую для реакции на событие, передается указатель на структуру `RDS_MOVEDATA`, содержащую новые и старые координаты блока и причину его перемещения (см. стр. 314).
 - ♦ “Рисование блока” – реакция на событие `RDS_BFM_DRAW`, возникающее у блоков, внешний вид которых рисуется программно, при обновлении окна подсистемы с этим блоком. В реакции на это событие модель блока должна нарисовать в окне подсистемы его изображение, используя для этого сервисные функции РДС или графические функции API Windows. Следует помнить, что для включения программного рисования в окне параметров блока должен быть установлен флаг “внешний вид блока – определяется функцией DLL”, в противном случае блок будет изображаться либо векторной картинкой, либо прямоугольником с текстом, и реакция на это событие вызываться не будет. Функции рисования и различные их особенности подробно рассматриваются в руководстве программиста, примеры моделей блока, рисующих изображения программно, приведены в §3.7.5 (стр. 173). В функцию, формируемую для реакции, передается указатель на структуру `RDS_DRAWDATA`, содержащую оконные координаты, по которым необходимо нарисовать изображение, а также другие необходимые для рисования параметры (см. стр. 315).

- ◆ “Дополнительное рисование блока” – реакция на событие `RDS_BFM_DRAWADDITIONAL`, возникающее у всех блоков при обновлении окна их родительской подсистемы после того, как все изображения уже нарисованы. В отличие от события рисования `RDS_BFM_DRAW`, которое возникает только у блоков, внешний вид которых рисуется программно, это событие возникает у всех блоков независимо от их внешнего вида: у рисуемых программно, у имеющих векторную картинку и у изображаемых прямоугольником с текстом. В реакции на него можно вывести поверх изображения блока какую-либо дополнительную информацию – например, иконки, сигнализирующие об ошибках. В функцию, формируемую для реакции на событие, тоже передается указатель на структуру `RDS_DRAWDATA`. Пример модели блока, реагирующего на это событие, приведен в §3.7.10 (стр. 223).
- Группа “сеть” – события, связанные с встроенным в РДС механизмом обмена данными по сети (более подробно сетевые механизмы РДС описаны в §2.15 руководства программиста, здесь они не рассматриваются):
 - ◆ “Получены данные” – реакция на событие `RDS_BFM_NETDATARECEIVED`, возникающее при получении блоком каких-либо данных по сети от другого блока. В функцию, формируемую для реакции, передается указатель на структуру `RDS_NETRECEIVEDDATA`, содержащую принятые данные и описание отправившего их блока и сервера, через который идет обмен данными (см. стр. 316).
 - ◆ “Соединение установлено” – реакция на событие `RDS_BFM_NETCONNECT`, возникающее после того, как соединение с сервером, запрошенное моделью блока при помощи сервисной функции РДС `rdsNetConnect`, успешно установлено. Обычно в этой реакции взводят какой-либо внутренний флаг в параметрах блока, указывающий на то, что теперь можно передавать данные другим блокам. В функцию, формируемую для реакции, передается указатель на структуру `RDS_NETCONNDATA`, содержащую параметры установленного соединения (см. стр. 317).
 - ◆ “Соединение разорвано” – реакция на событие `RDS_BFM_NETDISCONNECT`, возникающее после разрыва ранее установленного моделью блока соединения с сервером. Соединение может быть разорвано по разным причинам. Если соединение было разорвано по инициативе модели блока вызовом `rdsNetCloseConnection`, оно останется разорванным, в противном случае РДС будет пытаться самостоятельно восстановить это соединение без каких-либо запросов от модели. Обычно в этой реакции взводят какой-либо внутренний флаг в параметрах блока, указывающий на то, что передавать данные другим блокам сейчас нельзя. В функцию, формируемую для реакции, тоже передается указатель на структуру `RDS_NETCONNDATA`.
 - ◆ “Данные приняты сервером” – реакция на событие `RDS_BFM_NETDATAACCEPTED`, возникающее после того, как сервер, через который блок отправил свои данные, подтвердил их получение. Это событие возникает только в том случае, если, передавая данные, блок запросил у сервера подтверждение их приема в параметрах функций `rdsNetBroadcastData` или `rdsNetSendData`. Следует помнить, что подтверждение выдается при получении данных сервером, а не блоком-получателем данных – серверу еще только предстоит отправить данные получателю. Эту реакцию можно использовать для организации в блоке собственной очереди передачи данных, не связанной с очередью РДС, чтобы блок не передавал очередную порцию данных, пока не получит от сервера подтверждение приема предыдущей. В функцию, формируемую для реакции, передается указатель на структуру `RDS_NETACCEPTDATA`, содержащую параметры сетевого соединения и идентификатор блока данных, получение которого подтверждается (см. стр. 319).
 - ◆ “Ошибка сети” – реакция на событие `RDS_BFM_NETERROR`, возникающее у блоков, участвующих в приеме или передаче данных, при возникновении различных ошибок в

сетевом соединении. Большую часть таких ошибок РДС обрабатывает самостоятельно, поэтому, как правило, в моделях блоков реакция на это событие не требуется. В функцию, формируемую для реакции, передается указатель на структуру `RDS_NETERRORDATA`, содержащую параметры сетевого соединения и код ошибки (см. стр. 319).

- Группа “разное” – различные события, не вошедшие в одну из уже описанных групп:
 - ◆ “Переименование блока” – реакция на событие `RDS_BFM_RENAME`, возникающее после переименования блока. В функцию, формируемую для реакции, передается строка с именем блока до переименования. В реакции на это событие можно, например, изменить имена каких-либо связанных с блоком объектов, которые формируются из имени блока.
 - ◆ “Вызов настройки” – реакция на событие `RDS_BFM_SETUP`, возникающее в режиме редактирования при вызове функции настройки блока. Редактор модели позволяет достаточно простыми средствами создавать окно для задания настроечных параметров блока (см. §3.6.6 на стр. 60) без необходимости вручную писать программу для этой реакции. Реакцию на это событие можно использовать в тех случаях, когда встроенных в редактор модели возможностей окна настройки не хватает. Следует помнить, что если в редакторе модели одновременно описать настроечные параметры блока с окном их настройки и ввести текст для реакции на событие `RDS_BFM_SETUP`, сначала будет открыто окно настройки, созданное средствами редактора, а введенная программа реакции вызовется только после закрытия этого окна кнопкой “ОК” (кнопка “Отмена” заблокирует вызов реакции) – пример такого использования этой реакции приведен в §3.7.8 (стр. 206).
 - ◆ “Сообщение от управляющей программы” – реакция на событие `RDS_BFM_REMOTEMSG`, возникающее при управлении РДС из внешнего приложения (см. главу 3 руководства программиста) в момент вызова этим приложением данного блока схемы с помощью предназначенных для этого функций библиотеки `RdsCtrl.dll`. В функцию, формируемую для реакции, передается указатель на структуру `RDS_REMOTEMSGDATA`, содержащую переданные управляющим приложением данные (см. стр. 322).
 - ◆ “Реакция на таймер” – реакция на событие `RDS_BFM_TIMER`, возникающее при срабатывании таймера блока, созданного его моделью при помощи сервисной функции РДС `rdsSetBlockTimer`. В функцию, формируемую для реакции на это событие, передается идентификатор сработавшего таймера. Реакция на таймер может использоваться для периодического или отложенного на заданное время выполнения моделью блока каких-либо действий.
 - ◆ “Обновление окон блока” – реакция на событие `RDS_BFM_WINREFRESH`, возникающее при необходимости обновить открытые окна, принадлежащие данному блоку. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_WINREFRESHDATA`, в которой содержится идентификатор таймера, вызвавшего обновление окон (если обновление вызвано таймером), и в которую функция модели может записать время, потраченное ей на рисование содержимого этих окон, для учета в алгоритме подстройки частоты обновления (см. §2.18 части I). Создание моделей блоков, открывающих и поддерживающих собственные окна – довольно сложная задача, требующая использования блокировки данных, поэтому в автокомпилируемых моделях эта реакция практически не используется (желающие могут изучить §1.8 руководства программиста).
 - ◆ “Всплывающая подсказка” – реакция на событие `RDS_BFM_POPUPHINT`, возникающее при запросе у блока текста всплывающей подсказки, если курсор мыши задержался в пределах изображения этого блока (при этом вывод всплывающей подсказки должен

быть разрешен в параметрах блока). В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_POPUPHINTDATA`, в которой содержатся координаты курсора мыши и текущие параметры изображения блока, и через которую модель возвращает параметры показа подсказки, если это необходимо (см. стр. 324). Текст подсказки возвращается в РДС при помощи сервисной функции `rdsSetHintText`. В этой реакции можно формировать не только статичные всплывающие подсказки, но и подсказки, изменяющиеся в зависимости от положения курсора на изображении блока (см. §3.7.9 на стр. 211).

- ♦ “Вызов контекстного меню” – реакция на событие `RDS_BFM_CONTEXTPOPUP`, возникающее при открытии контекстного (то есть вызываемого по правой кнопке мыши) меню блока. В этой реакции модель может добавить в это меню свои собственные пункты. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_CONTEXTPOPUPDATA` (см. стр. 325), в которой содержится признак нахождения РДС в режиме редактирования (чаще всего пункты контекстного меню в режиме редактирования и в режимах моделирования и расчета не совпадают). Примеры моделей, добавляющих свои пункты в контекстное меню, приведены в §3.7.12 (стр. 238).
- ♦ “Выбор пункта меню” – реакция на событие `RDS_BFM_MENUFUNCTION`, возникающее при выборе пользователем одного из пунктов, добавленных моделью блока в его контекстное меню или в главное меню РДС. В функцию, формируемую для реакции на это событие, передается указатель на структуру `RDS_MENUFUNCDATA`, описывающую выбранный пользователем пункт (см. стр. 326).
- ♦ “После создания статических переменных” – фрагмент программы, вызываемый немедленно после автоматически формируемой модулем автокомпиляции реакции на событие проверки типа статических переменных `RDS_BFM_VARCHECK`, если фактические типы переменных блока совпали с заданными в редакторе модели. Если переменные блока отличаются по типам и последовательности от списка, заданного на вкладке “переменные” левой панели редактора (см. §3.6.2 на стр. 38), этот фрагмент, как и все остальные реакции на события, вызываться не будет. В моделях блоков крайне редко возникает необходимость выполнить какие-либо действия после успешной проверки соответствия переменных блока ожиданиям модели, поэтому данная реакция практически не используется.
- ♦ “Прочие события” – общая реакция на все остальные события, возникающие в схеме. Эти события используются значительно реже перечисленных выше, поэтому отдельные пункты в список для них не добавлены. В общую функцию, формируемую для реакции на эти события, передается идентификатор конкретного события и указатель общего вида, который нужно привести к типу, соответствующему параметрам события. Полный список событий приведен в приложении к руководству программиста.

Для перечисленных выше событий автоматически формируются функции следующего вида:

```
void rdsbcppBlockClass::имя_функции(параметры)
{
    текст пользователя
}
```

где `rdsbcppBlockClass` – имя класса блока (оно всегда одно и то же во всех моделях, несмотря на то, что содержимое этого класса изменяется от модели к модели), *имя_функции* – жестко заложенное в модуль автокомпиляции имя функции реакции на данное событие, *параметры* – жестко заданный для каждого типа события набор параметров, *текст пользователя* – введенный пользователем на вкладке редактора модели фрагмент текста программы. Все возможные варианты создаваемых функций перечислены в §3.8 (стр. 282). Заголовок такой функции со всеми ее параметрами отображается в верхней части вкладки, на

которой вводится текст реакции (рис. 337) – таким образом, пользователь может понять, какие параметры функции он может использовать в тексте.

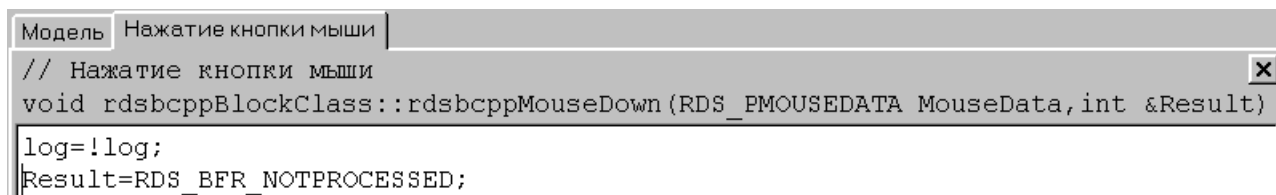


Рис. 337. Заголовок функции реакции на событие

Для описаний, в отличие от реакций на события, функции не генерируются – введенный пользователем текст вставляется непосредственно в программу внутри или снаружи описания класса блока (см. §3.8.1 на стр. 282).

§3.6.5. Функции блока

Описывается добавление в модель функций блока, при помощи которых блок может непосредственно, без использования связей, вызывать другие блоки и передавать им данные, а также реагировать на такие вызовы от других блоков.

Одним из способов взаимодействия блоков схемы друг с другом является непосредственный вызов модели одного блока моделью другого – в РДС это называется вызовом функции блока. Блок может выполнять несколько функций, каждая такая функция должна иметь уникальное текстовое имя, по которому ее можно отличить от других. Обычно, если разработчик модели решает, что его блок может выполнять какие-либо полезные действия по запросам от других блоков, он придумывает для этих действий имя функции – достаточно длинное, и сложное, чтобы избежать возможного пересечения с функциями, придуманными другими разработчиками – и встраивает в свою модель реакцию на выполнение этой функции. Имя функции блока в РДС не ограничено по длине, поэтому чаще всего в него включают имя разработчика, название выполняемой задачи, библиотеку, в модели из которой эта функция впервые появилась, и т.п. Общие рекомендации по выбору имен для новых функций даются в §2.13.1 руководства программиста [1].

Технически механизм вызова функций блоков устроен в РДС следующим образом. Чтобы модель блока могла вызывать функции других блоков и реагировать на вызов своих функций, эти функции должны быть зарегистрированы в РДС. Для этого модель должна передать в РДС имя функции и получить присвоенный этой функции целый идентификатор (для этого используется вызов `rdsRegisterFunction`), и вся дальнейшая работа с функцией ведется с использованием этого идентификатора. Для вызова функции другого блока необходимо передать РДС его идентификатор (или идентификатор подсистемы, если нужно вызвать функцию у всех ее блоков), идентификатор вызываемой функции и указатель на область данных параметров функции (как правило, это какая-либо структура). Вызванная модель блока получает идентификатор функции и указатель на область ее параметров, она должна сравнить этот идентификатор с идентификаторами поддерживаемых ей функций и выполнить необходимые действия, если он совпал с одним из них.

Стандартный модуль автокомпиляции автоматизирует регистрацию функций и распознавание вызванной функции: пользователю необходимо только добавить необходимые описания на вкладке “функции” дополнительной панели окна редактора модели (рис. 338, на рисунке дополнительная панель растянута по горизонтали, чтобы уместились длинные имена функций). Для каждой из функций в списке на этой вкладке в модель будет автоматически добавлен регистрирующий ее вызов и соответствующее ей событие в список событий (см. §3.6.4 на стр. 46, а также рис. 340).

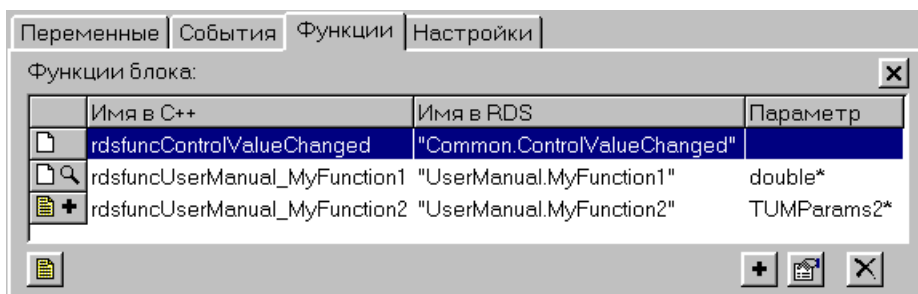


Рис. 338. Список функций блока

Список функций на вкладке состоит из четырех колонок:

- Безымянная колонка – отображает одну или две иконки. В левой части колонки белая иконка пустого листа означает, что текст реакции на вызов этой функции не введен в модель, а желтая иконка – что этот текст введен. На рисунке текст реакции введен только для третьей сверху функции. В правой части колонки изображается иконка дополнительных действий с функцией. Иконка с лупой (на рисунке – вторая сверху функция) указывает на то, что в модель данного блока будет добавлен автоматический поиск блока, зарегистрировавшегося в РДС в качестве исполнителя этой функции (см. §3.7.13.4 на стр. 268). Иконка со знаком “+” (третья сверху функция на рисунке) – на то, что модель регистрирует данный блок как исполнителя функции. Наконец, отсутствие иконки в правой части колонки означает, что никаких дополнительных действий с функцией модель производить не будет.
- “Имя в С++” – имя объекта, созданного для хранения идентификатора этой функции и работы с ней. Вызов функции у других блоков производится с использованием этого объекта – примеры различных способов вызова приводятся в §3.7.13 (стр. 245). Особенности использования объектов, создаваемых для работы с функциями, рассматриваются в §3.7.13.5 (стр. 279).
- “Имя в RDS” – имя функции блока, под которым она регистрируется в РДС. Другие модели для вызова этой функции или реакции на ее вызов должны зарегистрировать ее под этим же именем.
- “Параметр” – тип указателя на область данных параметров функции, или пустая строка, если у функции нет параметров. На рисунке первая функция не имеет параметров, вторая использует в качестве параметра вещественное число (поэтому в колонке отображается “double*”, то есть “указатель на double”), параметром третьей является некоторая структура или класс с именем “TUMParams2”.

Под списком располагаются кнопки, позволяющие добавлять, удалять и изменять функции, а также переходить к вводу текста реакции на их вызов:

Кнопка	Действие
	Открыть вкладку для ввода текста реакции на вызов выбранной функции.
	Добавить новую функцию (открывает отдельное окно, см. рис. 339).
	Изменить выбранную функцию (открывает отдельное окно).
	Удалить выбранную функцию.

При добавлении новой функции блока или изменении уже существующей ее параметры отображаются в отдельном окне (рис. 339). В верхней части окна можно либо установить флажок “стандартная функция” и выбрать в выпадающем списке одну из функций, поддерживаемых стандартными блоками, входящими в состав РДС, либо установить флажок “произвольная функция” и ввести все параметры функции вручную.

Рис. 339. Окно параметров функции блока

- Для произвольной функции в нижней части окна заполняются следующие поля:
- “Имя функции в RDS” – имя, под которым модель регистрирует функцию блока в РДС. Это же имя необходимо использовать для регистрации этой функции во всех остальных моделях, которые будут ее использовать.
 - “Идентификатор функции в программе” – имя объекта, который будет создан в программе модели для хранения идентификатора этой функции. Его имя должно удовлетворять требованиям языка С (содержать только латинские буквы, цифры и знак подчеркивания и не начинаться с цифры) и не должно совпадать с именами статических и динамических переменных блока. Запоминание в этом объекте идентификатора, полученного от РДС при регистрации функции, будет добавлено в программу автоматически. При первом вводе имени функции в окне параметров это поле заполняется само – по умолчанию имя объекта формируется добавлением к введенному имени функции приставки “rdsfunc” и заменой всех символов, недопустимых для имен переменных в языке С, на подчеркивания (на рисунке для функции “UserManual.MyFunction2” было автоматически сформировано имя переменной “rdsfuncUserManual_MyFunction2”). При желании, автоматически сформированное имя можно заменить на любое другое, не совпадающее с именами других глобальных объектов в программе.
 - “Имя функции реакции в классе блока” – имя функции-члена класса блока, которая будет автоматически вызываться в том случае, если какой-нибудь другой блок вызовет блок с этой моделью для выполнения данной функции. Текст в этом поле формируется автоматически и не может быть изменен: он получается добавлением к имени переменной из предыдущего поля слова “Event”.
 - “Тип параметра функции (указатель)” – тип указателя С++, к которому необходимо привести указатель на область параметров функции. Например, если параметром функции является вещественное число типа double, в это поле необходимо ввести “double*”. На рисунке в поле введено “TUMParams2*” – это означает, что параметром функции будет являться некоторая структура типа TUMParams2.
 - “Дополнительные действия” – одно из двух возможных действий, которые модель может выполнить для этой функции: либо зарегистрировать блок с этой моделью в РДС как исполнителя данной функции, чтобы все остальные блоки схемы могли его легко найти

(на рисунке выбран именно такой вариант), либо, наоборот, заставить модель найти блок, зарегистрированный как исполнитель функции, чтобы можно было вызывать его. Можно не указывать никаких действий – в этом случае модель все равно сможет вызывать функцию и реагировать на ее вызов, но все действия в этой модели по определению вызываемого блока и в других моделях по поиску этого должны будут выполняться вручную.

Нажатие кнопки “ОК” занесет новые параметры функции в список. При добавлении новой функции блока ее имя автоматически появляется в группе “функции блока” в списке событий на панели “события” (см. §3.6.4 на стр. 46), где можно будет ввести реакцию на ее вызов у блока с данной моделью. Реакции на вызовы функций из списка на рис. 338 в списке

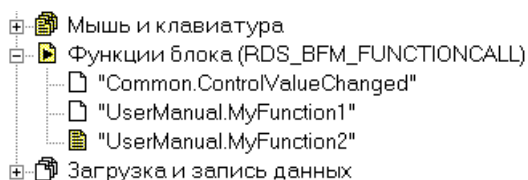


Рис. 340. Реакции на вызов функций в списке событий модели

событий изображены на рис. 340. Если модель должна не только вызывать эту функцию, но и реагировать на ее вызов, открыть вкладку для ввода текста реакции можно как из списка событий, дважды щелкнув на имени функции, так и из самого списка функций, выбрав в нем нужную функцию и нажав кнопку с желтым листком слева снизу под списком.

Для реакции на вызов функции без параметров автоматически формируется функция-член класса блока следующего вида:

```
void rdsbcppBlockClass::имя_функции(
    RDS_PFUNCTIONCALldata FData,
    int &Result)
{
    текст пользователя
}
```

где `rdsbcppBlockClass` – жестко заданное модулем автокомпиляции имя класса блока, *имя_функции* – автоматически сформированное имя функции реакции из поля “имя функции реакции в классе блока” окна параметров функции (см. рис. 339), `FData` – указатель на структуру `RDS_FUNCTIONCALldata` (см. §3.8.5 на стр. 301), описывающую вызов функции (какой блок ее вызвал, в каком режиме и т.п.), `Result` – ссылка на целую переменную, через которую функция возвращает результат, *текст пользователя* – введенный пользователем на вкладке редактора модели фрагмент текста программы. Для функции с параметрами в описание добавляется еще и параметр указанного при добавлении функции типа:

```
void rdsbcppBlockClass::имя_функции(тип_параметра Param,
    RDS_PFUNCTIONCALldata FData,
    int &Result)
{
    текст пользователя
}
```

где *тип_параметра* – указанный при создании функции тип указателя на область ее параметров, `Param` – приведенный к этому типу указатель.

Примеры моделей, вызывающих функции других блоков и реагирующих на такие вызовы, приведены в §3.7.13 (стр. 245).

§3.6.6. Настроечные параметры блока

Описывается добавление в модель параметров, которые пользователь сможет изменять в окне настройки блока, а также организация этого окна.

Величины, участвующие в расчете, выполняемом блоком, можно условно разделить на две категории: переменные и параметры. Переменные, как правило, изменяются часто, и значения их приходят от других блоков схемы. Параметры либо не изменяются вообще, либо

изменяются крайне редко, и, чаще всего, задаются пользователем. Разумеется, это деление достаточно условно. Рассмотрим, например, блок, задача которого – выдать на выход y значение входа x , умноженное на некоторую константу K (в модели блока будет записан оператор “ $y=K \cdot x$;”). Вход x и выход y блока должны быть статическими переменными – только статические переменные могут быть входами и выходами. Параметр K тоже можно сделать входом блока и подавать на него значение, например, с поля ввода. Однако, поскольку K меняется редко, загромождать схему дополнительным полем ввода и связью, соединенной с ним, может оказаться не лучшей идеей. Будет гораздо удобнее, если значение K можно будет вводить в отдельном окне, открываемом, например, по двойному щелчку на блоке. Большинство стандартных блоков в РДС позволяют задавать свои параметры именно таким образом, при помощи предусмотренной для каждого блока функции настройки. Такие параметры называют настроечными, а окно, в котором они вводятся, окном настройки блока. Стандартный модуль автокомпиляции позволяет достаточно простыми средствами добавлять в блок настроечные параметры и создавать окна с полями ввода, в которых пользователь сможет эти параметры задавать.

Настроечные параметры блока и поля ввода для их задания описываются на вкладке “настройки” дополнительной панели окна редактора модели (рис. 341). В верхней части вкладки вводится список самих параметров, в нижней задается состав полей ввода окна настройки (горизонтальную границу раздела между этими двумя списками можно перемещать вверх и вниз левой кнопкой мыши). Как правило, для каждого параметра в окне предусматривают поле ввода, однако, это не обязательно – некоторые параметры можно оставить без возможности редактирования пользователем.

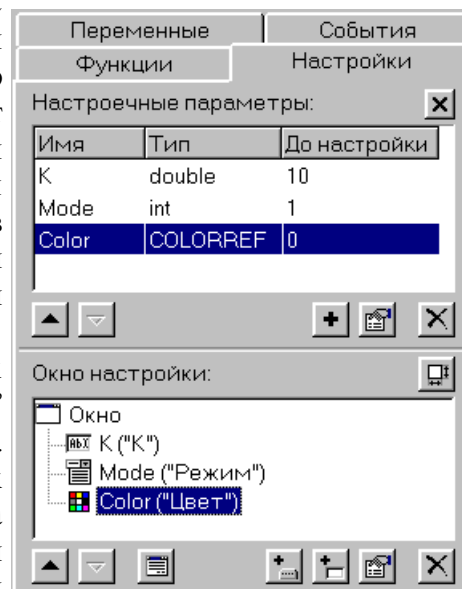




Рис. 341. Настроечные параметры блока

Список параметров блока в верхней части вкладки состоит из трех колонок:

- “Имя” – имя поля, которое будет создано в классе блока для хранения этого параметра. Это имя должно удовлетворять требованиям языка C++ (содержать только латинские буквы, цифры и знак подчеркивания и не начинаться с цифры) и не должно совпадать с именами статических и динамических переменных блока.
- “Тип” – тип параметра. Поддерживаются типы “double” (вещественное число), “int” (целое число), “BOOL” (логический тип, используемый в Windows), “COLORREF” (цвет, используемый в Windows) и “rdsbcppString” (собственный тип для работы со строками, используемый модулем автокомпиляции).
- “До настройки” – исходное значение параметра, которое он получает при подключении модели к блоку. Значение параметра может быть позднее изменено пользователем, если в окно настройки для этого параметра будет добавлено поле ввода.

Непосредственно под списком находятся кнопки, позволяющие добавлять, удалять и изменять параметры:

Кнопка	Действие
	Переместить выбранный параметр на одну позицию вверх по списку.
	Переместить выбранный параметр на одну позицию вниз по списку.
	Добавить новый параметр (открывает отдельное окно, см. рис. 342).

<i>Кнопка</i>	<i>Действие</i>
	Изменить выбранный параметр (открывает отдельное окно, см. рис. 343).
	Удалить выбранный параметр.

Порядок параметров в списке никак не отражается на работе модели, он нужен только для визуального упорядочения этих параметров в редакторе.

При добавлении нового параметра в список открывается окно, изображенное на рис. 342. В его верхней части задается имя поля внутри класса блока, в котором будет храниться параметр, тип этого параметра (выбирается из выпадающего списка) и его исходное значение. В нижней части окна, установив флажок “добавить для этой переменной поле ввода”, можно автоматически создать для добавляемого параметра поле ввода в окне настройки. Под этим флажком описывается тип поля ввода, его заголовок, размер и т.п. – все эти параметры будут объяснены далее, когда речь пойдет об окне настроек. Поле ввода для параметра не обязательно создавать вместе с самим параметром, его можно будет добавить позже.

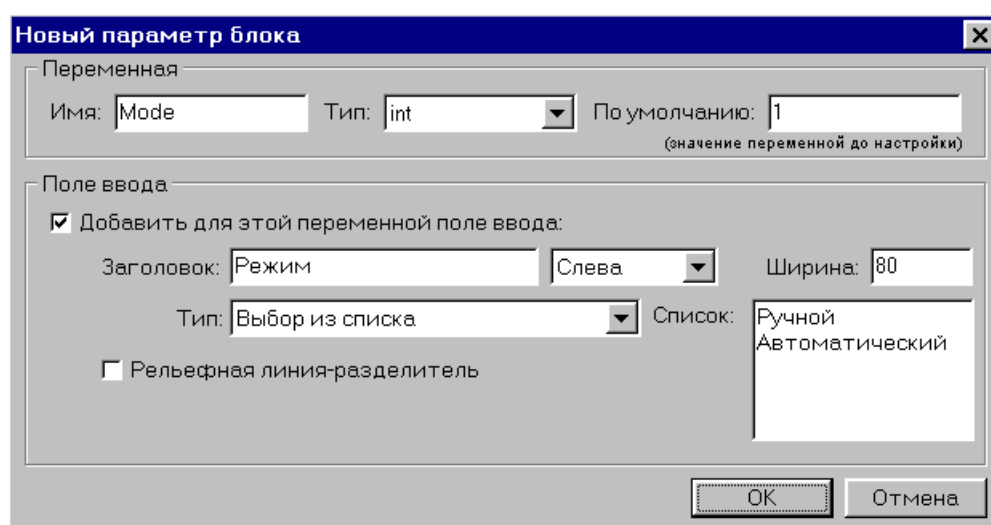


Рис. 342. Окно добавления нового параметра

В окне редактирования уже существующего параметра (рис. 343) можно изменить только имя, тип и значение параметра. Параметры его поля ввода в этом окне изменить нельзя – после создания это поле нужно редактировать в списке в нижней части вкладки.

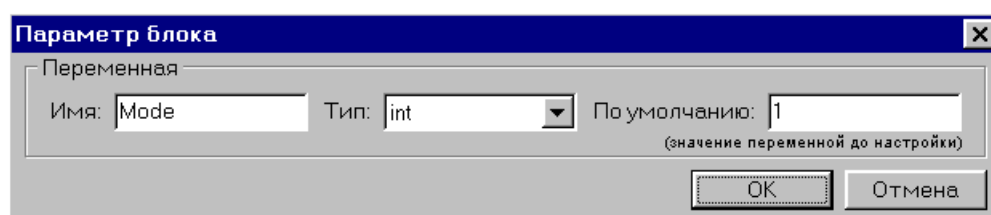


Рис. 343. Окно редактирования существующего параметра

Нижнюю часть вкладки занимает список полей ввода окна настройки блока. Список выводится в виде дерева – каждое поле ввода соединено линией с содержащим его элементом окна. Если в окно не были добавлены вкладки, все поля ввода будут расположены в списке друг под другом и соединены линиями с элементом “окно” (рис. 344). Если же вкладки были добавлены, каждое поле ввода в списке будет соединено с символом своей вкладки (рис. 345).

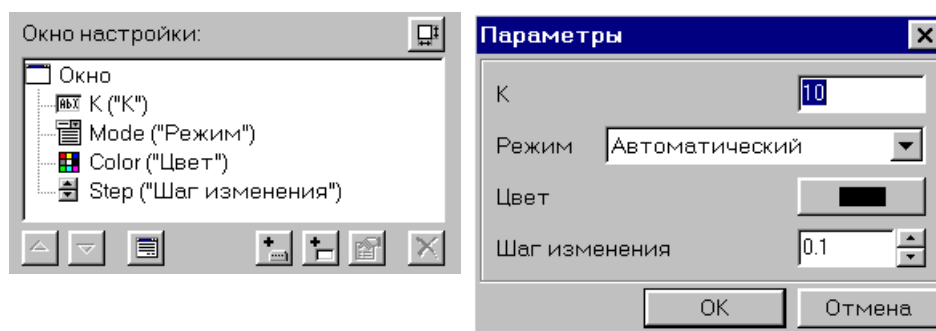


Рис. 344. Список полей ввода окна без вкладок (слева) и внешний вид окна (справа)

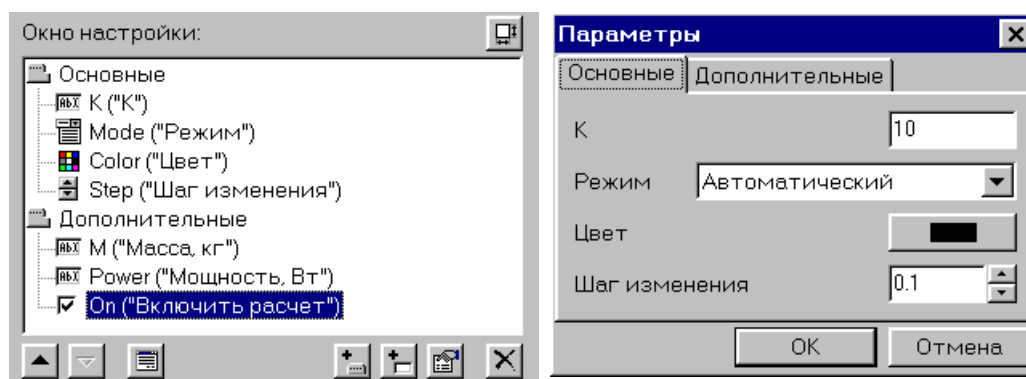



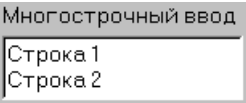

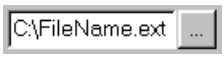

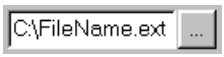


Рис. 345. Список полей ввода окна с вкладками (слева) и внешний вид окна (справа)







Для каждого поля ввода в списке изображается значок его типа, имя переменной, с которой связано поле, и, в скобках, заголовок этого поля, который видит пользователь. Порядок полей и вкладок в списке определяет их порядок в окне: поля будут располагаться в окне сверху вниз в том же порядке, что и в списке, а вкладки – слева направо. Модуль автокомпиляции поддерживает следующие типы полей ввода:

<i>Значок в списке</i>	<i>Описание</i>	<i>Внешний вид</i>	<i>Допустимые типы параметров</i>
	“Ввод” – простое поле ввода	<input type="text" value="10"/>	все типы
	“Ввод (+/-)” – поле ввода со стрелками для увеличения и уменьшения значения	<input type="text" value="0.1"/>	double, int
	“Ввод или выбор из списка” – поле ввода, в которое можно как вводить значение с клавиатуры, так и выбирать его из выпадающего списка	<div>Вариант 1 Вариант 1 Вариант 2</div>	все типы
	“Выбор из списка” – поле ввода, в котором можно выбирать значение из выпадающего списка (ввести с клавиатуры произвольное значение нельзя)	<div>Вариант 1 Вариант 1 Вариант 2</div>	все типы
	“Флаг” – флажок, который можно включать и выключать (заголовок этого поля всегда располагается справа от самого флажка)	<input checked="" type="checkbox"/> Заголовок	BOOL

<i>Значок в списке</i>	<i>Описание</i>	<i>Внешний вид</i>	<i>Допустимые типы параметров</i>
	“Выбор цвета” – кнопка, нажатие на которую открывает стандартный диалог выбора цвета		COLORREF
	“Многострочный ввод” – поле ввода, занимающее всю ширину окна или вкладки и позволяющее ввести несколько строк текста		rdsbcppString
	“Открытие файла” – поле ввода, в которое можно ввести имя файла с клавиатуры или выбрать это имя в стандартном диалоге открытия файла, вызываемом кнопкой с многоточием		rdsbcppString
	“Сохранение файла” – поле ввода, в которое можно ввести имя файла с клавиатуры или выбрать это имя в стандартном диалоге сохранения файла, вызываемом кнопкой с многоточием		rdsbcppString
T	“Надпись” – текстовая надпись, не имеющая поля ввода и не связанная с каким-либо настроечным параметром	Надпись	—

Для логических параметров (параметров типа BOOL) можно не только создавать отдельные поля ввода, но и использовать их для разрешения и запрещения ввода в другие поля. Пример такого использования приведен на рис. 346: логический параметр EnableStep привязан к разрешению поля ввода вещественной переменной Step с заголовком “шаг изменения”. При этом в списке полей ввода поле для EnableStep соединено линией с управляемым им полем, а в окне настроек перед полем ввода “шаг изменения” располагается флажок, значение которого связано с переменной EnableStep: ввести данные в это поле можно будет только тогда, когда флажок включен, то есть EnableStep имеет ненулевое значение (TRUE). Такая связь дополнительного флажка с полем устанавливается в параметрах управляемого поля ввода (см. рис. 349 на стр. 66, а также стр. 68).

Выше и ниже списка полей в нижней части вкладки “настройки” располагаются кнопки, позволяющие добавлять, удалять и редактировать поля ввода и вкладки в окне, а также задавать и просматривать внешний вид получающегося окна настройки:

<i>Кнопка</i>	<i>Действие</i>
	Показать или скрыть дополнительную панель для задания заголовка окна настройки и его размеров.
	Переместить выбранное поле или вкладку на одну позицию вверх по списку.
	Переместить выбранное поле или вкладку на одну позицию вниз по списку.
	Показать внешний вид окна настроек с заданными в списке полями ввода.
	Добавить новую вкладку (открывает отдельное окно, см. рис. 348).
	Добавить новое поле ввода (открывает отдельное окно, см. рис. 349).

Кнопка	Действие
	Изменить выбранное поле ввода или вкладку (открывает отдельное окно).
	Удалить выбранное поле ввода или вкладку.

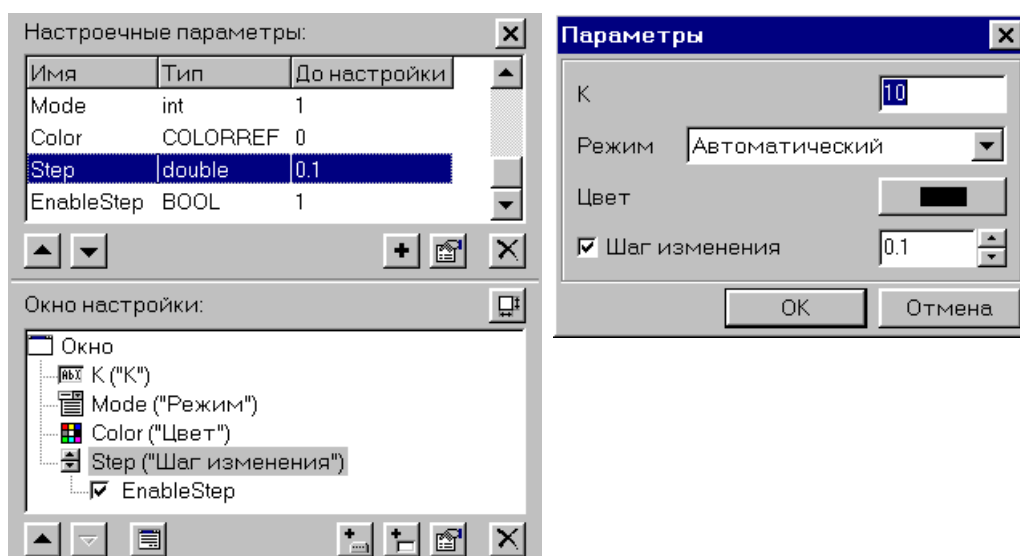


Рис. 346. Флаг разрешения для поля ввода “шаг изменения”: список параметров и полей в редакторе (слева) и внешний вид окна настройки (справа)

Кнопка, расположенная справа сверху от списка полей ввода, показывает или скрывает дополнительную панель, на которой можно ввести заголовок окна настроек, а также задать ширину и высоту окна в точках экрана (рис. 347). Первое нажатие кнопки показывает панель (кнопка при этом остается нажатой), второе – скрывает. Ширину и высоту окна задавать не обязательно: по умолчанию окно самостоятельно подстраивает свои размеры таким образом, чтобы все поля ввода уместились в нем. Тем не менее, если по каким-либо причинам необходимо точно задать один или оба размера окна, можно включить флажок “ширина” или “высота” и ввести соответствующий размер в поле рядом с ним.

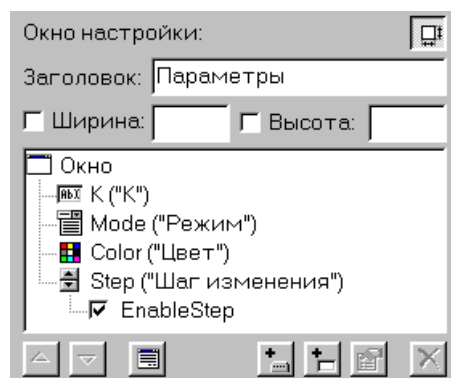


Рис. 347. Панель параметров окна настроек (над списком)

Кнопки перемещения полей и вкладок вверх и вниз по списку меняют положение этих полей и вкладок в окне. Чем выше поле ввода в списке, тем выше оно будет располагаться в окне или на своей вкладке, а чем выше в списке вкладка, тем левее она будет располагаться в окне. Вкладки в списке перемещаются вверх и вниз вместе со всеми своими полями. Поле ввода можно переместить и за пределы его вкладки, в этом случае оно перейдет на соседнюю. Если, например, поле ввода – самое верхнее на вкладке, нажатие кнопки перемещения вверх по списку переместит его в конец предыдущей вкладки.

Кнопка со всплывающей подсказкой “тест”, находящаяся справа от кнопок перемещения пунктов списка вверх и вниз, открывает окно настроек со всеми теми полями и вкладками, которые введены в список на данный момент. Все поля ввода и кнопки в этом окне работают, но измененные в нем параметры не запоминаются – это окно предназначено только для того, чтобы пользователь мог увидеть, как выглядят введенные им поля на экране.

Правее кнопки тестирования окна находятся кнопки добавления вкладок и полей. Исходно окно не содержит ни одной вкладки – корневой элемент списка называется “окно”, и все поля ввода в списке соединены линиями с ним (см. рис. 344 на стр. 63). Чтобы в окне появилась вкладка, необходимо нажать на кнопку добавления вкладки и ввести в открывшемся окне ее заголовок (рис. 348). При добавлении самой первой вкладки все уже имевшиеся в окне поля ввода помещаются на нее, следующие вкладки будут добавляться пустыми в конец списка.



Рис. 348. Окно для ввода заголовка вкладки

Чтобы добавить в окно поле ввода, необходимо нажать на соответствующую кнопку под списком и заполнить окно параметров добавляемого поля (рис. 349). Поля добавляются на выбранную в данный момент в списке вкладку или, если в списке выбрано поле ввода, на одну с ним вкладку в конец списка (после добавления поле можно перемещать вверх и вниз).

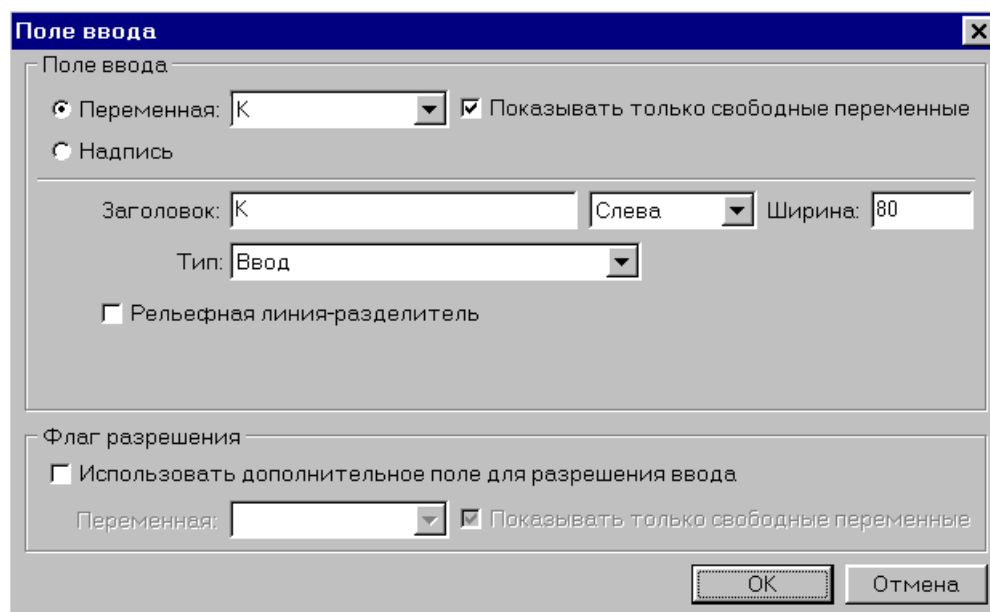


Рис. 349. Окно параметров поля ввода для простого поля

В верхней части этого окна необходимо выбрать один из взаимоисключающих флажков “переменная” или “надпись”. Выбор флажка “переменная” позволяет создать поле для ввода для какого-либо настроечного параметра, выбор флажка “надпись” создает в окне произвольную текстовую строку, никак не связанную с каким-либо параметром (в этой строке можно ввести, например, пояснения пользователю). Имя параметра, который будет редактироваться в этом поле, выбирается из выпадающего списка справа от флажка “переменная”. Если включен дополнительный флажок “показывать только свободные переменные”, в этот список попадут только те параметры, для которых еще не созданы поля ввода. Если флажок выключен, в список попадут все настроечные параметры блока. Следует иметь в виду, что стандартный модуль автокомпиляции не позволяет создавать для одного параметра несколько полей ввода, поэтому при выборе в выпадающем списке имени параметра, для которого уже есть поле ввода, старое поле ввода будет стерто (об этом будет выдано предупреждение пользователю).

В поле ввода “заголовок” вводится заголовок поля, то есть текст, который пользователь увидит рядом с полем ввода параметра. В выпадающем списке справа задается выравнивание текста заголовка в окне: слева (по левой границе окна или вкладки), по центру или справа (текст будет прижат к полю ввода параметра). В окне настроек заголовок будет располагаться слева от поля ввода, за исключением поля в виде флажка для логических параметров (у них заголовок располагается справа от флажка) и многострочных полей ввода (заголовок располагается над полем, а поле занимает всю ширину окна или вкладки). Если в верхней части окна выбран флажок “надпись”, заголовок и его выравнивание – единственное, что можно задать в окне.

Справа от заголовка указывается ширина поля ввода в точках экрана. Эту ширину обычно подбирают опытным путем – так, чтобы типичные значения параметров умещались в поле ввода. Проще всего, добавив поле ввода, вызывать его параметры и пробовать задавать разные значения ширины, каждый раз закрывая окно параметров кнопкой “ОК” и нажимая после этого кнопку “тест” для того, чтобы увидеть, как стало выглядеть окно настройки.

Непосредственно под заголовком из выпадающего списка выбирается тип поля ввода (см. стр. 63). Состав остальных полей окна зависит от выбранного типа. Для простого поля ввода, флага, выбора цвета и полей сохранения и загрузки файла окно параметров изображено на рис. 349, в нем нет дополнительных настроек, задающих его поведение. Для поля “ввод (+/-)” дополнительно задается шаг изменения параметра при нажатии на кнопки со стрелками, а также, если значение параметра необходимо ограничить, максимальное и минимальное его значения (рис. 350).

Заголовок:	Шаг изменения	Слева	Ширина:	80
Тип:	Ввод (+/-)		Шаг изменения:	0.1
<input type="checkbox"/> Рельефная линия-разделитель	<input checked="" type="checkbox"/> Минимум:			0
	<input checked="" type="checkbox"/> Максимум:			10

Рис. 350. Дополнительные настройки для поля типа “ввод (+/-)”

Для многострочного поля ввода задается его высота в точках экрана (рис. 351). Обычно ее подбирают так, чтобы в поле уместилось нужное число строк текста.

Заголовок:	Многострочный ввод	Слева	Ширина:	80
Тип:	Многострочный ввод		Высота:	40
<input type="checkbox"/> Рельефная линия-разделитель				

Рис. 351. Дополнительные настройки для поля типа “многострочный ввод”

Для полей “выбор из списка” и “ввод или выбор из списка” задается список вариантов – по одному варианту на строке (рис. 352). Если поле ввода “выбор из списка” связано с целым параметром, в этот параметр записывается индекс выбранного элемента списка (выбор первого варианта даст параметру значение 0, второго – 1, и т.д.), в противном случае в параметр будет записано само выбранное в списке значение.

Заголовок:	Режим	Слева	Ширина:	200
Тип:	Выбор из списка	Список:	Ручной Автоматический	
<input type="checkbox"/> Рельефная линия-разделитель				

Рис. 352. Дополнительные настройки для поля типа “выбор из списка”

Под выпадающим списком типа поля ввода находится флажок “рельефная линия-разделитель”. Если он включен, между этим полем ввода и следующим в окне будет нарисована горизонтальная линия во всю ширину окна или вкладки. Такими линиями можно разделять поля ввода, относящиеся к разным по смыслу группам параметров.

В нижней части окна параметров поля ввода (см. рис. 349 выше) располагается панель “флаг разрешения”, позволяющая добавить к заголовку поля ввода флажок, разрешающий и запрещающий ввод в это поле и связанный с отдельным логическим параметром. Пример такого флага разрешения приведен на рис. 346. Если включить на этой панели флажок “использовать дополнительное поле для разрешения ввода”, в выпадающем списке под ним можно будет выбрать один из уже созданных для блока логических (то есть типа `BOOL`) настроечных параметров. Как и в верхней части окна, справа от выпадающего списка находится дополнительный флажок “показывать только свободные переменные”: если он включен, в список попадут только те параметры, для которых еще не созданы поля ввода.

Заполнив список настроечных параметров блока и создав для них поля ввода (или используя их как флаги разрешения полей), пользователь автоматически получает работающее окно настройки, которое можно вызвать из контекстного меню блока. Изменив в параметрах блока реакцию на двойной щелчок в режиме редактирования (см. §2.9.1 части I), можно заставить это окно открываться по двойному щелчку. Никаких дополнительных действий по сохранению и загрузке настроечных параметров блоков выполнять не нужно: модуль автокомпиляции сам добавит необходимые функции в программу модели. Таким образом, блоки с такой моделью позволят пользователю менять их параметры и будут помнить их при сохранении и загрузке схемы. Несмотря на то, что у нескольких блоков схемы может быть одинаковая модель и, поэтому, одинаковый набор параметров, значения этих параметров у них будут разными, поскольку параметры сохраняются независимо для каждого блока. В функциях модели к настроечным параметрам можно обращаться по именам, как к обычным переменным – они оформляются модулем автокомпиляции как поля класса блока и доступны из всех реакция на события.

В §2.7.4 руководства программиста [1] описывается способ хранения настроечных параметров блока в значениях по умолчанию его статических переменных. Этот способ удобен тем, что 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. Многие стандартные блоки в РДС хранят настроечные параметры именно так. Стандартный модуль автокомпиляции не поддерживает этот способ. Дело в том, что все блоки с одной и той же моделью имеют один и тот же набор статических переменных, а, следовательно, и одинаковые значения по умолчанию для этих переменных. Если бы параметры хранились в значениях переменных по умолчанию, при каждой компиляции модели, когда модуль устанавливает структуру переменных для всех обслуживаемых блоков, введенные пользователем значения сбрасывались бы, что недопустимо.

Если сравнить приведенные выше типы полей ввода с §A.5.28.3 приложения к руководству программиста [2], можно заметить, что сервисные функции РДС поддерживают больше типов полей ввода, чем стандартный модуль автокомпиляции. Модуль автокомпиляции пользуется теми же самыми сервисными функциями, но в его интерфейс пользователя включены только самые простые поля. Если пользователю для каких-то целей необходимы не поддерживаемые модулем поля ввода для настроечных параметров, он может создавать окно настройки не с помощью заполнения вкладки “настройка” дополнительной панели окна редактора, а вручную, как описывается в §2.7.2 руководства программиста. Для этого в модель необходимо будет включить реакцию на вызов функции настройки (`RDS_BFM_SETUP`), и разместить в ней программу создания и открытия окна с нужными полями ввода.

§3.6.7. Параметры модели

Рассматривается ввод различных описаний и параметров модели, влияющих на ее общее поведение.

Возможности модели блока и особенности ее поведения зависят не только от реакций на события, введенных пользователем (см. §3.6.4 на стр. 46), но и от общих параметров модели, влияющих на формируемый модулем автокомпиляции текст программы. Не следует путать параметры модели с параметрами самого модуля, рассматриваемыми в §3.9 (стр. 328): хотя и те, и другие управляют формированием текста программы, параметры модели индивидуальны для каждой модели, а параметры модуля – общие для всех моделей, обслуживаемых этим модулем.

Чтобы изменить параметры модели, следует в окне ее редактора (см. §3.6.1 на стр. 32) вызвать пункт главного меню “модель | параметры модели”. При этом откроется окно с четырьмя вкладками, самая важная из которых – “компиляция” (рис. 353). На ней расположены флажки, большая часть которых управляет добавлением в программу модели тех или иных автоматически выполняемых действий.

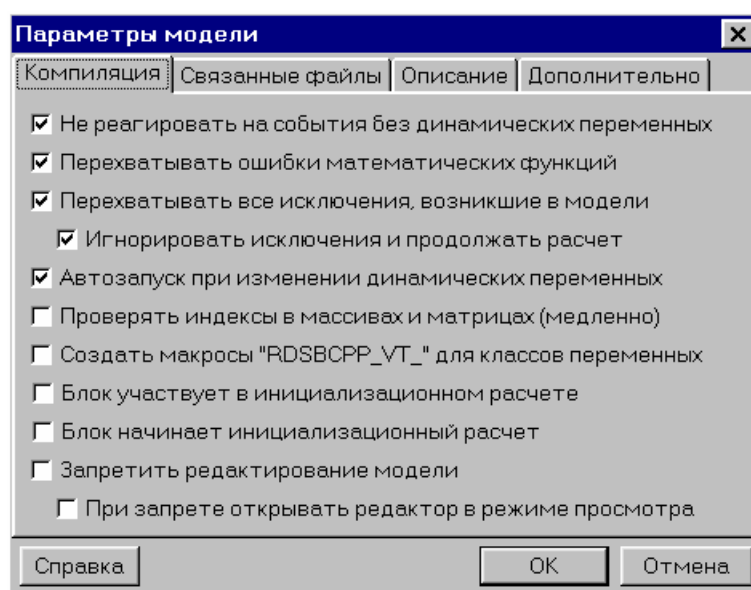


Рис. 353. Окно параметров модели – параметры исходного текста

Флажок “не реагировать на события без динамических переменных” отключает все реакции модели на события, если хотя бы одна из динамических переменных, занесенных пользователем в список на вкладке “переменные” (см. §3.6.3 на стр. 42), отсутствует. Динамические переменные, в отличие от статических, создаются не заранее, а в процессе работы схемы, и могут быть удалены создавшими их блоками в любой момент. Обращение к отсутствующей в данный момент динамической переменной вызывает критическую ошибку, поэтому модели блоков должны проверять их существование перед каждым обращением. Установка этого флажка добавляет в программу модели проверку существования всех указанных в модели динамических переменных перед вызовом реакции на каждое событие. Если эта проверка не проходит, программа модели немедленно завершается, и введенные пользователем фрагменты программ не выполняются. В этом случае внутри них можно обращаться к любым динамическим переменным, не заботясь об их существовании: если хотя бы одной переменной не будет, вся реакция не будет выполнена. Для большинства моделей такое поведение подходит – действительно, если одна из необходимых для работы блока переменных отсутствует, работа модели бессмысленна – поэтому в новых пустых моделях этот флажок установлен по умолчанию. Однако, если при отсутствии динамической переменной модель все равно должна предпринимать какие-либо действия (например,

выводить пользователю сообщение об ошибке, или брать данные из статической переменной), флажок следует отключить, и проверять существование каждой динамической переменной вручную, при помощи функции-члена ее класса `Exists` (см. стр. 133).

Флажок “перехватывать ошибки математических функций” добавляет в программу модели специальные функции, которые предотвращают прерывание выполнения модели при возникновении каких-либо ошибок в стандартной математической библиотеке. Какие именно функции необходимо вставить в программу при установке этого флажка, определяют параметры модуля автокомпиляции (см. §3.9.10 на стр. 356). По умолчанию в программу вставляется функция с названием `_matherr`, которая заменяет результат операции, которая не может быть выполнена, на ноль в случае ошибки потери точности, и на специальное значение, предусмотренное для индикации ошибки операции с вещественными числами, при любой другой ошибке (это значение хранится в глобальной переменной `rdsbcppHugeDouble`). Следует помнить, что данный флажок управляет только перехватом ошибок математической библиотеки, а не любых операций с вещественными числами: например, деление на ноль он отследить не в состоянии. Подробнее о перехвате математических ошибок можно прочесть в руководстве по используемому компилятору.

Флажок “перехватывать все исключения, возникшие в модели”, включает всю функцию модели блока внутрь оператора `try...catch`. Если внутри этого оператора возникнет ошибка, приводящая к прерыванию работы модели, пользователю будет выведено сообщение об этом, и расчет, если он запущен, будет остановлен. Если не устанавливать этот флажок, возникшая ошибка все равно будет перехвачена РДС, и пользователю будет выведено стандартное диагностическое сообщение (рис. 354) с предложением завершить работу.

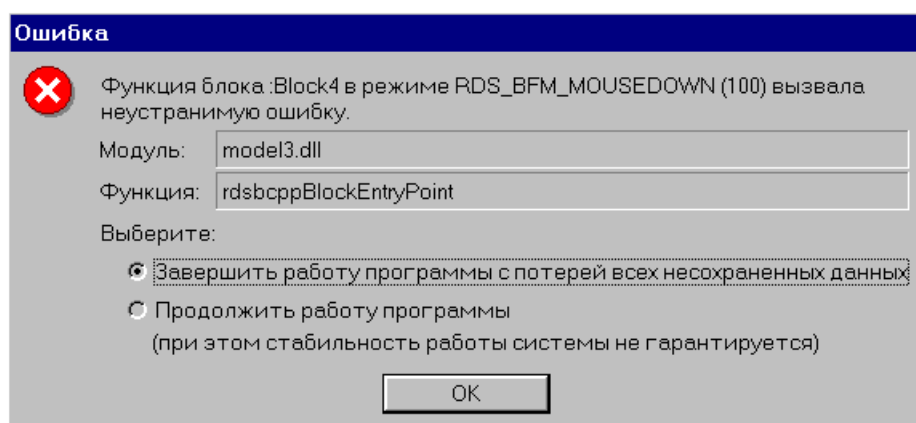


Рис. 354. Стандартное сообщение РДС о критической ошибке в модели

С установленным флажком поведение модели будет зависеть от дополнительного флажка “игнорировать исключения и продолжать расчет”: если он установлен, модель продолжит работу, несмотря на исключение, в противном случае пользователю будет выведено короткое сообщение, причем модель не позволит пользователю завершить работу с потерей всех не сохраненных изменений в схеме. Использование общего перехвата исключений, управляемого данными двумя флажками, следует рассматривать только как временную меру: если в модели содержатся ошибки, вызывающие исключения, разработчик должен самостоятельно перехватывать их именно в тех местах, где они возникают, и обрабатывать должным образом.

Флажок “автозапуск при изменении динамических переменных” добавляет в программу модели автоматическое присвоение единицы первой сигнальной переменной блока, всегда используемой в качестве команды на запуск модели, при изменении любой динамической переменной, внесенной пользователем в список на вкладке “переменные” дополнительной панели редактора (см. §3.6.3 на стр. 42). Очень многие блоки должны

выполнять одинаковые действия и при изменении их статических переменных (то есть при поступлении данных на их входы), и при изменении динамических переменных, с которыми эти блоки связаны. Чтобы не записывать одну и ту же программу два раза и в реакции на такт расчета (событие RDS_BFM_MODEL), и в реакции на изменение динамических переменных (событие RDS_BFM_DYNVARCHANGE (см. список событий блоков в §3.6.4 на стр. 46), можно включить этот флажок и записать необходимые действия только в реакции на такт расчета. При этом, как только какая-либо из связанных с блоком динамических переменных изменится, будет автоматически взведен сигнал запуска модели, и в ближайшем такте расчета блок выполнит свою программу. Разумеется, если в параметрах блоков с данной моделью включен запуск каждый такт, модель блока будет принудительно запускаться в каждом такте расчета независимо от изменений каких-либо переменных, и в установке данного флажка нет смысла.

Флажок “проверять индексы в массивах и матрицах (медленно)” добавляет в формируемую модулем автокомпиляции программу модели специальные функции для автоматической проверки допустимости индексов всех массивов и матриц, являющихся статическими и динамическими переменными блока. Если флажок выключен, то, как и принято в языке C, такая проверка не производится – это ускоряет работу программы. Однако, в этом случае при попытке обращения к элементу за пределами текущего размера массива или матрицы (например, при обращении к двадцатому элементу массива, в котором всего десять элементов), возникнет критическая ошибка. Правила написания моделей блоков для РДС требуют обязательной проверки допустимости индекса перед обращением к массиву или матрице, поэтому в правильно написанной модели устанавливать этот флажок не нужно: все проверки должны быть выполнены программистом явно. Тем не менее, если модель по непонятным причинам постоянно выдает исключения и отказывается работать, установка этого флажка может дать понять, в чем именно дело. Если после его установки модель начала выдавать сообщения о недопустимом индексе с указанием имени модели и имени массива или матрицы, значит, ошибки возникают именно из-за недопустимых индексов, и их проще будет найти и исправить. Следует, однако, помнить, что действие этого флажка распространяется только на статические и динамические переменные блока, для работы с которыми модуль автокомпиляции формирует специальные классы, в которые и добавляются функции автоматической проверки индексов. Для стандартных описаний массивов и матриц в синтаксисе языка C проверки не производится: если, например, внутри какой-либо реакции блока описать массив из десяти элементов оператором “double a[10];”, а затем выполнить оператор “a[20]=1;”, исключение возникнет независимо от состояния описываемого флажка, и никаких диагностических сообщений выдано не будет.

Флажок “создать макросы RDSBCPP_VT_ для классов переменных” добавляет для каждой статической и динамической переменной блока макроопределение для имени класса, сгенерированного модулем автокомпиляции для доступа к этой переменной (автоматическая генерация классов для доступа к переменным описана на стр. 86). Макроопределение имеет следующий вид:

```
#define RDSBCPP_VT_ИмяПеременной ИмяКласса
```

(здесь “ИмяПеременной” – имя переменной блока, “ИмяКласса” – имя класса, созданного для этой переменной). Например, для доступа к статическим переменным типа “матрица double” автоматически создается класс с именем “rdsbcstMDouble”. Если в блоке есть статическая матрица с именем “A”, то при включенном флажке создания макросов в текст программы будет добавлена следующая строка:

```
#define RDSBCPP_VT_A rdsbcstMDouble
```

В большинстве случаев разработчику модели можно не задумываться, как именно называется класс, созданный для работы с переменной, поскольку к переменным внутри реакций на события можно обращаться просто по именам. Однако, если необходимо создать отдельную

функцию, которая будет работать с переменными блока, для описания параметров этой функции необходимо будет указать типы переменных, что проще всего сделать с использованием автоматически созданных макросов. Пусть, например, требуется создать функцию, суммирующую все элементы статической матрицы вещественных чисел, причем в блоке есть переменная “A” нужного типа “матрица double”. Тогда функцию можно записать следующим образом:

```
double SumMatr(RDSBCPP_VT_A &matr)
{ double sum=0.0;
  for(int c=0;c<matr.Cols();c++)
    for(int r=0;r<matr.Rows();r++)
      sum+=matr[r][c];
  return sum;
}
```

Разместить эту функцию нужно будет в описаниях после класса блока (см. §3.8.1 на стр. 282), поскольку строки с макроопределениями добавляются после описания каждой переменной, т. е. внутри класса блока.

Флажок “блок участвует в инициализационном расчете” разрешает вызов модели блока для реакции на такт расчета не только в обычном режиме расчета, но и при проведении предварительного расчета в момент выхода из режима редактирования. Предварительный расчет обычно выполняется по запросу начальных блоков алгебраических цепочек (например, полей ввода) и позволяет уменьшить число тактов переходного процесса в таких цепочках. При этом блоки в цепочке последовательно вызываются для обработки значения, поступившего с предыдущего блока, и правильное значение на конечном блоке цепочки устанавливается не через несколько тактов после начала расчета, как обычно (см. рис. 10 в §1.4 части I), а непосредственно перед запуском расчета. Флажок предварительного расчета следует устанавливать только в том случае, если блок должен вычислять свои выходные значения немедленно при изменении входных. Если, например, блок моделирует какой-либо длящийся во времени процесс (см. §3.7.4 на стр. 145), включать его в предварительный расчет не следует, поскольку он должен вычислять выходы не при изменении входов, а при изменении динамической переменной времени. Установка этого флажка вызывает взведение флага RDS_INITCALC в поле Flags структуры данных блока RDS_BLOCKDATA (см. стр. 88). В абсолютном большинстве простых моделей можно обойтись и без предварительного расчета, поскольку задержка данных на несколько тактов, как правило, не влияет на работоспособность схемы.

Флажок “блок начинает инициализационный расчет” устанавливается у блоков, с которых обычно начинаются алгебраические цепочки, чтобы выходные значения этих блоков были обработаны этой цепочкой в момент выхода из режима редактирования. Установка этого флажка вызывает взведение флага RDS_INITCALCFIRST в структуре данных блока.

Флажок “запретить редактирование модели” блокирует все операции по изменению параметров, переменных, функций и фрагментов программ модели. После его установки модель можно только просматривать, но нельзя редактировать. Это может оказаться полезным при передаче модели другому пользователю, чтобы он случайно не изменил ее. При попытке вызвать редактор модели пользователю будет предложено либо открыть окно редактора в режиме просмотра, либо сделать копию модели для ее изменения. Если установить дополнительный флажок “при запрете открывать редактор в режиме просмотра”, окно редактора будет открываться в режиме просмотра без запроса. Чтобы снова редактировать модель, этот флажок необходимо будет сбросить.

Вкладка “связанные файлы” окна параметров модели (рис. 355) содержит список файлов, при изменении которых модуль автокомпиляции будет считать, что модель необходимо скомпилировать заново. Как правило, это файлы заголовков, включаемых в

программу модели. Если, например, в модели используются описания из файла “AuxDefs.h”, и в этот файл внесены изменения, модель необходимо компилировать заново, хотя сама модель не менялась.

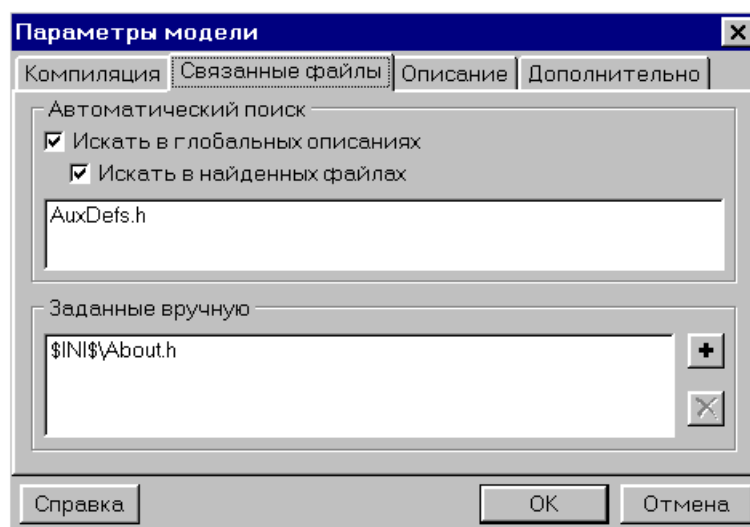


Рис. 355. Окно параметров модели – связанные файлы

Верхнюю часть вкладки занимает автоматически созданный список файлов, ссылки на которые (то есть команды препроцессора “#include”) модуль автокомпиляции нашел в глобальных описаниях, внесенных пользователем в модель (см. §3.8.1 на стр. 282). Для формирования этого списка должен быть включен флажок “искать в глобальных описаниях”. Если при этом включен еще и флажок “искать в найденных файлах”, в список попадут не только файлы, найденные в глобальных описаниях, но и файлы, на которые ссылаются эти найденные файлы – таким образом, в список будут добавлены все цепочки ссылающихся друг на друга файлов заголовков. Пользователь не может самостоятельно добавлять файлы в этот список.

Нижнюю часть вкладки занимает список связанных файлов, создаваемый пользователем вручную. В него обычно включаются файлы, которые модуль автокомпиляции не нашел автоматически, или которые связаны с моделью каким-либо образом, отличным от команды “#include” в описаниях. Для работы со списком используются кнопки:

<i>Кнопка</i>	<i>Действие</i>
	Добавить новый файл (открывается стандартный диалог выбора файла).
	Удалить выбранный в списке файл.

Для файлов, находящихся в стандартных папках РДС, имена этих папок заменяются на символические обозначения (полный список символических обозначений стандартных папок приводится в руководстве программиста [1]). На рис. 355, например, добавленный в список файл “About.h” находится в стандартной папке настроек РДС (обозначение “\$INIS”).

На вкладке “описание” окна параметров модели (рис. 356) вводится текст описания модели и информация о ее версии. Описание модели – это произвольный многострочный текст, он никак не влияет на работу и обычно используется для ввода каких-либо комментариев. Этот текст также отображается при создании модели по шаблону в нижней части окна выбора шаблона (см. рис. 316 на стр. 22).

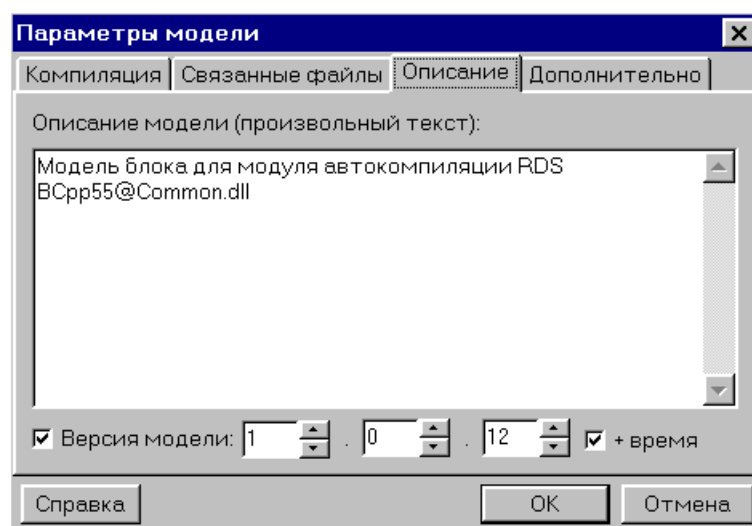


Рис. 356. Окно параметров модели – описание и версия

Для включения в модель информации о ее версии следует установить флажок “версия модели” и ввести в поля ввода справа от него три числа, которыми в РДС обозначаются версии всех моделей и библиотек. Первое число считается старшим номером версии (допустимый диапазон – 0...127), второе – младшим (0...255), третье – номером сборки (0...32767). Номер версии позволяет пользователю сравнивать разные версии модели друг с другом, чтобы выбрать из них самую последнюю. При сравнении версий сначала сравниваются старшие номера, и модель с большим старшим номером считается более поздней. Если старшие номера одинаковы, сравниваются младшие, и более поздней считается модель с большим младшим номером. Если же и младшие номера одинаковы, сравниваются номера сборки. Например, модель с версией “1.0.0” сделана позднее модели с версией “0.100.1”, а модель с версией “1.2.3” сделана позднее модели с версией “1.2.2”. Дополнительный флажок “+ время” позволяет включить в модель не только информацию о версии, но и информацию о времени последнего изменения. И номер версии, и время последнего изменения, если эта информация есть в модели, отображаются на вкладке “компиляция” окна информации о загруженной схеме (см. §2.17 части I). Кроме того, эта же информация сообщается РДС при загрузке модели блока в память (для этого используется сервисная функция `rdsReportVersion` и макросы `RDS_INTVERSION` и `RDS_DWORDVERDATE`), и, поэтому, также отображается на вкладке “используемые DLL” окна информации о схеме. Следует помнить, что номер версии не добавляется в параметры создаваемого файла DLL, поэтому, с точки зрения Windows, а также сторонних программ создания установочных пакетов, созданные модулем автокомпиляции файлы DLL не имеют версий.

На последней вкладке окна параметров модели, “дополнительно” (рис. 357), собраны редко используемые параметры, предназначенные для создания сложных блоков, программно изменяющих свои собственные модели. Здесь примеры таких блоков не приводятся.

Флажок “не останавливаться при предупреждениях” автоматически закрывает окно компиляции (рис. 320 на стр. 24), если в отчете, выданном компилятором, содержатся предупреждения, но нет ошибок. По умолчанию, т. е. если этот флажок не установлен, при наличии предупреждений окно компиляции остается на экране, отображая общее количество обнаруженных предупреждений и привлекая внимание пользователя, а после того, как он вручную закроет окно, открывается редактор модели со списком обнаруженных предупреждений. Если сложный блок каким-либо образом самостоятельно модифицирует

свою модель, в этой программно сформированной модели могут встретиться ненужные или дублирующие фрагменты, которые не влияют на работу модели, но распознаются компилятором и помещаются в его отчет как предупреждения: например, наличие не использованной переменной или повторное присвоение значения. Поскольку модель была сформирована программно, пользователю эти предупреждения ничего не скажут, и он, вероятнее всего, не сможет их исправить. Поэтому разработчикам самомодифицирующихся моделей блоков следует включать этот флажок.

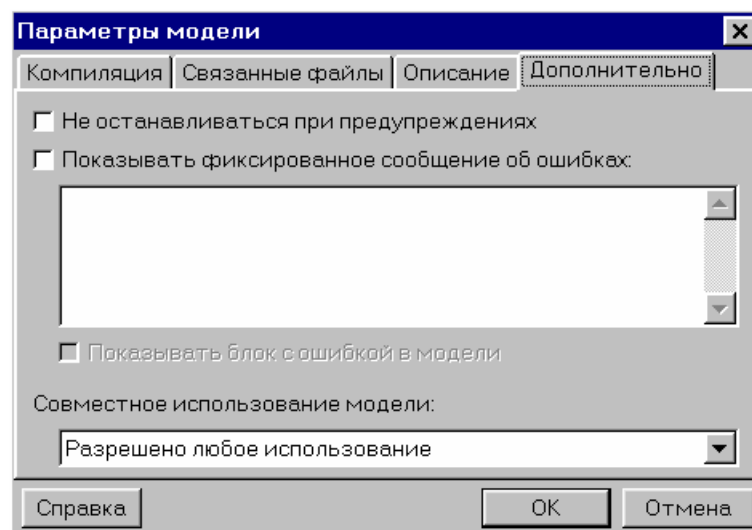


Рис. 357. Окно параметров модели – дополнительные параметры

Флажок “показывать фиксированное сообщение об ошибках” тесно связан с предыдущим. По умолчанию список ошибок, обнаруженных компилятором, выдается в нижней части окна редактора модели (см. §3.6.1 на стр. 32). Тексты сообщений об ошибках при этом формируются самим компилятором. Если модель создается блоком программно, сами сообщения и строки программы, к которым они относятся, вероятнее всего, ничего не скажут пользователю. Чаще всего блоки, программно формирующие свои модели, собирают эти модели из каких-то отдельных фрагментов, введенных пользователем, т. е., фактически, работают специализированной “надстройкой” над модулем автокомпиляции, где пользователь вводит не реакции на системные события, а, например, какие-либо функции, используемые блоком для формирования своей сложной вычислительной программы. Если собранная на основе этих фрагментов модель не компилируется, это указывает на ошибки во введенных пользователем текстах, но исправлять их нужно не в редакторе модели, а в специализированном интерфейсе, который блок предоставляет пользователю для ввода. В таких случаях лучше показать пользователю не сообщение об ошибке, выданное компилятором, а другое, поясняющее, что ошибки возникли в программно сформированной модели и предлагающее пользователю варианты действий по ее исправлению. Для вывода такого сообщения и служит данный флажок, а само сообщение вводится в многострочное поле ввода под ним. Если флажок включен, редактор модели не откроется, а вместо него появится окно с введенным текстом сообщения. Дополнительный флажок “показывать блок с ошибкой в модели” добавляет в это окно кнопку, нажав на которую пользователь сможет увидеть в схеме блок, при компиляции модели которого возникли ошибки.

Выпадающий список “совместное использование модели” управляет возможностью подсоединения одного файла модели к нескольким блокам в одной схеме и в разных схемах. Возможны следующие варианты:

- “Разрешено любое использование” – один и тот же файл модели может использоваться в нескольких блоках как в одной, так и в разных схемах. Это самый распространенный

вариант использования моделей, он установлен по умолчанию. При копировании блока с такой моделью пользователю выдается запрос, должна ли копия блока использовать тот же самый файл модели, или нужно сделать его копию (см. §3.5 на стр. 28). При стирании блока из схемы файл его модели всегда остается на диске.

- “Модель используется только этой схемой” – один и тот же файл модели может использоваться несколькими блоками, принадлежащими одной схеме, но не может использоваться в других схемах (обычно файл модели при этом находится в одной папке с файлом схемы). При копировании блока с этой моделью в пределах схемы пользователю выдается запрос на копирование файла модели, при копировании блока в другую схему файл модели копируется без запроса. При удалении последнего в схеме блока, использующего этот файл модели, файл модели стирается.
- “Модель используется единственным блоком этой схемы” – файл модели связан с конкретным блоком схемы (обычно файл модели при этом находится в одной папке с файлом схемы). При копировании этого блока автоматически создается и копия файла модели. При удалении блока файл модели тоже удаляется.
- “Модель используется всеми блоками без запросов” – один и тот же файл модели может использоваться в нескольких блоках как в одной, так и в разных схемах, при этом при копировании блока созданная копия автоматически подключается к тому же файлу модели без запросов пользователю.

§3.6.8. Установка параметров блоков с автокомпилируемой моделью

Описывается способ одновременной установки параметров всех блоков с редактируемой моделью при помощи окна групповой установки, вызываемого непосредственно из редактора модели.

Поведение блока в РДС зависит не только от функции его модели, но и от индивидуальных настроек параметров этого блока, которые могут изменять или запрещать некоторые его реакции. Например, будет ли блок выполнять какие-либо действия при щелчках мыши на его изображении зависит не только от того, есть ли в его модели реакция на нажатие кнопки мыши и что она делает, но и от того, разрешено ли блоку вообще реагировать на действия мышью. Если реакция блока запрещена, он не будет выполнять никаких действий при щелчках мышью, независимо от того, как написана его модель.

Параметры блоков с автокомпилируемыми моделями можно устанавливать точно так же, как и параметры любых других блоков – при помощи окна параметров (см. §3.4 на стр. 27). Однако, это не очень удобно: как правило, параметры всех блоков с одной и той же моделью делают одинаковыми, и открывать окно параметров для каждого из таких блоков и вносить в него одни и те же изменения было бы неразумной тратой времени. Гораздо лучше в этом случае воспользоваться одновременной установкой параметров нескольких блоков (см. §2.15.3 части I). Но и в этом случае необходимо, во-первых, предварительно выделить блоки с нужной моделью при помощи функции выделения блоков по критерию, и, во-вторых, если блоки с этой моделью находятся в разных подсистемах, использовать функцию пакетной обработки (эти функции описаны в части I в §2.15.1 и 2.15.4 соответственно). Чтобы упростить работу, в редактор модели встроена функция групповой установки параметров всех блоков с данной моделью во всех подсистемах загруженной схемы. Для вызова этой функции необходимо выбрать в редакторе пункт меню “модель | установка параметров блоков” или нажать соответствующую ему кнопку.

Окно, открывающееся при вызове групповой установки из редактора модели (рис. 358) – это то же самое окно, которое вызывается пунктом главного меню РДС “редактирование | групповая установка”. Однако, поскольку параметры будут устанавливаться только у блоков с автокомпилируемой моделью, часть вкладок в этом окне будет отсутствовать, и часть параметров будет недоступна для установки. Например, в нем нельзя изменить структуру переменных блока (это делается в самом редакторе модели на

вкладке “переменные” его боковой панели, см. §3.6.2 на стр. 38) или подключить к нему другую модель. Будут также отсутствовать все вкладки установки параметров связей. Окно групповой установки было подробно описано в §2.15.3 части I, поэтому здесь его функции будут описаны коротко.

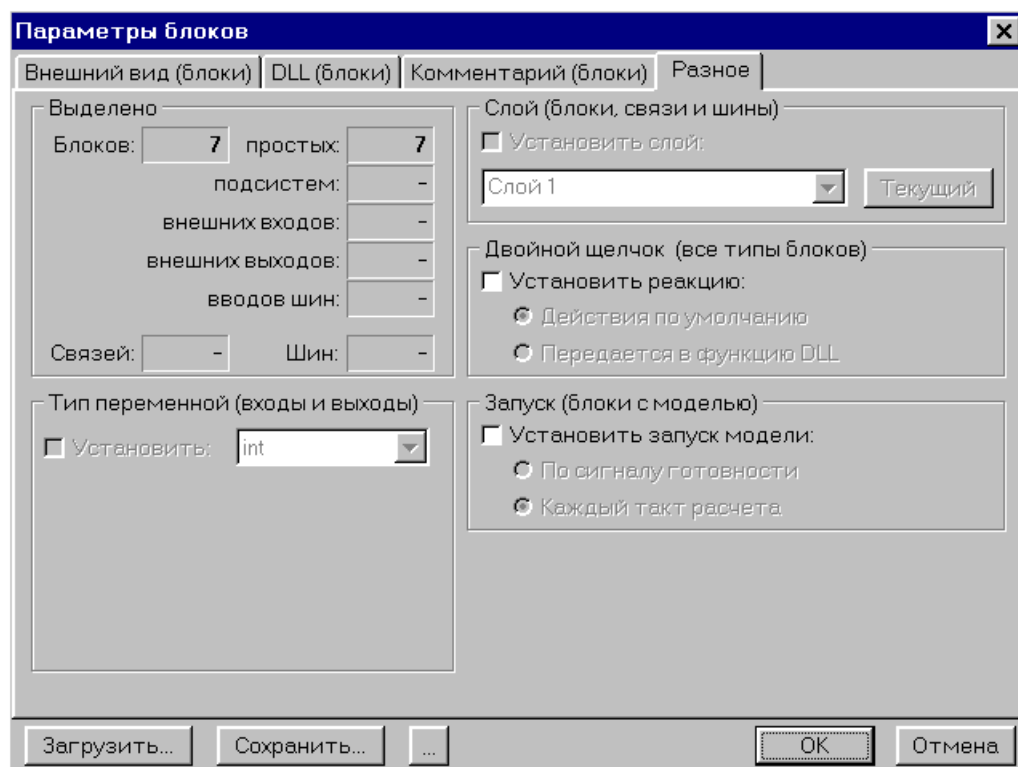


Рис. 358. Окно установки параметров блоков – вкладка “разное”

На вкладке “разное”, как обычно, отображается общее число выделенных блоков (в данном случае – блоков с данной моделью). На ней можно установить:

- реакцию блока на двойной щелчок в режиме редактирования: при выборе варианта “действия по умолчанию” будет открываться окно параметров блока, при выборе “передается в функцию DLL” – окно настройки, если оно у блока есть (создание окна настройки блока с автокомпилируемой моделью описано в §3.6.6 на стр. 60);
- способ запуска модели блока в режиме расчета: при выборе варианта “по сигналу готовности” модель будет запускаться только при ненулевом значении первой сигнальной переменной блока (то есть при явной установке сигнала запуска для данного блока), при выборе “каждый такт расчета” – принудительно в каждом такте.

Панель установки слоя блока отключена, поскольку блоки с данной моделью могут находиться в разных подсистемах с разным составом слоев, и поместить их все на один и тот же слой невозможно. Перемещать блоки со слоя на слой следует индивидуально в каждой подсистеме. Кроме того, на вкладке “разное” отключена и установка типа переменной для внешних входов и выходов, поскольку среди блоков с автоматически компилируемыми моделями внешние входы и выходы практически не встречаются.

На вкладке “внешний вид” (рис. 359) устанавливаются параметры, задающие изображение блоков в окне подсистемы: векторная картинка или простой прямоугольник с текстом и их настройки, имена переменных, управляющих перемещением, поворотом, масштабированием и скрытием изображения блока, включение и выключение отображения имени блока и т.п. Все эти параметры доступны для установки: тот факт, что они будут применены к блокам с автокомпилируемой моделью, находящимся в разных подсистемах, никак на них не влияет.

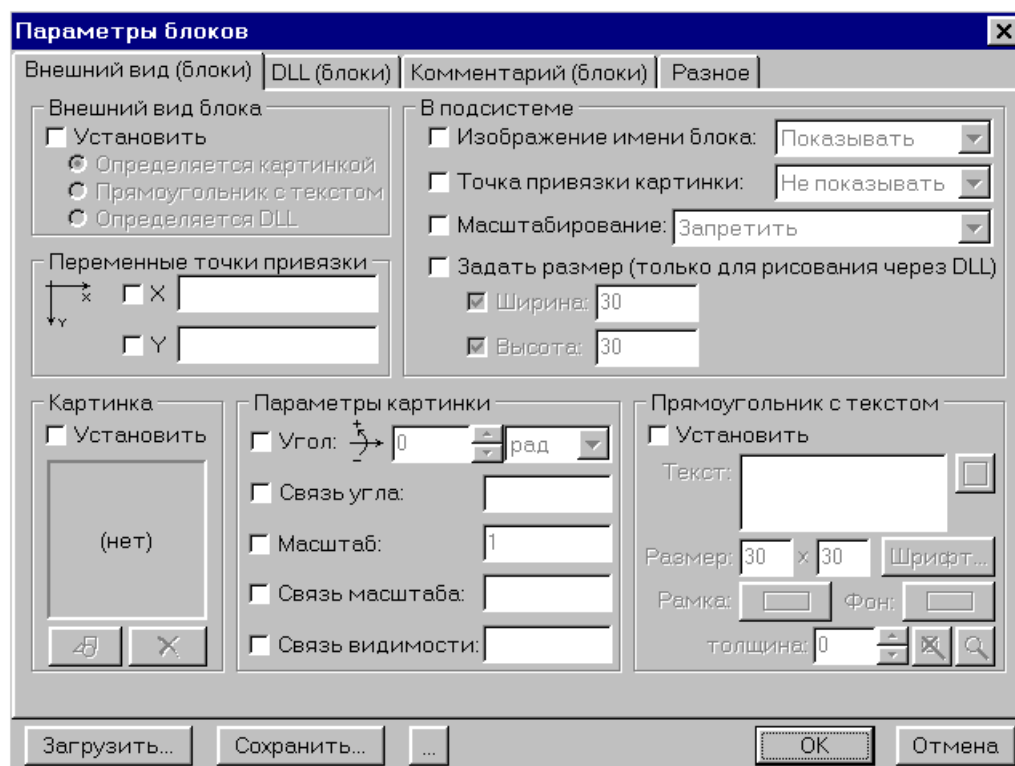


Рис. 359. Окно установки параметров блоков – вкладка “внешний вид”

На вкладке “DLL” (рис. 360) можно включить и отключить некоторые реакции модели блока.

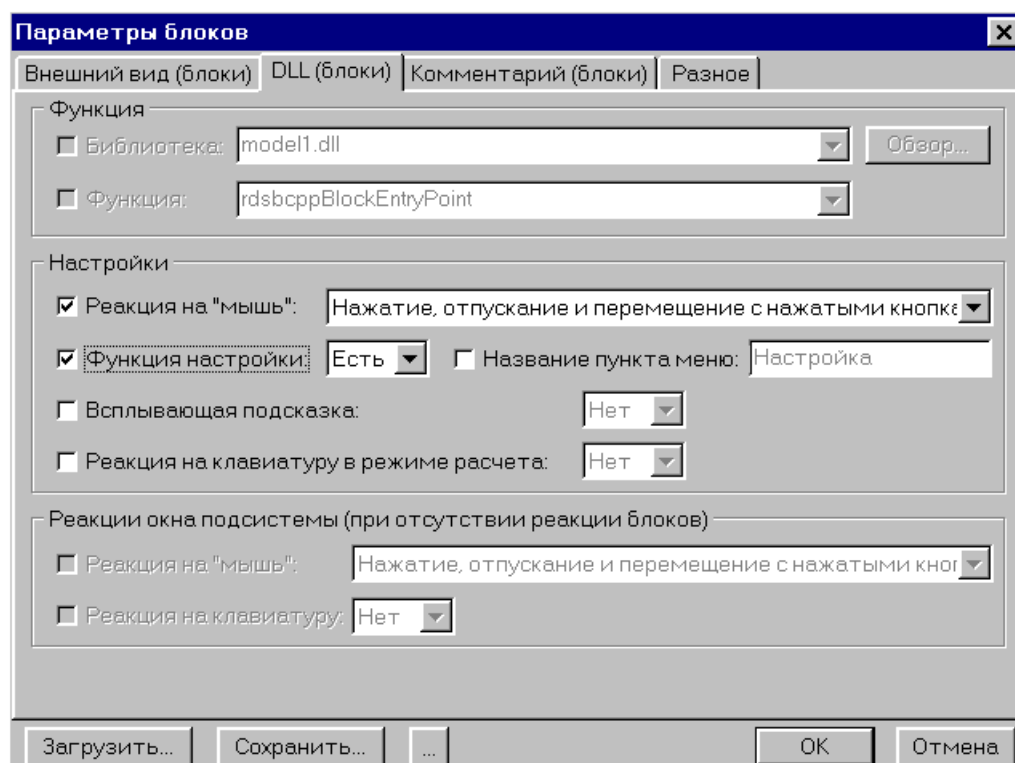


Рис. 360. Окно установки параметров блоков – вкладка “DLL”

Панель “функция”, на которой задается имя файла DLL с моделью блока и имя функции этой модели, будет отключена, поскольку для блоков с автокомпилируемыми

моделями эти параметры устанавливаются модулем автокомпиляции и пользователь не может в них вмешиваться. Изменить можно следующие параметры:

- “Реакция на мышь” (“отсутствует” / “действия с нажатыми кнопками” / “все действия”) – разрешает, запрещает или ограничивает реакцию блока на мышь в режимах моделирования и расчета. Для того, чтобы блок реагировал на действия пользователя мышью, кроме разрешения такой реакции необходимо ввести в его модель фрагмент программы, который будет выполняться при этих действиях (см. §3.7.11 на стр. 226).
- “Функция настройки” (“есть” / “нет”) – задает наличие у модели функции для открытия окна настройки параметров блока. В редакторе модели блока должны быть либо описаны параметры и окно настройки (см. §3.6.6 на стр. 60), либо введена реакция на событие RDS_BFM_SETUP, в которой это окно будет открываться вручную при помощи сервисных функций РДС.
- “Название пункта меню” – задает текст пункта меню, вызывающего окно настройки блока.
- “Всплывающая подсказка” (“есть” / “нет”) – разрешает или запрещает блоку реагировать на событие RDS_BFM_POPUPHINT, возникающее при задержке курсора над изображением блока. В реакции на это событие необходимо вернуть в РДС текст всплывающей подсказки (см. §3.7.9 на стр. 211).
- “Реакция на клавиатуру” (“есть” / “нет”) – разрешает или запрещает функции модели реагировать на нажатие и отпускание клавиш в режимах расчета и моделирования. В модель блока при этом необходимо ввести реакции на соответствующие события.

На панели “реакции окна подсистемы” можно управлять реакцией на мышь и клавиатуру окна подсистемы, если это окно открыто и ни один блок внутри него не среагировал на эти действия. Эта панель не будет отключена только для подсистем с автокомпилируемыми моделями, которые встречаются крайне редко.

Вкладка “комментарий” (рис. 361) позволяет изменить комментарий блоков с данной моделью.

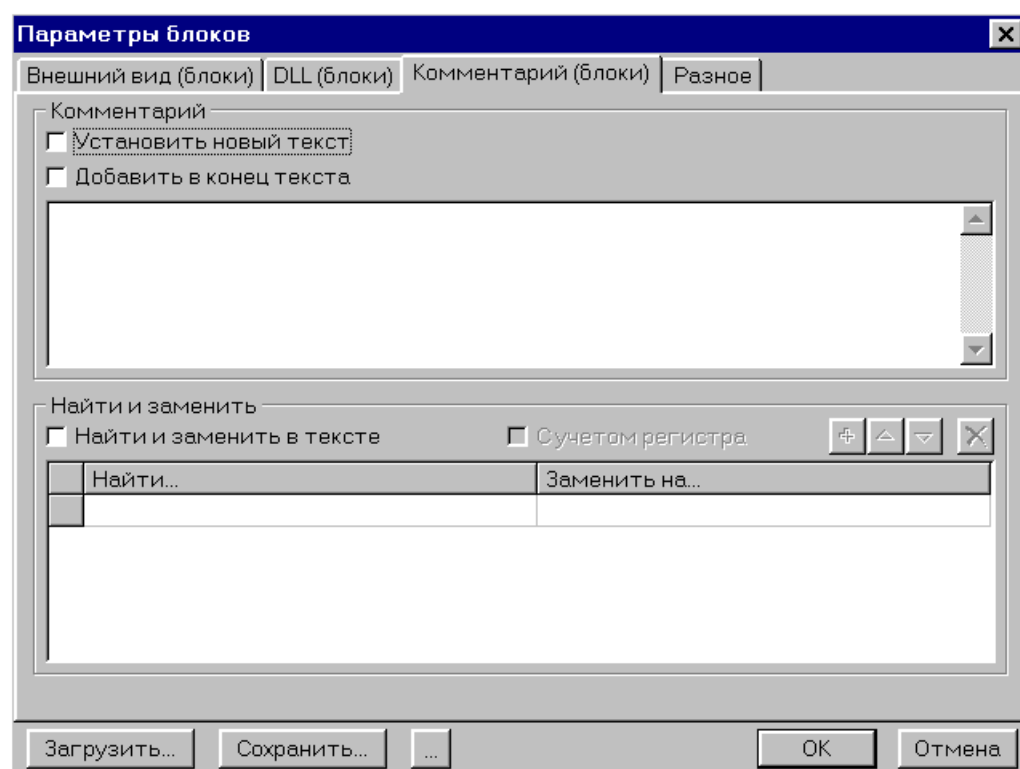


Рис. 361. Окно установки параметров блоков – вкладка “комментарий”

На этой вкладке можно либо ввести полный текст комментария или добавить к нему заданный текст (в верхней части вкладки), либо найти в тексте комментария заданные фрагменты текста и заменить их на другие (в нижней части). Обе панели на вкладке всегда включены – наличие автокомпилируемой модели никак не связано с комментарием блока.

Нажатие кнопки “ОК” изменит параметры всех блоков с данной моделью по сделанным в окне установкам. Как и обычную групповую установку параметров блоков, это изменение можно отменить пунктом главного меню РДС “система | отмена”, если в настройках РДС разрешена отмена действий пользователя.

§3.7. Принципы создания автокомпилируемых моделей блоков

Рассматривается задание поведения блока при помощи включения различных реакций на события в его автокомпилируемую модель. Приводятся примеры моделей, выполняющих различные действия.

§3.7.1. Устройство формируемой модулем программы

Описывается общая структура автоматически формируемого текста программы модели блока с точки зрения программиста C++. Пользователям, мало знакомым с программированием, можно бегло просмотреть этот параграф для лучшего понимания устройства создаваемых ими моделей.

При каждом изменении модели модуль автоматической компиляции на основе введенных пользователем фрагментов программ, описаний и настроечных параметров формирует общий текст программы, который передается компилятору для сборки файла динамической библиотеки (DLL). Затем функция модели из созданной компилятором библиотеки подключается к обслуживаемым блокам. Разработчикам моделей имеет смысл представлять себе, как устроен формируемый текст программы – это позволит лучше понять, что можно и что нельзя использовать в фрагментах программы, вводимых в редакторе модели. При создании простейших моделей это не так важно – достаточно просто понимать, что реакция блока на любое событие (см. §3.6.4 на стр. 46) оформляется модулем как функция, поэтому внутри вводимого фрагмента программы можно использовать любые конструкции языка C, оператором `return` можно немедленно прервать выполнение реакции, а все переменные, объявленные внутри фрагмента программы, будут локальными для этой автоматически сформированной функции. Свои функции внутри реакций на события описывать нельзя (в языке C запрещено описание функции внутри другой функции) – для этого служат глобальные описания. Описания (глобальные, внутри и после класса блока, см. там же в §3.6.4) как функции не оформляются, поэтому все переменные, объявленные в них, становятся глобальными для данной модели, и, естественно, оператор `return`, как и другие исполняемые операторы C, в этих описаниях использовать невозможно (зато можно описывать свои функции). Разработчики моделей, не собирающиеся пользоваться расширенными возможностями, предоставляемыми сервисными функциями РДС, могут пропустить этот параграф – все сведения, необходимые для добавления в модель реакций на стандартные события, будут приведены в следующих параграфах, начиная с §3.7.2.1. Список всех реакций и их параметров приводится в §3.8 (стр. 282).

В §3.3 (стр. 20) была рассмотрена одна из простейших моделей: модель сумматора, выдающего на вещественный выход “y” сумму вещественных входов “x1” и “x2”. Для ее создания достаточно было задать структуру переменных блока и ввести единственный оператор присваивания в реакцию на такт расчета. Теперь внимательно рассмотрим текст программы, которую модуль автокомпиляции сформировал для этой модели (его можно просмотреть в отдельной вкладке редактора модели, выбрав в меню редактора пункт “модель | показать текст C++”). Предполагается, что настройки модели (см. §3.6.7 на стр. 69) не изменялись пользователем и остались в состоянии по умолчанию.


```

//-----
// Автоматически сформированный текст для модели "new_model.mdl"
// Исходный файл модели: C:\Rds\Models\new_model.mdl
// Модуль автокомпиляции: Borland C++ 5.5
//-----
#line 7 "-1"
#include <windows.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

#define RDSBCPP_MODELNAME "new_model.mdl"

#define RDSCALL CALLBACK
#include <RdsDef.h>
#define RDS_SERV_FUNC_BODY rdsbcppGetService
#include <RdsFunc.h>
#include <CommonBl.h>
#include <CommonAC.hpp>

//-----
// Макросы обработки исключений
//-----
// Обработка исключений включена
#define RDSBCPP_EXCEPTIONS
// Оператор try
#define RDSBCPP_TRY __try
// Оператор catch
#define RDSBCPP_CATCHALL __except (EXCEPTION_EXECUTE_HANDLER)
//-----

// Ошибка математических функций
double rdsbcppHugeDouble;

//-----
// Перехват ошибок математики
//-----
int _matherr (struct _exception *a)
{ a->retval=(a->type==UNDERFLOW)?0.0:rdsbcppHugeDouble;
  return 1;
}
//-----

//-----
// Точка входа DLL
//-----
#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst,
                        unsigned long reason,void *lpReserved)
{ if(reason==DLL_PROCESS_ATTACH)
  { if(!RDS_SERV_FUNC_BODY())
    { MessageBox(NULL,
      "Невозможно получить доступ к сервисным функциям RDS",
      RDSBCPP_MODELNAME,MB_OK | MB_ICONERROR);
    return 0;
  }
}

```

```

        else // Есть доступ к сервисным функциям
        { rdsGetHugeDouble(&rdsbcppHugeDouble);
        }
    }
    return 1;
}
//-----

//-----
// Описания классов переменных блока
//-----
// Переменная double ("D")
RDSBCPP_STATICPLAINCLASS(rdsbcstDouble, double);
// Переменная char ("S")
RDSBCPP_STATICPLAINCLASS(rdsbcstSignal, char);
//-----

//-----
// Класс блока
//-----
class rdsbcppBlockClass
{ public:
    // Структура данных блока
    RDS_PBLOCKDATA rdsbcppBlockData;
    // Статические переменные
    rdsbcstSignal Start;
    rdsbcstSignal Ready;
    rdsbcstDouble x1;
    rdsbcstDouble x2;
    rdsbcstDouble y;

    // Инициализация переменных блока
    void rdsbcppInitVars(void *base)
    { // Статические переменные
        Start.Init(base, 0);
        Ready.Init(base, 1);
        x1.Init(base, 2);
        x2.Init(base, 10);
        y.Init(base, 18);
    };

    // Проверка существования динамических переменных
    BOOL rdsbcppDynVarsOk(void)
    { return TRUE; };

    // Функции реакции на события
    void rdsbcppModel(RDS_PINITIALCALCDATA InitialCalc);

    // Конструктор
    rdsbcppBlockClass(RDS_PBLOCKDATA data);
    // Деструктор
    ~rdsbcppBlockClass();
}; // class rdsbcppBlockClass
//-----

```

```

//-----
// Функция блока
//-----
#pragma argsused
extern "C" __declspec(dllexport)
int RDSCALL rdsbcppBlockEntryPoint(
    int CallMode,          // Режим вызова (сообщение RDS_BFM_*)
    RDS_PBLOCKDATA BlockData, // Данные блока
    LPVOID ExtParam        // Дополнительные данные
)
{
    int result=RDS_BFR_DONE; // Код возврата
    // Объект класса блока (хранится в личной области данных)
    rdsbcppBlockClass *data=
        (rdsbcppBlockClass*) (BlockData->BlockData);

    // Перехват ошибок математики - начало
    volatile unsigned int FPMask=_control87(0,0);
    _control87(MCW_EM,MCW_EM);

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

        case RDS_BFM_CLEANUP: // Очистка (вызывается перед удалением)
            delete data;
            break;

        case RDS_BFM_VARCHHECK:
            // Проверка допустимости структуры переменных
            if(strcmp((char*)ExtParam,"{SSDDD}" )!=0)
                return RDS_BFR_BADVARSMSG;
            break;

        case RDS_BFM_MODEL: // Один такт моделирования
            data->rdsbcppInitVars(BlockData->VarData);
            data->rdsbcppModel((RDS_PINITIALCALCDATA)ExtParam);
            BlockData->BlockData=data;
            // Предохраняет от случайного изменения пользователем
            break;
    } // switch(CallMode)

    // Перехват ошибок математики - конец
    _clear87();
    _control87(FPMask,MCW_EM);

    return result;
}
//-----

// Конструктор класса блока
rdsbcppBlockClass::rdsbcppBlockClass(RDS_PBLOCKDATA data)
{
    // Сохранение адреса структуры данных блока
    rdsbcppBlockData=data;
}
//-----

```

```

// Деструктор класса блока
rdsbcppBlockClass::~rdsbcppBlockClass()
{ }
//-----

//-----
// Функции реакции на события
//-----
// Один такт моделирования
void rdsbcppBlockClass::rdsbcppModel(RDS_PINITIALCALCDATA
                                     InitialCalc)
{
#line 1 "*0" // Со следующей строки – текст пользователя

y=x1+x2;

/* Служебный комментарий, предохраняющий от незавершенного
комментария в тексте пользователя */
#line 167 "*-1" // Со следующей строки – генерируемый текст
} // rdsbcppBlockClass::rdsbcppModel
//-----

```

Можно видеть, что, несмотря на простоту модели, текст получился довольно объемным. В него включены все необходимые для компиляции описания, для данных блока сформирован класс с названием `rdsbcppBlockClass`, добавлена главная функция (точка входа) DLL и т.п. Начинается текст программы с директив “`#include`” для включения в текст программы стандартных файлов заголовков, в которых содержатся описания, необходимые для программ Windows и для использования функций стандартных библиотек языка C. Директива “`#line`”, предвещающая их, используется модулем автокомпиляции для разбора сообщений об ошибках, выдаваемых компилятором – она будет неоднократно встречаться в этом тексте и на нее можно не обращать внимания. Далее идут директивы “`#define`”, описывающие некоторые специфичные для РДС константы, и директивы включения файлов заголовков РДС, в которых описаны необходимые для работы модели структуры и сервисные функции. Мы не будем подробно останавливаться на этих описаниях, желающие могут изучить руководство программиста [1]. За ними следуют макросы обработки исключений, взятые из параметров модуля автокомпиляции (см. §3.9.10 на стр. 356) – они будут использоваться в функции модели, если в параметрах этой модели будет установлен флажок “перехватывать все исключения, возникшие в модели” (см. §3.6.7 на стр. 69). Все перечисленные выше описания и директивы добавляются в каждую модель, независимо от ее параметров и состава ее реакций, и пользователь не может на них повлиять, не меняя настроек всего модуля: они находятся там, где и должны быть, и пользователь, чаще всего, не использует их напрямую, хотя и может пользоваться всеми типами и функциями из включенных файлов заголовков.

Далее в тексте программы находится объявление вещественной глобальной переменной следующего вида:

```
double rdsbcppHugeDouble;
```

Это весьма важная переменная, которую разработчикам моделей приходится использовать достаточно часто. В ней всегда хранится специальная константа `HUGE_VAL` из стандартных описаний языка C, обозначающая ошибку выполнения математической операции. Это значение может поступить на вход модели вместо вещественного числа, если блок, соединенный с этим входом, не смог выполнить какое-либо действие (например, в нем возникло переполнение или деление на ноль). Как правило, перед выполнением заложенных в них вычислений, модели проверяют значения на своих входах, сравнивая их со значением

переменной `rdsbcppHugeDouble`. Если, например, вход модели – вещественная переменная `x`, то, обычно, реакция на такт расчета в модели выглядит следующим образом:

```
if (x==rdsbcppHugeDouble)
{ // Действия при ошибке
}
else
{ // Обычные действия
}
```

Можно, конечно, сравнивать входы не с глобальной переменной, а с самой константой `HUGE_VAL`, но, поскольку переменная `rdsbcppHugeDouble` получает свое значение непосредственно из РДС, сравнение с ней гарантирует, что во всех моделях блоков в качестве признака ошибки будет использоваться одно и то же значение, независимо от того, какие версии описаний в них использовались и на каких языках программирования они написаны. В рассматриваемой нами модели нет такой проверки – в простейших моделях она не обязательна. Тем не менее, для иллюстрации обработки ошибок входов проверка будет добавлена в эту модель в §3.7.2.1 (стр. 92).

Следует обратить внимание на то, что имя описанной выше глобальной переменной имеет префикс “`rds`”. Этот префикс имеют все формируемые модулем автокомпиляции имена переменных, функций и типов, поэтому разработчику модели не следует начинать с этих символов свои идентификаторы, чтобы не пересечься со служебными именами. Лучше всего вообще не начинать никакие идентификаторы с символов “`rds`” и “`RDS`” – это гарантирует отсутствие конфликтов не только с именами, созданными модулем автокомпиляции, но и со всеми именами типов, констант и функций, описанных в файлах заголовков РДС.

Далее, поскольку в параметрах модели по умолчанию установлен флажок “перехватывать ошибки математических функций”, в текст программы вставлена функция обработки математических ошибок `_matherr`:

```
int _matherr (struct _exception *a)
{ a->retval=(a->type==UNDERFLOW)?0.0:rdsbcppHugeDouble;
  return 1;
}
```

Эта функция будет автоматически вызываться при возникновении ошибок в функциях математической библиотеки, ее вид задается в настройках модуля автокомпиляции (см. §3.9.10 на стр. 356). В данном случае она устроена очень просто: при потере точности результат считается нулем, а при всех прочих ошибках в качестве результата операции возвращается значение-индикатор ошибки из глобальной переменной `rdsbcppHugeDouble`. Сама функция возвращает ненулевое значение, сигнализируя о том, что ошибка обработана. Подробнее о перехвате ошибок в математических функциях можно прочесть в описании используемого компилятора.

После функции перехвата математических ошибок располагается автоматически формируемая функция `DllEntryPoint` – главная функция DLL, которую должна иметь каждая динамическая библиотека Windows. Имя и описание этой функции задается в параметрах модуля автокомпиляции (см. §3.9.9 на стр. 351). Структура и особенности главной функции DLL подробно описаны в §2.2 руководства программиста, здесь мы не будем рассматривать ее в деталях. Отметим только, что эта функция, в частности, вызывается при загрузке DLL в память РДС (при этом в ее параметре `reason` передается значение `DLL_PROCESS_ATTACH`). Текст функции сформирован модулем автокомпиляции так, чтобы в этот момент вызывалась другая, тоже автоматически сформированная, функция, служащая для получения доступа к сервисным функциям РДС (имя этой функции присвоено константе `RDS_SERV_FUNC_BODY`). Если доступ получен (он будет успешно получен в том случае, если DLL загружается в память РДС, и версия РДС соответствует описаниям из файлов заголовков, автоматически включенных в модель), будет вызвана сервисная функция

rdsGetHugeDouble, которая запишет в уже описанную глобальную переменную rdsbcppHugeDouble значение HUGE_VAL, используемое блоками для сигнализации об ошибке математики. Если бы в параметрах модели была включена информация о версии (см. рис. 356 на стр. 74), здесь же была бы вызвана функция, передающая в РДС номер версии модели.

Сразу за главной функцией DLL следуют макросы вида RDSBCPP_*CLASS, разворачивающиеся в описания классов для доступа к статическим и динамическим переменным, использованным в модели блока. Эти макросы описываются в файле “CommonAC.hpp”, в котором, для желающих разобраться в них, содержатся подробные комментарии. Структура переменных нашего блока включает только статические переменные типов “double” и “сигнал”, поэтому модуль включит в текст программы только два вызова макроса RDSBCPP_STATICPLAINCLASS: один для вещественного типа “double”, и один для сигнала. Если проанализировать эти макросы, то можно увидеть, что, например, макрос для вещественного типа

```
RDSBCPP_STATICPLAINCLASS(rdsbcstDouble,double);
```

разворачивается в описание класса следующего вида:

```
class rdsbcstDouble : public rdsbcppVarAncestor
{ protected:
    double *Ptr;
public:
    inline void Init(void *base,int offset)
        {Ptr=(double*) (((unsigned char*)base)+offset);};
    inline operator double() const {return *Ptr;};
    inline double * GetPtr(void) const {return Ptr;};
    inline double operator=(rdsbcstDouble &v)
        {*Ptr=(double)v; return *Ptr; };
    inline double operator=(double v) {*Ptr=v; return *Ptr;};
    inline double operator+=(double v) {(*Ptr)+=v; return *Ptr;};
    inline double operator-= (double v) {(*Ptr)-=v; return *Ptr;};
    inline double operator*=(double v) {(*Ptr)*=v; return *Ptr;};
    inline double operator/=(double v) {(*Ptr)/=v; return *Ptr;};
    inline double operator++() {++(*Ptr); return *Ptr;};
    inline double operator++(int)
        {double tmp=*Ptr; (*Ptr)++; return tmp;};
    inline double operator--() {--(*Ptr); return *Ptr;};
    inline double operator--(int)
        {double tmp=*Ptr; (*Ptr)--; return tmp; };
    rdsbcstDouble(void):rdsbcppVarAncestor(){};
};
```

Можно заметить, что в этом классе все математические операторы и оператор преобразования к типу double переопределены таким образом, чтобы работать с областью памяти, на которую указывает скрытое поле класса Ptr. Таким образом, если при помощи вызова функции-члена Init записать в поле Ptr указатель на какую-либо вещественную переменную в структуре переменных блока, объект этого класса можно будет использовать во всех математических выражениях и вызовах функций, в которых требуется число типа double. При этом объект будет автоматически выполнять все операции, в которых он участвует, с числом, на которое указывает Ptr. На этом принципе основан доступ к переменным блока из фрагментов программ, вводимых пользователем: в классе блока создаются объекты специальных классов с именами, соответствующими переменным блока, а пользователь использует эти объекты в математических выражениях, как будто это сами переменные привычных ему типов. В большинстве случаев пользователь может не задумываться о том, что переменная “x1”, которую он использует в модели, на самом деле

имеет не тип `double`, а тип `rdsbcstDouble`. Это будет работать до тех пор, пока ему для какой-либо цели не понадобится указатель на переменную блока: запись “&x1” будет иметь тип не “указатель на `double`”, как может ожидать пользователь, а “указатель на `rdsbcstDouble`”. Чтобы получить настоящий указатель на переменную блока, скрытую за объектом класса (например, для использования в функциях типа `scanf`), необходимо вызвать функцию-член класса `GetPtr`: вместо “&x1” нужно использовать “`x1.GetPtr()`”.

Следует учитывать, что, в некоторых, достаточно редких, случаях, компилятор не может сам выполнить операцию преобразования типов. Пусть, например, “x1” – вещественная переменная блока, для которой автоматически создан объект `x1` типа `rdsbcstDouble`. Если необходимо сформировать строку с текстовым представлением вещественного числа (например, для вывода сообщения), может показаться логичным включить в программу модели следующий фрагмент:

```
char buf[100]; // Вспомогательный буфер
sprintf(buf, "x1=%.2lf", x1);
```

Здесь вызывается функция `sprintf` из стандартной библиотеки C, выполняющая форматированный вывод во вспомогательный массив `buf`. Этот фрагмент программы будет скомпилирован без ошибок, однако, работать он не будет (значение переменной блока не запишется в массив). Дело в том, что в функции `sprintf` может быть произвольное число параметров разных типов, поэтому компилятор не поймет, что, в данном случае, объект `x1`, указанный в третьем параметре функции, необходимо привести к типу `double`. В таких случаях необходимо указать приведение типа явно:

```
char buf[100]; // Вспомогательный буфер
sprintf(buf, "x1=%.2lf", (double) x1);
```

В этом фрагменте перед `x1` стоит оператор приведения типа “`(double)`”, и разночтений в типе параметра уже не возникнет.

За исключением получения указателя на переменную и явного приведения типа, пользователю практически никогда не потребуется вызов функций-членов классов простых статических переменных, вроде приведенного выше класса `rdsbcstDouble`. В динамических переменных и в сложных статических (массивах, матрицах и т.п.) функции-члены, напротив, используются довольно часто: в динамических переменных с их помощью производится проверка существования переменной и уведомление других блоков о ее изменении (см. §3.7.3 на стр. 129), в массивах и матрицах – изменение размерности (см. §3.7.2.2 на стр. 99 и §3.7.2.3 на стр. 103). Примеры использования этих функций будут рассмотрены позже.

Если бы в нашей модели были глобальные описания пользователя, они разместились бы сразу за макросами классов переменных блока. Поскольку таких описаний в модели нет, за ними располагается описание самого класса блока с именем `rdsbcppBlockClass` (это имя жестко встроено в модуль автокомпиляции, его используют все модели). Объект этого класса будет создаваться для каждого блока с данной моделью в момент подключения модели (при загрузке схемы в память, загрузке блока из библиотеки или при первом подключении модели вручную) и будет существовать до тех пор, пока модель не будет отключена. Фактически, объект этого класса будет представлять собой личную область данных блока, рассматриваемую в §2.4 руководства программиста. Все реакции блока на события будут оформляться как функции-члены этого класса, все статические и динамические переменные блока (точнее, объекты для работы с этими переменными) и настроечные параметры будут полями этого класса. Команды создания и уничтожения объекта класса блока добавляются в модели автоматически, пользователю не нужно об этом заботиться.

Первое же поле в секции `public` этого класса (других секций у него нет) – это указатель на структуру данных, создаваемую в РДС для каждого блока:

```
RDS_PBLOCKDATA rdsbcppBlockData;
```

RDS_PBLOCKDATA – это указатель на структуру RDS_BLOCKDATA, описанную в файле “RdsDef.h”. Эта структура и ее поля подробно рассматриваются в §2.3 руководства программиста и А.2.3 приложения к нему [2]. Мы не будем разбирать ее подробно, приведем только ее описание и рассмотрим некоторые, самые важные поля.

```
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
```

Чаще всего в этой структуре используется поле Block – это уникальный идентификатор данного блока, который нужен для вызова некоторых функций РДС. Например, чтобы перебрать все блоки, соединенные связями с данным блоком (пример задачи, где это может потребоваться, приведен в §3.7.13.3 на стр. 258), необходимо вызвать функцию rdsEnumConnectedBlocks, в которую первым параметром передается идентификатор данного блока. Внутри какой-либо реакции на событие, вводимой в редакторе модели, это можно сделать следующим образом:

```
rdsEnumConnectedBlocks(rdsbcppBlockData->Block, ...);
```

Поскольку указатель на структуру данных блока записан в поле rdsbcppBlockData, а все реакции на события оформляются модулем автокомпиляции как функции-члены того же класса, во всех них поле класса rdsbcppBlockData доступно просто по имени, и идентификатор данного блока можно получить при помощи выражения rdsbcppBlockData->Block.

В поле Parent структуры данных блока хранится идентификатор родительской подсистемы данного блока. Он может понадобиться в функциях, выполняющих какие-либо операции с подсистемой. Например, чтобы программно открыть окно подсистемы, в которой находится данный блок, в его модели следует сделать вызов

```
rdsOpenSystemWindow(rdsbcppBlockData->Parent);
```

Поле BlockName содержит указатель на строку с именем блока, ее можно использовать, например, в сообщениях об ошибках. Поле Flags – одно из немногих полей в структуре данных блока, которое можно не только считывать, но и изменять внутри модели. Оно содержит набор битовых флагов, определяющих поведение блока. Например, в реакции на нажатие кнопки мыши блок может “захватить” мышь, взведя в этом поле флаг RDS_MOUSECAPTURE – после этого при любом перемещении курсора мыши будет вызываться реакция этого блока, даже если курсор покинет его изображение (см. пример блока-рукоятки в §3.7.11 на стр. 226). Остальные поля структуры используются реже. В большинстве случаев, при создании простых моделей структура RDS_BLOCKDATA вообще не используется, но, если она вдруг понадобится, следует иметь в виду, что доступ к ней можно получить через поле rdsbcppBlockData.

За полем rdsbcppBlockData в классе блока следуют автоматически добавляемые модулем автокомпиляции поля для статических и динамических переменных блока (см. §3.6.2 на стр. 38 и §3.6.3 на стр. 42 соответственно) и его настроечных параметров (см. §3.6.6 на стр. 60). В нашей модели всего пять статических переменных: два обязательных сигнала “Start” и “Ready” и вещественные переменные “x1”, “x2” и “y”. Для них будут созданы объекты двух классов, описания которых сформированы уже рассмотренными выше (стр. 86) макросами: два объекта класса rdsbcstSignal (сигнальные переменные) и три объекта типа rdsbcstDouble (вещественные). Затем в

класс блока автоматически вставляется функция, инициализирующая объекты доступа к переменным, в которой эти объекты настраиваются на конкретные переменные блока. Для нашей модели она выглядит так:

```
void rdsbcppInitVars(void *base)
{ // Статические переменные
  Start.Init(base,0);
  Ready.Init(base,1);
  x1.Init(base,2);
  x2.Init(base,10);
  y.Init(base,18);
};
```

Вызов этой функции автоматически добавляется модулем перед вызовом функций реакции блока на любое событие, поэтому внутри реакций на события (за исключением событий инициализации и очистки данных блока, в которых к статическим переменным обращаться запрещено) все объекты уже настроены на работу с переменными блока. Внутри функции для каждой статической переменной блока (динамических в нашей модели нет) у соответствующего объекта вызывается функция-член `Init`, в которую передается базовый адрес всех переменных блока и смещение к конкретной переменной (эти же смещения можно увидеть и в окне редактора переменных, см. колонку “начало” на рис. 323 на стр. 25). После функции инициализации внутри класса обычно располагается функция проверки наличия динамических переменных `rdsbcppDynVarsOk` – в нашей модели динамических переменных нет, поэтому эта функция всегда возвращает `TRUE`. И `rdsbcppInitVars`, и `rdsbcppDynVarsOk` создаются модулем автокомпиляции без всякого участия пользователя, и их вызовы тоже вставляются в нужные места формируемой функции модели автоматически, поэтому пользователю можно не задумываться об их существовании – вызывать их вручную ему не придется. Есть единственный случай, в котором пользователю может понадобиться функция проверки динамических переменных `rdsbcppDynVarsOk`: если в параметрах модели отключен флажок “не реагировать на события без динамических переменных” (см. §3.6.7 на стр. 69), перед использованием таких переменных необходимо проверить их наличие, а для этого можно вызвать `rdsbcppDynVarsOk` и, если она вернет `FALSE`, как-то среагировать на отсутствие динамических переменных (например, взять данные из другого источника).

Далее в класс блока добавляются описания всех функций-членов для введенных пользователем реакций на события. Наша модель реагирует на единственное событие – такт расчета, поэтому такая функция будет единственной (ее параметр сейчас не важен):

```
void rdsbcppModel(RDS_PINITIALCALCDATA InitialCalc);
```

Внутри класса вставляется только описание функции, ее тело будет добавлено в конец формируемого текста. За функциями реакции следуют автоматически формируемые описания конструктора и деструктора класса блока (сами эти функции располагаются в тексте дальше), а также описания пользователя внутри класса, если он их ввел (см. также §3.8.1 на стр. 282). В нашей модели таких описаний нет, поэтому класс на этом завершается. После его закрывающей фигурной скобки добавляются описания пользователя после класса блока (в нашей модели таких тоже нет).

Далее в тексте программы размещается экспортированная функция модели с именем `rdsbcppBlockEntryPoint` (общая структура любой функции модели блока описана в §2.3 руководства программиста), заголовок которой, необходимый для ее экспорта, задается в параметрах модуля автокомпиляции (см. §3.9.9 на стр. 351). Эта функция формируется полностью автоматически и, фактически, представляет собой один большой оператор `switch`, в метки `case` которого модуль автокомпиляции вставляет вызовы функций реакций на события, сформированных для каждого введенного пользователем фрагмента программы. Кроме того, реакции на некоторые события добавляются автоматически –

например, реакция на событие проверки структуры переменных `RDS_BFM_VARCHHECK`, в которой модель сравнивает структуру статических переменных блока, к которому ее подключают, с введенной пользователем, и выдает сообщение об ошибке, если они не совпадают. Команды создания и уничтожения объекта класса блока `rdsbcppBlockClass` тоже вставляются в функцию автоматически – в реакции на события инициализации `RDS_BFM_INIT` и очистки `RDS_BFM_CLEANUP` соответственно. Если в параметрах модели включен перехват ошибок математических функций, в начало и в конец функции `rdsbcppBlockEntryPoint` будут добавлены фрагменты, заданные в настройках модуля автокомпиляции (см. §3.9.10 на стр. 356). В тексте, приведенном выше, они присутствуют – их легко узнать по комментариям.

Следует обратить особое внимание на то, что указатель на создаваемый в реакции на инициализацию блока (событие `RDS_BFM_INIT`) объект класса `rdsbcppBlockClass` записывается не только во вспомогательную переменную `data`, но и в поле `BlockData` структуры `RDS_BLOCKDATA` (см. стр. 88):

```
case RDS_BFM_INIT: // Инициализация блока
    BlockData->BlockData=(data=new rdsbcppBlockClass(BlockData));
break;
```

В этом поле указатель на созданный объект будет храниться на всем протяжении жизни блока, поэтому использовать поле `BlockData->BlockData` для других целей нельзя. Даже если внутри реакции на какое-либо событие присвоить ему другое значение, прежнее значение будет восстановлено при завершении функции модели, как будет показано ниже. В начале функции модели значение этого поля приводится к нужному типу и переписывается во вспомогательную переменную `data`, поэтому везде внутри этой функции обращения к классу блока выглядят как “`data->...`”.

Наша модель имеет единственную введенную пользователем реакцию на событие – выполнение такта расчета, которой внутри оператора `switch` будет соответствовать следующий, автоматически вставленный, `case`:

```
case RDS_BFM_MODEL: // Один такт моделирования
    data->rdsbcppInitVars(BlockData->VarData);
    data->rdsbcppModel((RDS_PINITIALCALCDATA)ExtParam);
    BlockData->BlockData=data;
    // Предохраняет от случайного изменения пользователем
break;
```

Здесь сначала у созданного для данного конкретного блока объекта класса `rdsbcppBlockClass` (указатель на него записан в переменную `data` в самом начале функции модели) вызывается описанная выше функция инициализации переменных `rdsbcppInitVars` – теперь объекты `Start`, `Ready`, `x1`, `x2` и `y` ссылаются на настоящие статические переменные блока и их можно использовать в математических выражениях. Если бы в нашей модели были динамические переменные и в ее параметрах был включен флажок “не реагировать на события без динамических переменных”, далее была бы вызвана функция `rdsbcppDynVarsOk`, и, в случае возврата ей `FALSE`, выполнение реакции на событие было бы немедленно прервано оператором `break`. В нашей модели есть только статические переменные, поэтому после их инициализации сразу вызывается автоматически сформированная функция `rdsbcppModel`, внутрь которой будет вставлена введенная пользователем реакция на такт расчета (сама эта функция находится далее по тексту). В параметре функции передается указатель на структуру, используемую при инициализационном (предварительном) расчете, который в данной модели не используется. При реакции на такт расчета возвращенное моделью значение не анализируется РДС, поэтому функция `rdsbcppModel` ничего не возвращает. После ее вызова полю `BlockData` структуры `BlockData` присваивается указатель на объект класса блока из переменной `data`. Казалось бы, этот указатель и так должен находиться в этом поле: во-первых, он был

записан туда при создании объекта в реакции на событие RDS_BFM_INIT, и, во-вторых, в переменную data он попал из этого самого поля. Однако, во введенном пользователем тексте программы, который находится внутри вызванной функции rdsbcppModel, может оказаться оператор, присваивающий полю BlockData какое-либо другое значение (структура данных блока, содержащая это поле, доступна во всех функциях-членах класса, включая пользовательские реакции на события, через поле rdsbcppBlockData). В этом операторе не может быть никакого смысла, поскольку в автокомпилируемых моделях нельзя использовать поле BlockData структуры данных блока, однако, нельзя гарантировать, что пользователь по ошибке не попытается его изменить. От такого изменения и предохраняет оператор присваивания, находящийся непосредственно перед оператором break.

В конце функции модели располагается оператор “return result;”, возвращающий в РДС код завершения из переменной result. В самом начале функции модели этой переменной была присвоена константа RDS_BFR_DONE, означающая нормальное завершение модели. Единственная реакция в нашей модели ничего не возвращает, поэтому значение переменной result останется неизменным.

После функции модели записаны автоматически сформированные конструктор и деструктор класса блока, и, в самом конце, функция rdsbcppModel, внутрь которой вставлена пользовательская реакция на событие:

```
// Один такт моделирования
void rdsbcppBlockClass::rdsbcppModel(RDS_PINITIALCALCDATA
                                     InitialCalc)
{
#line 1 "0" // Со следующей строки - текст пользователя

y=x1+x2;

/* Служебный комментарий, предохраняющий от незавершенного
комментария в тексте пользователя */
#line 167 "-1" // Со следующей строки - генерируемый текст
} // rdsbcppBlockClass::rdsbcppModel
```

Если не обращать внимания на директиву “#line”, необходимую модулю автокомпиляции для обработки сообщений компилятора об ошибках, и на служебные комментарии, можно увидеть, что все тело функции представляет собой введенный пользователем фрагмент, а именно оператор “y=x1+x2;”. Так же устроены и любые другие реакции на события – весь введенный пользователем фрагмент программы вставляется внутрь некоторой функции-члена класса rdsbcppBlockClass, поэтому все описанные в этом фрагменте переменные будут локальными для данной функции и уничтожатся при ее завершении (если, конечно, они описаны без модификатора static). Многие функции реакций имеют параметры, через которые внутрь них передается информация о событии, на которое нужно среагировать (например, координаты курсора мыши или код нажатой клавиши), а наружу – результат обработки (например, среагировал ли блок на нажатие кнопки мыши). У функции rdsbcppModel тоже есть параметр, указывающий на проведение предварительного расчета, но в данном примере он не используется. Независимо от наличия параметров, все функции реакции имеют тип возврата void, поэтому их выполнение в любой момент можно прервать оператором return без параметра. Если функция что-то возвращает, возвращаемое значение будет передаваться через один из ее параметров, имеющий тип “ссылка на значение”. Параметры различных реакций на события будут описаны далее в примерах. Заголовок конкретной функции, в которую будет вставлен вводимый пользователем текст, всегда можно увидеть в верхней части вкладки редактора, на которой этот текст вводится (см. рис. 329 на стр. 32 и рис. 337 на стр. 57).

Пользователю, создающему автокомпилируемые модели, можно и не разбираться в том, как из вводимых им фрагментов собирается полноценный текст программы. Однако, при настройке модуля на нестандартный компилятор (см. §3.9.3 на стр. 333) эта информация может оказаться полезной. Кроме того, знание структуры этого текста может пригодиться при поиске ошибок в модели. Хотя модуль автокомпиляции, в большинстве случаев, может сам определить, к какому введенному пользователем фрагменту программы относится обнаруженная компилятором ошибка (при этом по двойному щелчку на сообщении об ошибке он автоматически откроет вкладку с нужным фрагментом), иногда ошибка может проявиться далеко от места своего возникновения. В этом случае модуль покажет весь автоматически сформированный текст, и пользователю желательно быть готовым к поиску своей ошибки внутри этого текста.

§3.7.2. Работа со статическими переменными блока

Описывается использование в моделях статических переменных блока, которые могут быть его входами и выходами. Отдельно рассматриваются разные типы переменных и их возможности.

§3.7.2.1. Модели с простыми статическими переменными

Описывается использование в моделях статических переменных простых типов: целых, вещественных и логических. Это наиболее часто используемые типы входов и выходов блока. Также рассматриваются особенности использования объектов, создаваемых для работы с переменными, и создание моделей, срабатывающих не в каждом такте расчета, а только при изменении входов блока.

Вернемся к уже рассмотренной в §3.3 (стр. 20) простейшей модели: сумматору, выдающему на вещественный выход “у” сумму вещественных входов “x1” и “x2”. Эта модель вполне работоспособна, но у нее есть пара недостатков.

Во-первых, сумматор с такой моделью будет работать только в том случае, если в параметрах блока будет установлен флаг запуска каждый такт (см. §2.9.1 части I). По умолчанию этот флаг устанавливается для каждого вновь созданного блока, поэтому в §3.3 он не упоминается. Такой способ запуска приводит к тому, что функция модели будет принудительно выполняться в каждом такте, вычисляя значение выхода, даже если значения входов не изменились. В данном случае модель очень простая, и ее лишние запуски не приведут к существенному замедлению работы схемы, однако, для более сложных моделей замедление может стать заметным. В РДС считается хорошим тоном писать модели блоков, выходы которых зависят только от входов, так, чтобы они запускались только при изменении этих входов.

Во-вторых, описанная модель не рассчитана на то, что на один из ее входов может вместо вещественного числа поступить специальная константа (в математической библиотеке языка С она называется `HUGE_VAL`), используемая в качестве признака математической ошибки. Такую константу выдают на выход стандартные блоки РДС, попытавшиеся выполнить недопустимую математическую операцию – например, деление на ноль. Как правило, при обнаружении на любом своем входе этой константы, блок, не выполняя вычислений, выдает ее же на свой выход, сообщая тем самым всем соединенным с ним блокам о произошедшей ошибке. По желанию разработчика, можно предусмотреть и другие возможные реакции. На самом деле, такая простая модель, несмотря на отсутствие в ней каких-либо проверок входов, все равно выдаст на выход константу `HUGE_VAL` при поступлении ее на один из входов: так устроена операция сложения. Кроме того, включенный по умолчанию флажок “перехватывать ошибки математических функций” в параметрах модели (см. §3.6.7 на стр. 69) автоматически заменит результат невыполнимой математической операции на эту константу. Однако, более сложным моделям необходимо опознавать ошибочные значения входов и предпринимать по этому поводу какие-либо действия, поэтому, для примера, мы добавим в нашу модель такие проверки.

Исправим созданную ранее модель сумматора. Сначала разберемся с запуском каждый такт – это не требует внесения изменений в текст введенного фрагмента программы модели, достаточно изменить некоторые параметры блока и его переменных. Загрузим схему с блоком, созданным в §3.3 (если она не сохранена, нужно просто проделать заново все описанные там шаги), и откроем окно редактора модели двойным щелчком на этом блоке. На левой панели редактора выберем вкладку “переменные” (см. рис. 321 на стр. 24, если левая панель отсутствует, следует включить ее пунктом меню “вид | переменные и события”) и нажмем кнопку “изменить” под списком статических переменных (см. рис. 322 на стр. 25). Откроется окно редактирования статических переменных блока, в котором нужно внести изменения, изображенные на рис. 362.

Начало	Имя	Тип	К-Н	Вход/Выход	Меню	Пуск	ABC	По умолчанию
0	Start	Сигнал	1	→ Вход	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1
1	Ready	Сигнал	1	→ Выход	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0
2	x1	double	8	→ Вход	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	0
10	x2	double	8	→ Вход	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	0
18	y	double	8	→ Выход	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0

Переменных: 5, тип: {SSDDD}

OK Отмена

Рис. 362. Изменения в статических переменных простого сумматора

Прежде всего, необходимо включить флажки в колонке “пуск” напротив входов блока “x1” и “x2”. Это приведет к тому, что при срабатывании любой связи, подключенной к этим входам, сигналу запуска блока, то есть его первой переменной (в данном блоке она называется “Start” – это ее обычное имя, даваемое ей при создании блока), будет автоматически присвоено значение 1, а это, в свою очередь, вызовет запуск модели блока в следующем такте расчета. Кроме того, переменной “Start” нужно дать единичное значение по умолчанию, чтобы при самом первом запуске расчета модель сложила начальные значения своих входов и выдала сумму на выход. Если этого не сделать, модель запустится только после первого срабатывания связи, а начальные значения входов не будут обработаны. Может показаться, что достаточно просто дать выходу значение по умолчанию, равное сумме значений входов, однако, это не так. Перед первым запуском расчета производится начальная передача данных по связям, о которой модели блоков не информируются: если, например, к входам блока будут подключены поля ввода, на момент самого первого запуска расчета (или запуска после сброса) значения входов “x1” и “x2” будут не нулевыми, а равными значениям этих полей ввода, причем сигнал “Start” при этой начальной передаче данных автоматически взведен не будет. По этой причине в моделях, полагающихся на автоматический запуск при срабатывании входных связей, сигналу запуска следует присваивать единицу либо в качестве значения по умолчанию, как в данном случае, либо принудительно в реакции на событие запуска расчета RDS_BFM_STARTCALC.

Внеся изменения в статические переменные блока, следует закрыть окно редактирования переменных кнопкой “OK”.

Теперь наша модель способна запускаться при срабатывании входных связей, но в параметрах блока, к которому она подключена, пока еще установлен ее запуск каждый такт. Необходимо перевести этот блок в режим запуска по сигналу, то есть запуска только при

ненулевом значении переменной “Start”. Поскольку этот блок – единственный, можно просто открыть окно его параметров и переключить флажок на вкладке “общие” (см. §2.9.1 части I и рис. 363). Если бы таких блоков было несколько, пришлось бы открывать окно параметров для каждого из них, поэтому переключим режим запуска модели блока при помощи функции групповой установки (см. §3.6.8 на стр. 76), которая обработает все блоки с нашей моделью во всех подсистемах. Для этого следует открыть окно групповой установки, выбрав в окне редактора модели пункт меню “модель | установка параметров блоков” или нажать соответствующую ему кнопку. В окне следует включить флажок “установить запуск модели” и подчиненный ему “по сигналу готовности” (рис. 364), после чего закрыть окно кнопкой “OK”.

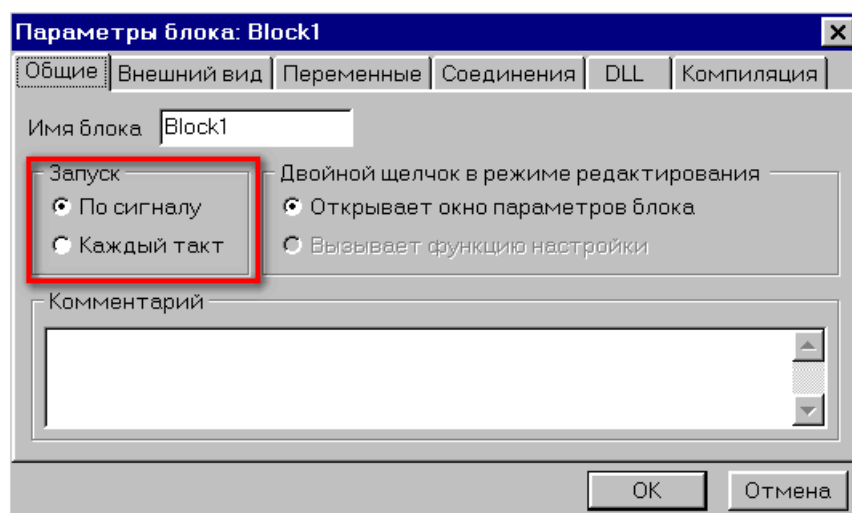


Рис. 363. Установка запуска по сигналу в окне параметров блока

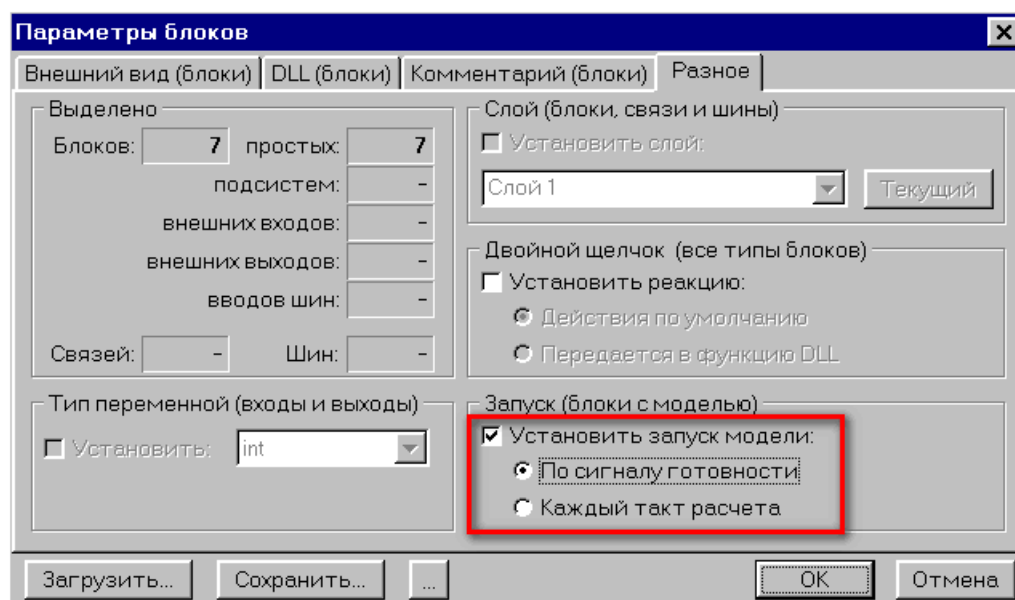


Рис. 364. Установка запуска по сигналу в окне групповой установки

Теперь модель блока будет запускаться один раз при первом запуске расчета, и далее каждый раз при срабатывании одной из входных связей. С точки зрения пользователя поведение сумматора не изменится: он складывает свои входы, как и раньше (простая схема для тестирования сумматора изображена на рис. 325, стр. 26), но теперь его работа меньше нагружает систему.

Добавим в нашу модель проверку значений входов на константу, сигнализирующую о математической ошибке. Как уже объяснялось выше, для такой простой модели, да еще и с включенным перехватом ошибок математики, в этой проверке нет необходимости. Здесь эта проверка приводится только в качестве примера. Она будет полезна, а, в некоторых случаях, и обязательна, в более сложных блоках. Например, если бы модель выполняла не сложение, а деление, и перехват ошибок математики был бы выключен, подача на оба входа константы ошибки привело бы к исключению в модели (впрочем, как и деление на ноль, на которое тоже пришлось бы делать проверку).

В автоматически компилируемых моделях блоков в качестве значения, указывающего на математическую ошибку, лучше всего брать не константу `HUGE_VAL`, а содержимое глобальной переменной `rdsbcppHugeDouble` (см. стр. 84). В этой переменной хранится константа `HUGE_VAL`, полученная из РДС при помощи сервисной функции `rdsGetHugeDouble`, которая подробно рассматривается в приложении к руководству программиста [2]. Вызов этой функции добавляется в программу модели без участия пользователя. Использование этой переменной вместо константы гарантирует, что во всех моделях блоков, независимо от того, какими компиляторами с какими версиями библиотек они созданы, в качестве ошибки математической операции используется одно и то же число. Следует учитывать, что эта переменная имеет тип `double`, и, поэтому, она пригодна только для работы с переменными блока того же типа. Для типа `float` в РДС нет специально выделенного значения ошибки, поэтому при создании автокомпилируемых моделей следует, по возможности, использовать именно тип `double`.

Поскольку занесение значения в глобальную переменную `rdsbcppHugeDouble` выполняется без нашего участия, все, что нам нужно сделать – это добавить в реакцию на такт расчета нашей модели (вкладка “модель” в редакторе, см. рис. 324 на стр. 26) оператор `if` (двойным подчеркиванием выделен добавленный в модель текст):

```
if(x1==rdsbcppHugeDouble || x2==rdsbcppHugeDouble)  
    y=rdsbcppHugeDouble;  
else  
    y=x1+x2;
```

Теперь, если хотя бы один из входов блока будет равен значению `rdsbcppHugeDouble`, выходу будет присвоено это же значение, и вычисления выполнены не будут. При желании, при ошибке на входе можно взводить какой-либо дополнительный выход ошибки, или выдавать на выход нулевое значение – все зависит от того, каких целей добивается создатель модели.

В программе модели можно использовать не только простые математические операции, но и любые функции языка C – например, тригонометрические. Рассмотрим еще одну модель, которая будет вычислять синус или косинус входа блока. В этом блоке тоже будут только простые статические переменные: вещественный вход “x”, вещественный выход “y” и целый вход “func”, на который будет подаваться ноль, если необходимо вычислить синус, и единица, если косинус. Сразу заложим в этот блок запуск при срабатывании входных связей и введем в него проверку на поступление на вход значения-индикатора ошибки.

Создадим новый пустой блок, переключим его в режим работы по сигналу и создадим для него новую модель, как это было описано в §3.3 на стр. 20 (в дальнейшем эта последовательность действий уже не будет повторно описываться). Необходимо:

1. нажать на свободном месте окна подсистемы правую кнопку мыши;
2. выбрать в открывшемся контекстном меню пункт “создать | новый блок”;
3. нажать на созданном блоке (он будет выглядеть как белый квадрат с черной рамкой) правую кнопку мыши;
4. выбрать в открывшемся контекстном меню пункт “параметры” (см. рис. 314 на стр. 21);

5. на вкладке “общие” открывшегося окна включить флажок “по сигналу” на панели “запуск” (см. рис. 363 на стр. 94);
6. на вкладке “компиляция” установить флажок “функция блока компилируется автоматически”, выбрать в выпадающем списке подключенный модуль автокомпиляции и нажать кнопку “новый” (см. рис. 315 на стр. 21);
7. в появившемся окне “новая модель” выбрать флажок “создать пустую модель” (см. рис. 316 на стр. 22) и закрыть окно кнопкой “ОК” (если в используемой версии РДС нет шаблонов моделей, этот шаг будет пропущен и сразу откроется диалог сохранения);
8. в диалоге сохранения модели ввести имя нового файла, в который будет записана создаваемая модель (см. рис. 318 на стр. 23), и нажать кнопку “сохранить”;
9. закрыть окно параметров блока кнопкой “ОК”.

После выполнения перечисленных выше действий откроется пустое окно редактора модели, в котором нужно создать структуру статических переменных блока и ввести реакцию на выполнение такта расчета. Для ввода статических переменных необходимо на вкладке “переменные” левой панели окна редактора нажать кнопку “изменить” (см. рис. 322 на стр. 25) и заполнить таблицу переменных в окне следующим образом:

<i>Имя</i>	<i>Тип</i>	<i>Вход/выход</i>	<i>Пуск</i>	<i>Начальное значение</i>
Start	Сигнал	Вход	✓	1
Ready	Сигнал	Выход		0
x	double	Вход	✓	0
func	int	Вход	✓	0
y	double	Выход		0

Теперь на вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) нужно ввести следующий фрагмент программы:

```

if (x==rdsbcppHugeDouble)
{
    y=rdsbcppHugeDouble;
    return;
}
switch (func)
{
    case 0: // Синус
        y=sin(x); break;
    case 1: // Косинус
        y=cos(x); break;
    default: // Ошибка
        y=rdsbcppHugeDouble;
}

```

Этот фрагмент будет выполняться в каждом такте расчета, если в этом такте блок получит сигнал запуска, то есть, если значение сигнала готовности (переменной Start) нашего блока не будет равно нулю. Мы дали этой переменной начальное значение, равное единице, поэтому наша модель обязательно выполнится при первом запуске расчета. Кроме того, мы поставили флажки в колонке “пуск” напротив входов блока x и func, поэтому при срабатывании связей, соединенных с этими входами, сигнал готовности автоматически взведется, и модель тоже выполнится. Рассмотрим введенный фрагмент программы подробно.

Самый первый оператор в этом фрагменте – сравнение значения входа блока x с глобальной переменной rdsbcppHugeDouble, в которой хранится значение-индикатор ошибки вычисления (HUGE_VAL). Если на входе нашего блока окажется это значение, оно же будет присвоено выходу y, и модель немедленно завершится оператором return. Таким

образом, при получении признака ошибки наш блок просто передает его на выход, не выполняя никаких вычислений. Значение входа `func` мы с признаком ошибки не сравниваем – это целый вход, а для целых чисел в РДС не предусмотрено каких-либо индикаторов ошибок вычисления.

Далее выполняется оператор `switch`, в котором, в зависимости от значения входа `func`, вычисляется выход блока. Если значение `func` – 0 или 1, выходу `y` присваивается синус или косинус входа соответственно. При любом другом значении `func` выходу присваивается значение-индикатор ошибки `rdsbcpHugeDouble`.

Теперь можно скомпилировать модель (пункт меню редактора “модель | компилировать” или кнопка с желтой шестеренкой) и проверить ее работу. Для тестирования модели можно собрать схему, изображенную на рис. 365 или подобную ей. На рисунке вход блока `x` соединен с выходом текущего времени `Time` стандартного блока-планировщика. Следует иметь в виду, что `Time` – скрытый выход, и он не будет отображаться отдельным пунктом в меню подключения связи (см. §2.7.1 части I), для его выбора придется использовать пункт “список” в этом меню.

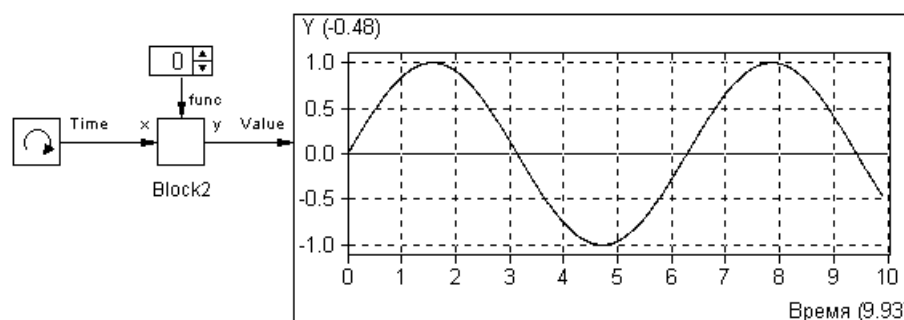


Рис. 365. Тестирование модели вычисления синуса и косинуса

Запустив расчет, можно будет наблюдать на графике синусоиду или косинусоиду, в зависимости от значения, поданного на `func`. Если вход `func` не будет равен нулю или единице, график рисоваться не будет, а в заголовке его вертикальной оси в скобках вместо текущего значения будет отображаться вопросительный знак – так в РДС выводится значение-индикатор ошибки.

В обоих приведенных выше примерах мы использовали переменные блока в программе так, как будто это переменные типов `int` и `double`, а не объекты специальных классов, за которыми скрыты настоящие значения (см. §3.7.1 на стр. 80). Об этом можно не задумываться до тех пор, пока где-нибудь в программе не потребуется указатель на переменную блока. В этом случае необходимо принимать специальные меры, что мы сейчас проиллюстрируем на примере.

Создадим модель блока, разбивающего поступившее на его вход “`x`” вещественное число на целую и дробную части и выдающего целую часть на выход “`integer`”, а дробную – на выход “`fraction`” (оба выхода сделаем вещественными). Блок будет иметь следующую структуру переменных:

<i>Имя</i>	<i>Тип</i>	<i>Вход/выход</i>	<i>Пуск</i>	<i>Начальное значение</i>
Start	Сигнал	Вход	✓	1
Ready	Сигнал	Выход		0
x	double	Вход	✓	0
integer	double	Выход		0
fraction	double	Выход		0

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (шаги, которые для этого необходимо выполнить, перечислены на стр. 95) и введем в модель приведенную выше структуру статических переменных. Для разбиения вещественного числа на целую и дробную части логично использовать функцию `modf` из стандартной математической библиотеки. В “`math.h`” она описана следующим образом:

```
double modf(double x, double *ipart);
```

Здесь `x` – разбиваемое число, `ipart` – указатель на переменную, в которую функция запишет целую часть числа, а дробную часть функция вернет по значению. Поскольку `x`, `integer` и `fraction` – статические переменные блока, имеющие тип “`double`”, возникает желание ввести на вкладку “модель” редактора следующий текст:

```
fraction=modf(x,&integer);
```

Однако, если скомпилировать такую модель, компилятор выдаст сообщение об ошибке примерно следующего содержания: “cannot convert ‘`rdsbcstDouble*`’ to ‘`double*`’ in function `rdsbcppBlockClass::rdsbcppModel`” (“невозможно преобразовать тип `rdsbcstDouble*` в тип `double*` в функции-члене `rdsbcppModel` класса `rdsbcppBlockClass`”). Действительно, несмотря на то, что в блоке переменная с именем “`integer`” имеет тип “`double`”, объект `integer` в классе блока, с которым мы работаем в программе, имеет тип `rdsbcstDouble` (результат работы макроса, создающего этот класс, приведен на стр. 86). Таким образом, выражение “`&integer`”, которое мы подставили во второй параметр функции `modf`, имеет тип “указатель на `rdsbcstDouble`”, а не “указатель на `double`”, и компилятор не знает, что с ним делать.

Из этой ситуации есть два выхода. Во-первых, можно использовать вспомогательную переменную для вызова `modf`, а затем присвоить ее значение объекту `integer`:

```
double i_aux;  
fraction=modf(x,&i_aux);  
integer=i_aux;
```

Здесь вводится локальная переменная `i_aux`, указатель на которую передается в `modf`. Это “настоящая” переменная типа `double`, поэтому никаких ошибок не возникнет. После вызова `modf` в `i_aux` будет находиться целая часть числа, и ее можно присвоить `integer` (операторы присваивания для класса `rdsbcstDouble` переопределены, никакие указатели здесь не используются, поэтому ошибок тоже не возникнет).

Во-вторых, и этот вариант будет гораздо короче, можно воспользоваться функцией-членом `GetPtr`, которая есть в любом классе, создаваемом модулем автокомпиляции для простых статических переменных: эта функция возвращает указатель на скрытую в объекте класса переменную блока. Для переменных блока типа “`double`”, обслуживаемых классом `rdsbcstDouble`, эта функция будет иметь тип “`double*`”, то есть “указатель на `double`”, что нам и требуется. Модель при этом будет выглядеть так:

```
fraction=modf(x,integer.GetPtr());
```

Вторым параметром в `modf` здесь передается указатель на вещественную переменную, скрытую в объекте `integer`, поэтому целая часть числа сразу будет записана в нужный выход блока.

На всякий случай, можно добавить в модель проверку входа на значение-индикатор ошибки. Если не добавлять эту проверку, функция `modf` запишет `HUGE_VAL` в переменную `integer`, а переменная `fraction` получит значение 0. Если такое поведение блока нам подходит, можно оставить в модели только вызов функции `modf`, как указано выше. Если же нужно сделать так, чтобы при поступлении значения ошибки на вход оба выхода получали это же значение, модель будет выглядеть так:

```
if (x==rdsbcppHugeDouble)  
{ fraction=integer=rdsbcppHugeDouble;  
  return;  
}
```

```
fraction=modf(x,integer.GetPtr());
```

Для тестирования созданной модели можно собрать схему, изображенную на рис. 366. Запустив расчет и вводя разные числа в поле ввода слева от блока, можно наблюдать на индикаторах справа целую и дробную части введенного числа.

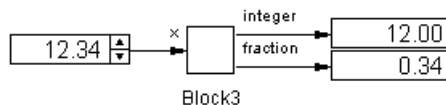


Рис. 366. Тестирование модели разбиения числа на целую и дробную части

§3.7.2.2. Модели с матрицами

Рассматриваются особенности использования матриц в моделях блоков. Описываются функции для работы с матрицами (в частности, для программного задания их размеров) и способ обращения к их элементам.

Матрица в РДС – это двумерная таблица переменных одного типа, в которой конкретный элемент определяется индексом строки и индексом столбца. Индексы строк и столбцов всегда начинаются с нуля. Тип элемента матрицы может быть любым, в том числе, и другой матрицей, но, чаще всего, используются матрицы простых типов – например, вещественных чисел. Как и любые статические переменные блока, матрицы могут быть входами, выходами и внутренними переменными. Размер матрицы, то есть число ее строк и столбцов, не фиксирован, и может изменяться в процессе расчета: матрицы-входы получают свои значения по связям, и их размеры определяются размерами подключенных к ним матриц-выходов других блоков, а размеры матриц-выходов и внутренних переменных задаются программно в модели блока. Матрица может быть пустой, то есть не иметь элементов (в этом случае считается, что ее размер – ноль строк на ноль столбцов).

Как и для простых статических переменных, для матриц модуль автокомпиляции создает специальные классы доступа и добавляет в класс блока `rdsbcppBlockClass` по одному объекту для каждой матрицы, причем имена этих объектов совпадают с именами переменных блока. В результате внутри фрагментов программ, вводимых пользователем, к матрицам, являющимся статическими переменными блока, можно обращаться просто по именам. Для доступа к конкретному элементу матрицы используется стандартный синтаксис языка С с квадратными скобками: если, например, матрица имеет имя `M`, ее элемент в строке `r` и столбце `c` записывается как `M[r][c]`. В классы матриц, создаваемые модулем автокомпиляции, включаются различные функции-члены для определения числа строк и столбцов в матрице, установки ее размера и т.п., все эти функции можно использовать в реакциях на события. Ниже приведены основные функции-члены классов матриц (во всех примерах предполагается, что `M` и `M1` – матрицы переменных типа `double`):

- *вспомогательный_тип* `operator[](int row)` – обращение к строке матрицы. Здесь `row` – целый номер строки, начинающийся с нуля, а *вспомогательный_тип* – специальный тип, такой, что применение к нему еще одного оператора `[col]` вернет элемент матрицы в строке `row` и столбце `col`. Например, для матриц вещественных чисел таким типом будет `double*`, т.е. “указатель на `double`”. Таким образом, `M[row][col]` позволяет обратиться к элементу матрицы `M`, находящемуся в строке `row` и столбце `col`. Такая запись может находиться как в левой, так и в правой части выражения, то есть можно не только получать значения элементов матриц, но и присваивать их. По умолчанию проверка индексов не производится, и попытка обратиться к элементу матрицы за пределами ее текущего размера вызовет критическую ошибку. Проверку индексов можно включить, установив в параметрах модели флажок “проверять индексы в массивах и матрицах (медленно)” (см. §3.6.7 на стр. 69), при этом попытка

обращения к элементу за пределами матрицы оператором “[]” вызовет остановку расчета и сообщение об ошибке. Следует учитывать, что включение этой проверки замедляет работу модели, поэтому проверять индексы в уже отлаженной модели следует вручную перед обращением к элементам матрицы. Примеры использования оператора:

```
M[r][c]=2.0;
double x=M[10][0];
double y=sin(M[r][c]);
```

- *тип_элемента* & Item(int row, int col) – обращение к элементу матрицы при помощи одной функции. Здесь row – номер строки, col – номер столбца (оба номера начинаются с нуля), *тип_элемента* – тип элемента матрицы (для матриц вещественных чисел, например, это будет тип double). Эта функция всегда выполняет проверку допустимости индексов независимо от установок параметров модели, поэтому она работает медленнее оператора “[]”. Как и указанный оператор, ее вызов может находиться и в левой, и в правой части выражения. Примеры использования функции:

```
M.Item(r,c)=2.0;
double x=M.Item(10,0);
double y=sin(M.Item(r,c));
```

- BOOL IsEmpty(void) – возвращает TRUE, если матрица пустая (0x0), и FALSE, если в ней есть элементы. Пример использования функции:

```
if (M.IsEmpty())
    return; // В матрице нет элементов
```

- BOOL HasData(void) – возвращает TRUE, если в матрице есть элементы, и FALSE, если она пустая. Пример использования функции:

```
if (M.HasData()) // Обработка данных матрицы
{ ... }
```

- int Cols(void) – число столбцов матрицы. Для пустой матрицы возвращается 0. Пример использования функции:

```
int ncolums=M.Cols();
```

- int Rows(void) – число строк матрицы. Для пустой матрицы возвращается 0. Пример использования функции:

```
int nrows=M.Rows();
```

- BOOL Resize(int rows, int cols, BOOL keep=FALSE) – изменить размер матрицы. Здесь rows – новое число строк матрицы, cols – новое число столбцов, необязательный параметр keep – TRUE, если при изменении размера нужно сохранить текущее содержимое матрицы, и FALSE, если ее всю нужно заполнить значением элемента по умолчанию. Если параметр keep не указан, после изменения размера вся матрица будет заполнена значением по умолчанию. Функция возвращает TRUE, если изменение размера выполнено успешно. В большинстве случаев результат возврата функции можно не проверять – по крайней мере, пока идет работа с матрицами обозримых размеров. Если передать в параметрах rows и cols нулевые значения, матрица станет пустой. Примеры использования функции:

```
// Установить размер 3x4
M.Resize(3,4);
// Добавить к матрице строку с сохранением содержимого
M.Resize(M.Rows()+1,M.Cols(),TRUE);
// Сделать матрицу пустой
M.Resize(0,0);
```

- *класс_матрицы* & operator=(const *класс_матрицы* &matr) – оператор присваивания, позволяющий копировать одну матрицу в другую (матрицы должны быть одинаковых типов). Здесь *класс_матрицы* – имя класса, созданного модулем автокомпиляции для матриц данного типа, matr – копируемая матрица. Необходимость полностью скопировать одну матрицу в другую возникает достаточно редко, тем не менее,

этот оператор позволяет выполнить ее без использования цикла по элементам. Следует учитывать, что обе матрицы должны обязательно быть одного и того же типа – нельзя, например, скопировать таким образом матрицу целых чисел в матрицу вещественных. Копирование матриц разного типа необходимо производить вручную поэлементно. Пример использования оператора:

`M1=M; // Скопировать матрицу M в матрицу M1`

В качестве примера создадим модель блока, который будет складывать матрицы вещественных чисел, поступающие на его входы “X1” и “X2” и выдавать результат на выход “Y”. Причем, если размеры “X1” и “X2” отличаются, будем считать недостающие в них строки и столбцы заполненными нулями – таким образом, размер выходной матрицы “Y” будет максимальным из размеров “X1” и “X2” и по числу строк, и по числу столбцов. Например, складывая матрицы 2x3 и 3x2, мы получим на выходе матрицу размером 3x3. Наш блок будет иметь следующую структуру переменных:

<i>Имя</i>	<i>Тип</i>	<i>Вход/выход</i>	<i>Пуск</i>	<i>Начальное значение</i>
Start	Сигнал	Вход	✓	1
Ready	Сигнал	Выход		0
X1	Матрица double	Вход	✓	[] 0
X2	Матрица double	Вход	✓	[] 0
Y	Матрица double	Выход		[] 0

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в его модель эту структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```
// Локальные переменные
int maxrows,maxcols;
double v1,v2;

// Определение максимального размера матриц X1 и X2
maxrows=max(X1.Rows(),X2.Rows());
maxcols=max(X1.Cols(),X2.Cols());

// Задание размера выходной матрицы
Y.Resize(maxrows,maxcols);

// Цикл по элементам
for(int r=0;r<maxrows;r++) // r - строка
    for(int c=0;c<maxcols;c++) // c - столбец
    { // v1 - элемент из X1
        if(r<X1.Rows() && c<X1.Cols()) // [r,c] - в X1
            v1=X1[r][c];
        else // [r,c] - за пределами X1
            v1=0.0;
        // v2 - элемент из X2
        if(r<X2.Rows() && c<X2.Cols()) // [r,c] - в X2
            v2=X2[r][c];
        else // [r,c] - за пределами X2
            v2=0.0;
        // Запись суммы в элемент Y
        Y[r][c]=v1+v2;
    }
```

Этот фрагмент программы будет выполняться в каждом такте расчета, перед которым сработали связи, подключенные к X1 или X2 (это обеспечат флажки в колонке “пуск”

напротив этих переменных). Сначала мы записываем во вспомогательную переменную `maxrows` наибольшее из чисел строк `X1` и `X2`, а в `maxcols` – наибольшее из чисел столбцов. Далее, вызывая функцию-член `Resize` у выходной матрицы `Y`, мы делаем размер этой матрицы равным `maxrows x maxcols`. Матрица `Y` готова, теперь нужно записать в ее элементы суммы соответствующих элементов матриц `X1` и `X2`.

Может показаться, что для поэлементного суммирования двух матриц и записи результата в третью можно использовать цикл следующего вида:

```
// Цикл по элементам
for(int r=0;r<maxrows;r++) // r - строка
    for(int c=0;c<maxcols;c++) // c - столбец
        Y[r][c]=X1[r][c]+X2[r][c];
```

Это было бы верно, если бы мы ограничились только суммированием матриц одинаковых размеров. Однако, мы решили при несовпадении размеров матриц дополнять их нулями, поэтому, прежде, чем обращаться к элементу “`X1[r][c]`” или “`X2[r][c]`”, нужно проверить, есть ли элемент (r,c) в каждой из этих матриц. Действительно, переменная `r` изменяется от нуля до `maxrows-1`, а `maxrows` – это максимальный из двух вертикальных размеров матриц. Если в `X1` будет две строки, а в `X2` – три, то `maxrows` будет равно трем, и, когда `r` примет значение `maxrows-1`, то есть два, выполнение оператора “`X1[r][c]`”, вероятнее всего, вызовет критическую ошибку, поскольку в `X1` всего две строки с индексами 0 и 1, и элемент `X1[2][c]` в ней отсутствует. Поэтому в цикле суммируются не непосредственно элементы матриц `X1` и `X2`, а вспомогательные переменные `v1` и `v2`, равные элементам соответствующих матриц, если в этих матрицах есть элемент (r,c) , и нулю в противном случае.

Можно заметить, что в этой модели нет проверки элементов входных матриц на значение `rdsbcppHugeDouble` (см. стр. 84). Можно не добавлять ее: единственная выполняемая в модели математическая операция – это сложение, а сложение значения `rdsbcppHugeDouble` с числом даст в результате то же самое значение `rdsbcppHugeDouble` (то есть `HUGE_VAL`).

Для тестирования созданной модели следует собрать схему, изображенную на рис. 367 (перед присоединением связей к созданному блоку необходимо скомпилировать модель, чтобы заданная в ней структура переменных была записана в этот блок). В этой схеме к входам блока присоединены стандартные библиотечные блоки ввода матриц вещественных чисел, а к выходу – стандартный блок отображения таких матриц. Эти блоки вводят и отображают матрицы в отдельных окнах, которые изображены на рисунке рядом с каждым блоком. Если теперь задать обе входные матрицы и запустить расчет, в окне выходной матрицы можно будет увидеть их сумму. На рисунке матрица `X1` имеет размер 2×3 , а `X2` – 3×2 , поэтому наш блок сложил эти матрицы, дополнив `X1` нулевой строкой снизу, а `X2` – нулевым столбцом справа, и результат получил размер 3×3 .

В описанном примере мы работали с матрицами вещественных чисел, точно так же можно работать с матрицами целых чисел, логических значений и других простых типов. Модуль автокомпиляции также позволяет использовать в блоках матрицы сложных типов: строк, структур и других матриц. Доступ к этим матрицам осуществляется точно так же, нужно только иметь в виду, что элемент такой матрицы сам по себе представляет собой сложный тип и имеет свои собственные функции-члены. Например, если `Z` – матрица матриц вещественных чисел, то допустимы следующие операции:

```
// Обращение к элементу [1,2] матрицы, находящейся в Z[3,4]
double x=Z[3][4][1][2];
// Задание размера матрицы, находящейся в Z[5,6]
Z[5][6].Resize(3,3);
```

```

// Задание размера Z и ее элемента и заполнение
// элементов этого элемента
Z.Resize(10,10);      // Задание размера Z
Z[0][0].Resize(5,5);  // Задание размера элемента Z[0,0]
for(int i=0;i<5;i++)  // Занесение значений
    for(int j=0;j<5;j++)
        Z[0][0][i][j]=i+j;

```

Если элементами матрицы являются структуры, у этих элементов можно обращаться к полям, как и у обычных переменных-структур. Точно так же, у строк, являющихся элементами матриц, можно вызывать те же функции-члены, что и у обычных переменных-строк. Работа со структурами и строками будет рассмотрена в §3.7.2.4 (стр. 107) и §3.7.2.5 (стр. 112) соответственно.

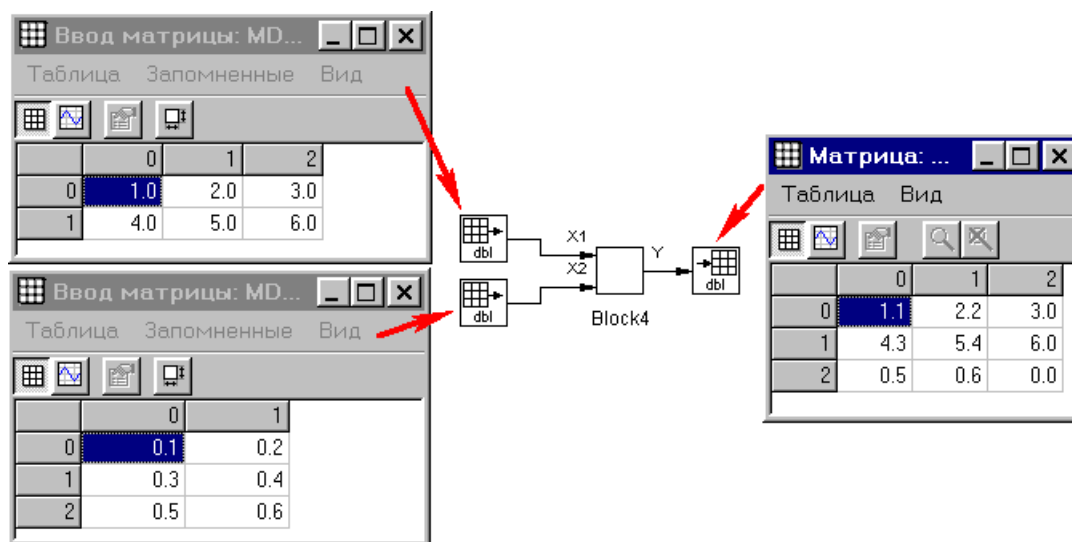


Рис. 367. Тестирование модели сложения матриц

§3.7.2.3. Модели с массивами

Рассматривается использование массивов в моделях блоков и описываются функции для работы с ними.

Технически, массивы в РДС – это матрицы, состоящие из единственной строки. Они хранятся в памяти точно так же, как и матрицы, и могут соединяться с матрицами при помощи связей. Однако, с точки зрения пользователя, массивы отличаются от матриц наличием одного индекса вместо двух: при подключении связи к отдельному элементу матрицы, как это описано в §2.7.3 части I, вводятся оба индекса элемента (например, “M[1,2]”), а при подключении связи к элементу массива – только один (например, “A[3]”).

Чаще всего массивы используются для создания блоков с произвольным числом входов и выходов, обрабатываемых одинаково. Сумматор, модель которого была рассмотрена в §3.3 (стр. 20), рассчитан только на два входа. Если пользователю потребуется складывать три числа, придется добавить в блок еще одну статическую переменную “x3” и изменить модель блока на “y=x1+x2+x3;”. Гораздо лучше было бы создать сумматор, число входов которого заранее не задано и определяется числом подсоединенных к нему связей. Этого можно добиться, сделав вход блока массивом вещественных чисел с нулевым значением элемента по умолчанию и выдавая на выход сумму всех элементов этого массива. Пользователь при этом сможет подключить столько связей к отдельным элементам этого входного массива, сколько ему будет нужно – размер массива будет определяться самым большим индексом элемента, к которому подключена связь. Если, например, подключить связи к элементам с номерами 0, 1, 2 и 5, размер массива окажется равным шести (к

элементам 3 и 4 связи не подключены, но они все равно будут присутствовать в массиве). Интерфейс РДС позволяет достаточно просто подключать связи к последовательным элементам массива – при присоединении очередной связи пользователю сразу предлагается следующий, еще не использованный, номер элемента – поэтому такой сумматор будет удобен в работе. Создадим такую модель, но сначала разберемся с классами, автоматически создаваемыми для доступа к массиву, и их функциями-членами.

Классы доступа для массивов в целом похожи на классы матриц, но отличаются от них поддержкой единственного индекса. В класс блока `rdsbcppBlockClass` добавляется по одному объекту для каждого массива, имена этих объектов совпадают с именами переменных блока. Для доступа к конкретному элементу массива используется стандартный синтаксис языка С с квадратными скобками: элемент n массива A записывается в программе как `A[n]`. Рассмотрим основные функции-члены классов массивов (во всех примерах предполагается, что X и Y – массивы переменных типа `double`):

- **тип_элемента & operator[] (int n)** – обращение к элементу массива. Здесь n – целый номер элемента, начинающийся с нуля, а **тип_элемента** – тип элемента массива (для матриц вещественных чисел, например, это будет тип `double`). Таким образом, `X[n]` позволяет обратиться к элементу массива X с номером n . Такая запись может находиться как в левой, так и в правой части выражения, то есть можно не только получать значения элементов массивов, но и присваивать их. Проверка допустимости индекса элемента по умолчанию не производится, и попытка обратиться к элементу за пределами текущего размера массива вызовет критическую ошибку. Проверку индексов можно включить, установив в параметрах модели флажок “проверять индексы в массивах и матрицах (медленно)” (см. §3.6.7 на стр. 69), при этом попытка обращения к элементу за пределами массива оператором `[]` вызовет остановку расчета и сообщение об ошибке. Следует учитывать, что включение этой проверки замедляет работу модели, поэтому проверять индексы желательно не автоматически, а вручную, перед обращением к элементам массива. Примеры использования оператора:

```
X[n]=2.0;
double x=X[5];
double y=sin(X[0]);
```

- **тип_элемента & Item(int n)** – обращение к элементу массива при помощи функции. Здесь n – целый номер элемента, начинающийся с нуля, **тип_элемента** – тип элемента массива. Эта функция всегда выполняет проверку допустимости индексов независимо от установок параметров модели, поэтому она работает медленнее оператора `[]`. Как и указанный оператор, ее вызов может находиться и в левой, и в правой части выражения. Примеры использования функции:

```
X.Item(n)=2.0;
double x=X.Item(5);
double y=sin(X.Item(0));
```

- **BOOL IsEmpty(void)** – возвращает `TRUE`, если массив пустой (не содержит элементов), и `FALSE` в противном случае. Пример использования функции:

```
if(X.IsEmpty())
    return; // В массиве нет элементов
```

- **BOOL HasData(void)** – возвращает `TRUE`, если в массиве есть элементы, и `FALSE`, если он пустой. Пример использования функции:

```
if(X.HasData()) // Обработка элементов массива
{ ... }
```

- **int Size(void)** – число элементов в массиве. Для пустого массива возвращается 0. Пример использования функции:

```
// Суммирование элементов
double s=0.0;
```


- ```

 for(int i=0;i<X.Size();i++)
 s+=X[i];

```
- `BOOL Resize(int size, BOOL keep=FALSE)` – изменить размер массива. Здесь `size` – новое число элементов, необязательный параметр `keep` – `TRUE`, если при изменении размера нужно сохранить текущее содержимое массива, и `FALSE`, если его нужно заполнить значением элемента по умолчанию. Если параметр `keep` не указан, после изменения размера весь массив будет заполнен значением по умолчанию. Функция возвращает `TRUE`, если изменение размера выполнено успешно (если не пытаться создавать массивы огромных размеров, не уместящиеся в память, результат возврата функции можно не проверять). Если передать в параметре `size` нулевое значение, массив станет пустым. Примеры использования функции:

```

// Установить размер в три элемента
X.Resize(3);
// Добавить в конец массива три новых элемента
X.Resize(X.Size()+3, TRUE);
// Очистить массив
X.Resize(0);

```

- `класс_массива & operator=(const класс_массива &arr)` – оператор присваивания, позволяющий копировать один массив в другой (оба должны иметь элементы одинаковых типов). Здесь `класс_массива` – имя класса, созданного модулем автокомпиляции для массивов с данным типом элементов, `arr` – копируемый массив. Следует учитывать, что оба массива должны обязательно быть одного и того же типа – нельзя, например, скопировать таким образом массив целых чисел в массив вещественных (копирование массивов разного типа необходимо производить вручную поэлементно). Пример использования оператора:

```
Y=X; // Скопировать массив X в массив Y
```

Рассмотрев функции, с помощью которых можно работать с массивами, создадим модель сумматора с произвольным числом входов, использующего входной массив вещественных чисел. Наш блок будет иметь следующую структуру переменных:

| <i>Имя</i> | <i>Тип</i>    | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|---------------|-------------------|-------------|---------------------------|
| Start      | Сигнал        | Вход              | ✓           | 1                         |
| Ready      | Сигнал        | Выход             |             | 0                         |
| X          | Массив double | Вход              | ✓           | [ ] 0                     |
| y          | double        | Выход             |             | 0                         |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели эту структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```

y=0.0;
for(int i=0;i<X.Size();i++)
 y+=X[i];

```

Модель получилась очень простой: сначала мы обнуляем выход `y`, а затем, в цикле, по очереди добавляем к нему все элементы входного массива `X`. Введенный нами фрагмент программы будет выполнен при первом запуске расчета (начальное значение сигнала запуска `Start` – единица), а также при срабатывании любой связи, соединенной с входом `X` или с каким-либо его отдельным элементом (это обеспечит флажок в колонке “пуск” напротив входа `X`). Здесь, как и в примере модели, работающей с матрицами (см. §3.7.2.2 на стр. 99), мы не сравниваем элементы входного массива со значением `rdsbcppHugeDouble` (см. стр. 84) – для операции сложения это не обязательно.

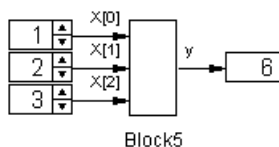


Рис. 368. Тестирование модели сумматора с входом-массивом

Для тестирования созданной модели можно собрать схему, изображенную на рис. 368, в которой к отдельным элементам массива  $X$  подключены поля ввода. Если запустить расчет, на индикаторе, подключенном к выходу  $y$ , появится сумма элементов массива. Для того, чтобы к блоку было удобнее подключать связи, параметры его внешнего вида изменены: в окне параметров (см. §2.9.1 части I) для него было назначено изображение в виде прямоугольника с текстом (в данном случае текст пустой) и разрешено масштабирование, после чего его вертикальный размер был увеличен.

Созданную нами модель можно использовать не только для суммирования чисел, поданных по связям на отдельные элементы массива. Поскольку в РДС разрешается соединять связями массивы и матрицы, мы можем подать на вход нашего блока какую-либо матрицу или массив с выхода другого блока – например, с выхода стандартного библиотечного блока ввода матрицы (рис. 369). В этом случае наша модель вычислит сумму всех элементов матрицы, даже если в ней не одна строка, а несколько: внутри модели матрица будет считаться одним длинным массивом. Например, если мы подадим на вход блока матрицу размером  $2 \times 3$  элемента, мы получим массив размером в шесть элементов.

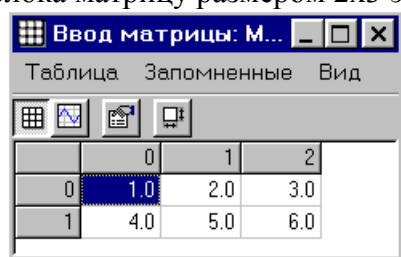


Рис. 369. Матрица на входе-массиве



В приведенном выше примере мы имели дело с массивом-входом, размер которого задается без нашего участия: его определяет либо число связей, подключенных к элементам массива, как на рис. 368, либо размер массива или матрицы на выходе блока, соединенного связью со всем входом  $X$ , как на рис. 369. Рассмотрим теперь модель, в которой

размер массива-выхода мы будем задавать самостоятельно. Создадим модель блока-демультиплексора, выходом которого будет массив вещественных чисел “ $Y$ ”. Блок будет иметь целый вход “ $N$ ” и вещественный вход “ $x$ ”, в процессе работы он будет копировать значение “ $x$ ” в элемент выходного массива с номером, определяемым входом “ $N$ ”.

Блок будет иметь следующую структуру переменных:

| Имя   | Тип           | Вход/выход | Пуск | Начальное значение |
|-------|---------------|------------|------|--------------------|
| Start | Сигнал        | Вход       | ✓    | 1                  |
| Ready | Сигнал        | Выход      |      | 0                  |
| x     | double        | Вход       | ✓    | 0                  |
| N     | int           | Вход       | ✓    | 0                  |
| Y     | Массив double | Выход      |      | [ ] 0              |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и установим для него эту структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```

if (N<0) return; // Недопустимый номер
// Проверка и увеличение размера массива
if (Y.Size()<N+1)
 Y.Resize(N+1,TRUE);
// Запись x в элемент массива N
Y[N]=x;

```

Введенный нами фрагмент программы, как и в предыдущей модели, будет выполнен при первом запуске расчета (начальное значение сигнала запуска *Start* – единица), а также при срабатывании любой связи, соединенной с входами блока (это обеспечат флажки в колонке “пуск” у обоих входов).

Первым оператором мы принудительно завершаем модель при отрицательных *N* – элементы массива не имеют отрицательных номеров. Затем мы проверяем, есть ли на данный момент элемент с номером *N* в массиве *Y*: чтобы он существовал, размер массива *Y.Size()* должен быть не меньше *N+1* (если в массиве, например, три элемента, самый последний его элемент имеет номер два, поскольку номера начинаются с нуля: 0, 1, 2). Если размер массива окажется меньше, мы принудительно устанавливаем его в *N+1*, вызывая у объекта *Y* функцию-член *Resize*. Во втором параметре этой функции передано значение *TRUE*, чтобы при увеличении размера массива его старое содержимое сохранилось, а не было заменено на значение по умолчанию. После этого мы просто присваиваем элементу *Y[N]* значение *x*. В этой модели не нужно никаких проверок входного значения *x* на равенство значению-индикатору ошибки *rdsbcppHugeDouble*: модель вообще не выполняет никаких математических вычислений, а просто переписывает на выход значение со своего входа.

Для тестирования модели можно собрать схему, изображенную на рис. 370. Здесь, как и в прошлом примере, размер нашего блока увеличен, чтобы к нему удобнее было подключать связи. К отдельным элементам массива *Y* подключены индикаторы, к входам *x* и *N* – поля ввода. Если запустить расчет и изменять значение на входе *x*, можно будет увидеть, что вместе с ним изменяется значение того индикатора, который подключен к элементу массива с номером *N*.

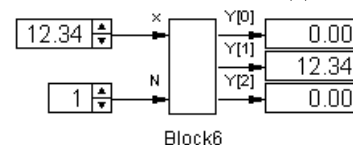


Рис. 370. Тестирование модели демультиплексора

Созданная нами модель имеет один недостаток: независимо от значения *N*, при ее срабатывании активируются все ее выходные связи. Так устроена логика работы блоков и связей в РДС – перед выполнением реакции блока на такт расчета его сигнал готовности *Ready* получает единичное значение, а в конце такта для всех блоков с ненулевыми сигналами готовности запускается передача выходов по связям (см. §1.3 части I). В схеме на рис. 370 к выходам блока подключены индикаторы, поэтому это не важно: многократного повторного срабатывания “лишних” связей мы не заметим, поскольку по ним будут передаваться те же самые числа. В частности, по связям, идущим от *Y[0]* и *Y[2]* при каждом изменении *x* будут повторно передаваться нули, несмотря на то, что *N* на рисунке равно единице. Но лучше подавлять передачу ненужных данных: во-первых, это ускоряет работу схемы, поскольку не активируются модели подключенных к “лишним” связям цепочек блоков, во-вторых, некоторые сложные блоки реагируют не только на число, пришедшее по связи, но и на факт ее срабатывания. Позже, в §3.7.2.8 (стр. 126), мы изменим нашу модель так, чтобы на выходе блока срабатывала только одна связь – та, которая подключена к элементу массива, соответствующему текущему значению *N*.

#### §3.7.2.4. Модели со структурами

Рассматривается использование структур в моделях блоков.

Структуры используются в тех случаях, когда необходимо передавать от блока к блоку несколько значений одновременно. Можно, конечно, сделать все эти значения отдельными выходами и входами блоков, но при этом пользователю придется проводить большое количество параллельных связей, которые, во-первых, загромождают схему, и, во-вторых, повысят вероятность совершения ошибки: если пользователь забудет провести одну из этих связей, схема не будет работать, а обнаружить эту пропущенную связь в сложной

схеме может оказаться не очень просто. В том случае, если все передаваемые значения имеют один и тот же тип, для этого можно использовать массив или матрицу. Например, в блоках, обрабатывающих трехмерные векторы, можно сделать входами и выходами массивы вещественных чисел. Однако, при этом разработчик модели такого блока должен все время контролировать размер массива на входе (он должен всегда содержать три элемента) и помнить, какой номер элемента какой именно координате вектора соответствует. Соответствие координат номерам элементов массива нужно помнить и пользователю, который будет работать с этими блоками: чтобы вывести на индикацию нужную ему координату, пользователь должен подключить индикатор к элементу массива с соответствующим ей номером. Если же кроме вещественных чисел между блоками должна передаваться какая-либо еще информация (например, единица измерения вектора или его текстовое название), передавать данные массивом уже не получится – все элементы массива должны иметь один и тот же тип.

Для одновременной передачи разнородных данных, каждый элемент которых имеет свое собственное имя, в РДС используются структуры (см. §2.14 части I). Структуры состоят из полей, которым можно дать имена, отражающие их назначение: например, структура, описывающая трехмерный вектор, может иметь поля с именами “x”, “y” и “z”, по названиям координатных осей. Запись “v1.x” при этом гораздо более удобна и информативна для пользователя, чем “v1[0]”, которую пришлось бы использовать при передаче векторов через массив. Поля структур могут иметь любой тип, кроме массива (вместо массивов можно использовать матрицы) – ими могут быть вещественные числа, строки, матрицы и другие структуры. В программе модели и при присоединении связей имя поля структуры отделяется от имени переменной, имеющей структурный тип, точкой, как принято в языке C: обращение к полю “x” структуры, находящейся в переменной “v”, записывается как “v.x”. Перед первым использованием структуры в блоке ее необходимо описать в РДС при помощи пункта главного меню “система | структуры” – необходимые для этого действия подробно рассмотрены в §2.14 части I. Если в схеме уже есть хотя бы один блок, использующий данную структуру, описывать ее не нужно: описание структуры автоматически загружается вместе с любым блоком, использующим ее.

Технически работа со структурами в автокомпилируемых моделях устроена так: для любой структуры, использованной в качестве типа какой-либо переменной блока, модуль автокомпиляции создает в программе модели специальный класс доступа, содержащий внутри себя все поля этой структуры, и добавляет в класс блока `rdsbcppBlockClass` по одному объекту этого созданного класса для каждой переменной-структуры с таким типом. Имена этих объектов совпадают с именами переменных блока. В результате к полям структуры внутри фрагментов программ, вводимых пользователем, можно обращаться, как и было указано выше, через точку: “*имя переменной.имя поля*”, где “*имя переменной*” – это имя созданного внутри класса блока объекта для переменной, а “*имя поля*” – имя поля специального класса доступа, созданного для поля структуры.

В качестве первого примера создадим собственную структуру для хранения трехмерного вектора и модель блока, который будет складывать два таких вектора. Начнем с описания структуры. Назовем ее “Vector3D”, и создадим в ней три вещественных поля: “x”, “y” и “z”. Чтобы добавить в РДС новую структуру, нужно выполнить следующие шаги:

- в режиме редактирования (должна быть загружена какая-либо схема или создана новая) выбрать пункт главного меню “система | структуры” – откроется окно со списком уже имеющихся в РДС структур;
- в окне структур нажать на кнопку со знаком “+” в правой части окна – откроется пустое окно редактирования структуры;
- ввести в окне редактирования имя типа структуры “Vector3D” и заполнить список полей согласно рис. 371;
- закрыть окно редактирования структуры кнопкой “ОК”;

- закрыть окно списка структур кнопкой “OK”.

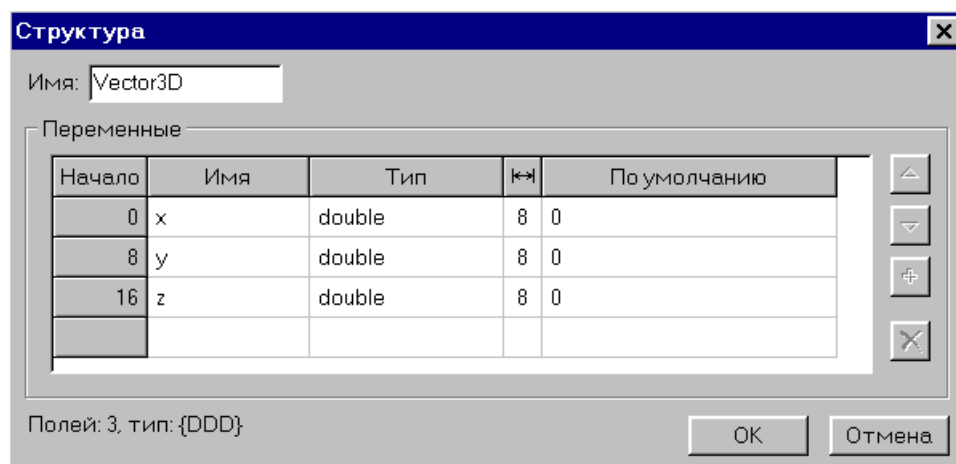


Рис. 371. Структура “Vector3D” в окне редактирования структуры

Теперь в текущей загруженной схеме есть структура с именем “Vector3D” и тремя вещественными полями. Это имя будет появляться в выпадающем списке типов при задании структуры переменных блока.

Создадим модель блока, который будет суммировать два трехмерных вектора, поступивших на входы “v1” и “v2”, и выдавать результат на выход “sum”. Он будет иметь следующую структуру переменных:

| Имя   | Тип      | Вход/выход | Пуск | Начальное значение |
|-------|----------|------------|------|--------------------|
| Start | Сигнал   | Вход       | ✓    | 1                  |
| Ready | Сигнал   | Выход      |      | 0                  |
| v1    | Vector3D | Вход       | ✓    | {0, 0, 0}          |
| v2    | Vector3D | Вход       | ✓    | {0, 0, 0}          |
| sum   | Vector3D | Выход      |      | {0, 0, 0}          |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе его модели указанную выше структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```
sum.x=v1.x+v2.x;
sum.y=v1.y+v2.y;
sum.z=v1.z+v2.z;
```

Здесь мы просто складываем одноименные поля структур в переменных v1 и v2 и присваиваем результат полю переменной sum с этим же именем. Имена полей отделяются от имен переменных точками. Как и во всех предыдущих примерах, наша модель будет запускаться автоматически при срабатывании любой связи, подключенной к входам v1 и v2 (в колонке “пуск” для этих входов установлены флажки).

Для тестирования модели можно собрать схему, изображенную на рис. 372 (для того, чтобы к блоку было удобнее подключать связи, ему был назначен внешний вид в виде прямоугольника с текстом, после чего его размер был увеличен).

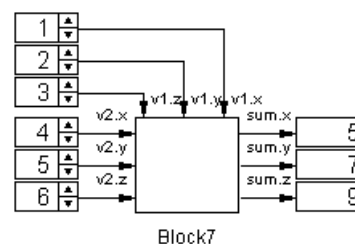


Рис. 372. Тестирование модели сложения векторов

покомпонентно просуммированы. Может показаться, что сделав входы и выход блока структурами, мы ничего не добились: к блоку все равно подключено большое количество связей – по одной связи на каждое поле каждой структуры. Однако, это произошло только потому, что у нас нет стандартного блока, позволяющего ввести или отобразить созданную нами структуру, и мы вынуждены использовать обычные вещественные поля ввода и индикаторы, подключая их к отдельным полям структур на входах и выходе нашего блока. Если бы у нас были такие блоки (их можно, при желании, создать), к блоку подходило бы всего три связи: по одной от гипотетических блоков ввода векторов, и одна – к блоку индикации вектора.

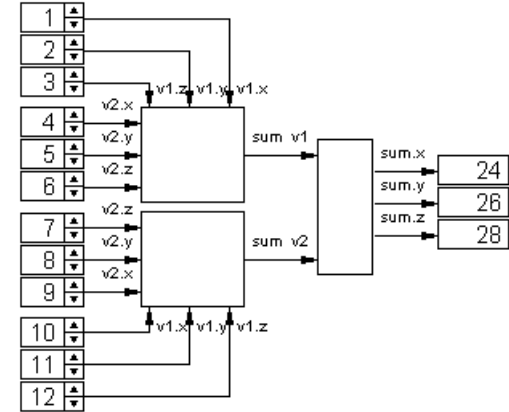


Рис. 373. Сложение четырех векторов

Соберем еще одну схему (рис. 373), в которой будем складывать четыре вектора при помощи трех блоков с созданной нами моделью. Уже созданный блок можно размножить, копируя его в буфер обмена клавишами Ctrl+C и вставляя оттуда клавишами Ctrl+V, при этом на запрос модуля автокомпиляции о том, нужно ли создавать новую модель для копии блока, следует отвечать “использовать тот же файл модели” (см. рис. 327 на стр. 30). В новой схеме четыре вектора разбиты на две пары: сначала два блока слева складывают векторы внутри каждой пары, а затем точно такой же блок справа складывает получившиеся суммы между собой. Можно заметить, что от каждого из двух левых блоков-сумматоров к

правому идет только одна связь: “sum → v1” и “sum → v2”. Каждая из этих связей передает всю трехкомпонентную структуру “Vector3D”, и это значительно удобнее, чем передавать все три ее поля отдельными связями.

Суммировать несколько векторов, строя из сумматоров каскады, как на рис. 373, не очень удобно, поэтому сделаем еще одну модель сумматора векторов, входом которого будет массив структур “Vector3D”. Так мы сможем проиллюстрировать обращение к полям структур, являющихся элементами массивов. Назовем вход нашего блока “V”, а выход – “sum”. Если мы будем создавать этот блок в той же схеме, в которой мы работали с предыдущим, описывать структуру “Vector3D” не понадобится, поскольку она в этой схеме уже есть. Если создавать новый блок в новой схеме, нужно либо повторить в ней описание структуры, либо просто вставить в нее (например, из буфера обмена) уже созданный нами ранее блок, использующий эту структуру (структура при этом вставится в схему вместе с блоком). Потом этот блок можно стереть, описание структуры останется в схеме.

Наш новый блок будет иметь следующую структуру переменных:

| Имя   | Тип             | Вход/выход | Пуск | Начальное значение |
|-------|-----------------|------------|------|--------------------|
| Start | Сигнал          | Вход       | ✓    | 1                  |
| Ready | Сигнал          | Выход      |      | 0                  |
| V     | Массив Vector3D | Вход       | ✓    | [ ]{0, 0, 0}       |
| sum   | Vector3D        | Выход      |      | {0, 0, 0}          |

Создадим уже неоднократно описывавшимся способом новый блок с новой автокомпилируемой моделью и зададим в этой модели приведенную выше структуру переменных. На вкладке “модель” введем следующий текст:

```
// Обнуление перед суммированием
sum.x=sum.y=sum.z=0;
```

```
// Сложение векторов в цикле
for(int i=0;i<V.Size();i++)
{
 sum.x+=V[i].x;
 sum.y+=V[i].y;
 sum.z+=V[i].z;
}
```

Здесь мы сначала обнуляем все три поля выхода блока, а затем, в цикле по всем элементам массива (см. §3.7.2.3 на стр. 103), добавляем к ним одноименные поля элементов этого массива. Обращение к полю  $x$  элемента  $i$  массива  $V$  при этом выглядит обычным для языка C образом: “ $V[i].x$ ”.

Для тестирования модели соберем схему, изображенную на рис. 374. На ней девять полей ввода слева подключены к полям отдельных элементов входного массива  $V$ . Эти поля не будут появляться в меню присоединения связи к блоку, поскольку они находятся слишком “глубоко” в иерархии переменных, поэтому при присоединении связи необходимо будет выбирать в меню пункт “список” и вручную вводить имена “ $V[0].x$ ”, “ $V[0].y$ ” и т.д. (см. §2.7.3 части I). Это снова происходит из-за того, что у нас нет стандартного блока для ввода нашей структуры и мы вынуждены пользоваться обычными вещественными полями ввода. Если бы наш сумматор получал данные с выходов других блоков, имеющих тип “Vector3D”, присоединять связи было бы проще – открывалось бы окно выбора элемента массива с заранее установленным первым свободным номером.

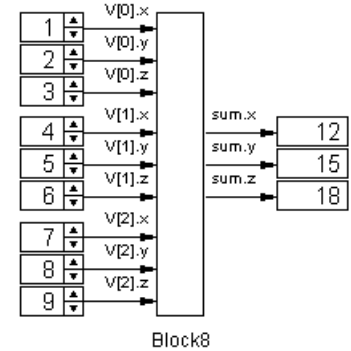


Рис. 374. Сложение массива векторов

В обоих примерах полями структуры были вещественные числа. Если бы полями были другие структуры или матрицы, индексы элементов матриц или поля этих вложенных структур записывались бы, как обычно, после имени поля-структуры или поля-матрицы. Представим себе, например, некоторую структуру с именем “SomeStruct”, имеющую поля “vector” типа “Vector3D” и “vmatr” типа “матрица Vector3D” (рис. 375).

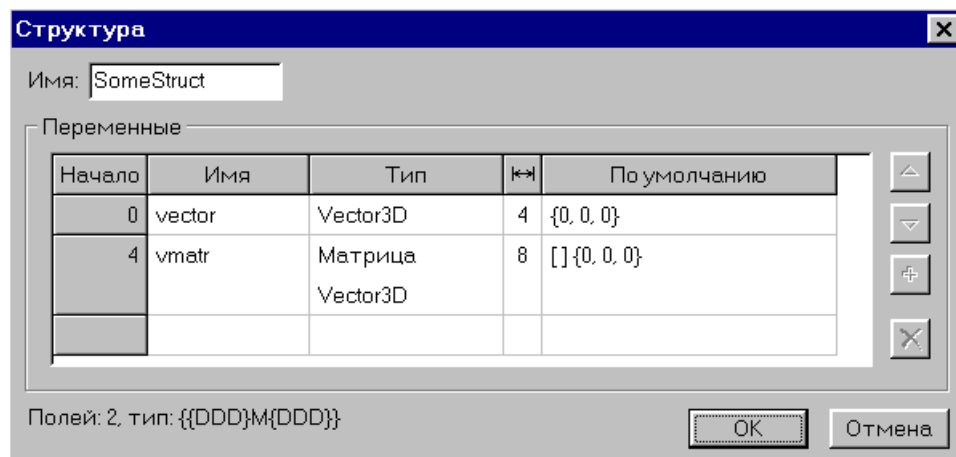


Рис. 375. Гипотетическая структура “SomeStruct”

Пусть в каком-либо блоке с автокомпилируемой моделью есть переменная “var” типа “SomeStruct”. В программе модели этого блока к отдельным полям и элементам этой сложной переменной можно обращаться следующим образом:

- “var.vector” – поле vector переменной var (это поле будет структурой типа “Vector3D”);
- “var.vector.y” – вещественное поле  $y$  структуры, находящейся в поле vector переменной var;

- “var.vmatr[1][2]” – структура типа “Vector3D”, находящаяся в строке 1 и столбце 2 матрицы из поля vmatr переменной var;
- “var.vmatr.Rows()” – число строк матрицы, находящейся в поле vmatr переменной var;
- “var.vmatr[1][2].x” – вещественное поле x структуры типа “Vector3D”, находящейся в строке 1 и столбце 2 матрицы из поля vmatr переменной var;
- и т.д.

Таким образом, способ обращения к полям структур во всех этих случаях ничем не отличается от стандартного синтаксиса языка C/C++.

### §3.7.2.5. Модели со строками

Рассматривается использование текстовых строк в переменных автокомпилируемых блоков. Описываются операторы и функции-члены класса для работы со строками.

Строки используются в качестве переменных блоков достаточно редко. Как правило, схемы, собираемые в РДС, предназначены для моделирования каких-либо технических процессов, в которых главную роль играют операции с числами, массивами и матрицами, которые могут сами быть входами и выходами блоков или содержаться внутри каких-либо структур. Тем не менее, РДС позволяет передавать строки по связям и работать с ними в моделях блоков. Чаще всего переменные-строки используются в блоках, отображающих какие-либо данные при помощи своих векторных картинок (см. §2.10 части I): блоки текста, являющиеся графическими элементами таких векторных картинок, могут быть связаны с переменными блока и автоматически отображать их значения. Если связать такой текстовый блок с переменной-строкой, на нем можно будет выводить любой текст по желанию разработчика модели, для этого достаточно присвоить нужный текст этой переменной.

Для работы со строками в автокомпилируемых моделях используется специальный класс с именем rdsbcppString. Для каждой переменной-строки в класс блока rdsbcppBlockClass добавляется по одному объекту класса rdsbcppString, через который и осуществляется работа с этой переменной. Имена этих объектов совпадают с именами переменных блока, поэтому внутри фрагментов программ, вводимых пользователем, к строкам, являющимся статическими переменными блока, можно обращаться просто по именам. Класс rdsbcppString имеет несколько переопределенных операторов и функций-членов, облегчающих работу со строками (во всех примерах далее будем считать, что str, str1 и str2 – это переменные-строки какого-либо блока):

- char \*c\_str(void) – указатель на стандартную строку “char\*”, хранящуюся внутри переменной блока (тип “char\*” используется в большинстве стандартных функций языка C, работающих со строками). Если строка пуста, возвращает указатель на статическую пустую строку, то есть эта функция никогда не возвращает значение NULL, даже если строка в переменной блока отсутствует. Пример использования функции:

```
// Преобразование строки str в целое число n
int n=atoi(str.c_str());
```

- BOOL IsEmpty(void) – возвращает TRUE, если строка пуста (не имеет символов) и FALSE в противном случае. Пример использования функции:

```
// Преобразование строки str в целое число n
if(str.IsEmpty())
{ // Строка str пуста
 ...
}
```

- int Length(void) – возвращает число символов в строке. Для пустой строки возвращается ноль. Пример использования функции:

```
int len=str.Length();
```



- `char & operator[](int n)` – символ строки номер `n` (нумерация начинается с нуля). Проверка допустимости индекса `n` не производится, попытка обратиться к символу за пределами строки (а также к любому символу пустой строки) вызовет критическую ошибку. Следует учитывать, что строки, как и положено в языке C, завершаются нулевым байтом. Выражение с этим оператором может находиться как в левой, так и в правой части оператора присваивания, то есть можно не только получать значения отдельных символов в строке, но и присваивать их. Примеры использования оператора:

```
// Запись первого символа строки str в переменную c
char c;
if(str.IsEmpty()) c=0;
else c=str[0];
// Замена всех символов в строке str на пробелы
for(int i=0; i<str.Length(); i++)
 str[i]=' ';
```

- `rdsbcppString operator+(const rdsbcppString &val)` – переопределенный оператор “+”, позволяющий складывать строки-переменные блоков. Пример использования оператора:
 

```
str=str1+str2;
```
- `rdsbcppString operator+(char *val)` – переопределенный оператор “+”, позволяющий складывать строку-переменную блока и обычную строку “char\*”. Пример использования оператора:
 

```
str=str1+"текст";
```
- `rdsbcppString & operator+=(const rdsbcppString &val)` – оператор “+=”, позволяющий добавить к концу строки в переменной блока другую строку из другой переменной. Пример использования оператора:
 

```
str+=str1;
```
- `rdsbcppString & operator+=(char *val)` – оператор “+=”, позволяющий добавить к концу строки-переменной блока обычную строку “char\*”. Пример использования оператора:
 

```
str+="текст";
```
- `rdsbcppString & operator=(const rdsbcppString &val)` – оператор присваивания, копирующий в данную переменную блока строку из переменной `val`. Пример использования оператора:
 

```
str=str1;
```
- `rdsbcppString & operator=(char *val)` – оператор присваивания, копирующий в данную переменную блока текст `val`. Пример использования оператора:
 

```
str="текст";
```

В большинстве случаев работа со строками в модели блока производится следующим образом: при помощи оператора `c_str()` считываются строки из входных переменных, затем при помощи стандартных функций библиотек C с этими строками выполняются какие-либо действия, результаты которых присваиваются выходным переменным блока. Если необходимо просто собирать длинный текст из нескольких строк, вместо библиотечных функций C можно использовать оператор сложения, переопределенный в классе строки `rdsbcppString`.

В качестве примера создадим блок, который будет отображать строку, представляющую собой разделенный запятыми список всех элементов его входного вещественного массива “X”, число знаков после десятичной точки в которых будет определяться целым входом “Dec”. Переменную блока, в которой будет записана сформированная строка, назовем “str”, ее можно сделать внутренней переменной или выходом блока. Таким образом, структура переменных у нашего блока будет такая:

| <i>Имя</i> | <i>Тип</i>    | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|---------------|-------------------|-------------|---------------------------|
| Start      | Сигнал        | Вход              | ✓           | 1                         |
| Ready      | Сигнал        | Выход             |             | 0                         |
| X          | Массив double | Вход              | ✓           | [ ]0                      |
| Dec        | int           | Вход              | ✓           | 0                         |
| str        | Строка        | Выход             |             |                           |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```
char buf[100]; // Вспомогательный массив

if(X.IsEmpty()) // Массив пуст
{ str=""; // Отображаем пустую строку
 return;
}
// В массиве есть элементы

if(Dec>10) // Выводим не более 10 символов после точки
 Dec=10;

str="("; // Список начнется со скобки
for(int i=0;i<X.Size();i++) // Цикл по всем элементам
{ if(i) // Если не первый элемент, добавляем запятую
 str+=", ";
 if(X[i]==rdsbcrpHugeDouble) // Признак ошибки
 str+="?";
 else
 { // Формируем текст в массиве buf
 sprintf(buf,"%.*lf",(int)Dec, (double)X[i]);
 str+=buf; // Дописываем buf к str
 }
}
str+=")"; // Закрывающая скобка списка
```

В этом фрагменте программы для преобразования вещественного числа в текст с заданной точностью используется стандартная библиотечная функция `sprintf`, описанная в файле “`stdio.h`”. Этот файл не включается в формируемый модуль автокомпиляции текст программы по умолчанию, поэтому директиву его включения необходимо записать вручную в раздел глобальных описаний. Для этого в редакторе модели необходимо выполнить следующие действия:

- на дополнительной левой панели редактора выбрать вкладку “события” (см. §3.6.4 на стр. 46);
- на этой вкладке раскрыть раздел “описания” (это самый первый раздел в списке), щелкнув левой кнопкой мыши на значке “+” слева от него;
- дважды щелкнуть на открывшемся подразделе “глобальные описания”, после чего в правой части окна редактора рядом с вкладкой “модель” появится новая вкладка “описания”.

На открывшейся вкладке “описания” следует ввести следующий текст из одной строки:

```
#include <stdio.h>
```

Теперь, когда директива включения недостающего файла заголовков добавлена, рассмотрим приведенную выше программу, введенную на вкладке “модель”. Эта программа будет

выполняться при каждом срабатывании связи, подключенной к любому из входов блока, то есть при изменении содержимого входного массива `X` или точности отображения его элементов `Dec`.

В самом начале программы описывается вспомогательный массив `buf` из ста символов. Он будет использоваться для преобразования числа в строку – девяносто девяти символов (с учетом завершающего строку нулевого байта) хватит для вывода числа типа `double` с любой разумной точностью. Поскольку текст реакции на любое событие, вводимый в редакторе модели, вставляется внутрь сформированных модулем автокомпиляции функций, массив `buf` будет локальным для функции реакции на такт расчета.

Далее, если во входном массиве `X` нет ни одного элемента (`X.IsEmpty()` вернет `TRUE`), переменной `str` присваивается пустая строка и выполнение реакции завершается оператором `return` – массив пуст, и отображать нечего. В противном случае значение числа символов после десятичной точки `Dec` ограничивается значением 10: если значение переменной `Dec > 10`, ей принудительно присваивается 10. Это ограничение сделано для того, чтобы строки, в которые преобразуются значения каждого элемента массива `X`, гарантированно не превысили размеров массива `buf`. Можно было бы написать программу так, чтобы такое ограничение не потребовалось (например, с использованием сервисной функции РДС `rdsDtoA`, описанной в пункте А.5.4.5 приложения к руководству программиста [2]), но мы не будем усложнять этот пример. Можно заметить, что мы присваиваем значение переменной, являющейся входом блока – это вполне допустимо, присвоенное значение останется в переменной `Dec` до следующего срабатывания подключенной к ней связи.

Далее мы присваиваем переменной `str` открывающую круглую скобку, а затем, в цикле по всем элементам массива `X` (см. §3.7.2.3 на стр. 103), добавляем в конец этой строки при помощи оператора `+="` текст для каждого элемента. Внутри этого цикла мы для каждого элемента, кроме первого (то есть при значении счетчика цикла `i`, не равном нулю), добавляем к строке запятую и пробел, отделяющий элементы друг от друга, и, если `X[i]` не равно значению ошибки `rdsbcppHugeDouble` (см. стр. 84), формируем в массиве `buf` текстовое представление вещественного числа с заданной точностью и добавляем этот текст к `str` (для значения ошибки вместо значения элемента добавляется вопросительный знак). После цикла к `str` добавляется закрывающая круглая скобка.

Вещественное число в цикле преобразуется в строку в массиве `buf` при помощи функции `sprintf`:

```
sprintf(buf, "%. *lf", (int)Dec, (double)X[i]);
```

Строка формата `"%. *lf"` указывает на то, что выполняется преобразование вещественного числа двойной точности (`"lf"`) с числом знаков после десятичной точки, указанным в параметре функции перед самим числом (`"." *`). В третьем и четвертом параметрах `sprintf` передаются точность представления числа `Dec` и само число `X[i]`. Поскольку `sprintf` – функция с переменным числом параметров, тип которых явно не указан в ее описании, компилятор не сможет самостоятельно преобразовать типы объектов, созданных модулем автокомпиляции для переменных блока (напомним, что, например, `Dec` – это именно объект некоторого специально сформированного класса, а не переменная типа `int`). Чтобы избежать разночтений, для `Dec` и `X[i]` явно вызываются операторы преобразования к типам `int` и `double` соответственно (см. также §3.7.1 на стр. 80).

Мы создали модель блока, преобразующую массив вещественных чисел в строку со значениями всех его элементов, но сам блок пока ничего отображать не может – он выглядит как пустой белый квадрат. Чтобы строка появлялась на изображении блока, необходимо задать для него векторную картинку согласно §2.10 части I. Создадим картинку, состоящую из прямоугольника и размещенного внутри него блока текста, связанного с переменной

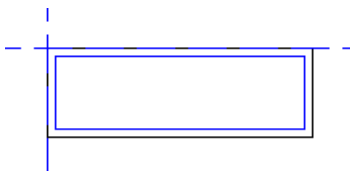


Рис. 376. Внешний вид блока в редакторе картинки

блока `str` (рис. 376). Картинку можно ввести из редактора модели через команду установки параметров блоков (см. §3.6.8 на стр. 76) или в окне параметров созданного нами блока. В последнем случае необходимо выполнить следующие шаги:

- в режиме редактирования нажать на изображении блока правую кнопку мыши и выбрать в контекстном меню пункт “параметры” – откроется окно параметров блока;
- на вкладке окна “внешний вид” установить флажок “определяется картинкой” и нажать кнопку “изменить” на панели “картинка” в левой нижней части вкладки – откроется редактор картинки;
- в редакторе картинки создать прямоугольник и блок текста, расположить блок текста поверх и внутри прямоугольника, как на рис. 376;
- дважды щелкнуть левой кнопкой мыши на блоке текста, выбрать в открывшемся окне его параметров вкладку “связи” и выбрать в выпадающем списке на этой вкладке переменную “`str`” (если это имя в выпадающем списке будет отсутствовать, а это возможно, если мы еще не компилировали модель, имя можно ввести вручную);
- закрыть окно параметров текста кнопкой “ОК”;
- закрыть окно редактора (как обычно в Windows, кнопкой с крестиком в правой верхней части окна);
- закрыть окно параметров блока кнопкой “ОК”.

Теперь наш блок в режимах моделирования и расчета будет отображать значение переменной `str`. Для тестирования созданной нами модели можно собрать схему, изображенную на рис. 377. В ней к входному массиву блока `X` подключен стандартный блок ввода матриц (в РДС матрицы можно соединять с массивами), а к целому входу `Dec` – обычное поле ввода. Задав на входе `X` какую-нибудь матрицу-строку и запустив расчет, на изображении блока можно будет увидеть текст из строки `str`, то есть элементы этой матрицы, перечисленные через запятую. Если размеров блока недостаточно для отображения строки, она будет обрезана по границам блока текста, заданного в редакторе картинки – в этом случае можно снова открыть редактор картинки блока и изменить размеры ее элементов.

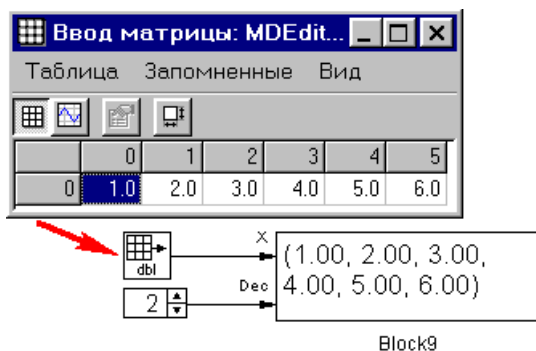


Рис. 377. Тестирование модели преобразования массива в строку

Следует иметь в виду, что строка в РДС – это всегда последовательность символов в кодировке Windows CP1251, завершающаяся нулевым байтом. Многобайтовые символы Unicode в РДС не поддерживаются.

### §3.7.2.6. Использование сигналов

Рассматривается работа с переменными сигнального типа, используемыми для передачи информации о факте наступления какого-либо события. Описываются особенности использования сигнала запуска Start и сигнала готовности Ready, которые есть у каждого простого блока.

Сигнальные переменные используются в РДС для передачи блокам информации о наступлении какого-либо события – нажатия кнопки, срабатывания устройства, обнаружения какого-либо специфического состояния схемы и т.п. Они очень похожи на логические – как и логические, сигнальные переменные могут принимать только значения 0 и 1 – но отличаются способом передачи значений по связям (см. §1.4 части I, а также, более подробно, §2.5.2 руководства программиста [1]). Этим отличий два:

- по связи передается только единичное значение сигнала-выхода, нулевое значение на входы соединенных блоков не передается;
- после передачи единичного значения сигнала-выхода оно автоматически сбрасывается в ноль (на соединенных с этим выходом входах остается единица).

Из-за такого поведения сигналов работа с сигнальными входами блоков отличается от работы с входами других типов: модель блока с сигнальным входом, обнаружив на этом входе единицу, должна, выполнив все необходимые действия, самостоятельно присвоить этому входу нулевое значение, подготовив его тем самым к приему следующей единицы. Действительно, нулевое значение сигнала по связи не передается, поэтому единица, поступившая на сигнальный вход, останется там навсегда, если блок-владелец этого входа принудительно не сбросит ее.

Задачи, решаемые с помощью сигналов, можно решать и при помощи логических переменных, но модели блоков при этом получаются несколько сложнее. В §2.5.2 руководства программиста рассмотрена задача подсчета числа нажатий пользователем кнопки, моделируемой блоком, и сложности, возникающие при создании блока-кнопки и блока-счетчика с использованием только логических переменных. Коротко повторим приведенные там рассуждения.

Если сделать выход блока-кнопки логическим и присваивать ему единицу при нажатии кнопки и ноль при отпускании, в блоке-счетчике необходимо будет увеличивать выход на единицу только при изменении значения логического входа с нуля на единицу. Просто увеличивать выход при единичном значении входа нельзя: если кнопка будет нажата в течение нескольких тактов расчета (а это весьма вероятно – такты следуют друг за другом очень быстро), на протяжении всех этих тактов на входе блока-счетчика будет единица, и в каждом такте его выход будет увеличиваться, хотя пользователь нажал на кнопку всего один раз. Таким образом, в блоке-счетчике необходимо отслеживать не само значение логического входа, а его изменение с нуля на единицу, для чего придется ввести дополнительную переменную для хранения значения входа на прошлом такте расчета: если в прошлом такте вход был нулем, а в текущем стал единицей, значит, кнопка была нажата, и выход нужно увеличивать. Таким образом, при решении задачи с помощью логических переменных модели получаются такими:

- в блоке-кнопке должно быть две реакции на события: при нажатии кнопки мыши логическому выходу нужно присваивать единицу, при отпускании – ноль;
- в блоке-счетчике необходимо ввести вспомогательную логическую переменную, в которую в конце реакции на такт расчета следует копировать текущее значение входа (то есть запоминать прошлое состояние входа), а в начале этой же реакции сравнивать значение входа с этой переменной и увеличивать выход, только если в переменной – ноль, а на входе – единица.

Теперь решим ту же задачу с помощью сигналов. В блоке-кнопке сделаем сигнальный выход, в который будем записывать единицу при нажатии кнопки (реакция на отпускание не понадобится). В блоке-счетчике сделаем сигнальный вход, при обнаружении единицы на

котором будем увеличивать выход и сбрасывать вход снова на ноль. Никакой дополнительной переменной для хранения прошлого значения входа не понадобится. Действительно, когда блок-кнопка установит единичное значение выхода при нажатии, после передачи этого значения на вход счетчика выход автоматически сбросится в ноль, и останется нулем, сколько бы пользователь ни держал кнопку нажатой. Счетчик, отреагировав на единицу на входе, увеличит выход и сбросит свой вход, на котором, таким образом, тоже останется ноль до следующего нажатия кнопки. Модели с использованием сигналов будут такими:

- в блоке-кнопке будет только реакция на нажатие кнопки мыши, в которой сигнальному выходу присваивается единица;
- в блоке-счетчике в реакции на такт расчета при обнаружении единицы на сигнальном входе без каких-либо проверок вход сбрасывается на ноль, а выход увеличивается.

Можно видеть, что во втором случае модели получаются проще, поскольку при использовании сигналов РДС, фактически, берет на себя работу по отслеживанию изменения состояния переменной: можно сказать, что по связи передается только передний фронт сигнала. При присвоении выходу единицы эта единица уходит на соединенные входы, автоматически сбрасывается, и передача на этом заканчивается – выход готов к приему следующей единицы. Платой за упрощение моделей является необходимость принудительно сбрасывать сигнальные входы в ноль после их обработки.

Создадим модель блока-счетчика, похожую на описанную выше (кнопку с сигнальным выходом возьмем из стандартных библиотечных блоков). Наш блок по входному сигналу “Clk” будет изменять свой целый выход “Count” с 0 до 9, а при поступлении следующего сигнала сбросит “Count” в ноль, выдаст на выход “Carry” сигнал переноса и продолжит счет с нуля. Кроме того, у блока будет сигнальный вход “Reset”, сбрасывающий счетчик (устанавливающий “Count” в ноль). Такая же модель, только без использования модуля автокомпиляции, подробно рассматривается в §2.5.2 руководства программиста.

Блок будет иметь следующую структуру переменных:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 1                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| Reset      | Сигнал     | Вход              | ✓           | 0                         |
| Clk        | Сигнал     | Вход              | ✓           | 0                         |
| Count      | int        | Выход             |             | 0                         |
| Carry      | Сигнал     | Выход             |             | 0                         |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```

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
 }
}

```

В этой модели мы сначала проверяем поступление сигнала *Reset*. Если *Reset* – не ноль (то есть сигнал поступил), мы сбрасываем его вручную, чтобы подготовиться к следующему приему сигнала, и вместе с ним обнуляем целый выход *Count* и все остальные сигнальные входы и выходы – сигнал *Reset* возвращает наш блок в исходное состояние. Если же *Reset* равен нулю, то есть сигнала сброса не было, мы проверяем поступление *Clk*. Если *Clk* – не ноль (поступил сигнал увеличения счетчика), мы сбрасываем его в ноль, подготавливаясь к поступлению и обработке следующего сигнала, и увеличиваем *Count* на единицу. Затем, если значение *Count* достигло десяти, мы сбрасываем его в ноль и присваиваем единицу сигналу переноса *Carry*.

Для тестирования этой модели можно собрать схему, изображенную на рис. 378. В ней – два блока с нашей моделью (*Block10* и *Block11*), соединенных последовательно: выход переноса *Carry* блока *Block10* соединен с входом счета *Clk* блока *Block11*. Когда *Block10* досчитает до десяти и выдаст сигнал переноса, *Block11* увеличит свой выход на единицу. При таком соединении выходы первого и второго блоков можно считать двумя разрядами двузначного десятичного числа – таким образом, вместе они могут досчитать до ста. Размножить созданный блок-счетчик можно скопировав его в буфер обмена нажатием *Ctrl+C*, а затем вставив его копию нажатием *Ctrl+V*, при этом на запрос модуля автокомпиляции, нужно ли создавать копию модели для вставляемого блока, следует ответить отрицательно (выбрать вариант “использовать тот же файл модели и для нового блока”, см. рис. 327 на стр. 30).

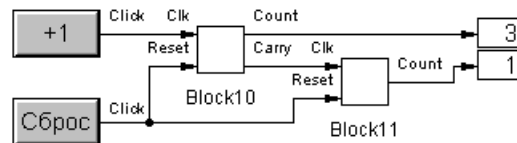


Рис. 378. Тестирование модели счетчика сигналов

Запустив расчет и щелкая по кнопке “+1” в приведенной выше схеме можно будет увидеть, как значение на верхнем числовом индикаторе справа будет увеличиваться. Когда оно дойдет до девяти, следующий щелчок сбросит его в ноль и увеличит значение на нижнем индикаторе. Щелчок по кнопке “сброс” установит оба индикатора в ноль.

Первые две переменных в каждом простом блоке всегда являются сигнальным входом и сигнальным выходом – по умолчанию они называются *Start* и *Ready* соответственно, но их, при желании, можно переименовать. В режиме расчета эти сигналы управляют работой блока и связей, подключенных к его выходам. Единица на входе *Start* запускает модель блока, если для блока не установлен запуск каждый такт (в противном случае модель будет запускаться всегда, какое бы значение вход *Start* ни имел). Единица на выходе *Ready* разрешает передачу данных выходов блока по связям. Модель может работать с этими переменными как с обычными сигналами. Следует помнить, что перед вызовом реакции модели на такт расчета РДС автоматически сбрасывает сигнал *Start* и взводит сигнал *Ready* (то есть считается, что блок уже сработал, и данные его выходов должны быть переданы по связям после такта). Если модели необходимо принудительно перезапустить себя в следующем такте независимо от срабатывания связей, присоединенных к входам блока и его настроек, она может самостоятельно взвести свой собственный сигнал *Start*. Модель может также, при необходимости, запретить передачу данных своих выходов по связям, сбросив сигнал *Ready*. Поскольку *Ready* автоматически взводится только при

реакции на такт расчета, во всех остальных реакциях модель должна самостоятельно взвести этот сигнал, чтобы передать свои изменившиеся выходы по связям. Например, если блок изменяет свой выход по щелчку мыши, в реакции на щелчок необходимо присвоить Ready единицу, иначе новое значение выхода не будет передано (пример такой модели рассматривается в §3.7.11 на стр. 226).

С сигналом Start мы уже сталкивались в предыдущих примерах: чтобы модели блоков принудительно запускались в самом первом такте расчета и обрабатывали исходные значения входов, мы давали этой переменной единичное значение по умолчанию. Однако, мы еще ни разу не подключали связи к этому входу. Такое подключение позволяет запускать реакцию модели на такт расчета по сигналу от другого блока – это позволяет либо организовать достаточно сложную логику работы схемы (примеры приведены в §1.4 части I), либо просто ускоряет работу схемы (модель не запускается, когда это не нужно) и, в некоторых случаях, упрощает программу модели. Рассмотрим сначала использование сигнала запуска непосредственно для запуска модели блока.

В §3.3 (стр. 20) и §3.7.2.1 (стр. 92) была рассмотрена модель простого сумматора, выдающего на вещественный выход  $y$  сумму вещественных входов  $x1$  и  $x2$ . Изменим немного эту модель так, чтобы суммирование осуществлялось только по команде от кнопки – то есть, по сигналу. Установим для сумматора запуск по сигналу на вкладке “общие” окна параметров блока (см. §2.9.1 части I и рис. 363) и изменим структуру переменных модели следующим образом (можно изменить ее в существующей модели, или создать копию модели с новой структурой переменных, скопировав блок через буфер обмена клавишами Ctrl+C и Ctrl+V и ответив “создать копию модели” на запрос модуля автокомпиляции, см. рис. 327 на стр. 30):

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| x1         | double     | Вход              |             | 0                         |
| x2         | double     | Вход              |             | 0                         |
| y          | double     | Выход             |             | 0                         |

По сравнению с моделью из §3.7.2.1, в эту структуру внесено два изменения. Во-первых, значение по умолчанию сигнала запуска Start изменено на ноль – нам не нужно, чтобы модель запускалась без команды в самом первом такте расчета. Во-вторых, флажки “пуск” у входов  $x1$  и  $x2$  выключены – автоматический запуск модели при срабатывании связей, подключенных к этим входам, нам тоже не нужен. Программа модели блока останется прежней – “ $y=x1+x2$ ;”.

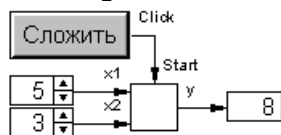


Рис. 379. Сумматор, складывающий по сигналу

Соберем с новым блоком схему, изображенную на рис. 379. Входы сумматора  $x1$  и  $x2$  подключены к полям ввода, выход  $y$  – к индикатору, а сигнальный вход Start – к сигнальному выходу Click стандартной библиотечной кнопки. Запустив расчет, можно будет увидеть, что при изменении значений в полях ввода выход нового блока, в отличие от блока из §3.7.2.1, не изменяется – модель блока не запускается.

Сумма входов появится на выходе только после нажатия кнопки, то есть после поступления единицы на вход Start. Разумеется, сигнал запуска может подаваться не только с кнопки, его источником может быть выход любого другого блока. Так в схемах одни блоки могут управлять запуском других.

Рассмотрим теперь другое использование сигнала запуска, иногда позволяющее несколько упростить программу модели. Выше в этом параграфе была рассмотрена модель



блока-счетчика с переносом, которая запускалась по сигналам счета Clk (увеличить значение) и сброса Reset (обнулить счетчик). Можно переписать эту модель так, чтобы в качестве сигнала счета выступал сигнал запуска Start. Мы переименуем его в Clk, и в программе модели будем проверять значение Reset: если Reset==1, значит, модель запустилась из-за сигнала сброса, а если Reset==0, то она запустилась по какой-то другой причине – а другой причиной запуска может быть только поступление единицы на сигнал запуска блока. Новый блок будет иметь следующую структуру переменных:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Clk        | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| Reset      | Сигнал     | Вход              | ✓           | 0                         |
| Count      | int        | Выход             |             | 0                         |
| Carry      | Сигнал     | Выход             |             | 0                         |

Первый сигнальный вход блока, ранее называвшийся Start, теперь переименован в Clk, а старый сигнал Clk удален из структуры переменных. В новом блоке **обязательно** нужно задать запуск по сигналу, причины этого будут объяснены ниже. Новая модель блока будет выглядеть следующим образом:

```

if (Reset) // Поступил сигнал Reset
{
 Reset=0; // Сброс этого сигнала
 // Сброс сигнала Clk здесь не нужен – сигнал запуска
 // блока сбрасывается автоматически
 Count=0; // Обнуление счетчика
 Carry=0; // Сброс сигнала переноса
 return;
}
// Сигнал Reset не поступал, но модель запустилась –
// значит, поступил Clk. Сбрасывать Clk не нужно –
// сигнал запуска блока сбрасывается автоматически
Count++; // Увеличение счетчика
if (Count>=10) // Досчитали до 10
{
 Count=0; // Обнуление счетчика
 Carry=1; // Выдача сигнала переноса
}

```

Новая модель, за исключением комментариев, получилась короче. Если собрать из новых блоков схему, изображенную на рис. 378 (см. стр. 119), она будет работать точно так же, как и схема со старыми блоками.

В новой модели счетчика следует обратить внимание на три важных момента. Во-первых, начальное значение сигнала запуска, то есть первой сигнальной переменной блока, теперь не единица, как в старой модели, а ноль. Мы теперь используем этот сигнал в качестве сигнала счета, поэтому единицу по умолчанию ему давать нельзя – в этом случае при первом запуске расчета счетчик увеличит свой выход лишней раз. Если бы нам необходимо было при первом запуске расчета обработать начальные значения входов блока, использовать сигнал запуска так, как мы это сделали, было бы нельзя – он должен был бы остаться выделенным сигналом, ответственным только за запуск модели в режиме расчета, чтобы мы могли дать ему единичное начальное значение.

Во-вторых, нигде в новой модели нет ни проверки, ни обнуления сигнала Clk. Проверять его значение бессмысленно, поскольку перед запуском модели сигнал запуска блока автоматически обнуляется. Сам запуск модели свидетельствует о том, что сигнал пришел либо на Clk, либо на Reset (у входа Reset установлен флажок в колонке “пуск”).

Reset автоматически не сбрасывается, поэтому, если модель запустилась из-за него, в нем останется единица (мы проверяем это в самом начале программы). Обнулять Clk не нужно по этой же причине – на момент вызова нашей реакции он уже обнулен.

В-третьих, в отличие от старой модели счетчика, новая будет работать только в том случае, если в параметрах блока установлен флажок “запуск по сигналу”. Если установить флажок “запуск каждый такт”, в каждом такте расчета модель будет запускаться, и, поскольку счетчик теперь увеличивает свой выход при каждом запуске, он будет считать непрерывно, независимо от поступления сигнала на вход Clk. В старой модели Clk был отдельным входом, значение которого проверялось в программе, а сам факт запуска модели ничего не значил, поэтому старая модель работала при любом способе запуска.

Теперь приведем пример использования выхода Ready для запрещения срабатывания связей, подключенных к выходу блока. Создадим модель блока, вставив который в разрыв связи, передающей вещественное значение, можно будет разрешать или запрещать передачу данных по этой связи. Такой блок есть в стандартной библиотеке РДС, но мы, в качестве примера, создадим его вручную.

Блок будет иметь вещественный вход *x*, который будет копироваться в вещественный выход *y*, если логический вход Enable не равен нулю, причем при Enable==0 значение выхода блока, оставшееся там с прошлых срабатываний, не должно передаваться по связи.

Структура переменных блока будет следующей:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 1                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| <i>x</i>   | double     | Вход              | ✓           | 0                         |
| Enable     | Логический | Вход              | ✓           | 0                         |
| <i>y</i>   | double     | Выход             |             | 0                         |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```
if(Enable) // Передача разрешена
 y=x;
else // Передача запрещена
 Ready=0;
```

В этой программе при ненулевом Enable мы копируем *x* в *y*, а при нулевом запрещаем работу всех выходных связей, записав 0 в Ready. При запуске модели блока для реакции на такт расчета второму сигнальному выходу блока, то есть Ready, автоматически присваивается единица, поэтому передача данных выходов блока по связям по умолчанию разрешена, и при Enable!=0 нам не нужно предпринимать никаких действий для ее разрешения, как и во всех рассмотренных ранее моделях.

Для проверки созданной модели можно собрать схему, изображенную на рис. 380. Здесь “Block12” и “Block13” – два одинаковых блока управления связью с нашей моделью, их входы *x* подключены к полям ввода, а выходы *y* – к одному и тому же числовому индикатору (обе связи соединены с его входом *v*). Входы Enable наших блоков соединены с выходами Down двух стандартных библиотечных кнопок, в настройках которых установлены флажки “переключающаяся кнопка”. В этом режиме по первому щелчку мыши кнопка нажимается и остается нажатой, и ее выход Down получает значение 1, а по второму – снова переходит в отпущенное состояние, и выход Down обнуляется. Если запустить расчет, на индикаторе будет отображаться значение того поля ввода, для которого нажата

соответствующая ему кнопка. Если изменять значение в поле ввода, подключенном к блоку, кнопка для которого не нажата (на рисунке – “Block13”), модель этого блока будет запускаться, поскольку у переменной *x* установлен флажок “пуск”, но, из-за того, что внутри модели в этом случае Ready обнуляется, выход *y* не передается по связи, и индикатор получает значение с другого соединенного с ним блока, выход Ready которого остался равным единице и, поэтому, связь сработала. Если нажать обе кнопки, будут срабатывать обе выходных связи, и какое именно значение будет отображаться на индикаторе, предсказать сложно.

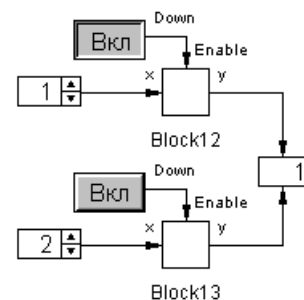


Рис. 380. Проверка блока управления связью

В приведенных примерах мы никогда ни к чему не присоединяли выход Ready. Его можно использовать не только для разрешения работы выходных связей, но и как признак того, что блок сработал – например, соединить его со входом Start следующего блока и, таким образом, задать последовательность работы блоков, если это необходимо. Пример подобной схемы приведен в §1.4 части I, здесь мы его рассматривать не будем, поскольку он не требует каких-либо специальных действий в программе модели – это стандартное для РДС поведение блоков.

### §3.7.2.7. 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.

В рассмотренных выше примерах моделей уже неоднократно использовался флажок “пуск”, установка которого для входа блока позволяла автоматически запустить модель этого блока в следующем такте расчета, если сработает связь, подключенная к этому входу. Отключив для модели блока запуск каждый такт и установив такие флажки для всех его входов, можно существенно снизить нагрузку на систему: модели будут запускаться только при срабатывании входных связей, то есть тогда, когда значения входов блоков изменились.

Однако, включение флажка “пуск” не позволяет модели узнать, какие именно из входных связей сработали. Если блок выполняет сложные и длительные вычисления с несколькими входами, было бы логично выполнять эти вычисления только для изменившихся значений. Можно, конечно, запоминать прежнее значение каждого входа и сравнивать его с текущим, выполняя вычисления только для тех входов, для которых эти значения не совпадают. Однако, этот способ годится только для входов простых типов. Если, например, входами блока являются несколько матриц, то для хранения их предыдущих значений может потребоваться много памяти (размер матрицы в РДС ограничен только свободной памятью), да и поэлементное сравнение сохраненной старой матрицы с новой может потребовать не меньше времени, чем сами вычисления с этой матрицей.

Для решения этой задачи в РДС встроен механизм, позволяющий модели блока узнать, какие именно входы получили по связям новые значения в конце предыдущего такта расчета. Для этого в редакторе переменных необходимо связать вход блока с переменной сигнального типа (она может быть входом, выходом или внутренней), задав для этого входа вместо роли “вход” роль “вход/сигнал” и указав имя связанной сигнальной переменной. При срабатывании связи, подключенной к этому входу, в связанную с ним сигнальную переменную автоматически запишется единица. Разумеется, если необходимо запустить модель при срабатывании связи, флажок “пуск” у этого входа тоже должен быть установлен. Таким образом, модель блока может проверить значения связанных с входами сигналов и, обнаружив в некоторых из них единицы, понять, какие именно входы получили значения по связям в прошлом такте расчета. Выполнив действия, связанные с изменившимися входами, модель должна обнулить все связанные сигналы, подготовив их к следующему

срабатыванию связей. Как и любые сигнальные переменные, связанные сигналы не могут получить нулевое значение при срабатывании связей – получив значение 1, сигнал сохраняет его до тех пор, пока модель блока самостоятельно не обнулит переменную (см. §3.7.2.6 на стр. 117). На самом деле, единица в связанной сигнальной переменной указывает не на изменение значения входа, а на то, что сработала связь, подключенная к этому входу: если по этой связи придет то же самое значение, связанный сигнал все равно взведется. Однако, поскольку модели блоков чаще всего пишут так, чтобы они срабатывали и активировали свои выходные связи только при необходимости передать новые значения, срабатывание присоединенной ко входу связи с большой вероятностью говорит о том, что что-то изменилось. В любом случае, обработка значения входа при срабатывании именно присоединенной к нему связи меньше нагружает систему, чем та же самая обработка при срабатывании любой связи, присоединенной к блоку. Несколько входов можно, при желании, связать с одним сигналом, если модель блока должна выполнять одинаковые действия при изменении любого из этих входов.

В качестве примера использования связанных сигналов создадим модель блока, входами которого будут вещественные матрицы M1 и M2, а выходом y – произведение минимального элемента матрицы M1 и максимального элемента матрицы M2 (именно эта модель приводится в примере в §2.5.7 руководства программиста [1]). Определение максимального и минимального элемента матриц требует перебора всех их значений, поэтому мы будем запоминать эти значения во вспомогательных переменных M1min и M2max, вычисляя их только при срабатывании связей, соединенных с входами M1 и M2 соответственно. Вход M1 мы свяжем с сигналом s1, вход M2 – с сигналом s2, по появлению единиц в этих сигналах мы будем узнавать о срабатывании соответствующих входных связей.

Таким образом, наш блок будет иметь следующую структуру переменных:

| <i>Имя</i> | <i>Тип</i>        | <i>Вход/выход</i>   | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|-------------------|---------------------|-------------|---------------------------|
| Start      | Сигнал            | Вход                | ✓           | 1                         |
| Ready      | Сигнал            | Выход               |             | 0                         |
| s1         | Сигнал            | Внутренняя          |             | 1                         |
| s2         | Сигнал            | Внутренняя          |             | 1                         |
| M1         | Матрица<br>double | Вход/сигнал<br>“s1” | ✓           | [ ] 0                     |
| M2         | Матрица<br>double | Вход/сигнал<br>“s2” | ✓           | [ ] 0                     |
| M1min      | double            | Внутренняя          |             | ?                         |
| M2max      | double            | Внутренняя          |             | ?                         |
| y          | double            | Выход               |             | ?                         |

Следует обратить внимание на то, что в качестве начальных значений переменных M1min, M2max и y указан вопросительный знак, то есть специальное значение, используемое в качестве индикатора ошибки вычисления (в модели его можно получить из глобальной переменной rdsbcppHugeDouble, см. стр. 84). Действительно, пока максимальный элемент M2 и минимальный элемент M1 не вычислены, мы ничего не можем присвоить ни внутренним переменным M1min и M2max, ни их произведению y.

В этой структуре переменных сигналы s1 и s2 расположены до входов M1 и M2, с которыми они связаны. Это не обязательно, но так удобнее редактировать переменные: при указании в редакторе для входа роли “вход/сигнал”, имя связанного сигнала выбирается из

выпадающего списка, в который включаются уже введенные на данный момент переменные сигнального типа, поэтому лучше, чтобы на момент ввода входа сигнал, который с ним будет связан, уже был введен в редактор переменных. Если, по каким-либо причинам, необходимо поместить связанные сигналы после входов, с которыми они связываются (например, если эти связанные сигналы – тоже входы блока, и разработчик хочет, чтобы в меню присоединения связей они были в конце), можно сначала ввести в редакторе переменных сигналы, а потом вставить входы в нужное место списка (см. §2.9.2 части I).

Сигналам *s1* и *s2* даны единичные начальные значения, чтобы при самом первом запуске расчета модель блока обработала исходные значения матриц-входов *M1* и *M2*, как будто связи, подключенные к ним, сработали перед запуском.

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```

if(s1) // Сработала связь к M1
{ s1=0; // Обнуление сигнала
 // Ищем минимальный элемент в M1 и записываем в M1min
 M1min=rdsbcppHugeDouble;
 for(int r=0;r<M1.Rows();r++)
 for(int c=0;c<M1.Cols();c++)
 if(M1min==rdsbcppHugeDouble || M1min>M1[r][c])
 M1min=M1[r][c];
}
if(s2) // Сработала связь к M2
{ s2=0; // Обнуление сигнала
 // Ищем максимальный элемент в M2 и записываем в M2max
 M2max=rdsbcppHugeDouble;
 for(int r=0;r<M2.Rows();r++)
 for(int c=0;c<M2.Cols();c++)
 if(M2max==rdsbcppHugeDouble || M2max<M2[r][c])
 M2max=M2[r][c];
}
// Перемножаем максимальный и минимальный элементы матриц
if(M1min!=rdsbcppHugeDouble && M2max!=rdsbcppHugeDouble)
 y=M1min*M2max;
else
 y=rdsbcppHugeDouble;

```

Эта программа состоит из трех частей. Сначала мы проверяем переменную *s1* – если ее значение не нулевое, значит, сработала связь, подключенная ко входу *M1*, и мы перебираем все элементы матрицы *M1* (работа с матрицами описана в §3.7.2.2 на стр. 99), ищем среди них минимальный и записываем во внутреннюю переменную *M1min*. Значение *s1* мы при этом обнуляем – кроме модели блока, это сделать некому. Затем, точно так же, при ненулевом значении *s2* мы находим максимальное значение *M2* и записываем его в *M2max*. Если какая-либо из входных матриц останется пустой, в соответствующей ей внутренней переменной останется значение *rdsbcppHugeDouble*, присвоенное ей перед циклом: для пустой матрицы значение максимального или минимального элемента не может быть определено, поэтому мы пишем в переменную значение-индикатор ошибки. В конце программы мы перемножаем *M1min* и *M2max*, если обе они не равны *rdsbcppHugeDouble*, то есть если определен и минимальный элемент первой матрицы, и максимальный элемент второй. В противном случае мы выдаем на выход значение-индикатор ошибки *rdsbcppHugeDouble*.

Таким образом, единственная операция, всегда выполняющаяся в такте расчета – это перемножение *M1min* и *M2max*. Поиск минимального элемента в *M1* выполняется только

при срабатывании подключенной к ней связи, так же как и поиск максимального элемента M2. Если значение на одном из входов меняется редко, перебор значений этой матрицы тоже будет выполняться редко, что увеличит производительность системы. В данном случае выигрыш будет незначительным, однако, если бы вычисления с матрицей были сложнее (например, необходимо было бы вычислять ее определитель или обратную ей матрицу), прирост быстродействия был бы ощутимым.

Для тестирования созданной модели можно собрать схему, изображенную на рис. 381. В ней к входам нашего блока присоединены стандартные блоки ввода матриц, а к выходу – числовой индикатор. Запустив расчет и изменяя значения в матрицах, можно наблюдать изменения на индикаторе.

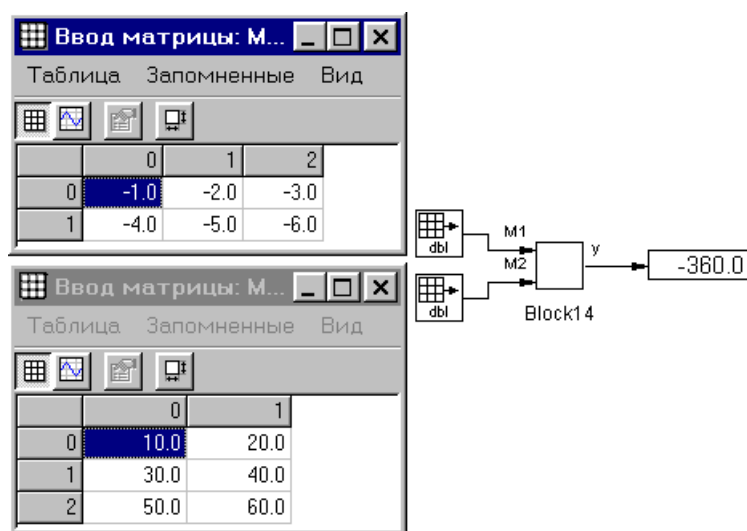


Рис. 381. Тестирование модели, вычисляющей произведение минимального элемента одной матрицы и максимального элемента другой

### §3.7.2.8. Использование выходов с управляющими переменными

Описывается использование управляющих переменных, которые позволяют либо запретить передачу по связям значения конкретного выхода блока, либо активировать только связь, подключенную к конкретному элементу выхода-массива.

В §3.7.2.6 (стр. 117) описывалось использование второй сигнальной переменной блока (выхода Ready) для запрещения срабатывания всех связей, подключенных к выходам блока в конце такта расчета. Перед выполнением реакции модели на такт расчета РДС присваивает этой переменной единицу, поэтому, если в модели не предпринять никаких действий, значения всех выходов блока будут переданы по связям на входы блоков, соединенных с ним. Если модель запишет в эту переменную ноль, данные передаваться не будут. Обычно модель блока запрещает работу выходных связей в тех случаях, когда значения на выходах блока не изменились, и нет смысла передавать их дальше, запуская тем самым модели соединенных блоков и зря тратя процессорное время.

Управление сигналом Ready позволяет запретить работу только всем выходным связям одновременно – это не всегда удобно. РДС позволяет разрешать и запрещать выходные связи индивидуально. Это может оказаться полезным при создании различных блоков-выключателей и демультиплексоров, управляющих передачей данных со входа на один или несколько выходов. Мы уже рассматривали простой демультиплексор в §3.7.2.3 (см. стр. 106) – в нем значение, поступавшее на вход, передавалось на выход с номером, определяемым дополнительным целым входом. Однако, в том блоке все равно срабатывали все выходные связи – это не мешало его работе, потому что “неактивные”

выходы просто еще раз передавали по связям свои старые значения. Тем не менее, это приводило к напрасному срабатыванию моделей во всех цепочках подключенных к этим выходам блоков и лишней трате процессорного времени. Далее в этом параграфе мы исправим модель так, чтобы в конце такта расчета срабатывала только связь, подключенная к выходу с нужным номером.

Чтобы разрешать и запрещать передачу данных отдельного выхода блока, необходимо ввести в его структуру переменных дополнительную логическую или целую переменную и связать с ней выход, задав для него тип “выход / логическая” вместо “выход” и указав имя этой дополнительной переменной. Если выход связан с логической переменной, то связи, подключенные к нему, 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 независимо от значений управляющих логических переменных отдельных его выходов. Если выход блока является массивом, можно связать его с целой дополнительной переменной – в этом случае будет разрешена работа связей, подключенных ко всему массиву как к одной сложной переменной и к элементу массива с номером, определяемым этой целой переменной. Связи, подсоединенные к элементам массива с номерами, отличными от значения целой связанной переменной, будут отключены.

Для иллюстрации управления отдельными связями создадим блок-переключатель, который будет выдавать свой вещественных вход  $x$  на выход  $y_0$ , если целый вход  $N$  будет равен 0, на выход  $y_1$ , если  $N$  будет равен 1, и на оба выхода одновременно для любого другого значения  $N$  (подобная модель рассматривается в §2.5.8 руководства программиста [1]). Для управления выходом  $y_0$  введем внутреннюю логическую переменную  $L_0$ , для управления  $y_1$  –  $L_1$ . Блок будет иметь следующую структуру переменных:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i>             | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход                          | ✓           | 1                         |
| Ready      | Сигнал     | Выход                         |             | 0                         |
| $x$        | double     | Вход                          | ✓           | 0                         |
| $N$        | int        | Вход                          | ✓           | 0                         |
| $L_0$      | Логический | Внутренняя                    |             | 0                         |
| $L_1$      | Логический | Внутренняя                    |             | 0                         |
| $y_0$      | double     | Выход/логический<br>“ $L_0$ ” |             | 0                         |
| $y_1$      | double     | Выход/логический<br>“ $L_1$ ” |             | 0                         |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```

y0=y1=x; // Копирование входа в оба выхода
switch (N)
{
 case 0: // N==0 - разрешить y0 через L0
 L0=1; L1=0; break;
 case 1: // N==1 - разрешить y1 через L1
 L0=0; L1=1; break;
}
```

```

 default: // Разрешить обе связи
 L0=L1=1;
}

```

В этой модели мы сразу копируем вход  $x$  и в  $y_0$ , и в  $y_1$ . Если бы мы не управляли выходными связями индивидуально, это привело бы к тому, что, независимо от  $N$ , блок передавал бы значение входа на оба выхода. Однако, далее, в операторе `switch`, в зависимости от значения  $N$ , мы присваиваем единицу или ноль управляющим логическим переменным  $L_0$  и  $L_1$ . При нулевом  $N$  единица будет только в  $L_0$  – работа связи, отходящей от  $y_0$ , будет разрешена, а от  $y_1$  – запрещена. При  $N=1$  единица будет только в  $L_1$  – будет разрешена только связь от  $y_1$ . При любом другом значении  $N$  (метка `default` в операторе `switch`) и  $L_0$ , и  $L_1$  получают значение 1, что разрешит обе выходных связи.

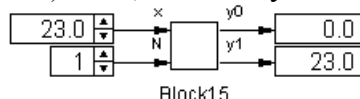


Рис. 382. Проверка переключателя

Для тестирования модели соберем схему, изображенную на рис. 382. Запустив расчет и изменяя значение в поле ввода, подключенном к  $x$ , можно наблюдать появление этого же значения на индикаторе, подключенном к выходу, соответствующему значению  $N$ . При этом, если остановить расчет, открыть окно параметров блока и просмотреть текущие значения его переменных на вкладке “переменные” (рис. 383), можно заметить, что, несмотря на то, что значения на индикаторах разные, значения обоих выходов блока одинаковые. Значение одного из выходов не передается в индикатор по связи из-за нулевого значения связанной с этим выходом управляющей переменной.

| Параметры блока: Block15                               |       |            |      |        |      |          |
|--------------------------------------------------------|-------|------------|------|--------|------|----------|
| Общие Внешний вид Переменные Соединения DLL Компиляция |       |            |      |        |      |          |
|                                                        | Имя   | Тип        | Пуск | Логика | Меню | Значение |
| →                                                      | Start | Сигнал     | ✓    |        | ✓    | 0        |
| →                                                      | Ready | Сигнал     |      |        | ✓    | 0        |
| →                                                      | x     | double     | ✓    |        | ✓    | 23       |
| →                                                      | N     | int        | ✓    |        | ✓    | 1        |
|                                                        | L0    | Логический |      |        |      | 0        |
|                                                        | L1    | Логический |      |        |      | 1        |
| →                                                      | y0    | double     |      | L0     | ✓    | 23       |
| →                                                      | y1    | double     |      | L1     | ✓    | 23       |

Входы Выходы Внутренние Изменить

OK Отмена

Рис. 383. Текущие значения переменных блока-переключателя

Вернемся теперь к описанной в §3.7.2.3 (см. стр. 106) модели демультиплексора и соберем с его использованием новую схему, изображенную на рис. 384.

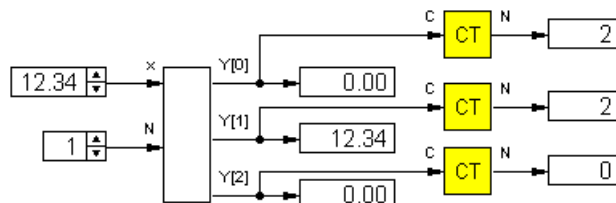


Рис. 384. Новая схема тестирования демультиплексора

Новая схема отличается от старой, изображенной на рис. 370, (стр. 107) тем, что каждая связь, идущая от выхода блока к индикатору, теперь разветвляется, и вторая ее ветвь



подключается к сигнальному входу счета С стандартного счетчика сигналов. В РДС к сигнальным входам можно подключать связи не только сигнального типа: в этом случае входу будет присваиваться единица при срабатывании связи, какое бы значение при этом по ней ни передавалось.

Запустив расчет и изменяя значение в поле ввода, подключенном к входу  $x$  демultipлексора, можно будет видеть, что срабатывает не только счетчик, подключенный к связи, отходящей от выхода блока с нужным номером, но и некоторые другие. Например, на рис. 384, несмотря на то, что  $N=1$ , при изменении значения в поле ввода  $x$  будет увеличиваться значение не только на счетчике, подключенном к выходу  $Y[1]$ , но и на счетчике, подключенном к  $Y[0]$ . Значение счетчика, подключенного к  $Y[2]$  изменяться не будет только по той причине, что в нашей модели мы изменяем размер выходного массива  $Y$  только при необходимости, и пока  $N$  не окажется большим или равным двум, элемент массива  $Y[2]$  просто не будет существовать, а, значит, и связь от этого элемента сработать не сможет. Мы наблюдаем вполне ожидаемую картину: независимо от того, что происходит внутри блока и в какой именно элемент массива он записывает значение своего входа, срабатывают все выходные связи этого блока.

Теперь изменим структуру переменных блока, связав его выход  $Y$  с целой управляющей переменной  $N$  (несмотря на то, что переменная целая, роль переменной  $Y$  в выпадающем списке все равно будет называться “выход/логический”):

| <i>Имя</i> | <i>Тип</i>    | <i>Вход/выход</i>           | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|---------------|-----------------------------|-------------|---------------------------|
| Start      | Сигнал        | Вход                        | ✓           | 1                         |
| Ready      | Сигнал        | Выход                       |             | 0                         |
| $x$        | double        | Вход                        | ✓           | 0                         |
| $N$        | int           | Вход                        | ✓           | 0                         |
| $Y$        | Массив double | Выход/логический<br>“ $N$ ” |             | [ ] 0                     |

Других изменений в модель блока вносить не нужно, и программа на вкладке “модель” редактора останется той же самой. Мы просто сделали вход  $N$  управляющей целой переменной для выходного массива  $Y$ , и теперь из всех связей, подключенных к элементам  $Y$ , будет срабатывать только та, которая отходит от элемента с номером, равным текущему значению  $N$ . Теперь при запуске расчета и изменении значения в поле ввода  $x$  считать будет только один единственный счетчик – связи, подключенные к остальным, срабатывать не будут.

### §3.7.3. Работа с динамическими переменными

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 получить доступ к одной и той же динамической переменной и использовать ее для связи. Имена динамических переменных обычно жестко задаются в модели блока – предоставление пользователю возможности указывать их имена описывается в §3.7.7 (стр. 203).

#### §3.7.3.1. Подключение к динамической переменной

Описывается получение данных из динамической переменной, созданной каким-либо другим блоком.

Принцип действия динамических переменных подробно рассматривается в §1.5 части I и еще подробнее – в §2.6 руководства программиста [1]. Коротко напомним: динамическая переменная – это скрытая от пользователя переменная, общая для нескольких блоков, через

которую они могут обмениваться информацией без использования связей. Какой-то один блок создает такую переменную, после чего он сам и другие блоки, зная имя этой переменной и ее тип, могут записывать и считывать ее значения. Блок, записавший в динамическую переменную новое значение, обычно информирует об этом остальные блоки, использующие эту переменную. При этом РДС вызывает у моделей этих блоков реакцию на специальное событие “изменение динамической переменной” (RDS\_BFM\_DYNVARCHANGE, см. §3.6.4 на стр. 46 и §3.8.3.7 на стр. 293), в которой они считывают новое значение и используют его в расчете. Динамическая переменная обычно создается внутри корневой подсистемы схемы или родительской подсистемы создавшего ее блока (блок может создать динамическую переменную и внутри самого себя, но этот вариант используется крайне редко), при этом она доступна всем блокам, находящимся в одной с ней подсистеме или в подсистемах, вложенных в эту подсистему на любом уровне. Получение блоком доступа к динамической переменной называется в РДС *подпиской* на переменную: модель блока сообщает имя переменной, ее тип и подсистему, в которой ее нужно искать, а РДС связывает блок с этой переменной, если она существует. Если переменная не найдена, РДС запомнит факт подписки и предоставит блоку доступ к переменной, как только она появится. Таким образом, не важно, какое событие произойдет в схеме первым: создание динамической переменной одним блоком или подписка на нее другим: в конце концов связь блока с переменной будет установлена. Чаще всего блоки используют подписку с поиском по иерархии: при этом переменная с заданными именем и типом сначала ищется в родительской подсистеме подписывающегося блока, затем – в родительской подсистеме этой подсистемы и т.д. вплоть до корневой подсистемы. Этот механизм позволяет разместить блок, использующий переменную, в любой подсистеме, вложенной в ту, в которой находится блок-создатель этой переменной. При этом в разных подсистемах схемы могут находиться динамические переменные с одинаковыми именами, и блоки в этих подсистемах и подсистемах, вложенных в них, будут видеть ближайшую к ним переменную.

Среди стандартных блоков, работающих с динамическими переменными, самую важную роль играет блок-планировщик динамического расчета. Он создает в своей подсистеме вещественную (double) переменную “DynTime”, в которую постоянно записывает значение условного системного времени (подробнее о моделировании длящихся во времени процессов с использованием этой переменной рассказывается в §3.7.4 на стр. 145). Планировщик изменяет это значение с шагом и скоростью, заданными в его настройках (рис. 385) – таким образом, можно настраивать дискретность изменения времени в системе и соотношение этого времени с реальным.

Параметр “значение шага расчета” задает дискретность изменения системного времени, то есть размер интервала между соседними его отсчетами. Если, например, задать шаг расчета равным 0.1, динамическая переменная “DynTime” будет принимать значения 0, 0.1, 0.2, 0.3 и т.д. Увеличение значения времени на шаг расчета будет происходить не в каждом такте расчета, даже если включен флажок “новый шаг – каждый такт расчета” (использование флажка “по сигналу готовности” требует дополнительных соединений и здесь не рассматривается). Во-первых, если включен флажок “дополнительные такты”, между изменениями “DynTime” будет выполняться по крайней мере указанное количество тактов расчета – это нужно для того, чтобы вычисленные после очередного изменения времени значения распространились по длинным цепочкам соединенных блоков. Во-вторых, если включен флажок “синхронизация с реальным временем”, планировщик будет согласовывать изменение “DynTime” с системными часами и введенным в настройках множителем задержки. При множителе задержки, равном единице, системное время будет примерно совпадать с реальным: увеличив “DynTime” на заданное значение шага, планировщик будет ждать, когда по системным часам пройдет это же самое время. При множителе задержки, меньшем единицы, время в системе будет идти быстрее реального (при множителе 0.1 системное время идет в 10 раз быстрее), при множителе, большем единицы –

медленнее (множитель 10 заставляет схему считать в 10 раз медленнее реального времени). Если флажок “синхронизация с реальным временем” выключен, схема будет считать с максимально возможной скоростью – это удобно, если нужно не наблюдать за протекающими в схеме процессами, а получить их графики или конечные значения как можно быстрее.

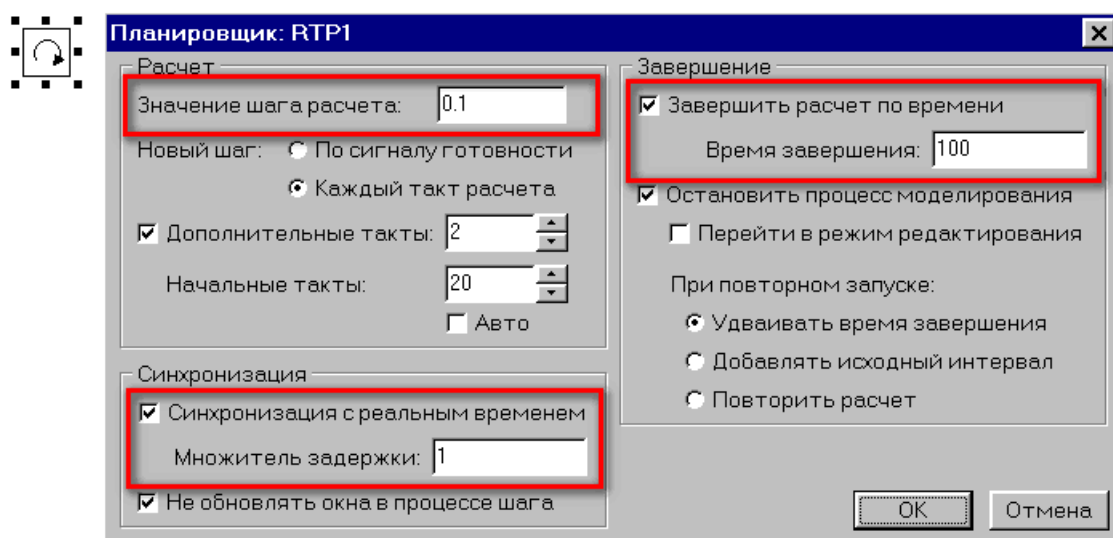


Рис. 385. Блок-планировщик и его самые важные параметры

Флажок “завершить расчет по времени” позволяет автоматически остановить расчет при достижении переменной “DynTime”, то есть системным временем, заданного значения. При этом, если включен дополнительный флажок “остановить процесс моделирования”, РДС выйдет из режима расчета и вернется в режим моделирования. За редкими исключениями, связанными с проведением итеративных расчетов (например, с использованием стандартных блоков оптимизации), этот дополнительный флажок следует оставлять включенным.

Многие библиотечные блоки в РДС (графики, генераторы, динамические блоки) полагаются на наличие в схеме блока-планировщика. Точнее, им необходимо наличие динамической переменной “DynTime”, из которой они берут текущее значение системного времени. Какой именно блок будет обеспечивать создание этой переменной и увеличение ее с некоторым шагом, этим блокам не важно. При желании, разработчик может написать собственную модель планировщика и использовать ее в схемах со стандартными блоками. При создании собственных моделей блоков, которым требуется значение системного времени, следует брать значение времени именно из переменной “DynTime” – так эти блоки смогут работать в одной схеме со стандартными динамическими и с блоком-планировщиком. В интерфейсе редактора модели можно достаточно просто добавлять связи блока с динамическими переменными (см. §3.6.3 на стр. 42). После создания такой связи в программе модели для этой переменной будет создаваться объект специального класса с переопределенными операторами присваивания и приведения типов. Таким образом, во вводимых в редакторе модели реакциях блока на события можно обращаться к динамической переменной по имени, получая ее значение и присваивая ей новое, а также вызывая функцию-член ее класса `NotifySubscribers`, уведомляющий все использующие ее блоки об изменении значения этой переменной (см. пример в §3.7.3.2 на стр. 136).

В качестве простейшего примера блока, использующего значение системного времени, создадим модель генератора, выдающую на выход синусоиду по формуле  $A \sin(\omega t)$ , где  $t$  – системное время. У нашего блока будет два входа  $A$  и  $\omega$  (амплитуда и частота сигнала соответственно) и выход  $y$ . Время мы будем брать из динамической переменной “DynTime”,

поэтому в структуре статических переменных блока переменная для времени не нужна. Структура переменных нашего блока будет такой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| A          | double     | Вход              |             | 0                         |
| w          | double     | Вход              |             | 0                         |
| y          | double     | Выход             |             | 0                         |

Здесь мы не устанавливаем флажки “пуск” у добавленных входов блока, поскольку он должен запускаться при изменении времени, а не при изменении входов (флажок у входа запуска блока Start всегда включен, убрать его нельзя).

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. Для того, чтобы связать блок с переменной времени “DynTime”, следует выполнить следующие шаги:

- на вкладке “переменные” левой панели редактора модели в нижней ее части (см. рис. 334 на стр. 42) нажать кнопку со знаком “+”;
- в открывшемся окне добавления динамической переменной (рис. 386) установить флажок “стандартная переменная” и выбрать в выпадающем списке справа от него пункт “динамический расчет – время”;
- закрыть окно добавления переменной кнопкой “ОК”.

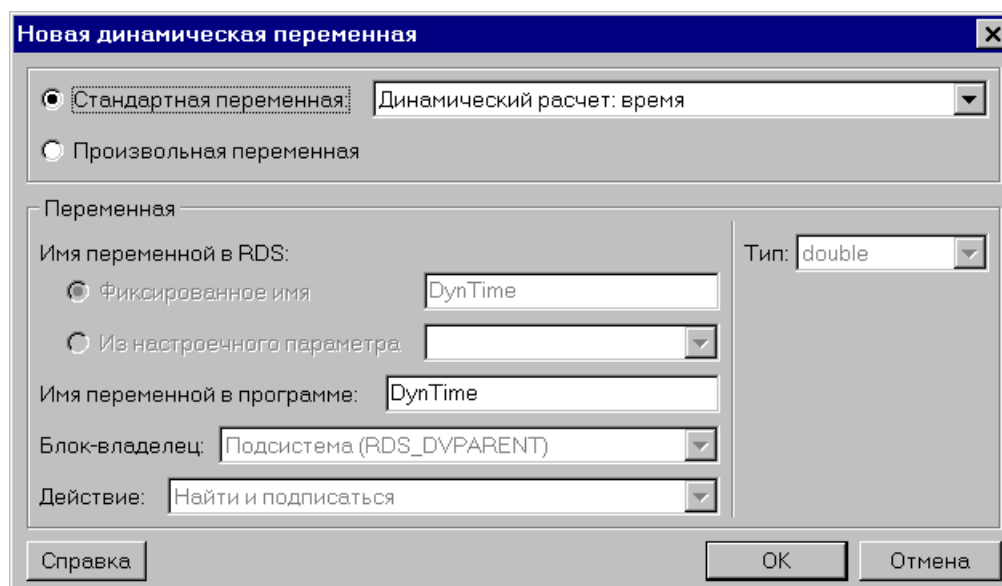


Рис. 386. Добавление переменной “DynTime” в блок

После этих действий в списке на нижней части вкладки “переменные” должна появиться переменная “DynTime” со значком лупы в левой колонке (этот значок указывает на то, что мы подписываемся на переменную, а не создаем ее, см. §3.6.3 на стр. 42). Поскольку мы добавили переменную как стандартную, все параметры связи ее с блоком уже настроены правильно, и в программе мы можем просто использовать ее имя DynTime.

На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```
y=A*sin(w*DynTime);
```

Здесь мы в качестве значения времени подставили имя переменной DynTime. После всех этих действий, если все выполнено правильно, окно редактора модели должно иметь вид, изображенный на рис. 387.

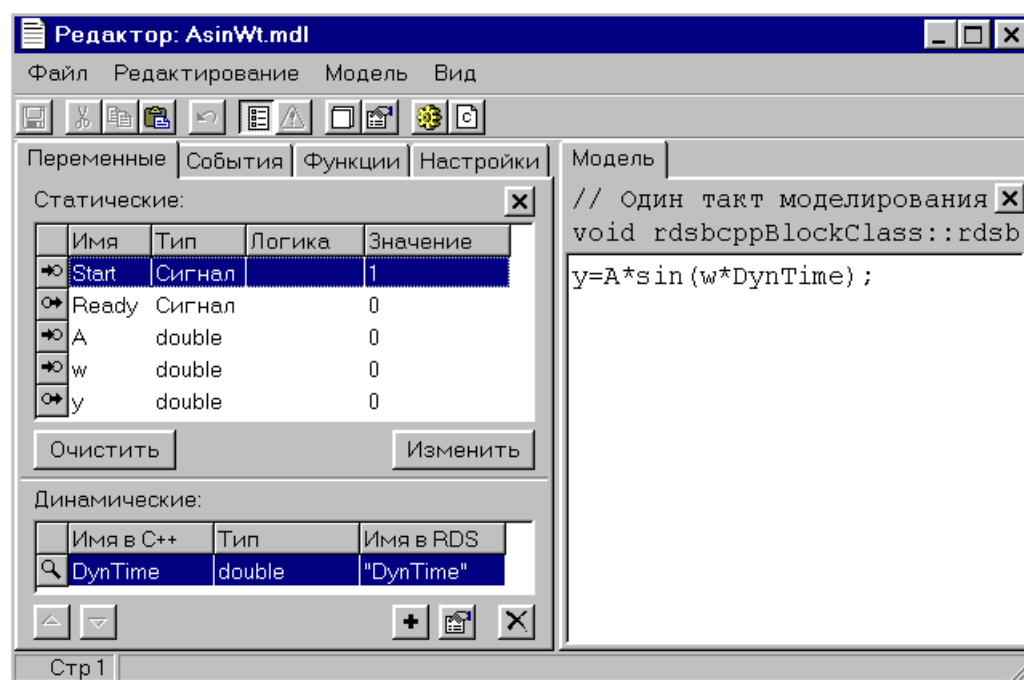


Рис. 387. Окно редактора модели генератора синусоиды

Прежде чем проверять работу созданной модели, следует обратить внимание на два важных момента. Во-первых, что будет, если в схеме, в которой находится наш блок, будет отсутствовать блок-планировщик, а, следовательно, и переменная “DynTime”? Во-вторых, для нашего блока задан запуск по сигналу, а не каждый такт, значит, реакция на такт расчета будет вызываться только при поступлении сигнала на вход Start – как же модель запустится при изменении DynTime?

Любая модель блока, работающая с динамическими переменными, должна в обязательном порядке проверять существование этих переменных перед обращением к ним – таковы правила создания моделей в РДС. Кроме того, крайне желательно, чтобы модель реагировала на событие RDS\_BFM\_DYNVARCHANGE, наступающее при изменении любой динамической переменной, на которую подписан блок. Мы не включили в модель ни проверку существования DynTime, ни реакцию на указанное событие, поскольку при параметрах модели по умолчанию (а мы их не меняли) блок автокомпиляции сделает это за нас. Вызовем из редактора окно параметров модели пунктом меню “модель | параметры модели” и рассмотрим те из них, которые связаны с динамическими переменными (рис. 388).

По умолчанию в модели установлен флажок “не реагировать на события без динамических переменных”, поэтому введенная нами реакция на такт расчета выполнится только в том случае, если переменная DynTime будет существовать. Именно этот флажок позволяет нам не вставлять в программу проверку существования DynTime, которая выглядела бы так:

```
if (DynTime.Exists())
 y=A*sin(w*DynTime);
```

Включение этого флажка позволяет не заботиться о таких проверках, и это сильно облегчает написание моделей. Тем не менее, это не всегда хорошо: в некоторых случаях при отсутствии переменной имеет смысл привлечь к этому факту внимание пользователя. Например, стандартный график, отображающий зависимость какого-либо значения от

времени, при отсутствии в схеме переменной “DynTime” рисует поверх себя иконку с восклицательным знаком и выводит сообщение об ошибке во всплывающей подсказке (пометки на блоках и всплывающие подсказки рассматриваются в §3.7.10 на стр. 223 и §3.7.9 на стр. 211 соответственно). Без этого пользователь не сразу смог бы догадаться, почему после запуска расчета график не строится. Чтобы нарисовать что-нибудь поверх изображения блока, модели необходимо реагировать на событие дополнительного рисования, а для этого нужно отключить указанный флажок, поскольку он блокирует реакции на все события, кроме инициализации и очистки. Разумеется, при этом все проверки в программах реакций придется выполнять вручную.

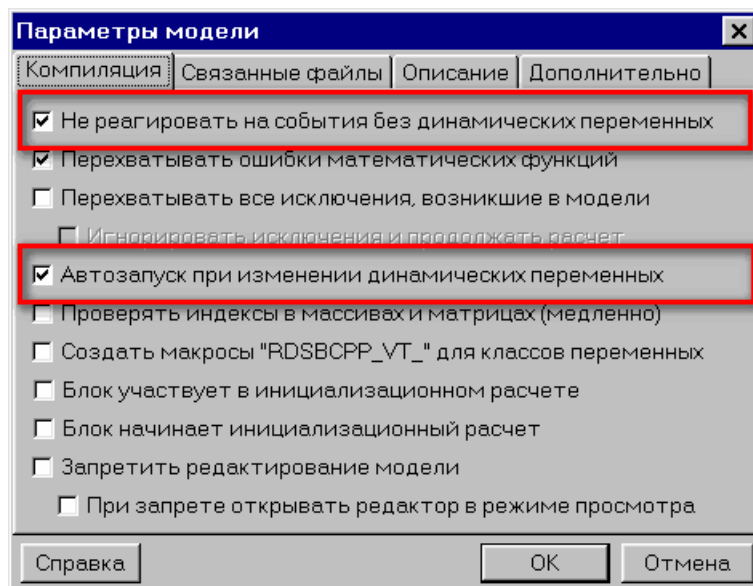


Рис. 388. Параметры, обеспечивающие работу модели с динамическими переменными

Флажок “автозапуск при изменении динамических переменных”, тоже включенный по умолчанию, автоматически добавляет в модель реакцию на событие изменения динамической переменной, в которой сигналу запуска блока Start присваивается единица. Эта реакция не отображается в редакторе модели, но, при желании, ее можно увидеть, просмотрев полный текст формируемой программы пунктом меню “модель | показать текст C++” и найдя в нем метку “case RDS\_BFM\_DYNVARCHANGE”. Таким образом, при изменении любой динамической переменной, на которую подписан блок, взведется его сигнал запуска, и в ближайшем такте расчета запустится его модель (в нашем случае выполнится оператор вычисления значения выхода блока  $y$ ). Отключать этот флажок и вручную вводить в модель реакцию на событие RDS\_BFM\_DYNVARCHANGE имеет смысл либо если блок подписан на несколько переменных и необходимо знать, какая из них изменилась, либо при отсутствии у блока реакции на такт расчета, когда все действия выполняются в реакции на изменение динамической переменной. Пример использования этой реакции будет приведен в конце этого параграфа.

Вернемся к нашему блоку, в параметрах модели которого оба описанных выше флажка остались установленными по умолчанию. Для проверки его работы соберем схему, изображенную на рис. 389.

В этой схеме к входам  $A$  и  $w$  нашего блока подключены поля ввода, а к выходу  $y$  – стандартный график зависимости значения от времени. Рядом с блоком расположен блок-планировщик (квадрат с изображением изогнутой стрелки). Он не соединен связями с другими блоками схемы, но без него ни созданный нами блок, ни график работать не будут. Планировщик можно разместить в любом месте схемы или в любой подсистеме выше по

иерархии – например, в корневой, если мы собрали нашу схему не в ней. Следует только иметь в виду, что планировщик в подсистеме может быть только один – две динамических переменных с именем “DynTime” в одной подсистеме создать невозможно, и один из планировщиков, если их будет два, не сможет работать.

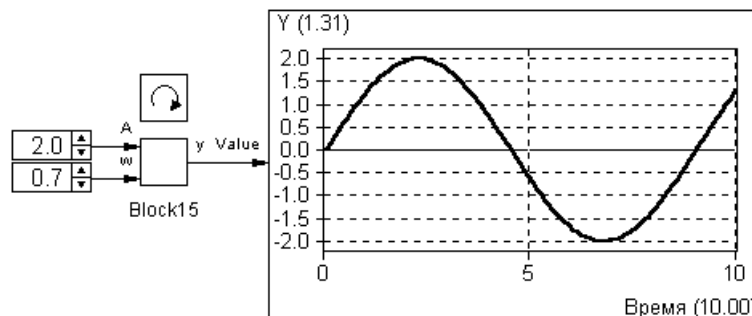


Рис. 389. Тестирование генератора синусоиды

Дважды щелкнув на планировщике, задав в его параметрах шаг расчета 0.1 и время остановки 10 секунд (синхронизацию с реальным временем можно включить или выключить по желанию) и запустив расчет, можно будет увидеть на графике изображение синусоиды с заданными в полях ввода амплитудой и частотой.

Модель нашего блока можно было бы построить и по-другому, выполняя все вычисления не в реакции на такт расчета, а в реакции на изменение динамической переменной. Сделаем это, и введем в модель ручную проверку существования переменной DynTime, как было описано выше. Для этого выполним следующие действия:

- сотрем в редакторе модели весь текст, введенный на вкладке “модель”;
- на левой панели редактора выберем вкладку “события” (см. рис. 336 на стр. 46), раскроем знаком “+” группу “моделирование и режимы” и дважды щелкнем на ее последнем подпункте “изменение динамической переменной” – в правой части редактора появится новая вкладка “изменение динамической”;
- на этой вкладке введем следующий текст программы:
 

```
if (DynTime.Exists()) // DynTime есть
 y=A*sin(w*DynTime);
else // DynTime нет
 y=rdsbcppHugeDouble;
Ready=1; // Выходные связи должны сработать
```
- отключим в параметрах модели (пункт меню редактора “модель | параметры модели”) флажки “не реагировать на события без динамических переменных” и “автозапуск при изменении динамических переменных” – на рис. 388 (стр. 134) они выделены рамками.

Теперь у нашей модели нет реакции на такт расчета. Вычисление значения выхода выполняется при изменении значения динамической переменной во введенной нами новой реакции: если переменная DynTime существует, будет вычислен синус, если нет, на выход будет подано значение переменной rdsbcppHugeDouble (см. стр. 84), указывающее на ошибку (это хуже, чем явное сообщение пользователю, но лучше, чем ничего). Затем мы присваиваем единицу сигналу готовности блока Ready, чтобы сработали связи, присоединенные к его выходу. Перед вызовом реакции на такт расчета РДС делает это автоматически, поэтому в предыдущем варианте модели мы не взводили Ready. Теперь мы работаем по изменению динамической переменной, и Ready автоматически не взводится – мы должны делать это вручную. Если убрать из программы последний оператор, выход блока будет вычисляться правильно (это можно увидеть, открыв окно его параметров и просмотрев текущие значения переменных после расчета), но на графике ничего не отобразится – вычисленное значение не поступит на график по связи.

Запустив расчет в схеме с новой моделью, мы увидим точно такой же график, что и со старой.

В созданной нами реакции на изменение динамических переменных мы не проверяем, какая именно переменная изменилась: наш блок подписан на единственную динамическую переменную “DynTime”, поэтому в такой проверке нет необходимости. Если бы блок одновременно работал с несколькими переменными и выполнял с ними какие-либо сложные вычисления, имело бы смысл не выполнять действия, связанные с переменными, которые не изменились – это сэкономило бы процессорное время. Это сделать достаточно просто: реакция на изменение динамической переменной вызывается для каждой переменной, на которую подписан блок, при этом в ее параметрах передается используемый в РДС идентификатор изменившейся переменной. Этот идентификатор является параметром функции `rdsbcppDynVarChange`, внутрь которой вставляется введенный пользователем на вкладке “изменение динамической” текст программы. В верхней части этой вкладки, непосредственно над областью ввода текста, можно увидеть заголовок этой функции, выглядящий следующим образом:

```
// Изменение динамической переменной
```

```
void rdsbcppBlockClass::rdsbcppDynVarChange(RDS_PDYNVARLINK Link)
```

Параметр `Link` типа `RDS_PDYNVARLINK` – это и есть тот самый идентификатор переменной, из-за изменения которой вызвана реакция. Идентификатор любой из динамических переменных блока можно получить функцией-членом `GetLink`, и, если сравнить результат этой функции с `Link`, можно узнать, не эта ли переменная изменилась. Оператор сравнения для переменной `DynTime` выглядел бы следующим образом:

```
if (DynTime.GetLink() == Link)
{ // Действия при изменении DynTime
 ...
}
```

Можно также использовать логическую функцию-член `CheckLink`, которая возвращает истину, если переданный в ее параметре идентификатор совпадает с идентификатором ее объекта:

```
if (DynTime.CheckLink(Link))
{ // Действия при изменении DynTime
 ...
}
```

Следует помнить, что параметр `Link` передается только в реакцию на изменение динамической переменной, поэтому в других реакциях (например, в реакции на такт расчета) приведенные выше проверки использовать нельзя. Если вычисления выполняются в реакции на такт расчета, необходимо вводить в переменные блока какие-либо дополнительные флаги, взводить их в реакции на изменение динамических переменных и проверять значения этих флагов в такте расчета.

Помимо упомянутых выше функций-членов `NotifySubscribers`, `Exists`, `GetLink` и `CheckLink`, у каждого объекта, создаваемого модулем автокомпиляции для динамической переменной, есть функции для создания, удаления, и получения доступа к переменной вручную. Они используются в тех случаях, когда нельзя жестко указать имя переменной в самой модели, как в рассмотренном примере. Эти функции и пример их использования описаны в §3.7.7 (стр. 203).

### §3.7.3.2. Создание динамических переменных

Описывается создание динамической переменной, к которой смогут обращаться другие блоки, и присвоение ей значений.

В §3.7.3.1 мы рассмотрели модель блока, считывающего значение из динамической переменной, за создание и присвоение значений которой отвечает другой блок –



стандартный блок-планировщик динамического расчета. Создадим теперь модель, которая сама будет создавать динамическую переменную, к которой будут обращаться другие блоки.

Пусть в создаваемой схеме необходимо моделировать какие-либо физические процессы, зависящие от температуры окружающей среды. Что это за процессы и как они будут моделироваться, в данном случае не важно – общие принципы моделирования любых процессов, протекающих во времени, описаны в §3.7.4 (стр. 145). В такой схеме, вероятно, будет очень много блоков, которым для расчета необходимо значение этой температуры. Передавать его по связям не очень удобно – таких связей будет слишком много, и они покроют собой всю схему. Гораздо лучше сделать температуру окружающей среды динамической переменной, к которой, при необходимости, смогут подключаться блоки – точно так же блоки, которым требуется текущее значение системного времени, подключаются к динамической переменной “DynTime”.

Дадим переменной, в которой будет храниться температура окружающей среды, имя “AmbientTemperature”, и создадим блок с автокомпилируемой моделью, которая будет создавать эту переменную и записывать в нее значение своего входа *t*. Структура переменных этого блока будет очень простой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 1                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| t          | double     | Вход              | ✓           | 0                         |

Модель блока будет запускаться при самом первом запуске расчета (у сигнала Start единичное начальное значение) и при поступлении нового значения на вход *t* (у этого входа установлен флажок “пуск”). Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных.

Теперь прикажем модели создавать в подсистеме, в которой находится ее блок, вещественную динамическую переменную с именем “AmbientTemperature”. Можно было бы создать эту переменную сразу в корневой подсистеме, чтобы ее видели все блоки схемы, однако, лучше использовать для этого подсистему блока-создателя – в этом случае переменная будет видна только в этой подсистеме и всех вложенных в нее (см. §1.5 части I), и, поэтому, в схеме можно будет разместить несколько подсистем с разными температурами, если это понадобится. Чтобы добавить в модель создание динамической переменной, следует выполнить следующие шаги:

- на вкладке “переменные” левой панели редактора модели в нижней ее части (см. рис. 334 на стр. 42) нажать кнопку со знаком “+”;
- в открывшемся окне добавления динамической переменной (рис. 390) установить флажок “произвольная переменная”;
- в поле “имя переменной в RDS” выбрать вариант “фиксированное имя” и ввести имя создаваемой переменной (в данном случае – “AmbientTemperature”);
- при переходе в поле “имя переменной в программе” в нем появится то же самое имя – его можно оставить без изменения, в программе для обращения к нашей переменной мы будем использовать это же имя AmbientTemperature;
- в выпадающем списке “блок-владелец” выбрать “подсистема (RDS\_DVPARENT)” – мы создаем переменную в одной подсистеме с блоком;
- в выпадающем списке “действие” выбрать “создать”;
- в выпадающем списке “тип” оставить уже находящийся там вариант “double” – наша переменная будет вещественной;
- закрыть окно добавления переменной кнопкой “ОК”.

Рис. 390. Добавление создания переменной “AmbientTemperature” в блок

После всех этих действий в списке на нижней части вкладки “переменные” должна появиться переменная “AmbientTemperature” со значком “+” в левой колонке (этот значок указывает на то, что мы создаем переменную, см. §3.6.3 на стр. 42).

На вкладке “модель” в правой части окна редактора (см. рис. 321 на стр. 24) введем следующий текст:

```
AmbientTemperature=t;
AmbientTemperature.NotifySubscribers();
```

В первой строчке программы мы записываем в динамическую переменную AmbientTemperature значение входа t, во второй – уведомляем все подписавшиеся на нее блоки об изменении переменной вызовом у объекта, связанного с этой переменной в нашей программе, функции-члена NotifySubscribers. На этом следует остановиться подробнее.

Событие изменения динамической переменной, на которое могут реагировать модели подключившихся к ней блоков, не возникает автоматически при присвоении этой переменной нового значения. Динамическая переменная – это, фактически, общая область памяти, разделяемая несколькими блоками, и запись данных в эту память не влечет за собой никаких дополнительных последствий. Блок, записавший что-то в динамическую переменную, должен явно сообщить об этом РДС, когда запись полностью завершена. В данном случае наша переменная имеет тип double, и мы присваиваем ей значение единственным оператором, поэтому запись начинается и кончается в одном и том же операторе. Если бы переменная была структурой или массивом, модель должна была бы сообщить РДС об изменении ее значения только после того, как все поля структуры или все элементы массива записаны (пример работы с динамической переменной сложного типа приведен в §3.7.3.3 на стр. 141). По этой причине уведомление РДС об изменении переменной не встроено в операторы присваивания объектов, через которые осуществляется доступ к этим переменным из автокомпилируемых моделей: когда запись нового значения окончена, у объекта динамической переменной нужно вручную вызвать функцию NotifySubscribers. В этот момент во всех блоках, подписавшихся на эту переменную, возникнет событие изменения динамической переменной RDS\_BFM\_DYNVARCHANGE: управление будет последовательно передано всем моделям этих блоков, которые смогут считать новое значение и как-то его использовать.

Если все описанные выше операции сделаны правильно, окно редактора модели созданного блока должно выглядеть примерно так, как на рис. 391.

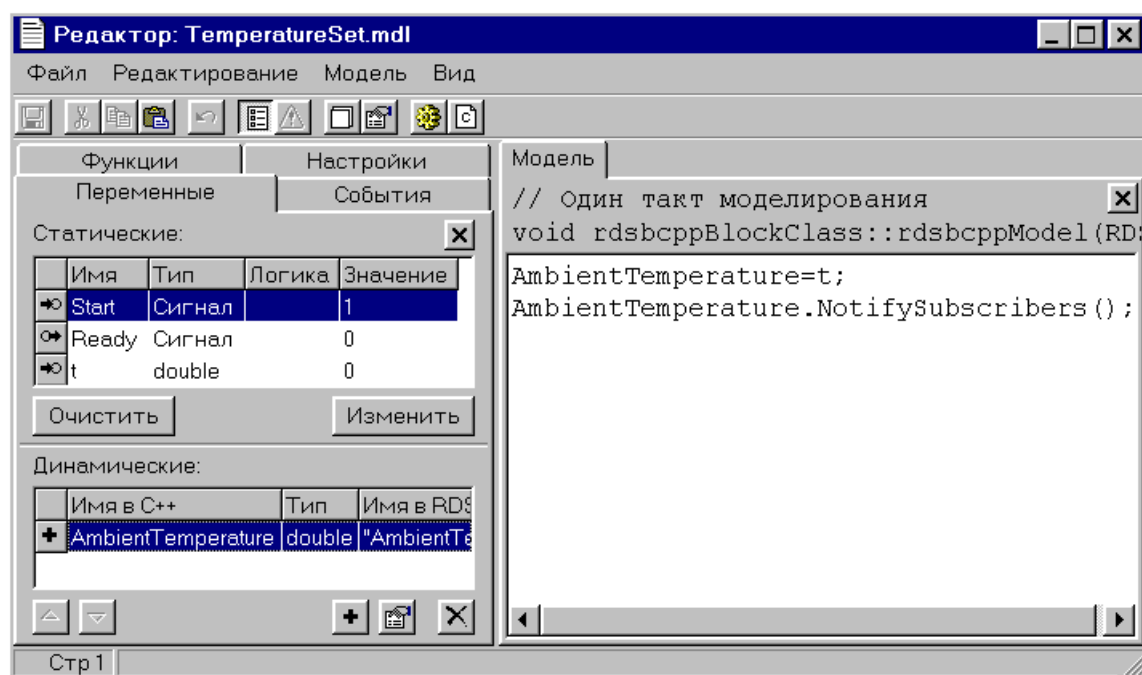


Рис. 391. Редактор модели блока задания температуры

Теперь у нас есть блок, создающий вещественную динамическую переменную “AmbientTemperature” и записывающий в нее значение своего входа при его изменении. Переменная будет создаваться и удаляться вместе с самим блоком – если просмотреть полный исходный текст программы, формируемый модулем автокомпиляции для этой модели (пункт меню “модель | показать текст C++”), можно будет увидеть команды создания и удаления переменной в конструкторе и деструкторе класса блока соответственно. Проверить работу блока мы пока не можем – у нас нет блоков, считывающих значение переменной. Можно, конечно, воспользоваться стандартными библиотечными блоками работы с динамическими переменными, настроив их на работу с нашей, но мы поступим иначе: создадим еще одну модель, которая будет выдавать значение “AmbientTemperature” на выход блока *t*. Это поможет проиллюстрировать подписку на нестандартную динамическую переменную – в §3.7.3.1 (стр. 129) мы подключались к стандартной переменной “DynTime”, и нам не приходилось настраивать параметры подписки.

Структура переменных блока, читающего динамическую переменную, будет такой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| t          | double     | Выход             |             | 0                         |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95), введем в модель эту структуру переменных и добавим в нее подписку на переменную “AmbientTemperature”. Для этого следует:

- на вкладке “переменные” левой панели редактора модели в нижней ее части нажать кнопку со знаком “+”;
- в открывшемся окне добавления динамической переменной (рис. 392) установить флажок “произвольная переменная”;
- в поле “имя переменной в RDS” выбрать вариант “фиксированное имя” и ввести имя переменной, на которую мы подписываемся, то есть “AmbientTemperature” – оно должно

точно совпадать с именем, которое мы вводили в предыдущем блоке при создании переменной;

Новая динамическая переменная

☐ Стандартная переменная: Динамический расчет: время

☒ Произвольная переменная

Переменная

Имя переменной в RDS:

☒ Фиксированное имя: AmbientTemperature

☐ Из настроечного параметра: [dropdown]

Имя переменной в программе: AmbientTemperature

Тип: double

Блок-владелец: Подсистема (RDS\_DVPARENT)

Действие: Найти и подписаться

Справка OK Отмена

Рис. 392. Добавление подписки на переменную “AmbientTemperature” в блок

- при переходе в поле “имя переменной в программе” в нем появится то же самое имя – его можно оставить без изменения;
- в выпадающем списке “блок-владелец” выбрать “подсистема (RDS\_DVPARENT)”, чтобы поиск переменной начинался в родительской подсистеме блока;
- в выпадающем списке “действие” выбрать “найти и подписаться”, чтобы, если переменная будет отсутствовать в родительской подсистеме, поиск продолжился вверх по иерархии;
- в выпадающем списке “тип” оставить уже находящийся там вариант “double” – у нас вещественная переменная;
- закрыть окно добавления переменной кнопкой “ОК”.

Вкладку “модель” редактора мы оставим пустой – наша модель не будет ничего вычислять в такте расчета. Вместо этого мы добавим в нее реакцию на изменение динамической переменной, как во втором примере §3.7.3.1. Для этого:

- на левой панели редактора выберем вкладку события (см. рис. 336 на стр. 46), раскроем знаком “+” группу “моделирование и режимы” и дважды щелкнем на ее последнем подпункте “изменение динамической переменной” – в правой части редактора появится новая вкладка “изменение динамической”;
- на этой вкладке введем следующий текст программы:

```
t=AmbientTemperature;
Ready=1; // Выходные связи должны сработать
```

В этой реакции мы переписываем значение из AmbientTemperature в выход блока t и взводим сигнал Ready, чтобы выходные связи блока сработали в следующем такте расчета. Выход Ready, разрешающий передачу данных выходов блока по связям, автоматически взводится только перед вызовом реакции на такт расчета, поэтому в любых других реакциях, где мы изменяем выходы блока, мы должны вручную присвоить ему единицу (после передачи данных он автоматически сбрасывается обратно в ноль). На самом деле, в данном случае модель работала бы, даже если бы мы не взводили Ready – параметры созданной модели мы не меняли, поэтому при изменении любой динамической переменной, на которую подписан блок, будет взведен сигнал запуска Start, а это, в свою очередь, приведет к тому, что в ближайшем такте расчета наша модель будет запущена (то, что мы не вводили реакцию на такт расчета, не повлияет на запуск), а перед запуском РДС автоматически взведет Ready.

Тем не менее, лучше взводить Ready вручную, а флаг “автозапуск при изменении динамических переменных” (см. рис. 388 на стр. 134) в параметрах модели при этом можно отключить.

Теперь у нас есть блок, записывающий значение температуры в динамическую переменную, и блок, считывающий его оттуда. Для проверки этих моделей соберем схему, изображенную на рис. 393. В этой схеме слева расположен блок с моделью, записывающий вход в динамическую переменную “AmbientTemperature” (к его входу  $t$  присоединено поле ввода), а справа – два одинаковых блока с моделью, подающей значения этой переменной на выход (к их выходам  $t$  присоединены числовые индикаторы). “Размножить” созданный блок со второй моделью можно, копируя его в буфер обмена клавишами Ctrl+C и вставляя оттуда клавишами Ctrl+V, при этом на запрос модуля автокомпиляции о том, нужно ли создавать новую модель для копии блока, следует отвечать “использовать тот же файл модели” (см. рис. 327 на стр. 30). Запустив расчет и изменяя значение в поле ввода, можно увидеть, что значения на индикаторах изменяются синхронно: информация о температуре передается между блоками через динамическую переменную. Теперь мы можем, при желании, включать подписку на “AmbientTemperature” в модель любого блока, которому для расчетов нужна температура окружающей среды – точно так же, как мы подписывались на значение времени, формируемое блоком-планировщиком.

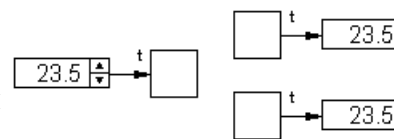


Рис. 393. Проверка блоков, работающих с “AmbientTemperature”

В рассмотренном примере один и тот же блок и создавал динамическую переменную, и записывал в нее значение. Это не является требованием РДС: записывать информацию в динамическую переменную и уведомлять остальные блоки о ее изменении (в автокомпилируемых моделях – вызовом NotifySubscribers) может не только блок-создатель, но и любой из подписавшихся на переменную блоков. Главное, чтобы блок, создавший переменную, был в подсистеме единственным: два блока не смогут создать в одной подсистеме две переменные с одинаковым именем. Можно, например, разделить функции следующим образом: один блок будет только создавать переменную (обеспечивать ее наличие в подсистеме), группа других – записывать в нее значения, еще одна группа – считывать эти значения. Один и тот же блок может и записывать, и считывать значение переменной, если этого требует логика его работы.

Следует обратить внимание на то, что в этом примере имя переменной “AmbientTemperature” жестко встроено в созданные нами модели, и, чтобы изменить его, придется изменить сами модели и скомпилировать их заново. Модуль автокомпиляции позволяет связать имя переменной с настроечным параметром блока, чтобы его мог изменять пользователь. Для этого необходимо при добавлении динамической переменной в редакторе модели в поле “имя переменной в RDS” выбрать вместо варианта “фиксированное имя” вариант “из настроечного параметра” (см. рис. 392) и ввести имя параметра, в котором будет храниться имя переменной. Подробнее этот вариант рассматривается в §3.7.7 (стр. 203).

### §3.7.3.3. Динамические переменные сложных типов

Рассматривается работа с динамическими переменными сложных типов – структурами, матрицами и т.п.

В примерах в §3.7.3.1 (стр. 129) и §3.7.3.2 (стр. 136) модели блоков работали с простыми динамическими переменными вещественного типа double. Динамическая переменная может быть любого типа, допустимого в РДС: структурой, матрицей, матрицей структур и т.п. В автокомпилируемых моделях обращение к таким сложным переменным производится по стандартным для языка C правилам: индексы массивов записываются в квадратных скобках, имена полей структур отделяются от имен самих переменных точкой.

Фактически, во фрагментах программ, вводимых пользователем в редактор модели, обращение к динамической переменной ничем не отличается от обращения к статической, описанного в §3.7.2 (стр. 92).

В предыдущем параграфе мы рассматривали блоки, обеспечивающие схему значением температуры окружающей среды, которая передавалась через вещественную динамическую переменную “AmbientTemperature”. Пусть блокам в этой схеме нужна не только температура среды, но и влажность и атмосферное давление. Можно было бы завести еще пару вещественных динамических переменных и записывать эти значения в них, но при этом в блоках, присваивающих значения переменным, потребовалось бы три вызова функции `NotifySubscribers` для уведомления остальных блоков об изменении значения – по одному вызову для каждой переменной. Вероятнее всего, все три этих значения будут изменяться одновременно, в момент считывания параметров окружающей среды или расчета динамики их изменения каким-либо блоком. Удобнее будет создать структуру, полями которой будут температура, влажность и давление, и динамическую переменную такого типа. Блок, вычисливший или получивший новые параметры окружающей среды, будет записывать их в поля структуры и делать единственный вызов `NotifySubscribers`.

Назовем структуру, в которой буду храниться параметры среды, “Ambience”, а ее поля для температуры, влажности и давления – “Temperature”, “Humidity” и “Pressure” соответственно. Для добавления в РДС этой структуры нужно выполнить следующие шаги:

- в режиме редактирования (должна быть загружена какая-либо схема или создана новая) выбрать пункт главного меню “система | структуры” – откроется окно со списком уже имеющихся в РДС структур;
- в окне структур нажать на кнопку со знаком “+” в правой части окна – откроется пустое окно редактирования структуры;
- ввести в окне редактирования имя типа структуры “Ambience” и заполнить список полей согласно рис. 394;
- закрыть окно редактирования структуры кнопкой “ОК”.

| Начало | Имя         | Тип    | К-т | По умолчанию |
|--------|-------------|--------|-----|--------------|
| 0      | Temperature | double | 8   | 0            |
| 8      | Humidity    | double | 8   | 0            |
| 16     | Pressure    | double | 8   | 0            |
|        |             |        |     |              |

Рис. 394. Структура “Ambience” в окне редактирования структуры

Как и в примере из §3.7.3.2 (стр. 136), создадим блоки с автокомпилируемыми моделями, одна из которых будет создавать динамическую структуру типа “Ambience” и записывать в ее поля значения входов блока, а другая – считывать поля структуры и выдавать их на свои выходы. Динамическую переменную назовем точно так же, как и тип структуры – “Ambience”.

Начнем с блока, записывающего свои входы в динамическую переменную. У него будет три входа: H – для влажности, T – для температуры, и P – для давления. Структура его переменных будет такой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 1                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| H          | double     | Вход              | ✓           | 0                         |
| T          | double     | Вход              | ✓           | 0                         |
| P          | double     | Вход              | ✓           | 0                         |

Модель блока будет запускаться при самом первом запуске расчета (у сигнала Start единичное начальное значение) и при поступлении нового значения на входы H, T и P (у них установлен флажок “пуск”). Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели его структуру переменных. Чтобы наш блок создавал динамическую структуру, выполним следующие шаги:

- на вкладке “переменные” левой панели редактора модели в нижней ее части (см. рис. 334 на стр. 42) нажмем кнопку со знаком “+”;
- в открывшемся окне добавления динамической переменной (см. рис. 390 на стр. 138) установим флажок “произвольная переменная”;
- в поле “имя переменной в RDS” выберем вариант “фиксированное имя” и введем имя создаваемой переменной – “Ambience”;
- при переходе в поле “имя переменной в программе” в нем появится то же самое имя, его можно оставить без изменения (в программе для обращения к нашей переменной мы тоже будем использовать имя Ambience);
- в выпадающем списке “блок-владелец” выберем “подсистема (RDS\_DVPARENT)” – мы создаем переменную в одной подсистеме с блоком;
- в выпадающем списке “действие” выберем “создать”;
- в выпадающем списке “тип” выберем “Ambience” – наша переменная будет структурой этого типа (то, что имя типа структуры совпадает с именем переменной, не важно, имя типа структуры РДС не используется в программе и, поэтому, не вызовет конфликта идентификаторов);
- закроем окно добавления переменной кнопкой “ОК”.

На вкладке “модель” окна редактора введем следующий текст:

```
// Запись входов в поля структуры
Ambience.Temperature=T;
Ambience.Humidity=H;
Ambience.Pressure=P;
// Уведомление подписчиков об изменении данных
Ambience.NotifySubscribers();
```

Здесь мы по очереди записываем в поля структуры входы блока, соответствующие этим полям по смыслу – имя поля (например, Humidity) отделяется от имени переменной (Ambience) точкой. В последней строчке программы мы, как и в предыдущих примерах, вызываем у динамической переменной функцию-член NotifySubscribers, которая уведомляет все блоки, подписавшиеся на эту переменную, о том, что мы записали в нее новые значения (при этом в моделях этих блоков вызывается реакция на изменение динамической переменной).

Теперь займемся блоком, который будет выдавать значения полей динамической структуры на выходы. Назовем выходы блока так же, как и поля структуры: Temperature (температура), Humidity (влажность) и Pressure (давление):

| <i>Имя</i>  | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|-------------|------------|-------------------|-------------|---------------------------|
| Start       | Сигнал     | Вход              | ✓           | 0                         |
| Ready       | Сигнал     | Выход             |             | 0                         |
| Temperature | double     | Выход             |             | 0                         |
| Humidity    | double     | Выход             |             | 0                         |
| Pressure    | double     | Выход             |             | 0                         |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95), введем в модель приведенную выше структуру переменных и добавим подписку на переменную “Ambience”. Для этого:

- на вкладке “переменные” левой панели редактора модели в нижней ее части нажмем кнопку со знаком “+”;
- в открывшемся окне добавления динамической переменной (см. рис. 392 на стр. 140) установим флажок “произвольная переменная”;
- в поле “имя переменной в RDS” выберем вариант “фиксированное имя” и введем имя переменной “Ambience”;
- при переходе в поле “имя переменной в программе” в нем появится то же самое имя – его можно оставить без изменения;
- в выпадающем списке “блок-владелец” выберем “подсистема (RDS\_DVPARENT)”, чтобы поиск переменной начинался в родительской подсистеме блока;
- в выпадающем списке “действие” выберем “найти и подписаться”, чтобы, если переменная будет отсутствовать в родительской подсистеме, поиск продолжился вверх по иерархии;
- в выпадающем списке “тип” выберем “Ambience” – мы подписываемся на структуру такого типа;
- закроем окно добавления переменной кнопкой “ОК”.

На вкладке “модель” редактора мы не будем ничего вводить – наша модель не выполняет вычислений в такте расчета. Вместо этого мы добавим в нее реакцию на изменение динамической переменной (см. стр. 140), в которую введем следующий текст программы:

```
// Запись полей структуры в выходы
Temperature=Ambience.Temperature;
Humidity=Ambience.Humidity;
Pressure=Ambience.Pressure;
Ready=1; // Выходные связи должны сработать
```

В этой реакции мы переписываем значения из полей динамической переменной Ambience в соответствующие этим полям выходы блока и взводим сигнал Ready, чтобы выходные связи блока сработали в следующем такте расчета.

Для проверки этих моделей соберем схему, изображенную на рис. 395. В этой схеме слева расположен блок с моделью, записывающий входы в динамическую структуру, а справа – блок, подающий поля этой структуры на выходы. Запустив расчет и изменяя значения в полях ввода левого блока, можно увидеть, что значения на индикаторах правого блока изменяются синхронно.

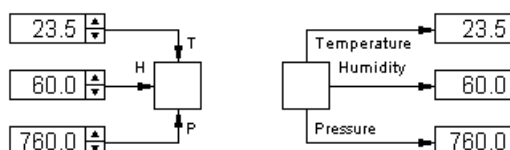


Рис. 395. Тестирование блоков, работающих с динамической структурой



В рассмотренном примере у нас было три отдельных вещественных входа, которые мы записывали в поля структуры, и три отдельных вещественных выхода, на которые мы подавали эти же поля в другом блоке. Это позволило показать, как в тексте программы осуществляется обращение к отдельным полям сложной динамической переменной, но, в данном простом примере, такое решение не оптимально. Если бы мы сделали у левого блока вход типа “Ambience”, а у правого – выход того же типа, в каждой из моделей можно было бы обойтись одним единственным оператором присваивания. Пусть вход левого блока называется X, а выход правого – Y, оба они – структуры типа “Ambience”. В этом случае модель левого блока выглядела бы так:

```
// Копирование структуры-входа в динамическую структуру
Ambience=X;
// Уведомление подписчиков об изменении данных
Ambience.NotifySubscribers();
```

Модель правого блока содержала бы обратный оператор присваивания:

```
// Копирование динамической структуры в выход
Y=Ambience;
Ready=1; // Выходные связи должны сработать
```

В объектах, создаваемых модулем автокомпиляции для доступа к статическим и динамическим переменным, переопределен оператор присваивания, поэтому приведенная выше запись допустима. В схеме с новой моделью поля ввода и индикаторы нужно было бы подключать уже не к отдельным входам и выходам, а к полям переменных X и Y (рис. 396).

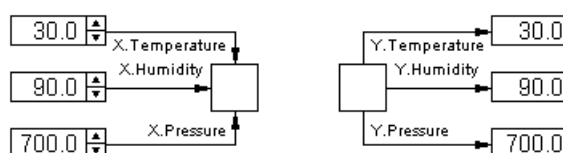


Рис. 396. Блоки, работающие с динамической структурой, входы и выходы которых – тоже структуры

Работа с динамическими массивами, матрицами и строками производится аналогичным образом: все операторы, применяемые к статическим переменным этих типов, могут применяться и к динамическим. Например, если DynArray – динамический массив, то к его i-му элементу можно обратиться оператором “DynArray[i]”. Функции NotifySubscribers, GetLink и CheckLink (см. §3.7.3.1 на стр. 129) при этом всегда применяются к самой динамической переменной, а не к ее отдельным элементам: например, для динамического массива DynArray нужно записывать “DynArray.NotifySubscribers()”.

### §3.7.4. Моделирование длящихся во времени процессов

Рассматривается создание динамических блоков – блоков, моделирующих какие-либо процессы, протекающие во времени и описываемые дифференциальными или разностными уравнениями. Приводятся примеры решения конкретных задач, описывается решения проблем с устойчивостью счета.

#### §3.7.4.1. Общие принципы численного моделирования непрерывных процессов

Описываются общие принципы перехода от описания процесса дифференциальными уравнениями к разностным уравнениям, которые можно закладывать в модель блока.

Достаточно часто приходится моделировать процессы, в которых величины, описывающие их, изменяются во времени по каким-либо законам. Случаи, когда для каждой величины известна формула закона ее изменения от времени, встречаются редко – как правило, известны связи параметров и состояний процесса друг с другом (причем эти связи могут быть довольно сложными), в результате действия которых процесс идет тем или иным

образом. К тому же, на процесс могут воздействовать различные внешние факторы, законы изменения которых не известны в принципе.

Рассмотрим простой пример: некоторый объект движется по прямой с заданной скоростью. Требуется определить положение объекта в заданный момент времени. Из курса физики средней школы известно, что при постоянной скорости положение объекта в любой момент времени определяется формулой

$$x(t) = x_0 + v t,$$

где  $x_0$  — начальное положение объекта на прямой,  
 $v$  — скорость объекта,  
 $t$  — время.

Если скорость не постоянна, эту формулу использовать нельзя. В этом случае приходится описывать движение объекта дифференциальным уравнением, то есть уравнением с участием производных. Скорость объекта  $v$  — это производная его координаты  $x$  по времени  $t$  (то есть, фактически, “быстрота изменения координаты”). При нулевом значении времени объект находится в положении  $x_0$ . Значит, можно записать дифференциальное уравнение

$$\frac{dx}{dt} = v, x(0) = x_0.$$

Таким образом, для определения координаты  $x$  объекта в интересующий нас момент времени  $t$  необходимо вычислить следующий определенный интеграл:

$$x(t) = x_0 + \int_{\tau=0}^t v(\tau) d\tau.$$

Для некоторых зависимостей скорости от времени  $v(t)$  этот интеграл может быть вычислен аналитически. Но, к сожалению, значение скорости в каждый момент времени чаще всего само получается в результате решения некоторого набора уравнений. Или, что тоже возможно, оно поступает снаружи — например, в схеме расположен блок-рукоятка для задания скорости, и пользователь все время двигает ее, изменяя скорость так, как ему хочется. В этом случае приведенное выше дифференциальное уравнение приходится решать одним из известных методов численного интегрирования, переходя от непрерывно текущего времени к дискретному. Ниже мы рассмотрим примеры такого решения.

Для численного решения дифференциальных уравнений мы заменяем непрерывное время дискретным, изменяющимся небольшими шагами. Чем меньше будут эти шаги, тем ближе будет дискретное время к непрерывному, и тем ближе численное решение уравнения будет к аналитическому. Шаги изменения дискретного времени могут быть как постоянными, так и изменяющимися. Для простоты будем считать их постоянными — тем более, что стандартный блок-планировщик, обеспечивающий схему текущим значением времени через динамическую переменную “DynTime” (см. §3.7.3.1 на стр. 129), изменяет это время с заданным постоянным шагом. Перейдя к такому дискретному времени, мы заменяем дифференциальные уравнения разностными, которые позволяют вычислить значение некоторой переменной на следующем шаге по значениям переменных (как данной, так и, возможно, нескольких других) на одном или нескольких предыдущих шагах. Существует множество различных методов численного интегрирования, то есть способов, позволяющих получить разностные уравнения из дифференциальных, но мы будем использовать самый простой из них — метод Эйлера [3]. Рассмотрим его на примере приведенной выше задачи определения координаты объекта, скорость движения которого задается извне.

Переходя к дискретному времени, мы заменяем непрерывное время  $t$  набором дискретных значений  $t_0, t_1, t_2, \dots, t_k, t_{k+1}, \dots$ . В каждый из уже прошедших моментов времени вплоть до текущего момента  $t_k$  нам должны быть известны значения скорости  $v_0 \dots v_k$  — скорость задается снаружи исследуемой нами системы, мы можем запоминать всю историю ее изменения, если это необходимо, и всегда знаем текущее ее значение. Мы не можем его вычислить, мы можем только получить его откуда-нибудь — например, оно может приходить

по связи на вход нашего блока от каких-либо других блоков, от рукоятки, которую двигает пользователь, от реального аппаратного датчика скорости и т.п. Наша задача – получить выражение координаты объекта  $x_{k+1}$  в следующий момент времени  $t_{k+1}$ , зная историю движения этого объекта, то есть, в общем случае, набор значений его координат  $x_0, x_1, \dots, x_k$  и скоростей  $v_0, v_1, \dots, v_k$  (мы увидим, что в простейшем случае достаточно иметь только предыдущие значения координаты и скорости, то есть  $x_k$  и  $v_k$ ). Будем считать, что на интервале времени  $t_k \dots t_{k+1}$  скорость объекта не меняется и равна  $v_k$ . Скорость  $v$  – это производная координаты  $x$  по времени  $t$ . Поскольку на указанном интервале она постоянна, можно записать следующее выражение:

$$\left. \frac{dx}{dt} \right|_{t \in [t_k, t_{k+1}]} = \frac{x_{k+1} - x_k}{t_{k+1} - t_k} = v_k.$$

Действительно, при постоянной скорости на интервале  $t_k \dots t_{k+1}$  эта скорость равна отношению изменения координаты к изменению времени. Отсюда можно получить необходимое нам выражение для вычисления  $x_{k+1}$ :

$$x_{k+1} = x_k + (t_{k+1} - t_k) v_k.$$

Это и есть разностное уравнение, полученное по методу Эйлера. Для вычисления очередного значения координаты  $x_{k+1}$  в момент времени  $t_{k+1}$  нам нужно знать прошедший с последнего вычисленного отсчета интервал времени  $t_{k+1} - t_k$ , прошлое значение координаты  $x_k$  и скорость ее изменения на момент начала интервала  $v_k$ . В общем виде, если у нас есть дифференциальное уравнение первого порядка (то есть с одной производной) вида

$$\frac{dy}{dt} = f(x, y, z, \dots, t), \quad y(0) = y_0,$$

где  $x, y, z$  – некоторые переменные,  $t$  – время, а  $f$  – известная функция, то разностное уравнение, полученное по методу Эйлера для этого дифференциального, будет выглядеть так:

$$y_{k+1} = y_k + (t_{k+1} - t_k) f(x_k, y_k, z_k, \dots, t_k).$$

При этом функция  $f$  может содержать внутри себя любые нелинейные функции или даже сложные алгоритмы вычисления. Нелинейность функции  $f$ , которая могла бы явиться препятствием для получения аналитического решения дифференциального уравнения, не мешает его численному решению. Нужно только правильно подобрать шаг расчета, то есть интервал между соседними отсчетами времени, чтобы моделирование процесса давало удовлетворительную точность (влияние шага расчета на численное решение показано в примере в §3.7.4.3 на стр. 159).

Если дифференциальное уравнение имеет больший порядок, надо заменить его на систему дифференциальных уравнений первого порядка любым известным способом, и привести эту систему в нормальную форму Коши, то есть в форму, в которой в левой части каждого уравнения находится первая производная, а в правых частях производных нет. После этого для каждого уравнения получившейся системы методом Эйлера записывается разностное. Допустим, у нас есть система дифференциальных уравнений в форме Коши:

$$\begin{cases} \frac{dy_1}{dt} = f_1(y_1, \dots, y_N, x_1, \dots, x_M, t) & y_1(0) = y_1^0 \\ \frac{dy_2}{dt} = f_2(y_1, \dots, y_N, x_1, \dots, x_M, t) & y_2(0) = y_2^0 \\ \dots & \dots \\ \frac{dy_N}{dt} = f_N(y_1, \dots, y_N, x_1, \dots, x_M, t) & y_N(0) = y_N^0 \end{cases},$$

где  $y_1, \dots, y_N$  – вычисляемые переменные (выходы и внутренние),  
 $x_1, \dots, x_M$  – поступающие снаружи переменные (входы),

- $f_1, \dots, f_N$  — известные функции,  
 $y_1^0, \dots, y_N^0$  — начальные значения вычисляемых переменных,  
 $t$  — время.

При помощи метода Эйлера из нее можно получить следующую систему разностных уравнений:

$$\begin{cases} y_1^{k+1} = y_1^k + (t^{k+1} - t^k) f_1(y_1^k, \dots, y_N^k, x_1^k, \dots, x_M^k, t^k) \\ y_2^{k+1} = y_2^k + (t^{k+1} - t^k) f_2(y_1^k, \dots, y_N^k, x_1^k, \dots, x_M^k, t^k) \\ \dots \\ y_N^{k+1} = y_N^k + (t^{k+1} - t^k) f_N(y_1^k, \dots, y_N^k, x_1^k, \dots, x_M^k, t^k) \end{cases}.$$

Для удобства записи номер отсчета мы здесь перенесли в верхний индекс:  $y_l^k$  – это не “ $y_l$  в степени  $k$ ”, это “ $k$ -й отсчет переменной  $y_l$ ”.

Вернемся к задаче вычисления координат объекта, движущегося с заданной скоростью, разностное уравнение для которой мы уже получили. Это разностное уравнение можно практически без изменений перенести в модель блока. Есть только одна сложность: в уравнении нам необходим интервал времени между прошлым отсчетом времени  $t_k$  и новым отсчетом  $t_{k+1}$ . Если новое значение времени  $t_{k+1}$  можно взять из динамической переменной “DynTime”, в которую его постоянно записывает блок-планировщик (см., например, §3.7.3.1 на стр. 129), то прошлый отсчет  $t_k$  в явном виде нам взять неоткуда. Придется в модели постоянно запоминать текущее значение времени в какой-нибудь внутренней переменной блока: когда планировщик изменит “DynTime” и наша модель запустится, в этой внутренней переменной окажется значение времени до последнего изменения, что нам и требуется.

Создадим блок, входом которого будет задаваемая скорость  $v$ , а выходом – координата объекта  $x$ . Проще значение времени для вычисления шага расчета  $t_{k+1} - t_k$  будем хранить во внутренней вещественной переменной  $t_0$ . Переменную для начального значения координаты объекта создавать не будем: начальное значение можно записывать в значение по умолчанию переменной  $x$  (далее в §3.7.4.2 на стр. 152 будет приведен пример модели, в которой начальные условия считываются из отдельных входов блока). Таким образом, структура переменных нашего блока будет такой:

| Имя   | Тип    | Вход/выход | Пуск | Начальное значение |
|-------|--------|------------|------|--------------------|
| Start | Сигнал | Вход       | ✓    | 0                  |
| Ready | Сигнал | Выход      |      | 0                  |
| $v$   | double | Вход       |      | 0                  |
| $x$   | double | Выход      |      | 0                  |
| $t_0$ | double | Внутренняя |      | 0                  |

В отличие от многих моделей, описанных ранее, здесь мы не даем сигналу запуска блока Start единичное начальное значение и не включаем флажок “пуск” у входа блока  $v$ : наша модель работает при изменении времени, и запускать ее при изменении входов бессмысленно. Пока время не изменится, новое значение координаты вычислено не будет, поскольку при одинаковых  $t_{k+1}$  и  $t_k$  разностное уравнение вырождается в “ $x_{k+1} = x_k$ ”, то есть координата объекта не изменяется – “лишний” запуск модели до приращения времени ничего не испортит, но и не даст ничего для расчета. Следует учитывать, что это верно для рассматриваемого здесь метода Эйлера, при использовании других методов численного интегрирования может оказаться важным не допускать запуска модели без изменения времени. В качестве начального значения переменной  $x$  мы записали ноль – считаем, что объект движется из начала координат – но можно ввести любое другое начальное значение.

Начальное значение переменной  $t_0$  **обязательно** должно быть нулем – на первом шаге предыдущее значение времени равно нулю.

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. Присоединим блок к динамической переменной “DynTime”, как описано на стр. 132. Параметры модели, которые она получила по умолчанию, изменять не будем – в результате в нашу модель будет автоматически добавлена проверка существования переменной “DynTime” и взведение сигнала запуска при ее изменении, что нам и нужно.

На вкладке “модель” редактора необходимо ввести программу, соответствующую полученному нами разностному уравнению

$$x_{k+1} = x_k + (t_{k+1} - t_k) v_k.$$

С учетом того, что предыдущее значение времени мы решили хранить во внутренней переменной  $t_0$ , модель будет выглядеть так:

```
x=x+(DynTime-t0)*v; // Разностное уравнение
t0=DynTime; // Запоминание предыдущего отсчета времени
```

Можно видеть, что формула для вычисления  $x$  в точности соответствует разностному уравнению. На каждом шаге расчета в DynTime будет находиться новое значение времени, для которого нужно вычислить новую координату  $x$ , а в  $t_0$  – значение времени, запомненное при прошлом вычислении (это обеспечивается второй строчкой программы).

Для тестирования созданного блока соберем схему, изображенную на рис. 397. В ней к входу скорости  $v$  присоединено поле ввода, а выход координаты объекта  $x$  подан на график. На второй график подано значение скорости с поля ввода, которое мы будем изменять в процессе расчета. В параметрах блока-планировщика (см. рис. 385 на стр. 131) задан шаг расчета 0.1 и время остановки 10. Синхронизация с реальным временем включена, иначе мы не успеем изменить скорость на входе блока за время расчета (можно даже замедлить расчет, увеличив в параметрах планировщика множитель задержки). На рисунке изображен результат расчета, в процессе которого сначала значение на входе блока равнялось двум, после третьей секунды оно было изменено на ноль, после пятой – на минус один (это видно на нижнем графике). В результате в течение первых трех секунд расчета координата объекта увеличивалась, затем он остановился (горизонтальная линия на верхнем графике), а затем, после пятой секунды, двинулся в обратном направлении с вдвое меньшей, чем в начале, скоростью.

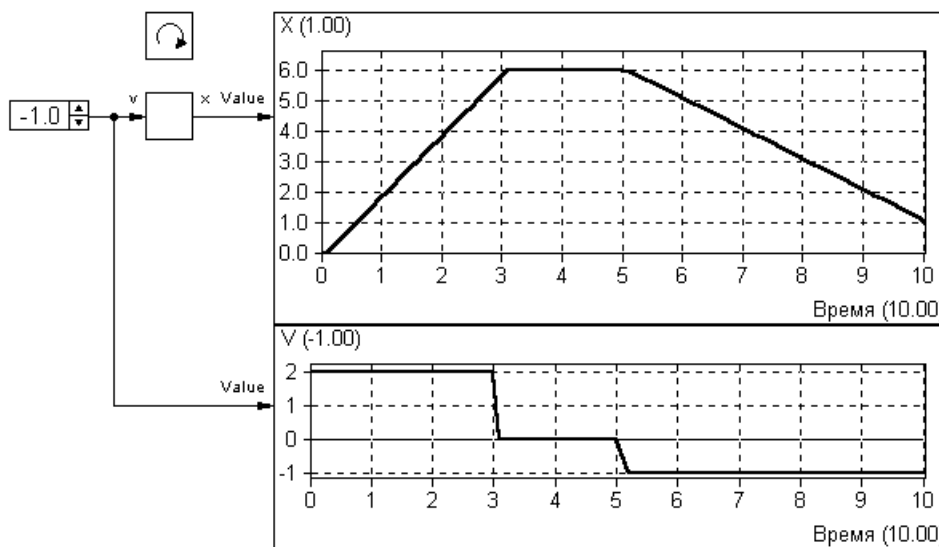


Рис. 397. Тестирование вычислителя координаты объекта, движущегося с задаваемой скоростью

Фактически, мы только что создали простейшую модель интегратора – линейного динамического блока, часто используемого при моделировании различных процессов. Из таких блоков можно собирать большие схемы, численно решающие различные системы дифференциальных уравнений, но, во многих случаях, гораздо удобнее закладывать всю систему уравнений в одну модель блока – такие примеры будут рассмотрены далее.

Разумеется, численно моделировать можно не только процессы в механических системах, вычисляя координаты, скорости и ускорения различных объектов или деталей механизмов. Описанным выше способом можно моделировать любые процессы, которые описываются системами дифференциальных уравнений: электрические, термодинамические и т.п. Рассмотрим, например, электрические процессы, протекающие в стандартной RC-цепи (рис. 398) и блок, моделирующий эти процессы.

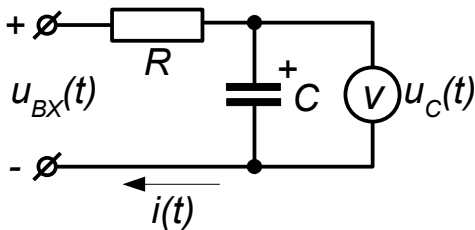


Рис. 398. Электрическая схема RC-цепи

Цепь представляет собой последовательное соединение резистора сопротивлением  $R$  и конденсатора емкостью  $C$ , подключенное к источнику напряжения  $u_{BX}(t)$ , значение которого будет задаваться на входе нашего блока. Напряжение на конденсаторе  $u_C(t)$  будет выходом нашего блока. При изменении напряжения  $u_{BX}$  конденсатор будет заряжаться или разряжаться – именно процесс его заряда и разряда мы и будем моделировать.

Сначала составим дифференциальные уравнения, описывающие электрическую схему. Резистор и конденсатор соединены последовательно, поэтому сила тока, протекающего через них, одинакова (ток через вольтметр, подключенный к конденсатору для измерения  $u_C$ , будем считать пренебрежимо малым):

$$i_R(t) = i_C(t) = i(t),$$

причем эта сила тока является функцией времени  $t$ . Сумма падений напряжения на резисторе и конденсаторе будет равна входному напряжению цепи:

$$u_R(t) + u_C(t) = u_{BX}(t).$$

Сила тока через конденсатор равна его емкости, умноженной на производную напряжения на нем по времени:

$$i(t) = C \frac{du_C(t)}{dt}.$$

Падение напряжения на резисторе равно произведению силы тока через него на его сопротивление. С использованием предыдущей формулы для силы тока можно записать:

$$u_R(t) = i(t)R = RC \frac{du_C(t)}{dt}.$$

Таким образом,

$$u_R(t) + u_C(t) = RC \frac{du_C(t)}{dt} + u_C(t) = u_{BX}(t).$$

Это и есть дифференциальное уравнение, описывающее процессы в RC-цепи. Переписав его в нормальной форме Коши, то есть переместив производную в левую часть уравнения, получим:

$$\frac{du_C}{dt} = \frac{1}{RC} (u_{BX} - u_C).$$

Для этого дифференциального уравнения должны быть заданы начальные условия – в данном случае, напряжение на конденсаторе в нулевой момент времени. Для простоты будем считать, что конденсатор исходно разряжен, то есть  $u_C(0) = 0$ . Переходя к дискретному

времени и заменяя по методу Эйлера производную напряжения на отношение его изменения за шаг расчета к изменению времени за этот же шаг, получим:

$$\left. \frac{du_C}{dt} \right|_{t \in [t^k, t^{k+1}]} = \frac{u_C^{k+1} - u_C^k}{t^{k+1} - t^k} = \frac{1}{RC} (u_{BX}^k - u_C^k).$$

Таким образом, мы получили следующее разностное уравнение для вычисления очередного ( $k+1$ -го) значения  $u_C$ :

$$u_C^{k+1} = u_C^k + (t^{k+1} - t^k) \frac{1}{RC} (u_{BX}^k - u_C^k).$$

По этому уравнению легко будет написать модель блока.

Создадим блок, входами которого будет напряжение источника  $u_{BX}(t)$  Uin в вольтах, сопротивление резистора R в омах и емкость конденсатора C в фарадах, а выходом – напряжение на конденсаторе Uс в вольтах. Конечно, удобнее было бы задавать емкость конденсатора в каких-нибудь более часто используемых единицах – например, микрофарадах – но мы, для упрощения модели, будем все задавать в единицах СИ. Прошлые значение времени для вычисления шага расчета будем, как и в предыдущем примере, хранить во внутренней вещественной переменной t0. Начальное напряжение на конденсаторе (мы решили считать его исходно разряженным) запишем в значение по умолчанию переменной Uс. Структура переменных нашего блока будет такой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| R          | double     | Вход              |             | 1                         |
| C          | double     | Вход              |             | 1                         |
| Uin        | double     | Вход              |             | 0                         |
| Uc         | double     | Выход             |             | 0                         |
| t0         | double     | Внутренняя        |             | 0                         |

Начальные значения входов R и C мы, на всякий случай, сделаем единицами: если мы забудем подключить к этим входам поля ввода, деление на их произведение на вызовет ошибки.

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в редакторе модели указанную выше структуру переменных. Присоединим блок к динамической переменной “DynTime”, как описано на стр. 132. Параметры модели изменять не будем – они останутся установленными по умолчанию, и наш блок будет автоматически запускаться при каждом изменении времени (то есть на каждом шаге расчета). На вкладке “модель” редактора введем программу, соответствующую полученному выше разностному уравнению для  $u_C$ :

```
Uc+=(DynTime-t0)*(Uin-Uc)/(R*C);
t0=DynTime;
```

В первой строчке мы вычисляем новое значение Uс на момент времени  $t^{k+1}$ , во второй – запоминаем текущее время в t0, чтобы на следующем шаге расчета вычесть его из изменившегося времени и получить величину шага. Для упрощения примера в этой модели мы не сравниваем входы блока Uс, R и C с признаком ошибки rdsbcppHugeDouble (см. стр. 84) и не проверяем R и C на равенство нулю.

Для тестирования модели соберем схему, изображенную на рис. 399. Зададим сопротивление резистора R равным 2 кОм (2000 Ом), емкость конденсатора C – 1000 мкФ (0.001 Ф). Шаг расчета в параметрах блока-планировщика (см. рис. 385 на стр. 131) зададим

равным 0.1 с, а время остановки – 10 с. Подадим на вход  $U_{in}$  напряжение 10 В и запустим расчет – на графике можно будет увидеть заряд конденсатора. Когда расчет остановится, подадим на вход блока ноль вольт и снова запустим расчет (время остановки при этом автоматически удвоится, то есть станет равным 20 с) – на графике будет видно, как разряжается ранее заряженный конденсатор.

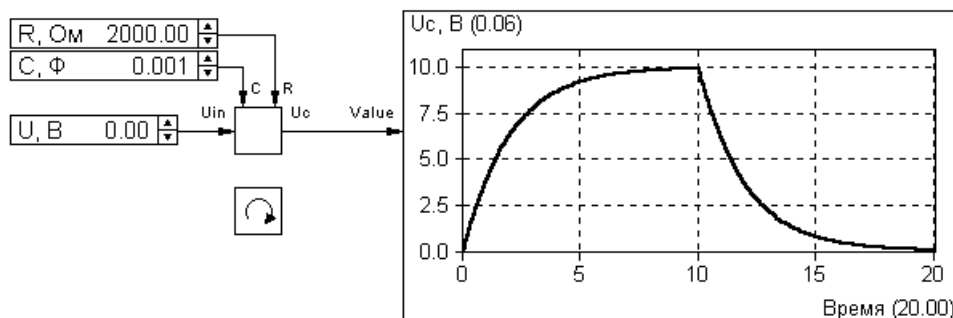


Рис. 399. Тестирование блока моделирования RC-цепи (заряд конденсатора напряжением 10 В в течение 10 секунд, затем разряд в течение еще 10 секунд)

### §3.7.4.2. Система дифференциальных уравнений и задание начальных условий

Рассматривается моделирование процессов, описываемых дифференциальным уравнением второго и выше порядка или системой дифференциальных уравнений. Описывается способ задания начальных условий при помощи отдельных входов блока.

Выше, в §3.7.4.1, были рассмотрены примеры процессов, описываемых одним дифференциальным уравнением первого порядка. Если процесс описывается уравнением более высокого порядка или системой уравнений, принципы моделирования не меняются: порядок уравнения понижается заменой его на систему уравнений первого порядка, система записывается в нормальной форме Коши, как было показано выше, после чего каждое из получившихся уравнений преобразуется в разностное одним из методов численного интегрирования – например, методом Эйлера [3].

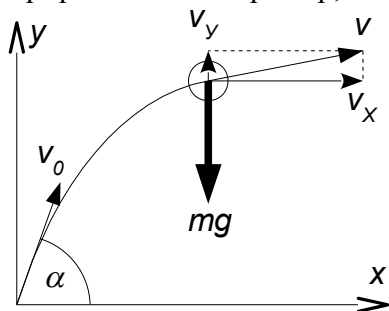


Рис. 400. Тело в свободном полете

Рассмотрим движение объекта в двух измерениях под действием силы тяжести. Будем рассчитывать полет тела, брошенного с заданной начальной скоростью  $v_0$  под заданным углом  $\alpha$  к горизонту (рис. 400). Направим ось  $x$  горизонтально в направлении полета,  $y$  – вертикально вверх. Чтобы упростить пример и не связываться с вычислением влияния сопротивления воздуха на полет тела, будем считать, что все это происходит в безвоздушном пространстве – например, на Луне (в §2.14.2 руководства программиста [1] решается такая же задача, но на Земле с учетом сопротивления воздуха). На тело, находящееся в свободном полете, действует только сила тяжести, которая сообщает ему ускорение, равное ускорению свободного падения и направленное вертикально вниз. В горизонтальном направлении никаких сил на тело не действует, и, поэтому, никаких ускорений в горизонтальном направлении телу не сообщается. Горизонтальное и вертикальное движение тела можно рассматривать независимо, раскладывая скорость  $v$  на горизонтальную и вертикальную проекции  $v_x$  и  $v_y$  соответственно, а ускорение  $a$  – на  $a_x$  и  $a_y$ . При этом горизонтальная проекция ускорения  $a_x$  на всем протяжении полета равна нулю (нет горизонтальных сил), а вертикальная  $a_y$  – ускорению свободного падения  $g$  со знаком минус (ось  $y$  направлена вверх, ускорение свободного падения – вниз):



$$\begin{cases} a_x = 0 \\ a_y = -g \end{cases}.$$

В каждой проекции ускорение – это вторая производная координаты по времени. Таким образом, мы получаем следующую систему дифференциальных уравнений, описывающих движение нашего брошенного тела в свободном полете (начальные условия пока опустим):

$$\begin{cases} \frac{d^2 x}{dt^2} = 0 \\ \frac{d^2 y}{dt^2} = -g \end{cases}.$$

Эта система состоит из двух уравнений второго порядка. Чтобы понизить порядок, вспомним, что скорость – это производная координаты по времени, а ускорение – производная скорости по времени. Введя в уравнение скорости, мы получим вместо двух уравнений второго порядка четыре уравнения первого:

$$\begin{cases} \frac{dv_x}{dt} = a_x = 0 \\ \frac{dx}{dt} = v_x \\ \frac{dv_y}{dt} = a_y = -g \\ \frac{dy}{dt} = v_y \end{cases}.$$

Теперь необходимо определить начальные условия для этой системы. Будем считать, что объект начинает движение из начала координат, поэтому  $x(0)=y(0)=0$ . Начальную скорость объекта разложим на проекции по осям координат  $v_{0x}$  и  $v_{0y}$  – эти значения и будут начальными условиями для скоростей:

$$\begin{cases} v_x(0) = v_{0x} = v_0 \cos \alpha \\ v_y(0) = v_{0y} = v_0 \sin \alpha \end{cases}.$$

Таким образом, мы получаем следующую систему дифференциальных уравнений, описывающих движение нашего брошенного тела в свободном полете:

$$\begin{cases} \frac{dv_x}{dt} = 0 & v_x(0) = v_0 \cos \alpha \\ \frac{dx}{dt} = v_x & x(0) = 0 \\ \frac{dv_y}{dt} = -g & v_y(0) = v_0 \sin \alpha \\ \frac{dy}{dt} = v_y & y(0) = 0 \end{cases}.$$

Эта система легко решается аналитически, но мы рассматриваем моделирование процессов с дискретным временем, поэтому будем решать ее численно. Система записана в нормальной форме Коши, и из нее легко получить разностные уравнения описанным в §3.7.4.1 (стр. 145) методом Эйлера:

$$\begin{cases} v_x^{k+1} = v_x^k & v_x^0 = v_0 \cos \alpha \\ x^{k+1} = x^k + h v_x^k & x^0 = 0 \\ v_y^{k+1} = v_y^k - h g & v_y^0 = v_0 \sin \alpha \\ y^{k+1} = y^k + h v_y^k & y^0 = 0 \end{cases}, \text{ где } h = t^{k+1} - t^k$$

(буквой  $h$  обозначен шаг расчета, то есть интервал между соседними моментами дискретного времени). Для удобства записи номер отсчета мы снова перенесли в верхний индекс:  $y^k$  – это не “ $y$  в степени  $k$ ”, это “ $k$ -й отсчет переменной  $y$ ”.

Из первого уравнения этой системы видно, что горизонтальная составляющая скорости  $v_x$  в процессе полета не изменяется и всегда равна  $v_0 \cos \alpha$ , следовательно, вычислять ее в модели блока на каждом шаге не нужно.

Создадим блок, который будет вычислять траекторию полета тела по полученным разностным уравнениям. Выходами нашего блока будут вещественные переменные  $x$  и  $y$ , в которых мы будем записывать вычисленные координаты тела. В вещественных переменных  $vx$  и  $vy$  мы будем хранить и вычислять проекции скорости тела – эти переменные можно сделать внутренними. Для упрощения примера вместо модуля начальной скорости  $v_0$  и ее угла к горизонту  $\alpha$  мы будем просто вводить вычисленные вручную значения проекций начальной скорости в значения по умолчанию переменных  $vx$  и  $vy$  (позже мы изменим модель так, чтобы можно было непосредственно подавать на вход блока угол и начальную скорость). Будем считать, что мы бросаем наше тело со скоростью 3 м/с под углом  $50^\circ$  к горизонту – при этом начальное значение  $vx$  должно равняться  $3 \cos 50^\circ = 1.928$ , а начальное значение  $vy$  –  $3 \sin 50^\circ = 2.298$ . Кроме указанных переменных нам еще потребуется внутренняя переменная для хранения значения времени предыдущего шага расчета – как и в рассмотренных ранее примерах, назовем ее  $t0$ .

Структура переменных нашего блока будет такой:

| <i>Имя</i> | <i>Tun</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| x          | double     | Выход             |             | 0                         |
| y          | double     | Выход             |             | 0                         |
| vx         | double     | Внутренняя        |             | 1.928                     |
| vy         | double     | Внутренняя        |             | 2.298                     |
| t0         | double     | Внутренняя        |             | 0                         |

Создадим блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95), введем в редакторе модели указанную выше структуру переменных и присоединим блок к динамической переменной “DynTime” (см. стр. 132). Параметры модели по умолчанию изменять не будем – она будет автоматически запускаться при изменении “DynTime” и блокировать реакции при отсутствии этой переменной в схеме. На вкладке “модель” редактора необходимо ввести программу, соответствующую полученным выше разностным уравнениям (за исключением уравнения для  $vx$  – эта переменная не изменяется, и уравнение для нее не нужно). В отличие от примеров из §3.7.4.1 (стр. 145), в этой модели будет три уравнения, а не одно, и, из-за этого, в написании программы для нее возникает особенность, на которую следует обратить внимание.

Кажется логичным просто перенести уравнения для  $x$ ,  $vy$  и  $y$  в модель в виде операторов присваивания следующим образом (будем считать, что в переменной  $g$  записано значение ускорения свободного падения):

```

x=x+(DynTime-t0)*vx; // x[k+1] = x[k] + h vx[k]
vy=vy-(DynTime-t0)*g; // vy[k+1] = vy[k] - h g
y=y+(DynTime-t0)*vy; // y[k+1] = y[k] + h vy[k]

```

Однако, если внимательно посмотреть на вторую и третью строчки этой программы, можно увидеть, что во второй строчке в переменную  $vy$  записывается вычисленное значение  $vy^{k+1}$ , а в третьей это же самое значение в правой части оператора присваивания умножается на шаг расчета  $(DynTime-t0)$  для вычисления  $y^{k+1}$ . Но ведь в правой части третьего уравнения должно находиться  $vy^k$ , а не  $vy^{k+1}$ . Во второй строчке мы потеряли значение  $vy$  на предыдущем шаге расчета, записав в эту же переменную новое значение, и теперь нам неоткуда взять его для правой части следующего уравнения. Получается, что третье уравнение будет “обгонять” остальные на один шаг расчета. Чем меньше шаг расчета, тем слабее будет выражен этот эффект – в такой простой системе мы, вероятнее всего, его даже не заметим. Но, формально, это является ошибкой, поэтому при численном интегрировании системы нескольких разностных уравнений нужно разделять переменные прошлого и следующего шага. Проще всего ввести в программе модели вспомогательные временные переменные и записать все значения  $(k+1)$ -го шага в них, а затем, по окончании расчета, переписать эти значения в соответствующие им статические переменные блока. В этом случае операторы присваивания в модели выглядели бы так:

```

double x_n, y_n, vy_n; // Временные переменные для (k+1)-го шага
// Расчет значений (k+1)-го шага
x_n=x+(DynTime-t0)*vx; // x[k+1] = x[k] + h vx[k]
vy_n=vy-(DynTime-t0)*g; // vy[k+1] = vy[k] - h g
y_n=y+(DynTime-t0)*vy; // y[k+1] = y[k] + h vy[k]
// Запись вычисленных значений в переменные блока
x=x_n;
y=y_n;
vy=vy_n;

```

Здесь мы не теряем значения  $k$ -го шага, поскольку вычисленные для  $(k+1)$ -го записываем в другие переменные.

В нашем случае система уравнений очень проста, и желаемого результата можно добиться просто переставив местами операторы вычисления  $y$  и  $vy$ : при этом мы сначала используем переменную  $vy$ , в которой хранится  $vy^k$ , для вычисления нового значения  $y$ , и только затем запишем в  $vy$  новое значение  $vy^{k+1}$ . Если бы система была сложнее и уравнения были бы сильнее связаны друг с другом, перестановка строк программы ничего бы не дала, и пришлось бы вводить временные переменные, как указано выше.

Таким образом, для расчета по полученным разностным уравнениям на вкладку “модель” можно ввести следующий текст программы:

```

double g=1.62; // Ускорение свободного падения на Луне
double h; // Вспомогательная переменная для шага расчета

h=DynTime-t0; // Шаг расчета
t0=DynTime; // Запоминание текущего времени - на следующем
// шаге оно станет предыдущим

// Вычисления новых значений переменных
x=x+h*vx;
y=y+h*vy;
vy=vy-h*g; // Новое vy вычисляем ПОСЛЕ нового y

```

Значение горизонтальной скорости тела не меняется, поэтому оператора присваивания для  $vx$  в этой модели нет – в течение всего полета тело сохраняет одну и ту же проекцию горизонтальной скорости. У нашего блока нет входов, поэтому мы ничего не сравниваем с признаком ошибки `rdsbcppHugeDouble` (см. стр. 84) – этому признаку просто неоткуда взяться в переменных блока.

Для тестирования созданной модели можно собрать схему, изображенную на рис. 401. Входов у нашего блока нет, а к его выходам присоединен график, отображающий зависимость двух переменных от времени: выход блока  $x$  подан на горизонтальную координату графика, выход  $y$  – на вертикальную. В параметрах блока-планировщика (см. рис. 385 на стр. 131) следует задать шаг расчета 0.01 (наше тело будет лететь довольно быстро, поэтому интервал между дискретными моментами времени должен быть небольшим) и время остановки 3 секунды. Синхронизацию с реальным временем можно включить или выключить по желанию. Если запустить расчет, на графике можно будет увидеть траекторию полета тела. Это парабола, как, согласно теории, и должно быть.

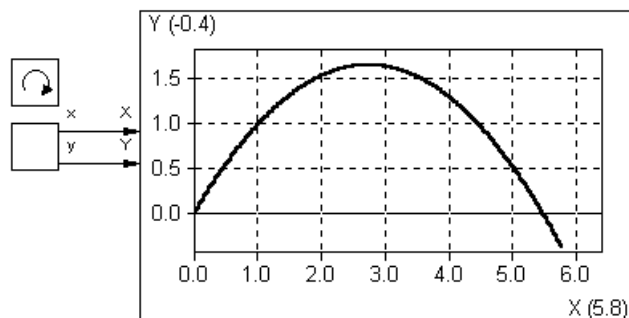


Рис. 401. Тестирование модели свободного полета брошенного тела

В этой модели нам пришлось вручную вычислять проекции начальной скорости тела на оси координат и вводить их в качестве начальных значений в структуру переменных блока – это не очень удобно. К тому же, если мы захотим увидеть траекторию полета тела при другой начальной скорости или другом угле броска, нам придется менять значения внутри структуры переменных и компилировать модель заново. Гораздо лучше сделать модуль начальной скорости  $v_0$  и ее угол к горизонту  $\alpha$  входами блока. Однако, поскольку это – начальные условия, нужно будет написать модель блока таким образом, чтобы значения этих двух входов считывались только в самом начале расчета. Этим мы сейчас и займемся.

Может показаться хорошей идеей считывать модуль и угол начальной скорости со входа блока в реакции модели на событие запуска расчета (`RDS_BFM_STARTCALC`, см. стр. 49). Однако, это – не самый лучший выбор по двум причинам. Во-первых, это событие возникает не только при самом первом, но и при повторном запуске расчета. Если остановить расчет, а затем запустить его, реакция модели на запуск будет вызвана снова. Чтобы отличить первый запуск от продолжения расчета, необходимо анализировать параметры события. Во-вторых, и это более важная причина, событие запуска возникает перед тем, как первый такт расчета будет выполнен. Если мы будем подавать значения на входы блока с полей ввода, все будет в порядке: перед расчетом производится начальная передача данных по связям, и значения с полей ввода поступят на входы. Если же начальные значения вычисляются цепочкой каких-либо соединенных блоков (например, значение угла с поля ввода будет подано на вход блока, переводящего радианы в градусы, а с его выхода – уже на вход нашего блока), эта цепочка не успеет вычислить правильное значение начального условия до того, как оно будет считано моделью с входа блока. Действительно, такты расчета, в которых модели всех блоков цепочки будут выполнять вычисления, начнутся уже после того, как блок среагирует на событие запуска расчета и считывает начальные значения со своих входов. Таким образом, вместо правильных начальных значений будут считаны значения по умолчанию выходов последнего блока цепочки, непосредственно соединенного с нашим.

Если модель динамического блока (то есть блока, вычисления которого связаны со временем) должна получать начальные значения с входов, лучше всего делать это на первом шаге расчета – именно на первом *шаге*, а не на первом *такте*. Шаг расчета – это изменение

динамической переменной “DynTime”, и на каждый шаг обычно приходится несколько тактов расчета. Число тактов на один шаг задается в параметрах блока-планировщика в поле ввода “дополнительные такты” (см. рис. 385 на стр. 131). Кроме того, при первом запуске расчета перед самым первым изменением “DynTime” блок планировщик выполняет так называемые *начальные такты*, то есть пропускает заданное в его настройках число тактов прежде чем в первый раз увеличивать значение времени. Эти начальные такты нужны как раз для того, чтобы цепочки блоков, вычисляющих начальные значения, успели сработать. Следовательно, чтобы считать с входов правильные начальные значения, модель должна дожидаться самого первого изменения переменной “DynTime”. Поскольку величина шага расчета модели заранее не известна, для того, чтобы определить момент первого шага лучше всего завести в переменных блока специальный логический флаг с начальным значением 1. Если “DynTime” изменилась, и при этом этот флаг равен единице, значит, это и есть самый первый шаг расчета – модель должна считать начальные условия и сбросить флаг в ноль, чтобы все последующие шаги не считались первыми.

Именно так мы и поступим. Добавим к структуре переменных нашего блока три новых: вещественные входы *v0* и *Alpha*, на которые будем подавать модуль начальной скорости и угол броска в градусах соответственно, и внутреннюю логическую переменную *Init* с начальным значением 1, которую будем использовать как флаг первого шага расчета. В начальные значения *vx* и *vy* не будем записывать никаких значений, вычисленных вручную – теперь все будет выполняться автоматически. Новая структура переменных блока будет такой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| x          | double     | Выход             |             | 0                         |
| y          | double     | Выход             |             | 0                         |
| vx         | double     | Внутренняя        |             | 0                         |
| vy         | double     | Внутренняя        |             | 0                         |
| t0         | double     | Внутренняя        |             | 0                         |
| v0         | double     | Вход              |             | 0                         |
| Alpha      | double     | Вход              |             | 0                         |
| Init       | Логический | Внутренняя        |             | 1                         |

Модель блока изменим следующим образом:

```
double g=1.62; // Ускорение свободного падения на Луне
double h; // Вспомогательная переменная для шага расчета

if(DynTime==t0) // Время не изменилось – нет нового
 return; // шага расчета

if(Init) // Самый первый шаг расчета (инициализация)
{ if(v0==rdsbcppHugeDouble || Alpha==rdsbcppHugeDouble)
 vx=vy=rdsbcppHugeDouble; // Ошибка на входе
else
{ double alpha_rad=Alpha*M_PI/180; // В радианы
 // Вычисление начальных проекций скорости
 vx=v0*cos(alpha_rad);
```

```

 vy=v0*sin(alpha_rad);
 }
 // Сброс флага инициализации
 Init=0;
}

if(vx==rdsbcppHugeDouble || vy==rdsbcppHugeDouble)
{ x=y=rdsbcppHugeDouble; // На входах была ошибка
 return;
}

h=DynTime-t0; // Шаг расчета
t0=DynTime; // Запоминание текущего времени - на следующем
 // шаге оно станет предыдущим

// Вычисление по разностным уравнениям
x=x+h*vx;
y=y+h*vy;
vy=vy-h*g; // Новое vy вычисляем ПОСЛЕ нового y

```

От предыдущей эта модель отличается тремя добавленными операторами `if`. Если значение `DynTime` (текущее время) равно значению `t0` (времени на момент последнего расчета), то шаг расчета не совершен – модель вызвана в один из дополнительных или начальных тактов. При этом ни чтения начальных условий, ни вычислений делать не нужно, мы немедленно завершаем реакцию оператором `return`. Может показаться, что проверка этого условия – лишняя, ведь наша модель вызывается не при изменении входов и не каждый такт, а при изменении динамических переменных, то есть ее единственной динамической переменной “`DynTime`”, и наша реакция, вроде бы, не должна выполняться без изменения времени. Однако, такая проверка позволит нашей модели правильно работать даже если, например, пользователь ошибочно задаст блоку запуск каждый такт – поскольку этот `if` не сильно усложняет модель, лучше сделать ее устойчивой к таким ошибкам.

Если время изменилось, реакция выполняется дальше, и проверяется значение флага инициализации `Init`. Если оно не нулевое (а по умолчанию мы дали этой переменной значение 1), мы проверяем входы блока на значение ошибки `rdsbcppHugeDouble`, и, если оба они не равны этому значению, значение входа `Alpha` будет переведено в радианы (библиотечные функции `sin` и `cos` требуют аргумента в радианах) и по нему и по значению модуля начальной скорости `v0` будут вычислены исходные значения проекций скорости `vx` и `vy`. Если хотя бы один из входов окажется равным `rdsbcppHugeDouble`, расчет будет невозможен, и мы присваиваем это же значение и `vx`, и `vy`. Затем флаг `Init` будет сброшен, чтобы инициализация не выполнялась на следующих шагах расчета.

После инициализации начинается фрагмент программы, выполняющийся на каждом шаге расчета. В нем мы сначала сравниваем `vx` и `vy` с `rdsbcppHugeDouble` – такое значение могло оказаться в этих переменных, если при инициализации на одном из входов было значение ошибки. В этом случае, мы не можем рассчитывать траекторию, поэтому обоим выходам блока присваивается `rdsbcppHugeDouble` и модель немедленно завершается.

Далее располагается уже знакомый нам фрагмент программы, в котором вычисляется шаг расчета `h` и значения вертикальной скорости и координат на новом шаге – он оставлен без изменений. При данной структуре программы этот фрагмент будет выполнен, только если `vx` и `vy` были успешно инициализированы.

Эта модель будет работать следующим образом. В тактах расчета, в которых время не изменилось, она будет завершаться, ничего не вычисляя. Когда время изменится в первый раз, сработает проверка “`if(Init)...`” и будут вычислены начальные значений

проекций скорости тела, а флаг `Init` будет сброшен, после чего будут вычислены новые значения координат и скорости тела, то есть значения на первом шаге. При всех последующих изменениях времени проверка для присвоения начальных условий срабатывать не будет из-за сброшенного флага, и модель будет просто вычислять значения переменных на новом шаге. Если остановить и снова запустить расчет, ничего не изменится – флаг `Init` останется сброшенным, и модель будет работать так, как будто расчет и не останавливался. Если же сбросить расчет, все переменные блока, включая `Init`, вернуться к исходным значениям, и при следующем запуске проверка “`if (Init)...`” снова сработает и снова присвоит проекциям скорости исходные значения.

Для тестирования созданной модели изменим схему: подключим к входам блока `v0` и `Alpha` поля ввода (рис. 402) и введем в них те же значения, которые использовались при проверке прошлой модели: 3 м/с и 50° соответственно. Параметры блока-планировщика оставим теми же. Запустив расчет, мы увидим ту же самую параболу.

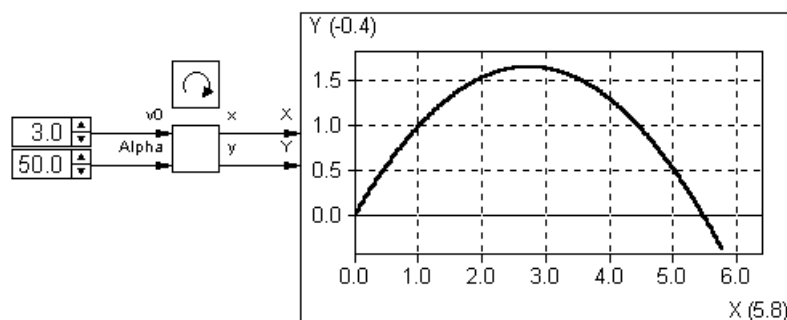


Рис. 402. Модель свободного полета с внешним заданием начальных условий

В состав стандартного модуля автокомпиляции входит универсальный шаблон модели динамического блока, позволяющий несколько быстрее создавать модели, подобные только что описанной. Использование этого шаблона будет рассмотрено в §3.7.4.4 (стр. 168).

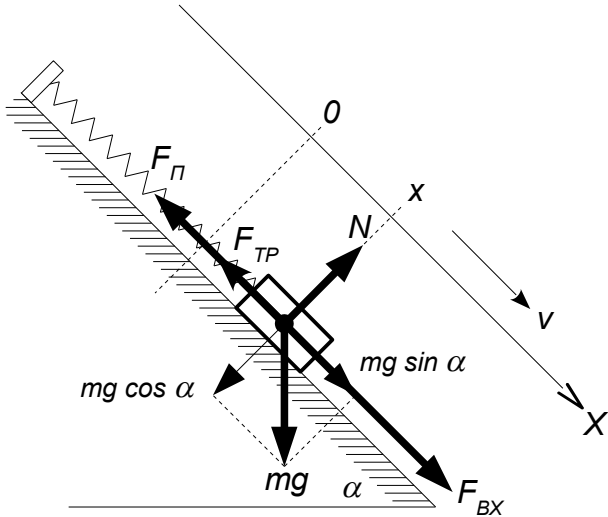
### §3.7.4.3. Важность правильного выбора шага расчета

Описываются проблемы, возникающие при выборе слишком большого шага расчета при численном решении дифференциальных уравнений.

При численном моделировании очень важно правильно выбрать шаг расчета. Независимо от используемого метода численного интегрирования, увеличение шага ведет к тому, что приближенное численное решение будет все сильнее отличаться от точного. Слишком большой шаг может привести не только к ухудшению точности моделирования, но и к качественному изменению поведения системы – полученное при этом решение не будет иметь ничего общего с моделируемым процессом (чаще всего это проявляется в виде расходящихся колебаний моделируемых величин). Существуют различные теоретические методы, позволяющие определить минимально допустимый шаг расчета для данной системы разностных уравнений и данного численного метода, однако проще всего промоделировать систему с разными шагами: изменение поведения системы укажет на слишком большой шаг.

В качестве примера рассмотрим еще одну механическую систему. Имеется груз массой  $m$ , лежащий на наклонной плоскости и прикрепленный к пружине, расположенной параллельно этой плоскости (рис. 403). Массу пружины и сопротивление воздуха учитывать не будем. Мы снова рассматриваем именно механическую систему, поскольку ее поведение представить проще, чем, например, поведение электрической, и правильность этого поведения легко проверить повседневным опытом.

На груз на наклонной плоскости будут действовать следующие силы:



- сила сжатия или растяжения пружины  $F_{\text{П}}$ , направленная вдоль плоскости в сторону положения конца пружины при отсутствии ее деформации;
- сила тяжести  $mg$ , направленная вертикально вниз;
- сила реакции опоры  $N$ , направленная перпендикулярно плоскости (предполагаем, что груз от плоскости не отрывается и всегда движется по ней);
- сила трения  $F_{\text{ТР}}$ , направленная против движения груза, если он движется (трение скольжения), или против приложенной силы, если он покоится (трение покоя);
- заданная внешняя сила  $F_{\text{ВХ}}$ , направленная вдоль плоскости.

Рис. 403. Груз с пружиной на наклонной плоскости

Расположим ось  $x$  вдоль плоскости, направив ее вниз и выбрав началом координат оси точку, в которой деформация пружины равна нулю (то есть точку, в которой будет находиться конец пружины без груза). Большая часть сил, действующих на груз, будет действовать вдоль этой оси. Исключение составят сила реакции опоры  $N$  и сила тяжести  $mg$ . Груз при движении не отрывается от плоскости, поэтому сумма проекций всех сил на ось, перпендикулярную плоскости, должна быть равна нулю, то есть перпендикулярная плоскости составляющая силы тяжести должна быть равна силе реакции опоры. Считая, что плоскость расположена под углом  $\alpha$  к горизонту, получаем

$$N = mg \cos \alpha.$$

Согласно закону Гука, сила пружины  $F_{\text{П}}$  будет пропорциональна величине ее деформации:

$$F_{\text{П}} = -K_{\text{П}}x,$$

где  $K_{\text{П}}$  – коэффициент жесткости пружины. Знак минус в формуле указывает на то, что сила пружины направлена против направления деформации.

Сила трения, препятствующая движению груза, вычисляется по-разному, в зависимости от того, движется груз или покоится. Если груз движется, на него действует сила трения скольжения, направленная против направления движения и пропорциональная силе, прижимающей груз к плоскости:

$$F_{\text{ТР}} \Big|_{v \neq 0} = -\text{sign}(v) K_{\text{ТР.СК}} mg \cos \alpha,$$

где  $v$  – скорость движения груза,  $K_{\text{ТР.СК}}$  – коэффициент трения скольжения, зависящий от трущихся материалов, а  $\text{sign}$  – функция знака, имеющая значение  $-1$  для отрицательных аргументов,  $1$  для положительных, и ноль для нулевого аргумента:

$$\text{sign}(x) = \begin{cases} 1 & , \quad x > 0 \\ 0 & , \quad x = 0 \\ -1 & , \quad x < 0 \end{cases}.$$

Согласно этой формуле, при движении груза сила трения направлена против вектора скорости тела и постоянна по модулю.

Если груз покоится, при приложении к нему сил на него будет действовать сила трения покоя, направленная против суммы проекций остальных приложенных сил на направление возможного движения, и равная ей по модулю, до тех пор, пока эта сумма не превысит максимальную силу трения покоя, вычисляемую по формуле



$$F_{TP.ПОК}^{MAX} = K_{TP.ПОК} N = K_{TP.ПОК} mg \cos \alpha ,$$

где  $N$  – сила реакции опоры (равная по модулю силе, прижимающей груз к плоскости), а  $K_{TP.ПОК}$  – коэффициент трения покоя, который, как и коэффициент трения скольжения, зависит от трущихся материалов. Пока сумма остальных сил меньше этой максимальной силы, груз будет оставаться в покое, поскольку сила трения покоя будет равна этой сумме по модулю и противоположна по направлению, то есть сумма всех сил, действующих на груз, будет равна нулю. Как только сумма остальных сил превысит максимальную силу трения покоя, груз начнет движение, и на него уже будет действовать сила трения скольжения. Таким образом, силу трения покоя можно вычислить по следующей формуле:

$$F_{TP}|_{v=0} = \begin{cases} -F_E & , \quad |F_E| \leq F_{TP.ПОК}^{MAX} \\ -\text{sign}(F_E) F_{TP.ПОК}^{MAX} & , \quad |F_E| > F_{TP.ПОК}^{MAX} \end{cases} ,$$

где  $F_E$  – сумма проекций на ось  $x$  всех сил, приложенных к грузу, кроме силы трения. В момент, когда  $F_E$  превысит максимальную силу трения покоя, сила трения, действующая на груз, скачкообразно изменится с максимальной силы трения покоя на силу трения скольжения, вычисляемую по другой формуле, приведенной несколькими абзацами выше.

Для простоты будем считать, что и груз, и плоскость сделаны из стали и между ними есть слой смазки – в этом случае коэффициент трения покоя можно считать равным коэффициенту трения скольжения (оба они будут равны 0.16). При этом скачок силы трения в момент страгивания груза с места будет отсутствовать, и сила трения скольжения будет равна максимальной силе трения покоя. С этим допущением силу трения, действующую на груз, можно вычислять по следующей формуле:

$$F_{TP} = \begin{cases} -\text{sign}(v) K_{TP} mg \cos \alpha & , \quad v \neq 0 \\ -F_E & , \quad (v=0) \& (|F_E| \leq F_{TP}^{MAX}) \\ -\text{sign}(F_E) F_{TP}^{MAX} & , \quad (v=0) \& (|F_E| > F_{TP}^{MAX}) \end{cases} ,$$

где  $v$  – скорость движения груза,  $K_{TP}$  – коэффициент трения (теперь общий для скольжения и покоя),  $F_E$  – сумма проекций на ось  $x$  всех *остальных* сил, приложенных к грузу, а максимальная сила трения покоя  $F_{TP}^{MAX}$  теперь вычисляется через коэффициент трения  $K_{TP}$ :

$$F_{TP}^{MAX} = K_{TP} mg \cos \alpha .$$

Верхняя строчка в формуле для  $F_{TP}$  относится к движению груза, средняя – к покою, нижняя – к моменту страгивания (перехода от покоя к движению).

В сумму проекций  $F_E$  входят сила пружины  $F_{П} = -K_{П}x$ , внешняя сила  $F_{ВХ}$  и проекция силы тяжести на ось  $x$  ( $mg \sin \alpha$ ):

$$F_E = F_{ВХ} - K_{П}x + mg \sin \alpha$$

Согласно второму закону Ньютона, сумма всех сил, приложенных к грузу, равна произведению его массы на ускорение. Рассматривая движение вдоль оси  $x$ , получаем дифференциальное уравнение второго порядка, описывающее нашу систему:

$$ma = m \frac{dv}{dt} = m \frac{d^2x}{dt^2} = F_E + F_{TP} ,$$

где  $a$  – ускорение груза, а  $v$  – его скорость вдоль оси  $x$ . Преобразуем это уравнение в систему из двух уравнений первого порядка относительно скорости и координаты груза, и добавим начальные условия для этих переменных (без начальных условий мы не сможем получить решение уравнения):

$$\begin{cases} \frac{dv}{dt} = \frac{F_E + F_{TP}}{m} & v(0) = v_0 \\ \frac{dx}{dt} = v & x(0) = x_0 \end{cases} .$$

Применив к этой системе метод Эйлера [3] (см. §3.7.4.1 на стр. 145), получим систему из двух разностных уравнений, позволяющий вычислить  $(k+1)$ -е состояние механической системы по ее  $k$ -му состоянию:

$$\begin{cases} v^{k+1} = v^k + h \frac{F_E^k + F_{TP}^k}{m} \\ x^{k+1} = x^k + h v^k \end{cases}, \text{ где } h = t^{k+1} - t^k - \text{ шаг расчета.}$$

Как и в предыдущих примерах, для удобства записи номер отсчета сделан верхним индексом:  $v^k$  – это не “скорость груза в степени  $k$ ”, это “ $k$ -й отсчет скорости груза”.

Теперь можно создать модель блока, который будет рассчитывать движение груза, но прежде решим, как мы будем определять, покоится груз или движется – это важно для вычисления силы трения. Проверять скорость на точное равенство нулю было бы плохой идеей: любые вычисления выполняются с некоторой погрешностью, поэтому вещественная переменная, в которой мы будем хранить значение скорости, вряд ли когда-нибудь получит в точности нулевое значение в результате вычислений. Вместо точной проверки равенства будем считать, что груз остановился, если модуль его скорости стал меньше очень маленького, заранее заданного, значения  $v_{MIN}$ . В качестве этого значения можно взять, например,  $10^{-6}$  м/с (один микрометр в секунду можно считать практически полной остановкой). Однако, есть еще одна проблема, которую введение такой проверки остановки не решит. Мы считаем в дискретном времени с шагом  $h$ , и, при достаточно большой скорости груза, за один шаг скорость может пройти через ноль, но при этом и значение в

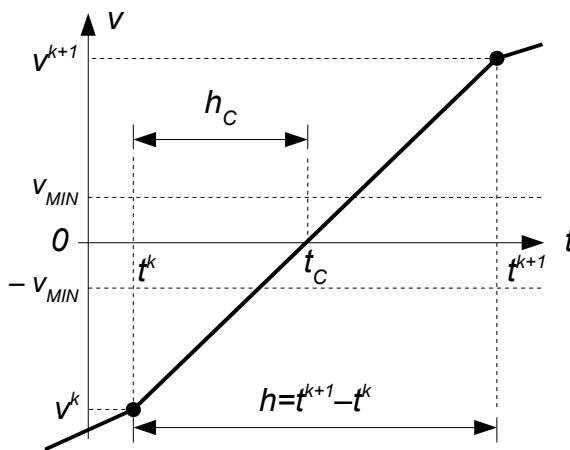


Рис. 404. Определение момента остановки

начале шага, и значение в его конце по модулю будут больше  $v_{MIN}$ . На рис. 404 изображены два отсчета скорости груза:  $v^k$  при времени  $t^k$  и  $v^{k+1}$  при времени  $t^{k+1}$ . Оба значения скорости по модулю больше  $v_{MIN}$ , поэтому проверка остановки не сработает ни в начале, ни в конце такта. При этом где-то внутри шага расчета скорость проходит через ноль, то есть груз останавливается. Если при этом сумма сил  $F_E$  окажется меньше максимальной силы трения покоя  $F_{TP}^{MAX}$ , груз останется в покое (приложенных к нему сил будет недостаточно для преодоления трения покоя), и скорость в момент времени  $t^{k+1}$  будет равна нулю, а не значению  $v^{k+1}$ , которое мы вычислили, считая, что груз будет продолжать двигаться весь шаг расчета.

Уменьшение шага расчета позволит обойти эту проблему: если шаг подобран так, чтобы за него при максимально возможном ускорении скорость груза изменялась бы не более чем на  $v_{MIN}$ , один из отсчетов около момента остановки обязательно попадет в интервал  $-v_{MIN} \dots v_{MIN}$ . Однако, есть более простое решение, не требующее подбора шага: если за шаг расчета скорость изменила свой знак (то есть прошла через ноль), можно проверить, достаточно ли на этом шаге приложенных сил, чтобы преодолеть трение покоя. Если окажется, что этих сил недостаточно, значит, в текущем шаге расчета груз остановится. Можно даже более точно определить время остановки  $t_c$ . На всем шаге расчета мы считаем ускорение постоянным, поэтому скорость будет меняться линейно – это видно из разностного уравнения для скорости:

$$v^{k+1} = v^k + h a^k, \text{ где } h = t^{k+1} - t^k, \quad a^k = \frac{F_E^k + F_{TP}^k}{m}.$$

Груз остановится через промежуток времени  $h_c$  после момента времени  $t^k$ , то есть скорость в этот момент будет равна нулю:

$$0 = v^k + h_c a^k.$$

Таким образом, интервал времени  $h_c$  после  $k$ -го отсчета времени, через который груз остановится (см. рис. 404), и момент времени остановки  $t_c$  вычисляются по формулам

$$h_c = -\frac{v^k}{a^k}, \quad t_c = t^k + h_c.$$

Деление на ускорение в первой формуле не вызовет проблемы: мы вычисляем время остановки только тогда, когда значения  $v^k$  и  $v^{k+1}$  лежат по разные стороны нуля, то есть скорость изменяется, а это значит, что ускорение не равно нулю. Координата груза в момент остановки вычисляется подстановкой  $h_c$  вместо  $h$  в разностное уравнение для  $x$ :

$$x_c = x^k + h_c v^k.$$

В момент времени  $t_c$  мы будем проверять, продолжит ли груз движение, то есть хватит ли приложенных к нему в положении  $x_c$  сил для преодоления трения покоя.

Создадим блок для моделирования поведения этой механической системы. Входами блока будут переменные  $x0$  (начальное положение груза на плоскости, м),  $v0$  (начальная скорость груза, м/с),  $\alpha$  (угол в основании наклонной плоскости, градусов),  $Ktr$  (безразмерный коэффициент трения),  $Kp$  (жесткость пружины, Н/м),  $m$  (масса груза, кг) и  $Fvh$  (внешняя сила, Н). Выходами будут  $x$  (текущее положение груза, м) и  $v$  (текущая скорость груза, м/с). Для вычисления шага расчета нам потребуется внутренняя переменная  $t0$ , в которой мы будем хранить прошлый отсчет времени  $t^k$  (см. примеры в §3.7.4.1 на стр. 145). Кроме того, поскольку мы считываем начальные условия с входов блока  $x0$  и  $v0$ , нам, как и в примере из §3.7.4.2 (см. стр. 156), потребуется внутренний логический флаг инициализации  $Init$ . В результате у нас получится следующая структура переменных блока:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| Kp         | double     | Вход              |             | 0                         |
| Ktr        | double     | Вход              |             | 0                         |
| m          | double     | Вход              |             | 1                         |
| Fvh        | double     | Вход              |             | 0                         |
| Alpha      | double     | Вход              |             | 0                         |
| x0         | double     | Вход              |             | 0                         |
| v0         | double     | Вход              |             | 0                         |
| Init       | Логический | Внутренняя        |             | 1                         |
| x          | double     | Выход             |             | 0                         |
| v          | double     | Выход             |             | 0                         |
| t0         | double     | Внутренняя        |             | 0                         |

Переменной  $m$  мы дали ненулевое начальное значение, чтобы, если мы забудем подключить к ней поле для ввода массы груза, деление на массу при расчете ускорения в программе модели не вызвало ошибки.

Создадим блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95), введем в редакторе модели указанную выше структуру переменных и присоединим блок к динамической переменной “DynTime” (см. стр. 132). Параметры модели по умолчанию изменять не будем – она будет автоматически запускаться при изменении “DynTime” и блокировать реакции при отсутствии этой переменной в схеме. На вкладке “модель” редактора введем следующий текст:

```
double h, Fe, a_rad, Ft, Ft0, vn, a; // Вспомогательные переменные
double vmin=1e-6; // Порог скорости (меньше - остановка)

// Макрос для функции sign
#define sign(x) ((x)<0.0?-1:1)

if(DynTime==t0) // Время не изменилось
 return;

if(Init) // Чтение начальных условий с входов
{ v=v0; x=x0;
 Init=0;
}

h=DynTime-t0; // Шаг расчета
t0=DynTime; // Запоминание времени очередного шага

// Угол наклонной плоскости в радианах
a_rad=Alpha*M_PI/180;

// Максимальная сила трения покоя
// (считаем ее равной силе трения скольжения)
Ft0=Ktr*m*9.8*cos(a_rad);

// Сумма сил кроме силы трения
Fe=m*9.8*sin(a_rad)+Fvh-Kp*x;

// Расчет силы трения скольжения/покоя
if(fabs(v)<vmin) // Покой
{ if(fabs(Fe)<Ft0) // Меньше страгивания
 Ft=-Fe;
 else // Страгивание
 Ft=-sign(Fe)*Ft0;
}
else // Движение
 Ft=-sign(v)*Ft0;

a=(Fe+Ft)/m; // Ускорение на этом шаге
vn=v+h*a; // Скорость в конце шага

// Проверка изменения знака скорости (т.е. остановки)
if(sign(v)!=sign(vn))
{ // Скорость изменила знак внутри шага расчета
 if(fabs(Fe)<Ft0) // Внешние силы меньше тах трения
 { // Полная остановка
 double hs;
 hs=-v*m/a; // Время остановки (после t0)
 x=x+hs*v; // Точка остановки
 v=0; // Остановились - обнуляем скорость
 }
}
```

```

 return;
 }
}
// Скорость не меняла знак или внешние силы больше
// т.е. силы трения – движение продолжается
x=x+h*v; // Новое положение груза
v=vn; // Новая скорость уже вычислена в vn

```

В самом начале этой программы мы вводим несколько вспомогательных переменных, которые будут использоваться далее. Среди них –  $v_{min}$  со значением  $1e-6$  (то есть  $1 \times 10^{-6}$ ), которое мы будем использовать как пороговое значение скорости для определения остановки груза. Затем мы описываем макрос для функции  $sign$ , возвращающей знак аргумента – она потребуется нам для вычисления силы трения и для обнаружения прохода скорости через ноль. Результат вычисления выражения в этом макросе будет равен  $-1$  для отрицательных аргументов и  $1$  для неотрицательных. При равенстве аргумента нулю выражение должно было бы давать нулевой результат, но в рассмотренных нами формулах это не принципиально, поэтому мы упростили вычисление  $sign$ . После описания макроса находятся два уже знакомых нам по модели из §3.7.4.2 (стр. 156) оператора  $if$ , необходимых для считывания начальных условий движения из входов  $x0$  и  $v0$ : первый оператор блокирует выполнение модели, если время не изменилось с прошлого расчета ( $DynTime==t0$ ), второй – переписывает  $x0$  и  $v0$  в  $x$  и  $v$  соответственно, если флаг инициализации  $Init$  взведен (флаг после этого сбрасывается). Далее, как обычно, вычисляется шаг расчета  $h$ , новое значение времени запоминается в переменной  $t0$  для следующего шага, после чего начинается основная часть программы, ответственная за моделирование движения груза.

Прежде всего, угол наклонной плоскости  $Alpha$  переводится из градусов в радианы (тригонометрические функции в стандартных библиотеках языка C работают именно с радианами) и записывается во вспомогательную переменную  $a\_rad$ . Максимальная сила трения покоя  $F_{TP}^{MAX}$  записывается в переменную  $Ft0$ , сумма всех сил вдоль оси  $x$  за исключением силы трения – в переменную  $Fe$ . Затем, в зависимости от состояния груза и приложенных сил, вычисляется сила трения  $Ft$ .

Если абсолютное значение скорости груза  $v$  меньше принятого нами порогового значения  $v_{min}$ , мы считаем, что груз покоится – в этом случае сумма приложенных сил  $Fe$  сравнивается с максимальной силой трения покоя  $Ft0$ . Если абсолютное значение  $Fe$  меньше  $Ft0$ , приложенных сил недостаточно для сдвигания груза с места, и сила трения в этом случае равна  $-Fe$  (сила трения компенсирует приложенные силы и груз остается в покое). Если же  $Fe$  превышает  $Ft0$ , груз начинает движение – сила трения, действующая на него, равна  $Ft0$  по модулю и имеет знак, обратный  $Fe$ .

Если абсолютное значение скорости больше порога  $v_{min}$ , значит, груз в данный момент движется, и на него действует сила трения скольжения, равная по модулю  $Ft0$  (то есть максимальной силе трения покоя, поскольку мы решили считать коэффициенты трения покоя и скольжения равными) и имеющая знак, обратный скорости.

Зная силу трения  $Ft$  и сумму остальных сил вдоль оси  $x$   $Fe$  (все силы рассчитаны на момент начала шага расчета), мы рассчитываем ускорение груза  $a$  (сумма всех сил, деленная на массу) и, по разностному уравнению для скорости, скорость груза на конец шага расчета  $vn$ . Теперь нужно проверить, не остановился ли груз на данном шаге расчета и, если это так, вычислить момент времени остановки согласно рис. 404 (стр. 162).

Если за шаг расчета скорость поменяла знак (знак  $v$  не равен знаку  $vn$ ), значит, скорость груза прошла через ноль. В этом случае сумма сил  $Fe$  сравнивается с максимальной силой трения покоя  $Ft0$ : если трение покоя превышает сумму остальных сил, груз останется стоять. При этом вычисляется интервал времени  $hs$  внутри шага расчета  $h$  до момента остановки. Затем этот интервал подставляется в разностное уравнение для координаты  $x$  и

вычисляется положение груза на момент остановки – поскольку он останется стоять, по крайней мере, до конца шага (и далее, если приложенная сила  $F_{vh}$  на следующем шаге не изменится), это и будет его координатой на момент конца шага. Его скорость на конец шага будет нулевой, поэтому переменной  $v$  присваивается ноль, и модель завершается оператором `return` – до начала следующего шага расчета больше нечего вычислять.

Если же скорость не меняла знак или максимальная сила трения покоя меньше суммы остальных сил, груз будет продолжать движение – в этом случае его новая координата  $x$  вычисляется согласно разностному уравнению, а уже полученная новая скорость переписывается из вспомогательной переменной  $v_n$  в переменную блока  $v$ .

Для тестирования созданной модели соберем схему, изображенную на рис. 405. Внешнюю силу  $F_{vh}$ , начальное положение груза  $x_0$  и его начальную скорость  $v_0$  зададим нулевыми – пусть груз движется из положения покоя в начале координат. Угол наклона плоскости установим в  $45^\circ$ , массу груза – 1 кг, коэффициент трения – 0.16 (сталь по стали со смазкой), жесткость пружины – 5 Н/м (специально возьмем пружину малой жесткости, чтобы смещение груза было побольше). К выходам координаты  $x$  и скорости  $v$  присоединим графики.

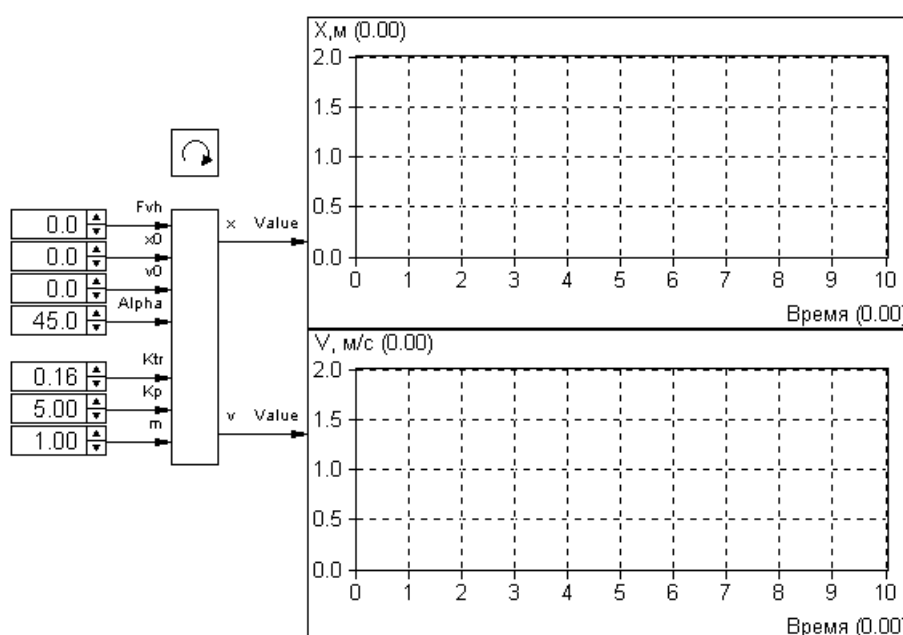


Рис. 405. Схема для тестирования модели движения груза на пружине

Установим в параметрах блока-планировщика (см. рис. 385 на стр. 131) время остановки 10 с и шаг расчета 0.1 с. Запустив моделирование, на графиках мы увидим расходящиеся колебания (рис. 406). Если не останавливать расчет после десяти секунд, их амплитуда будет увеличиваться бесконечно.

Но откуда в такой механической системе без подвода энергии извне (мы обнулили внешнюю силу) могут взяться расходящиеся колебания? В системе есть трение, поэтому, по всем законам физики, груз должен со временем остановиться, а не наращивать амплитуду колебаний. Дело в том, что шаг расчета в одну десятую секунды, установленный в параметрах блока-планировщика, слишком велик для заданных нами параметров груза, плоскости и сил при выбранном методе численного интегрирования. Слишком велики интервалы, на которых мы считаем скорость и ускорение груза постоянными, поэтому моделирование теряет не только точность, но и устойчивость – полученная картина уже не имеет ничего общего с поведением моделируемой системы. В этом легко убедиться, уменьшив шаг расчета в десять раз – при шаге в 0.01 с колебания затухают и груз останавливается (рис. 407).

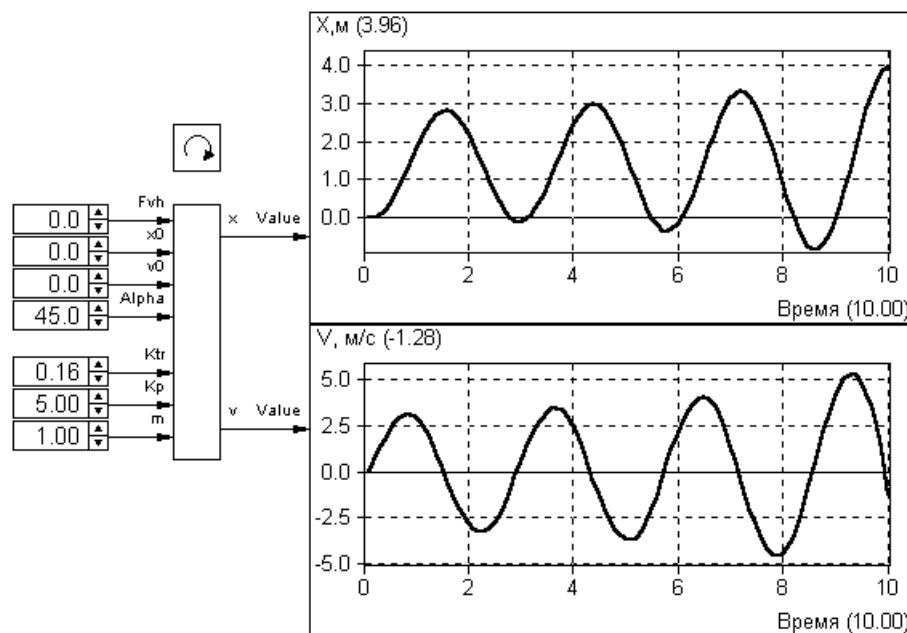


Рис. 406. Расходящиеся колебания при моделировании с шагом расчета 0.1 секунды

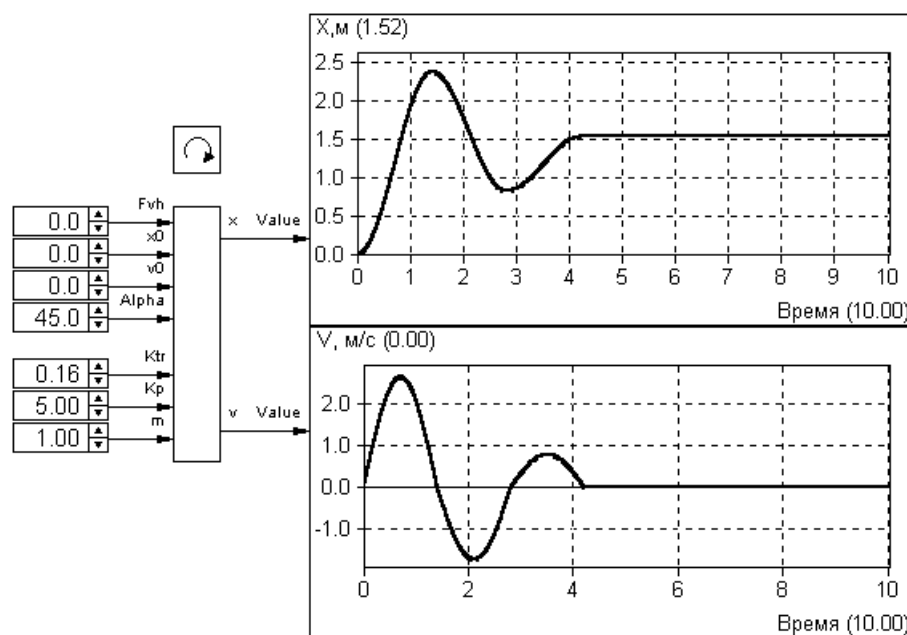


Рис. 407. Моделирование с шагом расчета 0.01 секунды

С таким значением шага можно моделировать движение груза и из других начальных условий, и меняя внешнюю силу  $F_{vh}$ . Например, на рис. 408 начиная с пятой секунды к уже остановившемуся грузу прикладывается постоянная сила в 5 Н – груз снова начинает двигаться и останавливается уже в другом положении.

Следует учитывать, что изменение других параметров системы, входящих в разностные уравнения (угла наклона плоскости, коэффициента трения и т.п.) может потребовать подбора другого шага расчета.

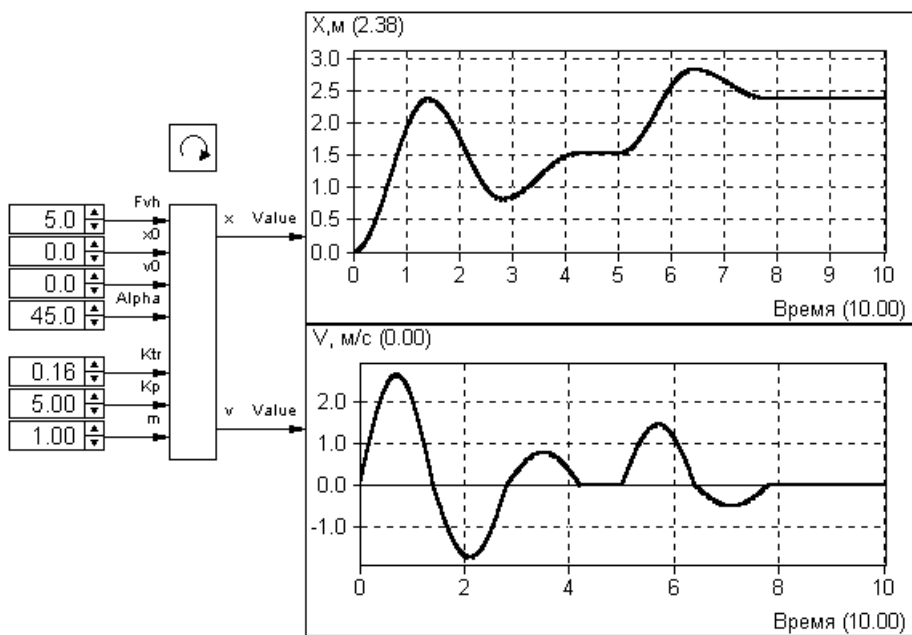


Рис. 408. Моделирование с изменением внешней силы

Рассмотренный, достаточно простой, пример показывает важность правильного выбора шага расчета. Слишком большой шаг приводит к потере устойчивости счета. С другой стороны, слишком маленький шаг приводит к замедлению вычислений. Получив разностные уравнения, теоретически можно определить, как максимально допустимый шаг расчета зависит от параметров системы. Однако, на практике такие вычисления могут оказаться слишком сложными, поэтому можно промоделировать систему несколько раз с разными шагами расчета – при недопустимо большом шаге поведение системы будет существенно другим, и это можно будет заметить по графикам процессов в ней.

#### §3.7.4.4. Создание динамического блока по шаблону

Описывается создание динамического блока при помощи стандартного шаблона модели, входящего в состав РДС. Использование шаблона упрощает написание модели блока, поскольку в этот шаблон уже включена связь с динамической переменной времени “DynTime”, вычисление шага расчета и возможность чтения начальных условий с входов блока.

В §3.7.4.2 (стр. 152) и §3.7.4.3 (стр. 159) мы создавали модели блоков, которые численно рассчитывали процессы в системах, описываемых дифференциальными уравнениями (которые мы переводили в разностные), причем начальные условия этих уравнений подавались на входы блока. В каждой из этих моделей мы вычисляли шаг расчета как разность текущего времени и времени прошлого шага, а для определения правильного момента считывания начальных условий мы использовали специальный логический флаг инициализации. Таким образом, в наших программах моделей динамических блоков содержатся похожие фрагменты, а в структуре статических переменных блока – переменные одинакового назначения. В составе стандартного модуля автокомпиляции есть шаблон динамического блока, в котором эти фрагменты и переменные уже введены, и разработчику остается только добавить свои переменные и дописать внутри модели две подпрограммы: чтения начальных условий с входов и выполнения вычислений для шага расчета. Во многих случаях это удобнее создания модели “с нуля”.

Создание модели по шаблону отличается от создания пустой модели только на одном шаге: после нажатия кнопки “новый” на вкладке “компиляция” окна параметров блока, для которого создается модель (см. §3.3 на стр. 20) в окне “новая модель” нужно вместо флажка



“создать пустую модель” установить флажок “создать модель по шаблону” и выбрать название шаблона в списке под флажком (рис. 317).

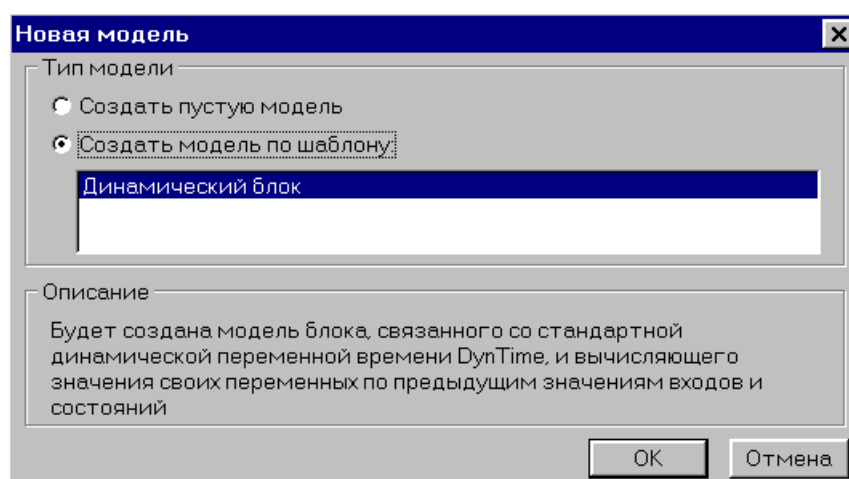


Рис. 317. Выбор создания модели по шаблону (повтор рисунка со стр. 22)

В §3.7.4.2 рассматривалась модель блока, рассчитывающего траекторию тела, брошенного под углом к горизонту при наличии тяготения и отсутствии сопротивления воздуха. В окончательном варианте этой модели (стр. 156) начальная скорость и угол броска считывались с входов блока  $v_0$  и  $\alpha$  соответственно. Создадим точно такую же модель, но с использованием стандартного шаблона. Для этого необходимо выполнить следующие действия:

- нажать на свободном месте окна подсистемы правую кнопку мыши;
- выбрать в открывшемся контекстном меню пункт “создать | новый блок”;
- нажать на созданном блоке (он будет выглядеть как белый квадрат с черной рамкой) правую кнопку мыши;
- выбрать в открывшемся контекстном меню пункт “параметры” (см. рис. 314 на стр. 21);
- на вкладке “общие” открывшегося окна включить флажок “по сигналу” на панели “запуск” (см. рис. 363 на стр. 94);
- на вкладке “компиляция” установить флажок “функция блока компилируется автоматически”, выбрать в выпадающем списке подключенный модуль автокомпиляции и нажать кнопку “новый” (см. рис. 315 на стр. 21);
- в появившемся окне “новая модель” выбрать флажок “создать модель по шаблону”, выбрать в списке шаблон “динамический блок” (см. рис. 317), а затем закрыть окно кнопкой “OK” (это единственное отличие от последовательности действий, приведенной на стр. 95, которая выполняется при создании пустой модели);
- в диалоге сохранения модели ввести имя нового файла, в который будет записана создаваемая модель (см. рис. 318 на стр. 23), и нажать кнопку “сохранить”;
- закрыть окно параметров блока кнопкой “OK”.

После выполнения перечисленных выше действий откроется окно редактора, в котором вместо пустой вкладки “модель”, которая открывалась при создании моделей “с нуля”, будет открыта частично заполненная вкладка “описания в классе” (рис. 409).

На этой вкладке размещается следующий текст:

```
//-----
// ВНИМАНИЕ! Переменные rdsbcppTime0, rdsbcppFirstStep и
// DynTime нужны для работы блока, их не следует изменять
// или удалять.
//-----
```

```
// Функции, заполняемые пользователем
```

```

#pragma argsused
void InitVars(double h)
{
 // Здесь можно считать начальные условия с входов блока,
 // если это необходимо.
 // h - шаг расчета.
}
//-----

#pragma argsused
void DoStep(double h)
{
 // Здесь можно вычислить новые значения выходов блока на
 // момент времени DynTime (интервал времени с прошлого
 // вычисления - h)
}
//-----

```

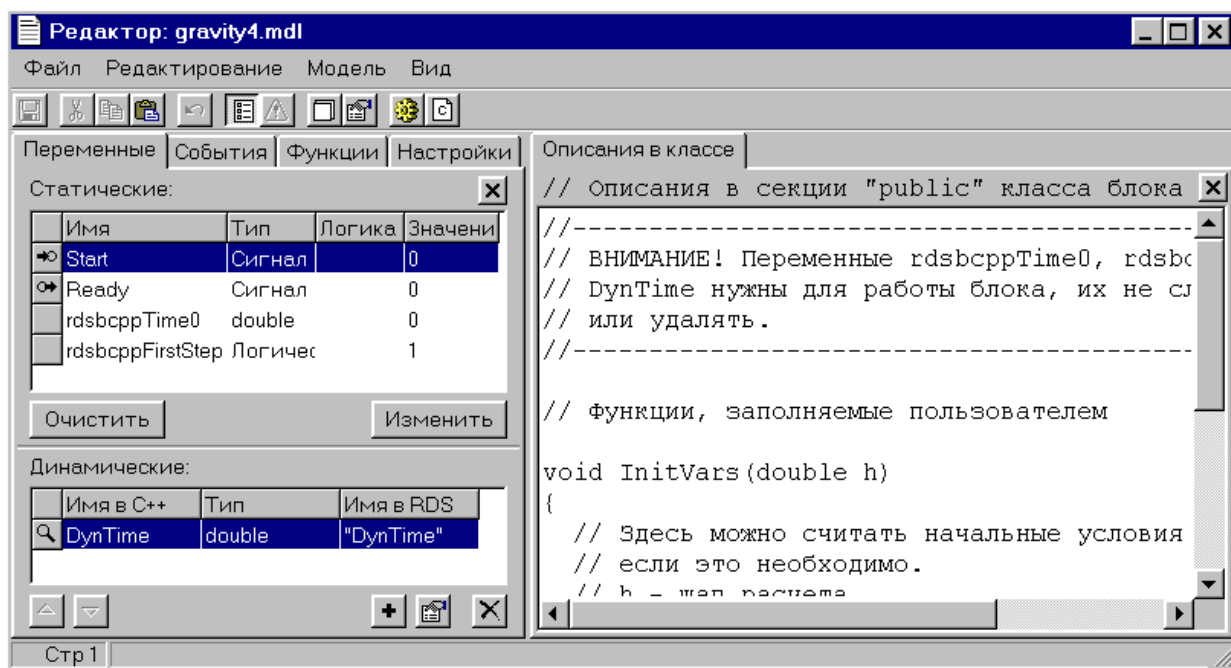


Рис. 409. Окно редактора модели, созданной по стандартному шаблону

В описаниях внутри класса блока в этом шаблоне находятся две пустых функции, внутрь которых пользователь должен вставить свои фрагменты программы: в функцию `InitVars` записывается чтение начальных условий с входов блока, в функцию `DoStep` – вычисления в шаге расчета. В обе этих функции шаг расчета передается в параметре `h`.

К модели блока уже подключена динамическая переменная времени “`DynTime`”, структура статических переменных блока тоже уже частично заполнена. В ней находятся следующие переменные:

| Имя              | Тип        | Вход/выход | Пуск | Начальное значение |
|------------------|------------|------------|------|--------------------|
| Start            | Сигнал     | Вход       | ✓    | 0                  |
| Ready            | Сигнал     | Выход      |      | 0                  |
| rdsbcppTime0     | double     | Внутренняя |      | 0                  |
| rdsbcppFirstStep | Логический | Внутренняя |      | 1                  |

Помимо стандартных сигналов Start и Ready, которые есть у любого простого блока, в эту структуру добавлены две внутренние переменные: вещественная `rdsbcppTime0` и логическая `rdsbcppFirstStep`. В переменной `rdsbcppTime0` хранится значение времени при прошлом срабатывании блока, необходимое для вычисления значения шага расчета – в моделях, которые мы создавали ранее, аналогичную переменную мы называли `t0`. Логическая переменная `rdsbcppFirstStep` с единичным начальным значением используется как флаг инициализации блока – в предыдущих примерах моделей для этих целей мы использовали переменную `Init`.

Эти добавленные переменные необходимы для работы модели, созданной по шаблону динамического блока. Во фрагменте программы на вкладке “описания в классе” они не используются (там есть только комментарий об их необходимости), они используются в реакции блока на такт расчета: хотя вкладка “модель” и не открылась по умолчанию, соответствующий ей фрагмент программы присутствует в модели, и его можно увидеть в редакторе, открыв эту вкладку вручную. Сделаем это: выберем на левой панели редактора вкладку “события” (рис. 410), раскроем на ней группу “моделирование и режимы” и дважды щелкнем на пункте “модель”. В результате этого в правой части окна редактора появится уже знакомая нам вкладка “модель” со следующим текстом:

```
double h;
if(DynTime==rdsbcppTime0) // Время не изменилось
 return;
// Время изменилось
h=DynTime-rdsbcppTime0; // Вычисление шага расчета

if(rdsbcppFirstStep) // Инициализация
{
 InitVars(h);
 rdsbcppFirstStep=0;
}

// Выполнение шага расчета
DoStep(h);

// Запоминание времени последнего расчета
rdsbcppTime0=DynTime;
```

Этот текст уже знаком нам по предыдущим моделям, только раньше мы писали его вручную, а здесь он является частью стандартного шаблона. Как и раньше, сначала текущее время `DynTime` сравнивается с временем прошлого шага расчета `rdsbcppTime0` и, если время не изменилось, модель немедленно завершается. Если время изменилось, вычисляется шаг расчета `h` и проверяется значение переменной `rdsbcppFirstStep`. Если оно равно единице, значит, выполняется самый первый шаг расчета – при этом вызывается функция `InitVars` из описаний в классе, внутрь которой мы должны вставить программу чтения начальных значений, после чего `rdsbcppFirstStep` сбрасывается. Затем вызывается функция `DoStep`, внутрь которой мы вставим вычисления значений переменных блока на новом шаге, и новое время запоминается в переменной `rdsbcppTime0`. При желании, в текст этой программы можно внести какие-либо изменения, но все, что необходимо для расчета, в ней уже есть, поэтому можно вернуться на вкладку “описания в классе” и

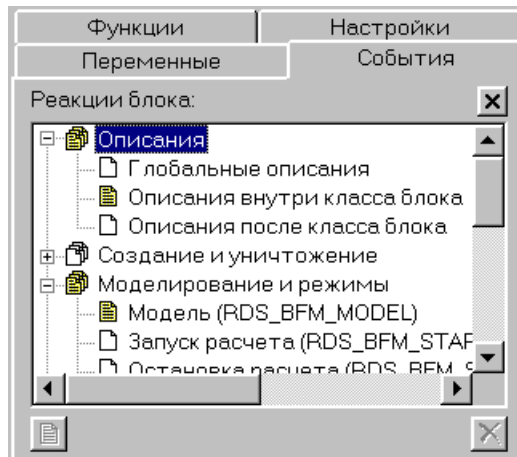


Рис. 410. События и описания модели, созданной по шаблону

наполнить содержимым две ее пустые функции. Но сначала добавим в блок необходимые для нашей модели переменные.

В модели на стр. 156 блок имел входы  $v_0$  и  $\alpha$ , на которые подавались начальная скорость в м/с и угол броска к горизонту в градусах соответственно, выходы  $x$  и  $y$ , на которые выдавались текущие координаты летящего тела, и внутренние переменные  $v_x$  и  $v_y$ , в которых вычислялась его текущая скорость, необходимая для расчета координат. Добавим эти переменные в структуру нашего блока, сохранив в ней те, которые являются частью шаблона. В результате этого наш блок будет иметь следующую структуру переменных:

| <i>Имя</i>       | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------------|------------|-------------------|-------------|---------------------------|
| Start            | Сигнал     | Вход              | ✓           | 0                         |
| Ready            | Сигнал     | Выход             |             | 0                         |
| rdsbcppTime0     | double     | Внутренняя        |             | 0                         |
| rdsbcppFirstStep | Логический | Внутренняя        |             | 1                         |
| $v_0$            | double     | Вход              |             | 0                         |
| $\alpha$         | double     | Вход              |             | 0                         |
| $x$              | double     | Выход             |             | 0                         |
| $y$              | double     | Выход             |             | 0                         |
| $v_x$            | double     | Внутренняя        |             | 0                         |
| $v_y$            | double     | Внутренняя        |             | 0                         |

Внутри функции `InitVars` мы вставим фрагмент старой модели, отвечающий за чтение начальных условий:

```

if (v_0 ==rdsbcppHugeDouble || α ==rdsbcppHugeDouble)
 v_x = v_y =rdsbcppHugeDouble; // Ошибка на входе
else
 { double alpha_rad= α *M_PI/180; // В радианы
 // Вычисление начальных проекций скорости
 v_x = v_0 *cos(alpha_rad);
 v_y = v_0 *sin(alpha_rad);
 }

```

В нем мы вычисляем начальные значения проекций скорости  $v_x$  и  $v_y$  по данным на входах блока  $v_0$  и  $\alpha$ , если оба входа не содержат признака ошибки `rdsbcppHugeDouble`. Параметр функции `h`, в котором передается шаг расчета, для чтения начальных условий нам не нужен (директива препроцессора “`#pragma argsused`”, расположенная перед описанием функции `InitVars`, подавит вывод предупреждения о неиспользуемом параметре).

Внутри функции `DoStep` мы вставим фрагмент, вычислявший в старой модели новые значения  $x$ ,  $y$  и  $v_y$  ( $v_x$  в данной задаче не изменяется со временем, поскольку в горизонтальном направлении на тело не действует никаких сил):

```

double g=1.62; // Ускорение свободного падения на Луне
 x = x + h * v_x ;
 y = y + h * v_y ;
 v_y = v_y - h *g; // Новое v_y вычисляем ПОСЛЕ нового y

```

Таким образом, если убрать поясняющие комментарии, находившиеся внутри функций `InitVars` и `DoStep`, содержимое вкладки “описания в классе” должно теперь выглядеть следующим образом:

```

//-----
// ВНИМАНИЕ! Переменные rdsbcppTime0, rdsbcppFirstStep и

```

```

// DynTime нужны для работы блока, их не следует изменять
// или удалять.
//-----
#pragma argsused
void InitVars(double h)
{
 if(v0==rdsbcppHugeDouble || Alpha==rdsbcppHugeDouble)
 vx=v0=rdsbcppHugeDouble; // Ошибка на входе
 else
 { double alpha_rad=Alpha*M_PI/180; // В радианы
 // Вычисление начальных проекций скорости
 vx=v0*cos(alpha_rad);
 vy=v0*sin(alpha_rad);
 }
}
//-----

#pragma argsused
void DoStep(double h)
{ double g=1.62; // Ускорение свободного падения на Луне
 x=x+h*vx;
 y=y+h*vy;
 vy=vy-h*g; // Новое vy вычисляем ПОСЛЕ нового y
}
//-----

```

Если теперь собрать с нашим новым блоком схему, аналогичную изображенной на рис. 402 (см. стр. 159), и запустить расчет, на графике мы увидим ту же самую параболу: новая модель, созданная по шаблону, будет работать точно так же, как и старая, созданная вручную.

Важно понимать, что после создания по какому-либо шаблону, модель, с точки зрения РДС и модуля автокомпиляции, ничем не будет отличаться от модели, созданной вручную “с нуля”. Никакой связи между моделью и шаблоном не сохраняется, шаблон – это просто некоторое начальное содержимое модели, которое может быть потом как угодно изменено пользователем. В рассмотренной модели, например, можно было бы стереть все описания в классе, а содержимое функций, которое мы туда вставили, разместить непосредственно на вкладке “модель” вместо вызовов этих функций. Можно было бы, при желании, переименовать переменные `rdsbcppTime0` и `rdsbcppFirstStep`, являющиеся частью шаблона (например, дать им имена `t0` и `Init`, как раньше), и в структуре статических переменных блока, и на вкладке “модель”, где они используются. Ни то, ни другое на работоспособность блока не повлияло бы.

Пользователь может в любой момент сохранить любую свою модель в качестве шаблона, выбрав в меню редактора пункт “файл | сохранить как шаблон” (см. §3.6.1 на стр. 32). Редактировать, добавлять и удалять шаблоны можно в настройках модуля автокомпиляции, эти действия рассматриваются в §3.9.2 (стр. 331).

### §3.7.5. Блоки, программно рисующие свое изображение

Рассматривается создание моделей, программно формирующих изображение блока в подсистеме. Такие модели используются для сложных анимированных блоков-индикаторов. Также описывается введение в модель реакции на изменение размера блока, позволяющей корректировать размеры, заданные пользователем, сохраняя пропорции блока неизменными.

В схемах РДС внешний вид любого блока задается либо прямоугольником с произвольным текстом внутри него, либо векторной картинкой, либо рисуется программно моделью блока. Прямоугольники с текстом обычно используются в блоках, внешний вид

которых остается неизменным на протяжении всего расчета – так выглядят все стандартные математические и логические блоки. Векторные картинки чаще всего используются в блоках с анимированными изображениями, поскольку отдельные элементы картинки можно связать с переменными блока, в результате чего в процессе расчета эти элементы будут поворачиваться, перемещаться, менять размер и цвет, появляться или исчезать (векторные картинки подробно рассматриваются в §2.10 части I). Программное рисование тоже используется в блоках, внешний вид которых меняется в процессе расчета, если векторной картинки для его задания недостаточно или она получается слишком сложной и неудобной для редактирования. К таким блокам относятся, например, графики, сложные рукоятки и индикаторы и т.п. Во многих случаях проще написать небольшую программу рисования вместо того, чтобы тщательно подгонять друг к другу подвижные элементы векторной картинки и рассчитывать их координаты и размеры.

Для того, чтобы модель блока программно рисовала его изображение в подсистеме, должны быть выполнены два условия: в окне параметров блока на вкладке “внешний вид” должен быть включен флажок “внешний вид – определяется функцией DLL” (рис. 411), а в модели должна быть введена реакция на событие “рисование блока”. Эта реакция находится на вкладке “события” левой панели окна редактора модели (см. рис. 336 на стр. 46) в группе “внешний вид блока”, и ей соответствует константа RDS\_BFM\_DRAW. Если разрешить программное рисование флажком в окне параметров, но не ввести соответствующую реакцию модели, блок станет невидимым: РДС будет вызывать его модель для рисования блока в окне подсистемы, а модель будет игнорировать этот вызов – в результате, ничего не будет нарисовано. Если ввести реакцию модели на рисование, но в окне параметров блока вместо флажка “внешний вид – определяется функцией DLL” оставить включенным “определяется картинкой” или “прямоугольник с текстом”, РДС не будет вызывать модель для программного рисования, несмотря на то, что в ней есть соответствующая реакция, и блок будет выглядеть согласно установленным флажкам.

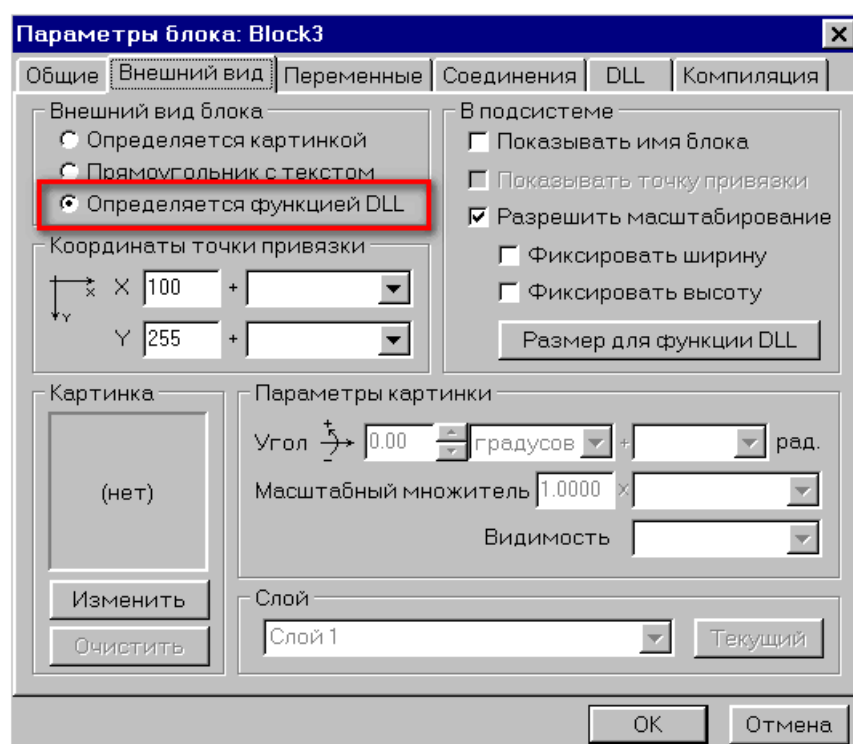


Рис. 411. Флажок, разрешающий программное рисование в окне параметров блока

Как правило, модели блоков, которые рисуют себя программно, пишут таким образом, чтобы изображение подстраивалось под размер блока, заданный пользователем. В этом

случае вместе с флажком “внешний вид – определяется функцией DLL” на вкладке “внешний вид” окна параметров имеет смысл включить и флажок “разрешить масштабирование”, чтобы пользователь мог задать размеры блока, растягивая мышью маркеры его выделения. Можно также, при желании, нажать кнопку “размер для функции DLL” и ввести точный размер блока в точках экрана для масштаба 100%.

В реакции модели блока на рисование можно пользоваться всеми графическими функциями Windows API, а также специальными графическими функциями РДС, описанными в разделе А.5.18 приложения к руководству программиста [2] (все эти функции начинаются с символов “rdsXG”). Графические функции РДС позволяют проще задавать цвет заполнения, толщину линии, шрифт и другие параметры рисования, в остальном же они почти полностью повторяют функции Windows API. Разработчик модели выбирает, какими из них пользоваться, исходя из собственных предпочтений (допускается смешивать в одной модели вызовы и тех, и других, хотя это может запутать не очень опытного программиста). В функцию рисования, создаваемую для реакции, передается один параметр с именем DrawData – это указатель на стандартную структуру РДС RDS\_DRAWDATA, в которой содержатся параметры, необходимые для выполнения рисования. Эта структура рассматривается в разделе А.2.6.3 приложения к руководству программиста, здесь же мы кратко перечислим ее поля и их назначение.

| <i><b>Поле<br/>структуры</b></i> | <i><b>Назначение</b></i>                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HDC dc                           | Контекст устройства Windows, на котором модель рисует изображение. Этот параметр используется только в графических функциях Windows API, в графических функциях РДС его указывать не нужно.                                                                                                                                                                                                                  |
| BOOL<br>CalcMode                 | TRUE, если РДС находится в режимах моделирования или расчета, и FALSE, если РДС находится в режиме редактирования (в режиме редактирования сложные блоки часто рисуют в наиболее общей их форме и со всеми включенными дополнительными элементами, если эти элементы могут исчезать в процессе расчета).                                                                                                     |
| <b>int</b> BlockX,<br>BlockY     | Координаты верхнего левого угла изображения блока в текущем масштабе окна его подсистемы. В этих координатах уже учтена возможная связь положения блока с его переменными, которая может быть задана в окне параметров на панели “координаты точки привязки” (см. рис. 411 выше).                                                                                                                            |
| <b>double</b><br>DoubleZoom      | Текущий масштаб окна родительской подсистемы блока в долях единицы: 1 – 100%, 0.5 – 50%, 2 – 200% и т.п. Если какие-то внутренние элементы программно рисуемого изображения блока (например, шрифт надписи) должны масштабироваться вместе с ним, при рисовании размеры этих элементов необходимо умножать на значение этого поля.                                                                           |
| BOOL<br>RectValid                | Признак изменения размеров прямоугольника блока. Если в результате рисования модель изменяет размер блока (например, если его высота или ширина отражают его внутреннее состояние), в это поле необходимо записать значение TRUE и занести в следующие четыре целых поля (Left, Top, Width и Height) новые координаты и размеры блока. В автокомпилируемых блоках эта возможность используется крайне редко. |

| <i>Поле<br/>структуры</i>                 | <i>Назначение</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>int</b> Left,<br>Top, Width,<br>Height | Координаты левого верхнего угла (Left, Top) прямоугольной области, занимаемой блоком, ее ширина (Width) и высота (Height) в текущем масштабе с учетом возможной связи положения блока с его переменными. Эти значения можно непосредственно использовать в функциях рисования. В эти же поля модель может записать новые координаты и размеры блока, если, по какой-либо причине, она решит изменить их при рисовании (при этом также необходимо записать TRUE в поле RectValid). |
| RECT<br>*VisibleRect                      | Указатель на структуру RECT Windows API, в которой записаны координаты видимой в данный момент в окне части рабочего поля подсистемы. Модель может использовать их для того, чтобы уменьшить потери времени на обновление окна, не рисуя части блока, не попавшие в видимую в данный момент область. Делать это не обязательно – рисование за пределами видимой в окне области отсекается автоматически.                                                                          |
| BOOL<br>FullDraw                          | Логическое поле, указывающее на необходимость полной перерисовки всего изображения блока (TRUE) или только тех его частей, которые изменились с момента последнего рисования (FALSE). Его использование позволяет снизить потери времени на обновление окна, не рисуя не изменившиеся части изображения блока без необходимости. В автокомпилируемых блоках эта возможность используется крайне редко, подробно она рассматривается в §2.10.2 руководства программиста [1].       |

Поскольку параметр функции реакции DrawData – это указатель на данную структуру, все обращения к полям структуры нужно предварять “DrawData->”. Например, чтобы узнать масштаб окна подсистемы рисуемого блока, необходимо записать “DrawData->DoubleZoom”.

Создадим модель простейшего блока-индикатора – индикатора уровня. Блок будет выглядеть как прямоугольник, разделенный по вертикали на две части: верхняя часть будет белой, нижняя – синей, причем высота синей части будет пропорциональна значению входа блока: при нуле на входе весь блок будет белым, при значении входа 100 весь блок будет синим, при значении 50 граница раздела будет находиться точно в середине блока. Таким образом, блок будет рисовать синий вертикальный столбик, высота которого в процентах относительно полной высоты блока равна значению входа (подобный пример рассматривается в §2.10.1 руководства программиста). Для того, чтобы блок лучше выглядел, будем рисовать вокруг него черную рамку, причем в масштабе 100% и меньше рамка будет иметь толщину в одну точку экрана, а в крупных масштабах ее толщина будет увеличиваться согласно выбранному масштабу (если этого не сделать, на печати в высоком разрешении рамка в одну точку будет практически не заметна).

Прежде всего, создадим новый пустой блок, переключим его в режим работы по сигналу и создадим для него новую пустую модель (шаги, которые необходимо для этого выполнить, описаны на стр. 95). После этого откроется окно редактора модели с пустой вкладкой “модель” – на ней мы ничего вводить не будем, поскольку у нашего блока-индикатора не будет реакции на такт расчета. Зададим для блока структуру статических переменных – кроме обязательных сигналов Start и Ready у него будет единственный вещественных вход x:



| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| x          | double     | Вход              |             | 0                         |

Флаг “пуск” у входа x мы не устанавливаем, поскольку нам не нужно запускать модель при изменении этого входа. Модель нашего блока будет вызываться при обновлении окна его подсистемы, в этот момент она и будет считывать текущее значение входа.

Модель блока-индикатора будет состоять из единственной реакции на событие рисования. По умолчанию вкладка для этой реакции не открыта, необходимо открыть ее вручную. Для этого на левой панели окна редактора следует выбрать вкладку “события” (см. §3.6.4 на стр. 46), раскрыть на ней раздел “внешний вид блока” и дважды щелкнуть на его подразделе “рисование блока (RDS\_BFM\_DRAW)” (рис. 412). При этом значок подраздела станет желтым, а в правой части окна появится новая пустая вкладка “рисование”. На этой вкладке мы и запишем текст программы, которая будет рисовать внешний вид индикатора.

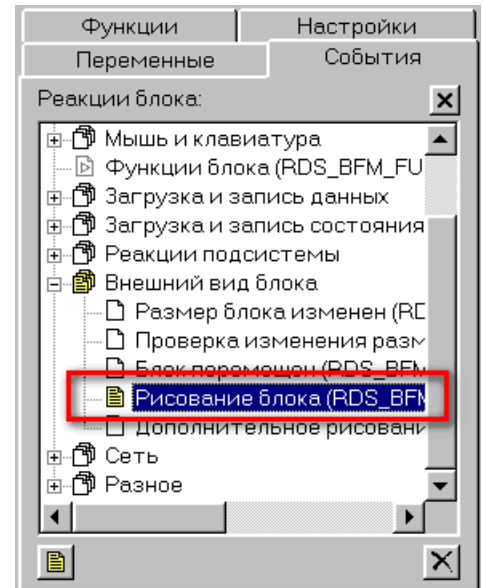


Рис. 412. Реакция на рисование блока в списке событий

Рисовать индикатор мы будем при помощи графических функций РДС (они несколько проще, чем стандартные функции Windows API) в следующей последовательности:

- сначала проверим вход блока x на значение-признак ошибки rdsbcppHugeDouble, и, если они совпадают, нарисуем красный прямоугольник с черной рамкой и завершим реакцию;
- вычислим высоту синей (нижней) части блока в точках экрана согласно значению x;
- нарисуем в верхней части блока белый прямоугольник шириной во весь блок и высотой, равной разности высоты блока и вычисленной высоты синей части;
- в нижней части блока нарисуем синий прямоугольник вычисленной высоты и шириной во весь блок;
- вокруг всего блока нарисуем черную рамку, толщина которой равна текущему масштабу в долях единицы, но не меньше одной точки.

Выполняя эти действия, нам нужно будет проверять значение входа x на попадание в диапазон [0...100]: если x меньше нуля, весь блок будет белым, если больше ста – синим. Мы не будем разрешать границе раздела синей и белой частей выходить за пределы заданного в подсистеме размера блока.

Можно было бы несколько упростить приведенную последовательность действий и нарисовать блок не в три приема (белая часть, синяя часть, рамка), а в два: сначала белый прямоугольник с черной рамкой размером во весь блок, а затем, в его нижней части и поверх него, синий прямоугольник нужной высоты. Однако, в этом случае пришлось бы учитывать утолщение рамки в крупных масштабах и подстраивать размеры синего прямоугольника так, чтобы он не перекрыл собой эту рамку. Описанная же последовательность действий позволяет не заботиться об этом, поскольку рамка будет нарисована самой последней и гарантированно не будет ничем перекрыта.

Для выполнения указанных выше действий на вкладке “рисование” редактора модели нужно ввести следующий текст:

```

// Вспомогательные переменные
int right,bottom,h;

// Правый нижний угол области блока
right=DrawData->Left+DrawData->Width;
bottom=DrawData->Top+DrawData->Height;

// Проверка наличия ошибки на входе
if (x==rdsbcppHugeDouble) // Заливаем красным
{ // Красный фон
 rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xff);
 // Черная линия
 rdsXGSetPenStyle(0,PS_SOLID,
 DrawData->DoubleZoom,0,R2_COPYPEN);
 // Прямоугольник - черная рамка и красный фон
 rdsXGRectangle(DrawData->Left,DrawData->Top,right,bottom);
 // Завершаем реакцию, больше ничего не рисуем
 return;
}

// Высота синего столбика в точках экрана
h=x*DrawData->Height/100.0;
// Ограничение
if (h<0)
 h=0;
else if (h>DrawData->Height)
 h=DrawData->Height;

// Отключаем рамку
rdsXGSetPenStyle(RDS_GFSTYLE,PS_NULL,0,0,0);

// Рисуем верхнюю (белую) часть
if (h!=DrawData->Height) // Есть белое
{ // Белый фон
 rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xffffffff);
 // Верхняя часть - на h меньше высоты блока
 rdsXGRectangle(DrawData->Left,
 DrawData->Top,
 right,
 bottom-h+1);
}

// Рисуем нижнюю (синюю) часть
if (h!=0)
{ // Синий фон
 rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xff0000);
 // Нижняя часть - от h до высоты блока
 rdsXGRectangle(DrawData->Left,
 bottom-h,
 right,
 bottom);
}

// Рисуем рамку
// Черная линия, толщина - с учетом масштаба
rdsXGSetPenStyle(0,PS_SOLID,DrawData->DoubleZoom,0,R2_COPYPEN);
// Отключаем заливку
rdsXGSetBrushStyle(RDS_GFSTYLE,RDS_GFS_EMPTY,0);

```

```
// Прямоугольник рамки
```

```
rdsXGRectangle (DrawData->Left, DrawData->Top, right, bottom);
```

В этой модели мы использовали три графических функции РДС: функцию установки параметров рисуемой линии `rdsXGSetPenStyle`, функцию установки цвета заполнения геометрических фигур `rdsXGSetBrushStyle` и функцию рисования прямоугольника `rdsXGRectangle` (эти функции со всеми их параметрами подробно описаны в §A.5.18.25, §A.5.18.20 и §A.5.18.18 приложения к руководству программиста соответственно). Рассмотрим подробно, что именно делает эта программа.

Сначала мы вычисляем координаты правого нижнего угла прямоугольной области, занимаемой блоком, и записываем их во вспомогательные целые переменные `right` (горизонтальная координата) и `bottom` (вертикальная координата). Координаты левого верхнего угла этой области у нас уже есть: они записаны РДС в поля `Left` и `Top` структуры, указатель на которую передан в нашу функцию реакции в параметре `DrawData`. В полях `Width` и `Height` этой структуры записаны ширина и высота области соответственно, поэтому для вычисления координаты правой границы области (`right`) нужно сложить `DrawData->Left` и `DrawData->Width`, а для вычисления координаты нижней границы (`bottom`) – `DrawData->Top` и `DrawData->Height`. Следует помнить, что рисование ведется в оконных координатах `Windows`, поэтому горизонтальная ось всегда направлена слева направо, а вертикальная – сверху вниз.

Далее мы сравниваем вход блока `x` с признаком ошибки `rdsbcppHugeDouble` (см. стр. 84). Если они равны, мы должны нарисовать красный прямоугольник. Для этого сначала мы устанавливаем красный цвет заливки вызовом

```
rdsXGSetBrushStyle(0, // Маска параметров
 RDS_GFS_SOLID, // Стил
 0xff); // Цвет
```

В первом параметре этой функции указывается маска устанавливаемых параметров (мы передаем ноль, что означает установку всех параметров одновременно), во втором – стиль заливки (константа `RDS_GFS_SOLID` означает сплошную заливку цветом), в третьем – цвет заливки в формате `COLORREF`, используемом в `Windows`. В данном случае, шестнадцатеричное число `0xff` (десятичное 255) означает красный цвет: младший байт числа `COLORREF` задает интенсивность красной компоненты цвета (у нас – максимально возможная), второй и третий байты – интенсивности зеленой и синей компонент соответственно (в числе 255 они нулевые). После этого вызова все рисуемые замкнутые геометрические фигуры будут иметь сплошной красный цвет максимальной интенсивности.

Затем мы устанавливаем параметры линии рамки геометрических фигур вызовом

```
rdsXGSetPenStyle(0, // Маска параметров
 PS_SOLID, // Стил
 DrawData->DoubleZoom, // Толщина
 0, // Цвет
 R2_COPYPEN); // Режим
```

В первом параметре опять передается маска устанавливаемых параметров (0 – все параметры устанавливаются одновременно). Второй параметр задает стиль линии (`PS_SOLID` – сплошная линия), третий – ее толщину (в данном случае передается `DrawData->DoubleZoom`, то есть текущий масштаб подсистемы), в четвертом – цвет линии (переданный ноль означает черный цвет – все три компоненты цвета имеют нулевую интенсивность). Наконец, в пятом параметре передается режим рисования, или “растровая операция”. Этот режим задает способ смешения цвета фона и цвета линии, переданная константа `R2_COPYPEN` указывает на то, что цвет фона должен просто заменяться на заданный цвет линии. После вызова `rdsXGSetPenStyle` с этими параметрами все рисуемые геометрические фигуры будут иметь видимую сплошную рамку черного цвета,

толщина которой будет равна текущему масштабу подсистемы в долях единицы, то есть одной точке при масштабе 100%, двум точкам при масштабе 200% и т.д.

Может возникнуть вопрос: что будет с толщиной линии рамки при масштабах, меньших 100% – например, при 25%? В качестве толщины мы передаем масштаб подсистемы в долях единицы `DrawData->DoubleZoom`, для 25% это значение будет равно 0.25. Третий параметр функции `rdsXGSetPenStyle` имеет тип `int`, то есть 0.25 будет округлено до нуля и передано в функцию – получается, что мы устанавливаем толщину линии в ноль точек. В РДС ноль в качестве толщины линии указывает на минимально возможную толщину, то есть толщину в одну точку экрана. Таким образом, в крупных масштабах толщина рамки нашего прямоугольника будет увеличиваться вместе с масштабом, а в мелких (при значении `DoubleZoom`, меньшем единицы) – оставаться равной одной точке, как мы и хотели.

После того, как параметры заливки и рамки установлены, вызов функции `rdsXGRectangle` рисует прямоугольник с этими параметрами:

```
rdsXGRectangle(DrawData->Left, // Левая граница
 DrawData->Top, // Верхняя граница
 right, // Правая граница
 bottom); // Нижняя граница
```

В параметрах этой функции передаются координаты левого верхнего (`DrawData->Left` и `DrawData->Top`) и правого нижнего (`right`, `bottom`) углов рисуемого прямоугольника. Поскольку это рисование выполняется при ошибке на входе, после вызова `rdsXGRectangle` мы немедленно завершаем реакцию оператором `return`.

Если же на входе не было ошибки, мы вычисляем высоту синей части блока как  $x\%$  от полной высоты блока и записываем ее во вспомогательную переменную `h`:

```
h=x*DrawData->Height/100.0;
```

Чтобы ограничить рисование границами области блока, вместо того, чтобы проверять попадание  $x$  в диапазон  $[0...100]$ , мы проверяем значение `h`. Если значение  $x$  отрицательно, значение `h` тоже будет отрицательным – в этом случае мы принудительно присваиваем `h` ноль, что не даст синей части блока выйти вниз за границы его области. Если значение  $x$  больше ста, значение `h` будет больше высоты блока `DrawData->Height` – в этом случае мы принудительно присваиваем `h` значение высоты блока, что не даст синей части выйти за границу области блока вверх.

Теперь можно рисовать белую и синюю части блока. Сначала мы отключаем рисование рамки вызовом уже знакомой функции `rdsXGSetPenStyle`, но с другими параметрами:

```
rdsXGSetPenStyle(RDS_GFSTYLE, PS_NULL, 0, 0, 0);
```

В первом параметре функции передается битовый флаг `RDS_GFSTYLE`, указывающий на то, что мы меняем только стиль линии. Во втором параметре передается константа стиля `PS_NULL`, задающая невидимую линию. Остальные параметры функции – нулевые: из-за того, что в первом параметре передан единственный флаг установки стиля, они все равно будут проигнорированы РДС, и в них можно указать любые значения.

Если высота синей части `h` не равна полной высоте блока `DrawData->Height`, в верхней части области блока необходимо нарисовать белый прямоугольник. Белый цвет заливки мы устанавливаем вызовом

```
rdsXGSetBrushStyle(0, RDS_GFS_SOLID, 0xffffffff),
```

в котором шестнадцатеричное число `0xffffffff` (десятичное  $16777215 = 255 \times 65536 + 255 \times 256 + 255$ ) задает белый цвет: все три байта этого числа равны 255, то есть красная, зеленая и синяя компоненты цвета имеют максимально возможные интенсивности). Затем мы рисуем прямоугольник, верхняя граница которого совпадает с верхней границей всей области блока, а нижняя отстоит от нижней границы блока на `h`:

```

rdsXGRectangle (DrawData->Left, // Левая граница блока
 DrawData->Top, // Верх блока
 right, // Правая граница блока
 bottom-h+1); // На h выше нижней границы

```

Прямоугольник будет нарисован без рамки, поскольку рамку мы отключили предыдущим вызовом `rdsXGSetPenStyle`. В последнем параметре функции передается “bottom-h+1”, а не “bottom-h”, чтобы не было зазора между верхней и нижней частями прямоугольника. Иногда по результатам рисования приходится увеличивать или уменьшать координаты рисуемых элементов на одну точку, чтобы точно подогнать их друг к другу – это как раз такой случай. Если убрать из параметра “+1”, можно будет видеть, что между белым прямоугольником и синим, который мы нарисуем позже, окажется полоска цвета фона окна подсистемы высотой в одну точку экрана.

Если высота синей части *h* не равна нулю, мы точно так же рисуем в нижней части блока синий прямоугольник. Для этого мы устанавливаем синий цвет заливки вызовом

```

rdsXGSetBrushStyle(0, RDS_GFS_SOLID, 0xff0000),

```

где шестнадцатеричное число `0xff0000` (десятичное `16711680 = 255 x 65536`) задает синий цвет: третий байт числа, указывающий синюю компоненту цвета, равен 255, остальные – нулевые, то есть красная и зеленая компоненты имеют нулевую интенсивность. Затем рисуется нижний прямоугольник блока:

```

rdsXGRectangle (DrawData->Left, // Левая граница блока
 bottom-h, // На h выше нижней границы
 right, // Правая граница блока
 bottom); // Нижняя граница

```

По вертикали нижний прямоугольник начинается там, где закончился верхний: в точке `bottom-h`. Эта точка и будет границей раздела цветов нашего индикатора, перемещающейся вверх и вниз при изменении входа блока.

Теперь, когда оба цветных прямоугольника внутри блока нарисованы, можно нарисовать вокруг него черную рамку. Для этого сначала мы задаем черную сплошную линию рамки, связывая ее толщину с масштабом подсистемы, в которой находится блок:

```

rdsXGSetPenStyle(0, // Маска параметров
 PS_SOLID, // Стиль
 DrawData->DoubleZoom, // Толщина
 0, // Цвет
 R2_COPYPEN); // Режим

```

Точно такой же вызов мы уже рассматривали выше, при описании действий в момент обнаружения на входе блока значения-индикатора ошибки `rdsbcppHugeDouble`. Затем мы отключаем заливку вызовом

```

rdsXGSetBrushStyle(RDS_GFSTYLE, RDS_GFS_EMPTY, 0);

```

Здесь в первом параметре функции мы передаем битовый флаг `RDS_GFSTYLE`, указывающий на то, что мы меняем только стиль заливки, во втором – новый стиль заливки `RDS_GFS_EMPTY`, отключающий ее, и в третьем – ноль, поскольку цвет заливки при передаче флага `RDS_GFSTYLE` мы не устанавливаем (третий параметр функции будет проигнорирован РДС). После этих двух вызовов все рисуемые геометрические фигуры будут иметь черную сплошную рамку и не будут иметь заливки внутри. Теперь мы можем нарисовать рамку блока, то есть прямоугольник размером во всю область, занимаемую блоком:

```

rdsXGRectangle (DrawData->Left, // Левая граница блока
 DrawData->Top, // Верхняя граница блока
 right, // Правая граница блока
 bottom); // Нижняя граница блока

```

Модель нашего блока-индикатора создана – можно скомпилировать ее и закрыть редактор. Однако, внешний вид блока после этого останется таким же, как и после его

создания. Для того, чтобы функция модели смогла самостоятельно рисовать внешний вид блока, мало ввести в нее соответствующую реакцию, нужно еще и разрешить ей делать это в окне параметров блока. Можно было бы сделать это сразу, в момент создания блока, однако, в этот момент у модели еще не было реакции рисования, и блок стал бы невидимым: модель вызывалась бы для рисования и не рисовала бы ничего. Чтобы не искать потом этот невидимый блок в подсистеме, лучше сначала написать, хотя бы частично, реакцию рисования, и только потом разрешить это рисование в параметрах блока.

Откроем окно параметров созданного блока (например, пунктом “параметры” контекстного меню) и выберем в нем вкладку “внешний вид”. На этой вкладке на панели “внешний вид блока” необходимо включить флажок “определяется функцией DLL” (рис. 413). Имеет смысл также включить флажок “разрешить масштабирование”, чтобы пользователь мог задавать размер блока перетаскиванием маркеров его выделения.

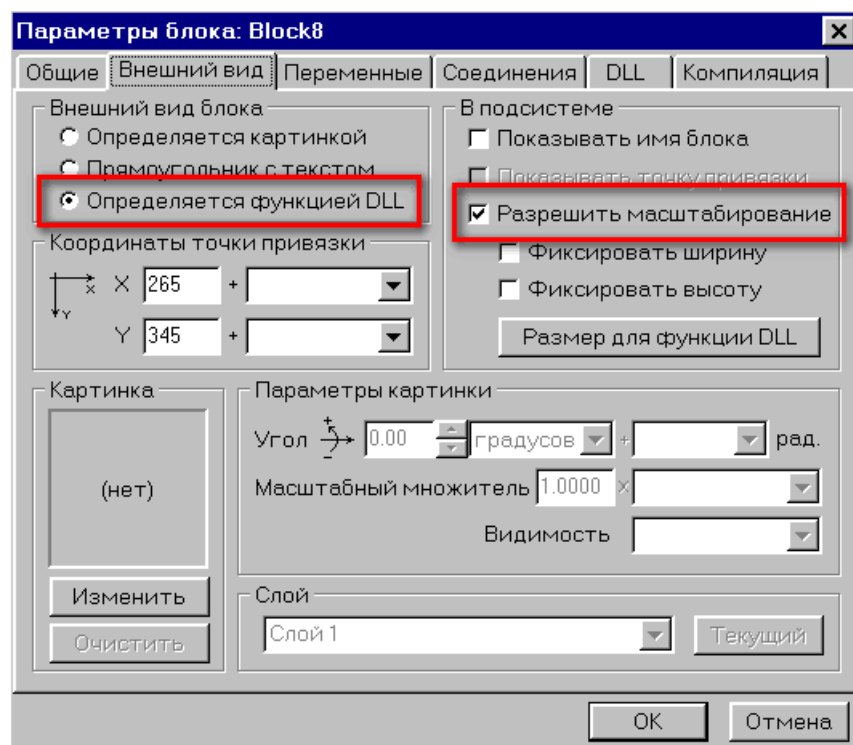


Рис. 413. Включение программного рисования в окне параметров блока

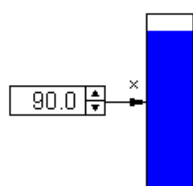


Рис. 414. Тестирование индикатора уровня

После того, как окно будет закрыто кнопкой “ОК”, за рисование нашего блока будет отвечать его собственная модель. Чтобы проверить ее, можно собрать схему, изображенную на рис. 414. В ней блок-индикатор вытянут мышью по вертикали, а к его входу  $x$  подключено поле ввода. Если запустить расчет и вводить в это поле значения от 0 до 100, высота закрашенной части индикатора будет изменяться. При вводе в поле отрицательных значений весь индикатор будет белым, при вводе значений, больших 100 – синим.

Подобный индикатор можно было бы создать и не прибегая к программному рисованию – в редакторе векторной картинки есть элемент “прямоугольник” (см. §2.10.2 части I), вертикальный масштаб которого можно связать с переменной блока. Однако, в этом случае пользователь не смог бы свободно менять размер блока: у векторной картинки нельзя изменить высоту, не изменив ширину, ее можно только увеличить или уменьшить всю целиком.

Рассмотрим теперь модель более сложного блока, изображение которого не удалось бы создать при помощи векторной картинки. Сделаем индикатор с круглой шкалой и стрелкой (рис. 415), причем начало и конец шкалы, шаг чисел на этой шкале, число знаков в дробной части этих чисел и количество дополнительных делений между числами на шкале будут входами блока, чтобы их можно было изменять в процессе расчета.

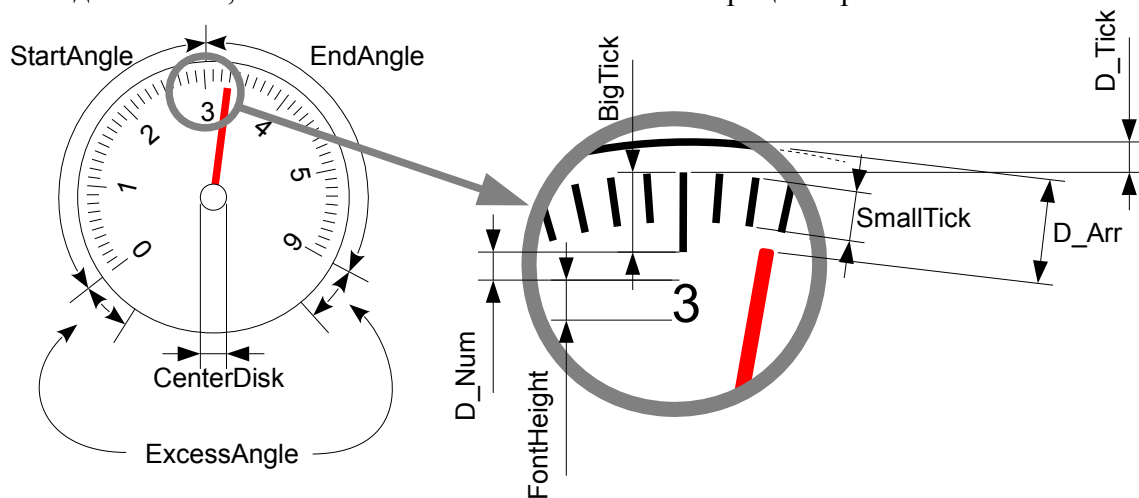


Рис. 415. Предполагаемый вид и параметры стрелочного индикатора

Таким образом, мы дадим нашему блоку следующие входы:

- начало шкалы – вещественный вход Min;
- конец шкалы – вещественный вход Max;
- шаг чисел шкалы – вещественный вход Step;
- количество знаков в дробной части чисел шкалы – целый вход Dec;
- число дополнительных делений (мелких рисок) между числами шкалы – целый вход Sub;
- индицируемое значение (положение стрелки) – вещественный вход x.

На рисунке индикатор изображен для значений Min=0, Max=6, Step=1, Dec=0, Sub=10 и x=3.28.

Кроме указанных выше, для рисования индикатора нам потребуются дополнительные параметры, задающие размер различных его элементов: рисок шкалы, стрелки, чисел и т.п. (все эти параметры изображены на рис. 415). Вместо того, чтобы делать их входами блока, мы просто опишем их как локальные переменные в модели. Это приведет к тому, что все индикаторы с этой моделью будут иметь одинаковый размер рисок и чисел шкалы. При желании, можно сделать эти параметры настраиваемыми пользователем (см. §3.7.6 на стр. 194), но для упрощения примера мы не будем этого делать.

Внешний вид нашего индикатора будет определяться следующими дополнительными параметрами (все угловые величины будем задавать в радианах, все линейные – в точках экрана для масштаба 100%):

- StartAngle – угол между вертикалью и началом шкалы;
- EndAngle – угол между вертикалью и концом шкалы;
- ExcessAngle – дополнительный угол, на который мы разрешим стрелке отклоняться наружу от диапазона шкалы (это позволит пользователю увидеть, что значение на входе блока вышло за отображаемый диапазон);
- FontHeight – высота шрифта чисел шкалы;
- BigTick – размер больших рисок шкалы, около которых будут выводиться числа;
- SmallTick – размер маленьких рисок шкалы, располагающихся между большими;
- D\_Tick – расстояние между внешним краем прибора и рисками;
- D\_Num – расстояние между внутренним концом большой риски и числом;

- D\_Arr – расстояние между внешним краем прибора и стрелкой;
- CenterDisk – диаметр центрального круга прибора.

Независимо от заданных пользователем размеров блока, мы всегда будем рисовать прибор круглым в центре прямоугольной области, занимаемой блоком: диаметр круга будет равен наименьшему из двух размеров блока. При изменении масштаба подсистемы мы будем соответствующим образом подстраивать линейные размеры всех элементов блока, включая размер шрифта чисел, а также толщину всех линий.

Теперь, когда мы определились с внешним видом и параметрами индикатора, можно приступать к созданию модели. Создадим новый пустой блок, переключим его в режим работы по сигналу и создадим для него новую пустую модель (см. стр. 95). Зададим для блока следующую структуру статических переменных:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| Min        | double     | Вход              |             | 0                         |
| Max        | double     | Вход              |             | 100                       |
| Step       | double     | Вход              |             | 20                        |
| Dec        | int        | Вход              |             | 0                         |
| Sub        | int        | Вход              |             | 10                        |
| x          | double     | Вход              |             | 0                         |

Мы снова не устанавливаем флаг “пуск” у входов – наш блок не будет ничего вычислять в такте расчета, и нам не нужно запускать его модель при изменении входов. Вкладка реакции на рисование блока по умолчанию закрыта – откроем ее, как было описано в предыдущем примере: на левой панели окна редактора выберем вкладку “события”, раскроем раздел “внешний вид блока” и дважды щелкнем на подразделе “рисование блока” (см. стр. 177). На открывшейся вкладке “рисование” введем следующий текст:

```
//----- Параметры рисования -----
// Углы начала и конца шкалы от направления "вверх"
double StartAngle=0.66*M_PI, // Начало шкалы (+120гр)
 EndAngle=-0.66*M_PI; // Конец шкалы (-120гр)
double ExcessAngle=0.055*M_PI; // По краям шкалы (10гр)
// Размеры цифр и рисок в точках экрана для масштаба 100%
int FontHeight=17; // Высота шрифта
int BigTick=10; // Размер большой риски
int SmallTick=4; // Размер маленькой риски
int D_Tick=5; // Зазор между внешней границей и большой риской
int D_Num=2; // Зазор между числом и большой риской
int D_Arr=2; // Зазор между внешней границей и стрелкой
int CenterDisk=10; // Диаметр центрального диска
//-----

// Макрос для расчета угла от направления "вверх"
// по значению входа
#define V_TO_A(v) \
 (((v)-Min)*(EndAngle-StartAngle)/(Max-Min)+StartAngle)
```



```

// Макрос для расчета координат точки по радиусу и углу
#define CALCPPOINT(r,a,x,y) \
 x=CenterX-(r)*sin(a); \
 y=CenterY-(r)*cos(a);

// Вспомогательные переменные
int CenterX,CenterY,left,top,right,bottom,R,r1,r2,
 ix1,iy1,ix2,iy2,rf,w,rs,ra;
double a,v,e_min,e_max;
char *str;
BOOL x_error,scale_error;

// Допустимы ли значения шкалы?
scale_error=(Min==rdsbcppHugeDouble ||
 Max==rdsbcppHugeDouble ||
 Step==rdsbcppHugeDouble ||
 Max<=Min || Step<=0.0);
// Допустимо ли значение на входе?
x_error=(x==rdsbcppHugeDouble);

// Центр изображения блока
CenterX=DrawData->Left+DrawData->Width/2;
CenterY=DrawData->Top+DrawData->Height/2;

// Выделяем центральный квадрат блока
if(DrawData->Width>DrawData->Height)
{ // Блок вытянут по ширине
 R=DrawData->Height/2;
 left=CenterX-R;
 right=CenterX+R;
 top=DrawData->Top;
 bottom=DrawData->Top+DrawData->Height;
}
else
{ // Блок вытянут по высоте
 R=DrawData->Width/2;
 left=DrawData->Left;
 right=DrawData->Left+DrawData->Width;
 top=CenterY-R;
 bottom=CenterY+R;
}
// R - радиус большого круга

// Рисуем белый (красный при ошибке) круг с черной рамкой
rdsXGSetPenStyle(0,PS_SOLID,DrawData->DoubleZoom,0,R2_COPYPEN);
rdsXGSetBrushStyle(0,RDS_GFS_SOLID,
 (scale_error||x_error)?0xff:0xffffffff);
rdsXGEllipse(left,top,right,bottom);

if(scale_error) // Без шкалы ничего больше не нарисовать
 return;

// Расширенный диапазон (на ExcessAngle больше шкалы)
a=ExcessAngle*(Max-Min)/fabs(EndAngle-StartAngle);
e_min=Min-a;
e_max=Max+a;

```

```

// Отключаем заливку
rdsXGSetBrushStyle(RDS_GFSTYLE,RDS_GFS_EMPTY,0);

// Вычисляем вспомогательные радиусы
r1=R-D_Tick*DrawData->DoubleZoom;// Внешний радиус большой риски
r2=r1-BigTick*DrawData->DoubleZoom;// Внутр. радиус большой риски
rs=r1-SmallTick*DrawData->DoubleZoom; // Внутренний радиус
 // маленькой риски
rf=r2-D_Num*DrawData->DoubleZoom;// Внешний радиус чисел
ra=R-D_Arr*DrawData->DoubleZoom;// Радиус стрелки

// Рисуем шкалу в цикле
for (v=Min;v<=Max+Step*0.001;v+=Step)
{ // Угол, соответствующий v
 a=V_TO_A(v);
 // Расчет точек большой риски
 CALCPOINT(r1,a,ix1,iy1)
 CALCPOINT(r2,a,ix2,iy2)
 // Рисование большой риски
 rdsXGMoveTo(ix1,iy1);
 rdsXGLineTo(ix2,iy2);
 // Выводимое число (строку нужно потом освободить rdsFree)
 str=rdsDtoA(v,Dec,NULL);
 // Не повернутый шрифт
 rdsXGSetFont(RDS_GFFONTALLHEIGHT,"Arial",
 FontHeight*DrawData->DoubleZoom,
 0,DEFAULT_CHARSET,
 0,
 FALSE,FALSE,FALSE,FALSE);
 // Ширина числа
 rdsXGGetTextSize(str,&w,NULL);
 // Повернутый шрифт
 rdsXGSetFont(RDS_GFFONTALLHEIGHT,"Arial",
 FontHeight*DrawData->DoubleZoom,
 0,DEFAULT_CHARSET,
 a*180.0/M_PI,
 FALSE,FALSE,FALSE,FALSE);
 // Координаты центра верхней линии шрифта
 CALCPOINT(rf,a,ix1,iy1)
 // Вычисление левого верхнего угла текста числа
 ix2=ix1-0.5*w*cos(a);
 iy2=iy1+0.5*w*sin(a);
 // Вывод числа
 rdsXGTextOut(ix2,iy2,str);
 // Освобождение памяти, отведенной в rdsDtoA
 rdsFree(str);
 // Рисование маленьких рисок
 for(int i=1;i<Sub;i++)
 { a=v+i*Step/Sub; // Значение риски
 if(a>Max) break;
 a=V_TO_A(a); // Угол риски
 CALCPOINT(r1,a,ix1,iy1)
 CALCPOINT(rs,a,ix2,iy2)
 rdsXGMoveTo(ix1,iy1);
 rdsXGLineTo(ix2,iy2);
 } // for(int i=1;...)
} // for(v=Min;...)

```

```

// Рисуем стрелку
if(!x_error)
{ // Устанавливаем красный цвет линии
 rdsXGSetPenStyle(0,PS_SOLID,
 DrawData->DoubleZoom,0xff,R2_COPYPEN);
 // Ограничиваем по расширенному диапазону шкалы
 if(x<e_min)
 v=e_min;
 else if(x>e_max)
 v=e_max;
 else
 v=x;
 // Вычисляем координаты конца стрелки
 CALCPOINT(ra,V_TO_A(v),ix1,iy1)
 // Рисуем линию стрелки
 rdsXGMoveTo(CenterX,CenterY);
 rdsXGLineTo(ix1,iy1);
}

// Рисуем центральный диск
w=0.5*CenterDisk*DrawData->DoubleZoom;
rdsXGSetPenStyle(0,PS_SOLID,DrawData->DoubleZoom,0,R2_COPYPEN);
rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xffffffff);
rdsXGEllipse(CenterX-w,CenterY-w,
 CenterX+w,CenterY+w);

```

Эта модель тоже пользуется для рисования графическими функциями РДС. Помимо уже рассмотренных ранее функций `rdsXGSetPenStyle` и `rdsXGSetBrushStyle`, в ней вызываются функции рисования эллипса, линии, функция установки шрифта и функция определения размеров заданного текста в точках экрана. Кроме того, в ней используется пара функций работы со строками, при помощи которых формируются текстовые строки с числами, выводимыми на шкале. Все эти функции подробно описаны в приложении к руководству программиста.

В самом начале модели описаны переменные, соответствующие параметрам внешнего вида индикатора, изображенным на рис. 415 (стр. 183). Переменным `StartAngle` и `EndAngle` мы присваиваем значения  $0.66\pi$  и  $-0.66\pi$  радиан соответственно – шкала нашего прибора будет начинаться в точке, отстоящей от вертикали на  $120^\circ$  против часовой стрелки, а заканчиваться – в точке, отстоящей от вертикали на  $120^\circ$  по часовой стрелке (значение `StartAngle` положительно, а `EndAngle` – отрицательно, потому что положительные углы во всех математических функциях откладываются против часовой стрелки). Переменной `ExcessAngle` присвоено значение  $0.055\pi$  радиан: разрешенный выход стрелки за шкалу будет составлять примерно  $10^\circ$ . Далее задаются размеры в точках экрана (в масштабе подсистемы 100%) для всех элементов изображения.

Сразу после параметров рисования описаны два макроса, которые будут использоваться для пересчета значения на входе блока в угловое положение точки на шкале и ее экранные координаты. Можно было бы сделать эти макросы обычными функциями, но, поскольку они не будут использоваться нигде за пределами реакции модели на рисование блока, а для написания функций пришлось бы открывать новую вкладку глобальных описаний, ввести макросы будет проще.

Макрос `V_TO_A(v)` с единственным параметром `v` вычисляет угол относительно вертикали, который на шкале прибора соответствует отображаемому значению `v`. В макросе записано обычное линейное преобразование диапазона `[Min...Max]` в диапазон `[StartAngle...EndAngle]`. Макрос `CALCPOINT(r,a,x,y)` записывает в параметры `x` и

у горизонтальную и вертикальную координаты точки, находящейся под углом  $\alpha$  к вертикали на окружности радиуса  $r$ , центр которой совпадает с центром прибора (в макросе используются координаты центра прибора `CenterX` и `CenterY`, которые будут вычислены позже). С помощью этого макроса будут вычисляться и координаты концов всех рисок, и координаты конца стрелки. Сразу за макросами описывается некоторое количество вспомогательных переменных, которые будут использованы в программе.

Первое действие, которое мы делаем в программе модели, это проверка допустимости значений на входах блока. Если хотя бы один из входов `Max`, `Min` или `Step`, описывающих шкалу, будет равен признаку ошибки `rdsbcppHugeDouble` (см. стр. 86), или конец шкалы `Max` будет меньше или равен началу шкалы `Min`, или шаг шкалы `Step` будет равен нулю или отрицателен, логической (BOOL) переменной `scale_error` будет присвоено значение TRUE. Эту переменную мы будем использовать как признак ошибки шкалы – при такой ошибке мы не можем ни нарисовать шкалу на приборе, ни, очевидно, отобразить на нем значение входа. Если вход блока  $x$  равен признаку ошибки, значение TRUE получит переменная `x_error` – при такой ошибке мы можем нарисовать шкалу, но не сможем нарисовать стрелку.

Записав признаки ошибок в логические переменные (они потребуются нам позже, при рисовании), мы вычисляем `CenterX` и `CenterY` – координаты центра прямоугольной области, занимаемой блоком, которая будет также центром нашего круглого прибора. Для этого мы используем уже знакомые по прошлому примеру поля `Left`, `Top`, `Width` и `Height` из структуры описания события рисования `RDS_DRAWDATA` (см. стр. 175), указатель `DrawData` на которую передается в нашу функцию реакции. Горизонтальная координата центра – это левая граница плюс половина ширины блока, а вертикальная – верхняя граница плюс половина его высоты. Напомним, что в структуре `RDS_DRAWDATA` все поля заполнены уже с учетом текущего масштаба подсистемы, поэтому здесь нам не нужно умножать координаты на масштабный коэффициент (поле `DoubleZoom` той же структуры) самостоятельно.

Определив координаты центра прибора, мы вычисляем радиус  $R$  его круга и координаты `left`, `top`, `right` и `bottom` квадрата в центральной части блока, в который будет вписан этот круг. Для этого мы сравниваем ширину блока `DrawData->Width` с его высотой `DrawData->Height`. Если ширина блока больше высоты (то есть блок вытянут по горизонтали), верхняя граница `top` и нижняя граница `bottom` его центрального квадрата будут совпадать с границами всего блока, радиус круга будет равен половине высоты, а левая `left` и правая `right` границы квадрата будут отстоять от центра на  $R$ . Если же высота блока больше ширины (блок вытянут по вертикали), `left` и `right` будут совпадать с левой и правой границами блока, радиус круга  $R$  будет равен половине ширины, а `top` и `bottom` будут на  $R$  отстоять от центра прямоугольника.

Теперь мы можем нарисовать большой круг прибора, поверх которого будут выводиться шкала и стрелка. Сделаем этот круг красным, если на входах блока есть какие-либо ошибки (то есть либо `x_error`, либо `scale_error` имеют значение TRUE), а его рамку сделаем черной и сплошной, и будем, как и у рассмотренного ранее индикатора уровня, увеличивать ее толщину вместе с масштабом. Сначала уже рассматривавшейся ранее графической функцией РДС `rdsXGSetPenStyle` мы устанавливаем параметры рамки (в качестве толщины рамки передается текущий масштабный множитель подсистемы `DrawData->DoubleZoom`), затем функцией `rdsXGSetBrushStyle` устанавливаются параметры заливки (в качестве цвета заливки передается результат выполнения условного оператора “?:” проверяющего признаки ошибок, и возвращающего либо красный цвет `0xff`, либо белый `0xffffffff`). После этого вызывается функция рисования эллипса `rdsXGEllipse`, очень похожая на рассматривавшуюся ранее функцию рисования прямоугольника `rdsXGRectangle`:

```

rdsXGEllipse(left, // Левая граница
 top, // Верхняя граница
 right, // Правая граница
 bottom); // Нижняя граница

```

Поскольку left, top, right и bottom – границы центрального квадрата блока, эта функция нарисует эллипс с осями одинаковой длины, то есть круг.

Нарисовав круг прибора, мы проверяем значение признака ошибки шкалы `scale_error`. Если оно истинно, мы немедленно завершаем модель: без параметров шкалы мы ничего больше не сможем нарисовать. Таким образом, если параметры шкалы заданы неверно, наш прибор будет выглядеть просто как красный круг с черной рамкой.

Если же параметры шкалы прошли все проверки, мы продолжаем выполнение модели и вычисляем расширенный диапазон шкалы с учетом добавочного разрешенного угла отклонения стрелки `ExcessAngle`. Сначала в переменную `a` записывается интервал значений, соответствующий на шкале углу `ExcessAngle` (это линейное преобразование диапазона `[StartAngle...EndAngle]` в диапазон `[Min...Max]`, обратное выполняемому макросом `V_TO_A`, но без знака и смещений). Вычитая этот интервал из `Min` и добавляя его к `Max`, мы получаем границы расширенного на дополнительный угол диапазона входных значений `e_min` и `e_max` соответственно. В этих пределах входное значение будет преобразовываться в угол отклонения стрелки.

Далее мы отключаем заливку геометрических фигур вызовом `rdsXGSetBrushStyle` (если не сделать этого, числа на шкале будут выведены в закрашенных прямоугольниках) и вычисляем радиусы концентрических окружностей, на которых располагаются различные элементы шкалы и конец стрелки:

- `r1` – радиус окружности, на которой лежат внешние концы всех рисок;
- `r2` – радиус окружности, на которой лежат внутренние концы больших рисок;
- `rs` – радиус окружности, на которой лежат внутренние концы маленьких рисок;
- `rf` – радиус окружности, на которой лежат внешние границы чисел шкалы;
- `ra` – радиус окружности, на которой лежит конец стрелки.

Все эти радиусы вычисляются согласно изображенным на рис. 415 (стр. 183) параметрам рисования, но при этом эти параметры умножаются на `DrawData->DoubleZoom` для учета текущего масштаба подсистемы. Например, согласно рисунку, в масштабе 100% внешняя граница рисок отстоит от края прибора на `D_Tick` точек экрана. Масштаб 100% соответствует `DrawData->DoubleZoom=1`. Значит, в произвольном масштабе риски будут отстоять от края прибора на произведение `D_Tick` и `DrawData->DoubleZoom` – радиус окружности `r1` будет короче радиуса всего прибора `R` на эту величину.

Теперь можно нарисовать шкалу. Мы будем рисовать ее в цикле по вещественной переменной `v`, изменяющейся от `Min` до `Max` с шагом `Step`:

```

for (v=Min; v<=Max+Step*0.001; v+=Step)

```

Здесь в качестве конца цикла выбрано не `Max`, а значение, большее `Max` на одну тысячную шага изменения: если диапазон кратен шагу, из-за погрешностей вычисления последнее значение `v` может оказаться чуть больше `Max`, и при обычной проверке `v<=Max` самая последняя большая риска шкалы могла бы пропасть. Например, при `Min=0`, `Max=100`, `Step=20` на шкале должно быть выведено шесть больших рисок с числами: 0, 20, 40, 60, 80 и 100. Однако, после пятого прибавления `Step` (20) к `v` может получиться не 100, а 100.00000001. При этом, поскольку это число больше `Max`, риска не была бы нарисована. Добавление к концу шкалы небольшой доли шага позволяет справиться с этой проблемой.

Внутри цикла мы при помощи макроса `V_TO_A` вычисляем угол `a`, который на нашей шкале соответствует величине `v`, и, по этому углу и радиусам `r1` и `r2`, макросами `CALCPOINT` вычисляем координаты концов большой риски, соответствующей значению `v`. Эти координаты записываются в пары переменных `(ix1, iy1)` и `(ix2, iy2)`. В РДС нет

графической функции, позволяющей одним вызовом нарисовать линию между двумя точками, поэтому, чтобы нарисовать риску, приходится делать два вызова: сначала мы задаем координаты начала линии, вызвав функцию `rdsXGMoveTo(ix1, iy1)`, а затем вызываем функцию `rdsXGLineTo(ix2, iy2)`, которая нарисует линию из этой точки в точку `(ix2, iy2)`. Эти функции описываются в §A.5.18.14 и §A.5.18.13 приложения к руководству программиста соответственно. Линия будет нарисована с параметрами, заданными последним вызовом `rdsXGSetPenStyle`, то есть риска шкалы будет черной сплошной линией с толщиной, равной масштабу подсистемы.

Теперь рядом с этой риской нужно вывести число `v`, причем это число должно быть повернуто на угол риски `a`, как на рис. 415 (стр. 183). Для этого придется последовательно выполнить следующие действия:

- преобразовать число `v` в текстовый вид, оставив в его дробной части `Dec` знаков;
- установить шрифт для вывода текста – пока без поворота (будем использовать шрифт “Arial”, высота которого будет равна произведению параметра `FontHeight` на масштаб подсистемы);
- вычислить ширину числа в точках экрана, которую оно займет, если вывести его текущим установленным шрифтом;
- установить шрифт, повернутый на угол риски шкалы `a`;
- вычислить координаты левой верхней точки выводимого текста с учетом его поворота;
- вывести текст.

Для того, чтобы преобразовать вещественное число `v` в строку, можно, например, использовать стандартную функцию `sprintf`, как в примере из §3.7.2.5 (стр. 112). Однако, ее использование требует добавления в раздел глобальных описаний команды для включения файла “`stdio.h`”, а для этого придется открыть еще одну вкладку в редакторе модели. Чтобы не делать этого, воспользуемся сервисной функцией РДС `rdsDtoA` (см. §A.5.4.5 приложения к руководству программиста), которая формирует в динамической памяти строку с текстовым представлением числа с заданным количеством знаков в дробной части:

```
str=rdsDtoA(v, // Преобразуемое число
 Dec, // Знаков в дробной части
 NULL); // Через этот указатель функция может вернуть
 // длину созданной строки
```

В первом параметре функции передается преобразуемое число (`v`), во втором – необходимое число знаков после десятичной точки (`Dec`), в третьем можно было бы передать указатель на целую переменную, в которую функция запишет длину созданной ей строки, но нам эта длина не нужна, поэтому в третьем параметре мы передаем `NULL`. Функция возвращает указатель (`char*`) на созданную строку, который мы записываем во вспомогательную переменную `str`. Потом, когда эта строка уже не будет нам нужна, ее нужно будет обязательно удалить при помощи функции `rdsFree` (см. пункт A.5.4.8 приложения к руководству программиста).

Получив текст, который мы будем выводить рядом с риской шкалы, мы устанавливаем шрифт для вывода текста функцией `rdsXGSetFont`, описанной в пункте A.5.18.22 приложения к руководству программиста:

```
rdsXGSetFont(RDS_GFFONTALLHEIGHT, // Маска установки
 "Arial", // Имя шрифта
 FontHeight*DrawData->DoubleZoom, // Размер шрифта
 0, // Цвет шрифта
 DEFAULT_CHARSET, // Набор символов
 0, // Угол поворота (гр)
 FALSE, // Жирность
 FALSE, // Курсив
 FALSE, // Подчеркивание
 FALSE); // Перечеркивание
```

В первом параметре этой функции передается набор битовых флагов, указывающий на то, какие именно параметры шрифта мы устанавливаем. Константа `RDS_GFFONTALLHEIGHT` объединяет флаги всех параметров, причем размер шрифта при ее использовании задается в точках экрана, а не в типографских точках. Второй параметр функции задает имя шрифта (у нас – “Arial”), третий – размер (высоту `FontHeight` для масштаба 100% мы умножаем на масштаб подсистемы `DrawData->DoubleZoom`), четвертый – цвет символов (0 означает черный). В пятом параметре указывается используемый набор символов шрифта: мы будем выводить только цифры, десятичную точку и знак минуса, поэтому в этом параметре указана стандартная константа Windows API для набора по умолчанию – `DEFAULT_CHARSET` (если бы, например, нам нужно было вывести текст на русском языке, необходимо было бы передать в нем `RUSSIAN_CHARSET`). В шестом параметре передается угол поворота шрифта в градусах – мы передаем ноль, поскольку мы его еще не поворачиваем, нам пока предстоит определить ширину не повернутого текста в точках экрана. Наконец, в последних четырех параметрах передаются логические признаки жирности, курсива, подчеркивания и перечеркивания шрифта.

Шрифт установлен, теперь мы вызываем функцию `rdsXGGetTextSize` (см. §A.5.18.10 приложения к руководству программиста), которая вернет нам ширину строки текста `str`, выведенной текущим шрифтом, в точках экрана:

```
rdsXGGetTextSize(str, // Текст
 &w, // Сюда запишется ширина
 NULL); // Сюда могла бы записаться высота
```

В первом параметре функции передается указатель на строку, ширину которой мы хотим узнать, во втором – указатель на целую переменную, в которую функция запишет ширину строки, в третьем можно было бы передать указатель на переменную для высоты строки, но высота нас не интересует, поэтому мы передаем в нем `NULL`.

Теперь можно установить повернутый шрифт. Для этого мы снова вызываем `rdsXGSetFont` с теми же параметрами, только в качестве угла поворота теперь передается не ноль, а  $a \cdot 180.0 / M\_PI$  (то есть угол `a`, переведенный в градусы). Затем, при помощи макроса `CALCPOINT`, мы вычисляем точку, находящуюся на одной линии с рисккой, но на расстоянии `rf` от центра круга, и записываем ее координаты в переменные (`ix1`, `iy1`). Чтобы число на шкале располагалось симметрично относительно рискки, на эту точку должна приходиться середина верхней границы прямоугольной области, занимаемой его текстом. Поскольку в функциях вывода текста место вывода задается левым верхним углом первого символа строки, необходимо вычислить этот левый верхний угол. Если бы текст не был повернут, левый верхний угол первого символа отстоял бы от (`ix1`, `iy1`) влево на половину ширины строки, то есть на `w/2`. Чтобы учесть поворот строки на угол `a`, необходимо умножить `w/2` на синус (для вертикальной координаты) и косинус (для горизонтальной) этого угла. Таким образом, левый верхний угол первого символа повернутого текста будет находиться в точке (`ix2`, `iy2`):

```
ix2=ix1-0.5*w*cos(a);
iy2=iy1+0.5*w*sin(a);
```

Теперь можно выводить число. Для этого используется функция `rdsXGTextOut` (см. §A.5.18.27 приложения к руководству программиста), в которую передаются вычисленные координаты `ix2` и `iy2` и выводимая строка `str`. После вызова этой функции строка с числом нам уже не нужна, и мы освобождаем ее память вызовом `rdsFree`.

Большая риска с числом нарисована – теперь необходимо вывести после нее мелкие риски. В одном большом интервале шкалы содержится `Sub` маленьких интервалов, то есть необходимо вывести `Sub-1` мелких рисков (в начале самого первого маленького интервала уже выведена большая риска). Мы выводим их в цикле по целой переменной `i`, изменяющейся от 1 до `Sub-1`. Внутри цикла мы вычисляем значение, соответствующее `i`-й мелкой рискке, по формуле  $v + i \cdot \text{Step} / \text{Sub}$ : `v` – это значение большой рискки шкалы в начале

интервала, а Step/Sub – шаг мелких рисок. Если значение получилось больше верхнего предела шкалы Max, мы досрочно прерываем цикл, поскольку дальше рисковать не нужно. В противном случае мы вычисляем угол мелкой риски при помощи макроса V\_TO\_A, а затем макросами CALCPOINT рассчитываем координаты концов риски, лежащих на радиусах r1 и rs, после чего рисуем риску вызовами rdsXGMoveTo и rdsXGLineTo.

После того, как вся шкала нарисована, можно нарисовать стрелку, если, конечно, вход блока x не содержит признака ошибки rdsbcppHugeDouble (в этом случае переменная x\_error будет иметь значение TRUE). Для рисования стрелки мы меняем цвет линии на красный (0xff) вызовом

```
rdsXGSetPenStyle(RDS_GFCOLOR, 0, 0, 0xff, 0)
```

(константа RDS\_GFCOLOR в первом параметре указывает на то, что мы меняем только цвет линии, а ее толщина, стиль и режим вывода остаются неизменными). После этого мы, как и для рисок шкалы, при помощи макросов V\_TO\_A и CALCPOINT вычисляем координаты конца стрелки, соответствующие ее положению для значения на входе блока x, и рисуем линию из центра прибора в эту точку. Нарисовав стрелку, мы снова устанавливаем черный цвет линии, включаем сплошную белую заливку и рисуем в центре прибора круг, радиус которого вычисляется как половина параметра CenterDisk (диаметр этого круга для масштаба 100%), умноженная на текущий масштаб подсистемы блока.

Теперь можно скомпилировать модель и закрыть редактор. В параметрах внешнего вида блока следует включить флажки “внешний вид – определяется функцией DLL” и “разрешить масштабирование” (см. рис. 413 на стр. 182), после чего можно будет растянуть маркеры выделения блока, задав ему любой желаемый размер.

Для проверки работы модели можно собрать схему, изображенную на рис. 416. В ней к входам x и Max индикатора подключены поля ввода. Если запустить расчет и изменять значение x, стрелка прибора будет двигаться. Если изменять значение Max, будет меняться диапазон шкалы, а стрелка будет оставаться на одной и той же риске.

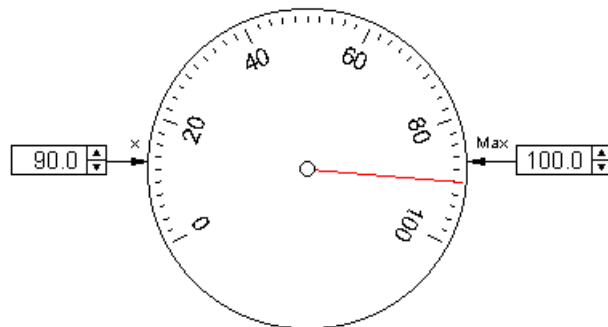


Рис. 416. Тестирование модели стрелочного индикатора

Стрелочный индикатор, похожий на созданный, можно было бы нарисовать и при помощи векторной картинки блока, но при этом его шкала была бы фиксированной, на ней не могли бы появляться новые риски и числа при изменении входов Min, Max и Step. В нашем же блоке шкала формируется программно и может быть какой угодно.

В моделях, рисующих внешний вид блока программно, часто вводят еще и реакцию на изменение размеров блока – при этом можно пересчитывать какие-либо параметры рисования, или каким-либо образом вмешиваться в изменение размеров. Созданный нами стрелочный индикатор, например, всегда имеет форму круга, поэтому было бы логично не давать пользователю вытягивать его по горизонтали и вертикали. Сделаем так, чтобы, как бы ни перетаскивал пользователь маркеры выделения нашего блока, его область всегда оставалась квадратной.

Моделью блока поддерживаются две реакции на события, связанные с изменениями размеров блока: реакция на перетаскивание маркеров выделения пользователем, и реакция



на окончание изменения размеров (подробно они описаны в приложении к руководству программиста в §A.2.7.7 и §A.2.7.6 соответственно, а также в §2.12.8 самого руководства программиста). Первое событие возникает при каждом движении мыши в процессе перетаскивания маркеров, второе – когда пользователь отпускает кнопку мыши по окончании перетаскивания, или когда размер блока задан явно вводом ширины и высоты в окне его параметров. Для упрощения примера мы введем в нашу модель только реакцию на первое событие, поскольку оно, как и второе, позволяет скорректировать размер блока так, как нам надо, и дает пользователю визуальную обратную связь: если модель вмещается в изменение размеров блока, это немедленно отразится на размере прямоугольника, который пользователь видит при перетаскивании меток масштабирования.

Откроем вкладку события перетаскивания маркеров выделения: на левой панели окна редактора выберем вкладку “события” (см. §3.6.4 на стр. 46), раскроем на ней раздел “внешний вид блока” и дважды щелкнем на его подразделе “проверка изменения размера (RDS\_BFM\_RESIZING)” (рис. 417). При этом значок подраздела станет желтым, а в правой части окна появится новая пустая вкладка “проверка размера”.

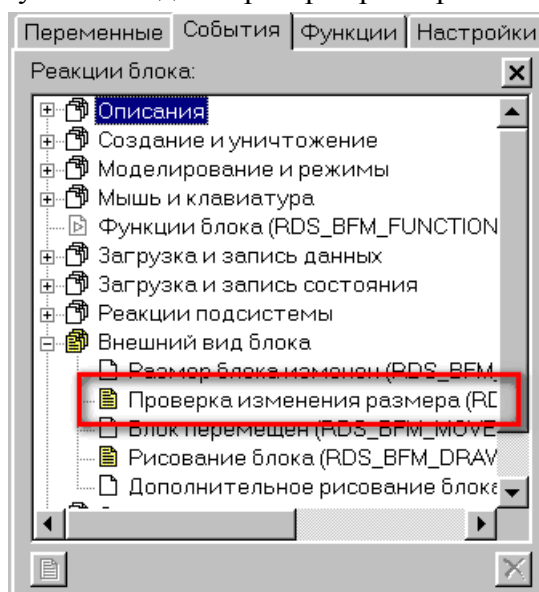


Рис. 417. Реакция на изменение размеров блока в списке событий

Введем на этой вкладке следующий текст (точно такая же реакция используется в примере из §2.12.8 руководства программиста):

```
if(ResizeData->HorzResize && (!ResizeData->VertResize))
 // Перетаскивается левый или правый маркер
 ResizeData->newHeight=ResizeData->newWidth;
else if((!ResizeData->HorzResize) && ResizeData->VertResize)
 // Перетаскивается верхний или нижний маркер
 ResizeData->newWidth=ResizeData->newHeight;
else // Перетаскивается угловой маркер
{ // Вычисляем среднее арифметическое размеров
 int avg=(ResizeData->newWidth+ResizeData->newHeight)/2;
 // Присваиваем его и ширине и высоте
 ResizeData->newWidth=ResizeData->newHeight=avg;
}
```

В эту реакцию передается указатель ResizeData на структуру RDS\_RESIZEDATA, которая описывает выполняемое изменение размеров блока. Она подробно описывается в §A.2.7.6 приложения к руководству программиста, а нас в этой структуре будут интересовать только четыре поля: целые newWidth и newHeight, содержащие новые значения ширины

и высоты блока, и логические `HorzResize` и `VertResize`, указывающие на то, какие маркеры выделения перетаскивает пользователь. Различные сочетания значений `HorzResize` и `VertResize` соответствует следующим ситуациям:

| <i><b>HorzResize</b></i> | <i><b>VertResize</b></i> | <i><b>Способ изменения</b></i>                                 |
|--------------------------|--------------------------|----------------------------------------------------------------|
| 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 выделения |

На момент вызова нашей реакции в полях структуры `ResizeData->newWidth` и `ResizeData->newHeight` находятся ширина и высота блока, которые попытался установить пользователь. Если перетаскивается левый или правый маркер, значит, пользователь пытается изменить ширину блока, и мы копируем `newWidth` в `newHeight`, чтобы высота блока оставалась равной ширине. Если перетаскивается верхний или нижний маркер, значит, пользователь меняет высоту блока, и мы, наоборот, копируем `newHeight` в `newWidth`, делая ширину блока равной высоте. Если же пользователь перетаскивает один из угловых маркеров, значит, он пытается одновременно изменить и ширину, и высоту. В этом случае мы присваиваем `newWidth` и `newHeight` среднее арифметическое заданных пользователем размеров – при этом поведение блока будет выглядеть более-менее естественным.

Теперь, если скомпилировать модель, область, занимаемая блоком, при перетаскивании маркеров выделения будет все время оставаться квадратной, и круглый индикатор, нарисованный внутри нее, будет заполнять ее полностью. Пользователь все еще может сделать область блока прямоугольной, войдя в окно его параметров, выбрав там вкладку “внешний вид”, нажав на панели “в подсистеме” кнопку “размер для функции DLL” (см. рис. 413 на стр. 182) и введя произвольные размеры. Работу индикатора это не нарушит – его модель написана так, чтобы он всегда оставался круглым. Можно, при желании, заблокировать и такое изменение размеров, просто скопировав введенную нами реакцию в реакцию на событие “размер блока изменен”, которое находится в том же разделе “внешний вид блока” (см. рис. 417 на стр. 193) на первом месте. В эту реакцию передается точно такой же указатель `ResizeData` на структуру `RDS_RESIZEDATA`, поэтому в тексте программы ничего не нужно изменять.

### §3.7.6. Блоки с настраиваемыми пользователем параметрами

Рассматривается добавление в модель блока окна настроек, позволяющего пользователю задавать различные индивидуальные параметры блока. Разные блоки с одной и той же моделью могут иметь разные значения настроечных параметров.

Чтобы сделать ввод различных параметров, управляющих работой блока, более удобным, модуль автокомпиляции позволяет достаточно просто создавать окна настроек, в которых пользователь в режиме редактирования может вводить или выбирать из списков значения этих параметров. Интерфейс создания таких окон подробно описан в §3.6.6 (стр. 60). Рассмотрим несколько простых примеров блоков с настройками.

Сначала создадим блок, умножающий вещественный вход  $x$  на константу  $K$  и выдающий полученное произведение на выход  $y$ , причем  $K$  сделаем параметром блока, который пользователь сможет вводить в окне настройки, открываемом по двойному щелчку на блоке. Создадим в схеме новый блок, запускающийся по сигналу (см. стр. 95), и зададим ему следующую структуру статических переменных:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 1                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| x          | double     | Вход              | ✓           | 0                         |
| y          | double     | Выход             |             | 0                         |

Модель блока будет запускаться при самом первом запуске расчета (у сигнала Start единичное начальное значение) и при поступлении нового значения на вход x (у этого входа установлен флажок “пуск”). “К” будет настроечным параметром, поэтому в структуру переменных блока переменная с таким именем не входит. На вкладку “модель”, то есть в реакцию на такт расчета, введем единственный оператор:

$$y=K \cdot x;$$

Осталось добавить в список настроечных параметров переменную K и создать для нее поле ввода.

Выберем в левой части окна редактора вкладку “настройки” (см. рис. 341 на стр. 61) – в данный момент и список параметров, и список полей ввода окна настройки на ней пусты. Нажмем в верхней части вкладки (в списке параметров) кнопку со знаком “+” и заполним открывшееся окно согласно рис. 418.

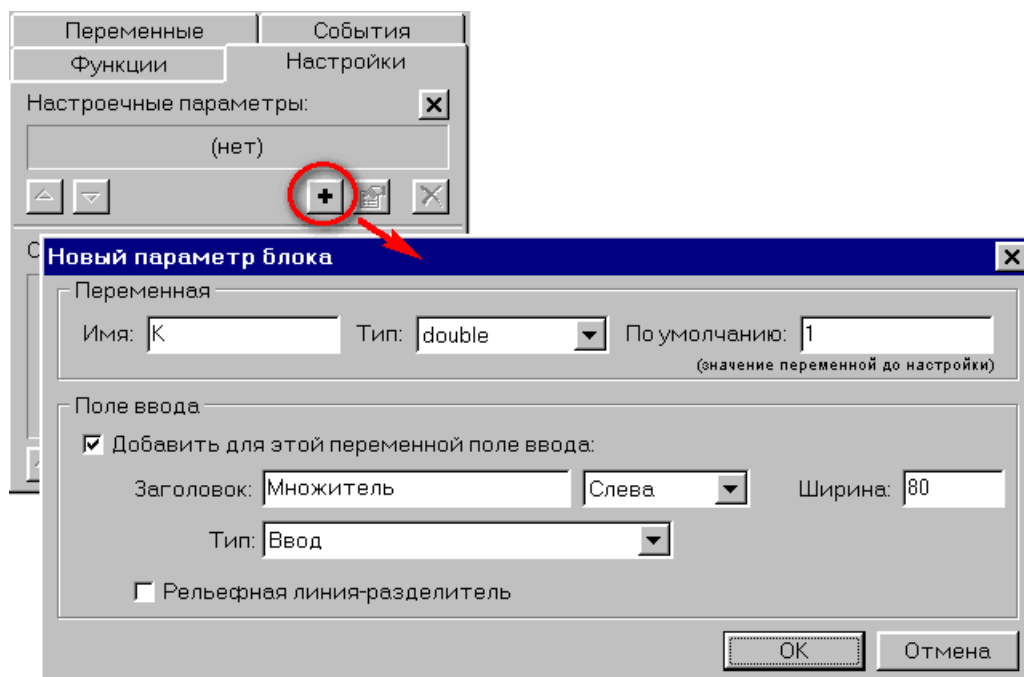


Рис. 418. Добавление настроечного параметра “К” и поля ввода для него

На панели “переменная” введем имя “К” (это имя будет именем переменной в нашей программе, поэтому введенная буква “К” должна быть буквой латинского алфавита), выберем в списке тип “double” и дадим параметру значение по умолчанию “1”. Одновременно с самим параметром создадим и поле ввода для него – на панели “поле ввода” установим флажок “добавить для этой переменной поле ввода” и введем заголовок “Множитель”. Теперь окно можно закрыть кнопкой “ОК”. Вкладка “события” окна редактора модели после этого примет вид, изображенный на рис. 419: в списке параметров появилась переменная “К”, а в списке полей окна настройки – поле ввода для нее с заголовком “множитель”.

Чтобы пользователю было понятнее, настройки какого блока он вызвал, дадим окну настройки заголовок. Для этого на панели “окно настройки” в нижней части вкладки

“настройки” нажмем кнопку справа сверху от списка полей (она останется нажатой) и введем заголовок “умножение на константу”, как показано на рис. 420. Ширину и высоту окна вводить не будем.

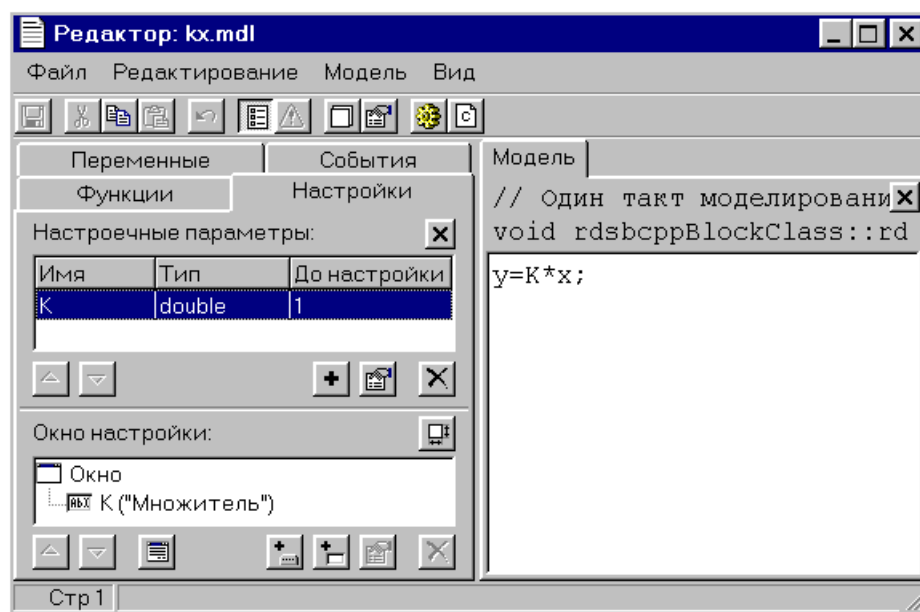


Рис. 419. Окно редактора модели с добавленным параметром

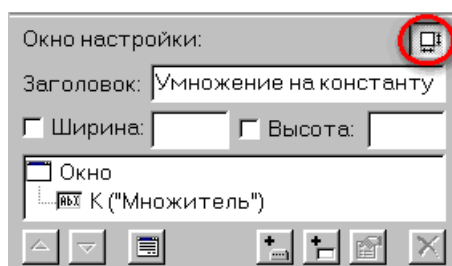


Рис. 420. Ввод заголовка окна настроек

Теперь можно посмотреть, как будет выглядеть созданное нами окно настроек. Нажмем кнопку “тест” под списком полей ввода, и созданное нами окно откроется (рис. 421).

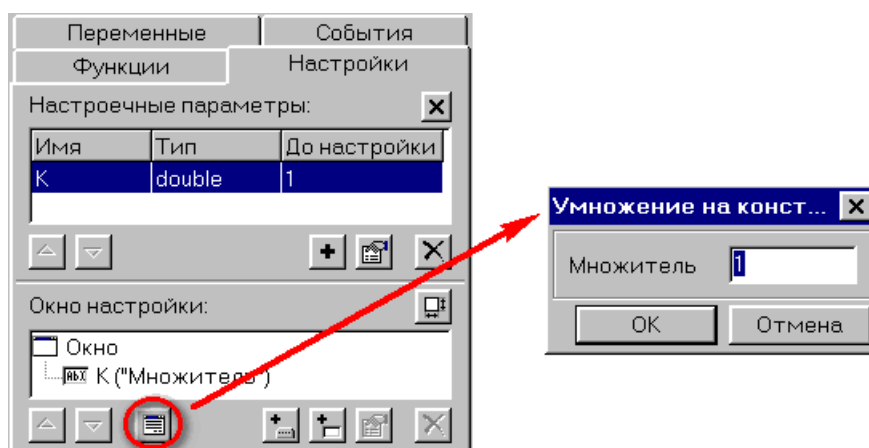


Рис. 421. Тестирование окна настроек

При некоторых настройках Windows (например, если установлены крупные шрифты), введенный нами текст заголовка окна может не уместиться в его верхней части – именно такой случай изображен на рисунке. Можно ввести другой, более короткий, заголовок, а

можно просто увеличить ширину окна, установив флажок “ширина” над списком полей ввода (см. рис. 420, если панель с заголовком закрылась, ее снова можно открыть кнопкой над списком полей) и введя желаемую ширину окна в точках экрана. Можно, например, установить ширину в 300 точек и снова нажать кнопку “тест”, чтобы проверить, уместился ли заголовок в окно. Если он все равно не умещается, можно ввести большую ширину и снова протестировать окно, и т.д. до тех пор, пока окно с заголовком не приобретет желаемый внешний вид.

Выполненных действий достаточно, чтобы у блока появилось окно настройки, в котором можно будет вводить значение множителя  $K$ . Это значение будет сохраняться вместе со схемой, причем независимо для всех блоков с этой моделью: если в схеме будет несколько таких блоков, значения  $K$  у них могут быть разными, несмотря на то, что их обслуживает одна и та же модель. Модуль автокомпиляции автоматически добавляет в модель команды сохранения и загрузки введенных пользователем значений параметров при сохранении и загрузке схемы или самого блока. Этим настроечные параметры отличаются от, например, значений по умолчанию статических переменных блока: у всех блоков с одной и той же моделью значения переменных по умолчанию будут одинаковыми, поскольку структура переменных блока с автокомпилируемой моделью запоминается в файле модели, а не в файле схемы, в котором хранятся другие данные блока.

Теперь можно скомпилировать модель и закрыть окно редактора. Если нажать на созданном блоке правую кнопку мыши и выбрать в контекстном меню пункт “настройка”, откроется созданное нами окно, в котором можно будет ввести значение множителя  $K$  для этого блока.

В отличие от большинства стандартных блоков, у которых окно настройки открывается по двойному щелчку, в нашем блоке по двойному щелчку открывается редактор модели. Можно исправить это двумя способами. Если блок с этой моделью – единственный, как в нашем случае, проще всего открыть окно его параметров (пункт “параметры” в его контекстном меню), и на вкладке “общие” включить флажок “двойной щелчок – вызывает функцию настройки” (рис. 422).

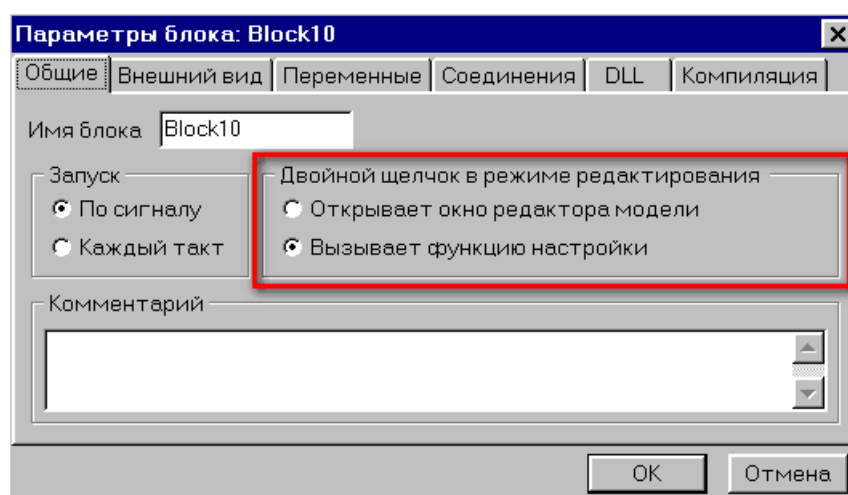


Рис. 422. Включение настройки по двойному щелчку в окне параметров блока

Если же блоков с данной моделью несколько, лучше включить настройку по двойному щелчку для них всех одновременно. Для этого нужно открыть редактор модели и выбрать пункт меню “модель | установка параметров блоков” (см. §3.6.8 на стр. 76): при этом откроется окно групповой установки, в котором можно изменить параметры всех блоков с этой моделью. На вкладке “разное” этого окна на панели “двойной щелчок” следует включить флажки “установить реакцию” и “передается в функцию DLL” (рис. 423). В данном

случае, “передается в функцию DLL” означает, что с двойным щелчком будет разбираться модель блока, а не РДС (окно параметров блока) и не модуль автокомпиляции (редактор модели). А модель блока по двойному щелчку может только открывать окно настройки, что и будет происходить. Новая реакция на двойной щелчок вступит в силу после закрытия окна кнопкой “ОК”.

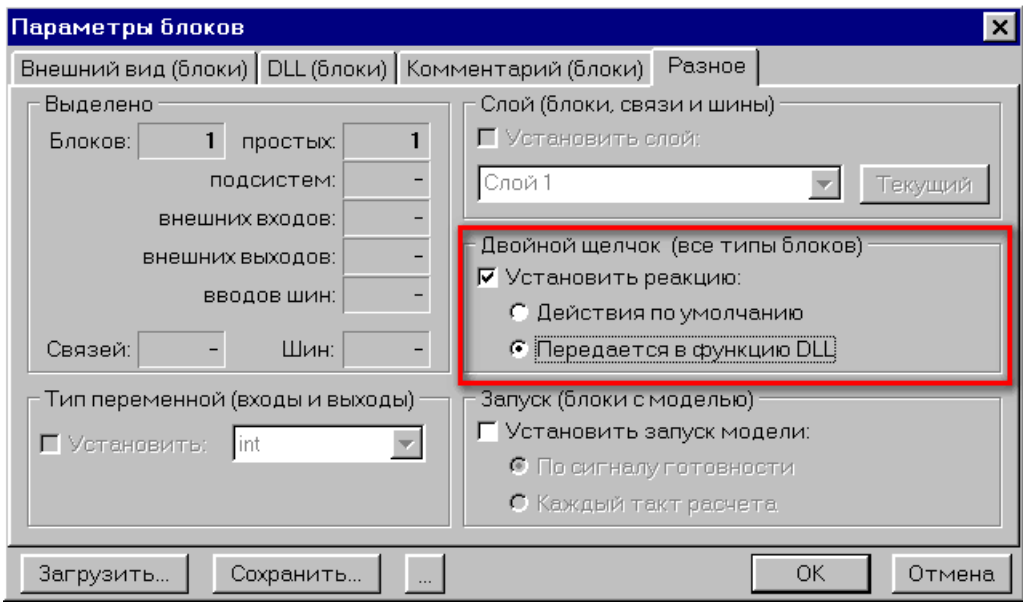


Рис. 423. Включение настройки по двойному щелчку в окне групповой установки

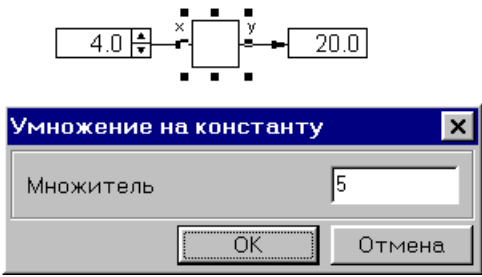


Рис. 424. Блок умножения на константу и окно его настройки

Для проверки работы созданной модели и ее настройки можно подключить поле ввода к входу блока *x* и числовой индикатор к выходу *y* (рис. 424). Двойной щелчок на блоке откроет окно настройки, в котором вводится множитель. Запустив расчет и изменяя значение на входе блока, можно будет видеть на выходе значение произведения входа и введенной в окне константы.

Параметрами блока могут быть не только вещественные переменные, но и целые, логические, строки, а также переменные типа COLORREF, который в

API Windows используется для описания цвета. Последние часто используют в качестве параметров блоков, изображения которых программно рисуются их моделями, чтобы пользователь мог изменять цвета этих изображений.

В §3.7.5 (стр. 176) мы создали простой индикатор уровня, рисующий на белом фоне вертикальный синий столбик, высота которого пропорциональна значению входа блока, причем вход мог изменяться от нуля (столбик нулевой высоты) до ста (столбик высотой во весь блок). Изменим модель так, чтобы диапазон изменения входа и все используемые при рисовании цвета стали настроечными параметрами.

Откроем редактор ранее созданной модели и, описанным выше образом, добавим на вкладке “настройки” следующие настроечные параметры, одновременно создавая для них поля ввода:

| Имя | Тип    | По умолчанию | Заголовок поля ввода | Тип поля ввода |
|-----|--------|--------------|----------------------|----------------|
| Min | double | 0            | Минимум              | Ввод           |
| Max | double | 100          | Максимум             | Ввод           |

| <i>Имя</i>  | <i>Тип</i> | <i>По умолчанию</i> | <i>Заголовок поля ввода</i> | <i>Тип поля ввода</i> |
|-------------|------------|---------------------|-----------------------------|-----------------------|
| UpColor     | COLORREF   | 0xffffffff          | Верхняя часть               | Выбор цвета           |
| DownColor   | COLORREF   | 0xff0000            | Нижняя часть                | Выбор цвета           |
| BorderColor | COLORREF   | 0                   | Рамка                       | Выбор цвета           |
| ErrorColor  | COLORREF   | 0xff                | Ошибка                      | Выбор цвета           |

В качестве заголовка окна настройки введем “уровень”. В результате вкладка “настройки” окна редактора и окно настройки блока (его можно увидеть, нажав кнопку “тест” под списком полей ввода) будет выглядеть так, как на рис. 425.

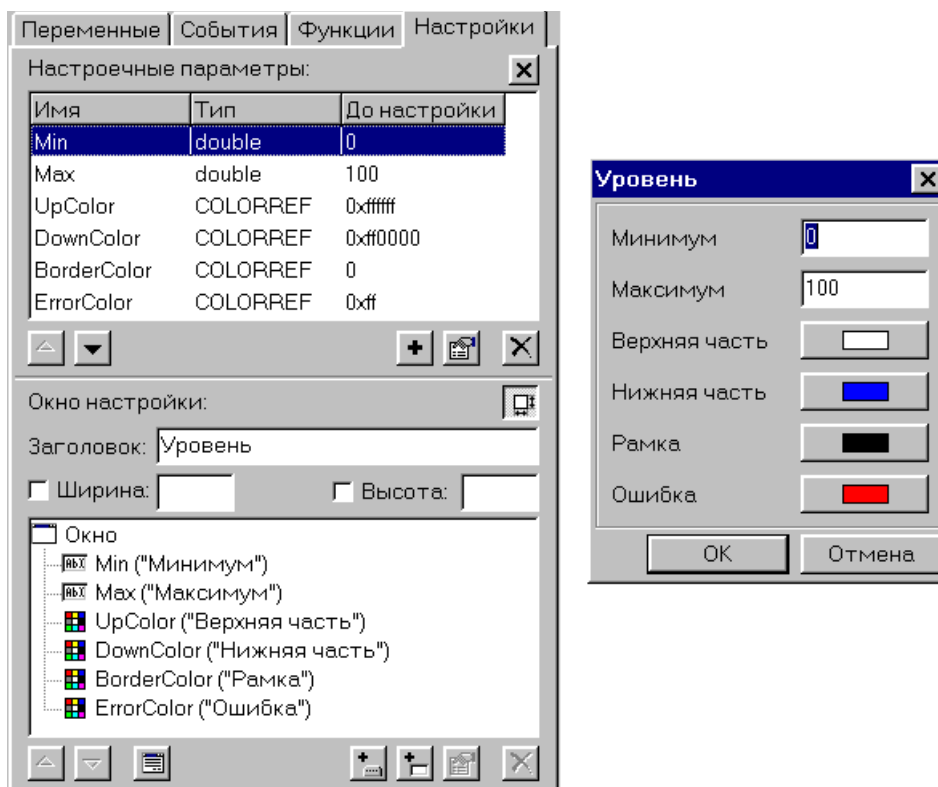


Рис. 425. Добавленные настроечные параметры и тест окна настройки

У созданного окна настройки есть небольшой недостаток: разные по смыслу параметры изображаются вместе. Пользователю будет удобнее, если дать ему понять, что “минимум” и “максимум” относятся к диапазону входа, а кнопки – к заданию цветов блока. Можно добавить в окно пояснительные надписи, но от этого оно станет слишком высоким. Лучше всего разбить параметры на две вкладки: поля ввода диапазона входа будут находиться на вкладке “диапазон”, а цвета – на вкладке “цвета”.

Сейчас в нашем окне нет ни одной вкладки. Создадим первую из них: нажмем кнопку “добавить вкладку” под списком полей ввода (рис. 426) и введем ее заголовок “диапазон” в открывшемся окне. После нажатия кнопки “ОК” все уже созданные поля ввода окажутся на новой вкладке. Еще раз нажмем кнопку “добавить вкладку” и введем заголовок “цвета” – новая пустая вкладка “цвета” появится в конце списка полей (рис. 427).

Теперь нам нужно по очереди переместить кнопки выбора цветов на вторую вкладку. Сначала выберем в списке поле ввода для параметра “UpColor”. Под списком полей слева находятся две кнопки со стрелками вниз и вверх, предназначенные для изменения порядка полей и вкладок. Будем нажимать кнопку со стрелкой вниз до тех пор, пока поле “UpColor”

не переместится ниже символа вкладки “цвета”. Теперь поле ввода будет располагаться на этой вкладке.

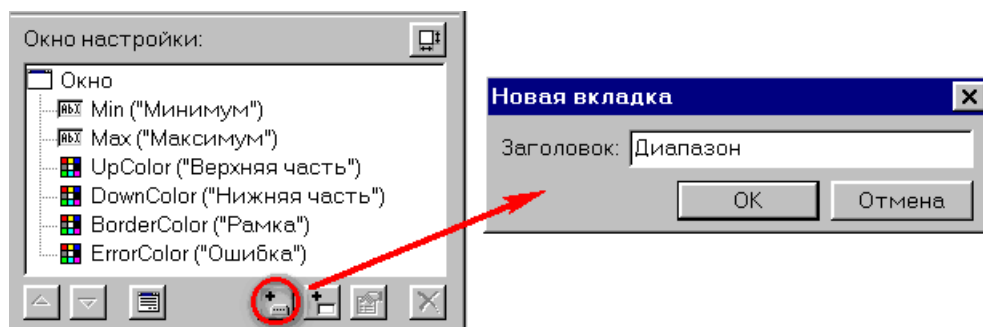


Рис. 426. Добавление вкладки

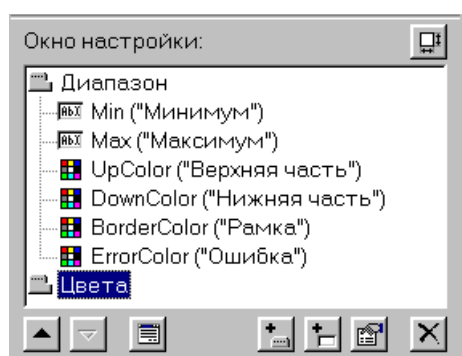


Рис. 427. В окно добавлены вкладки “диапазон” и “цвета”

открывшемся окне параметров

последовательно переместим все остальные поля задания цветов на вкладку “цвета” таким же образом, выстроив их в том же порядке, в котором мы их добавляли: “UpColor”, “DownColor”, “BorderColor” и “ErrorColor”. Чтобы визуально отделить цвет “ErrorColor”, используемый для индикации ошибки входа, от остальных трех цветов, использующихся при нормальной работе блока, добавим после “BorderColor” (то есть, перед “ErrorColor”) рельефную линию. Для этого дважды щелкнем на “BorderColor” в списке полей окна настройки (то есть в списке на нижней части вкладки – на верхней части вкладки располагается список самих параметров, а не полей ввода для них) и в поля ввода установим флажок “рельефная линия-разделитель” (рис. 428).

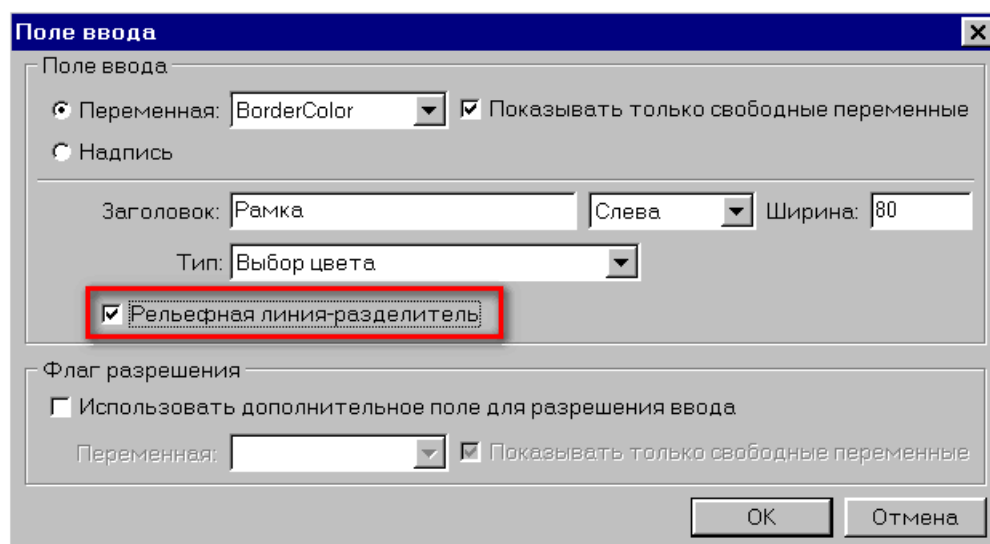


Рис. 428. Добавление рельефной линии в окне параметров поля ввода

Теперь поля ввода распределены по двум вкладкам. Если нажать под списком полей кнопку “тест”, можно будет увидеть окно с вкладками “диапазон” и “цвета” и линией между полями “рамка” и “ошибка” на последней (рис. 429).



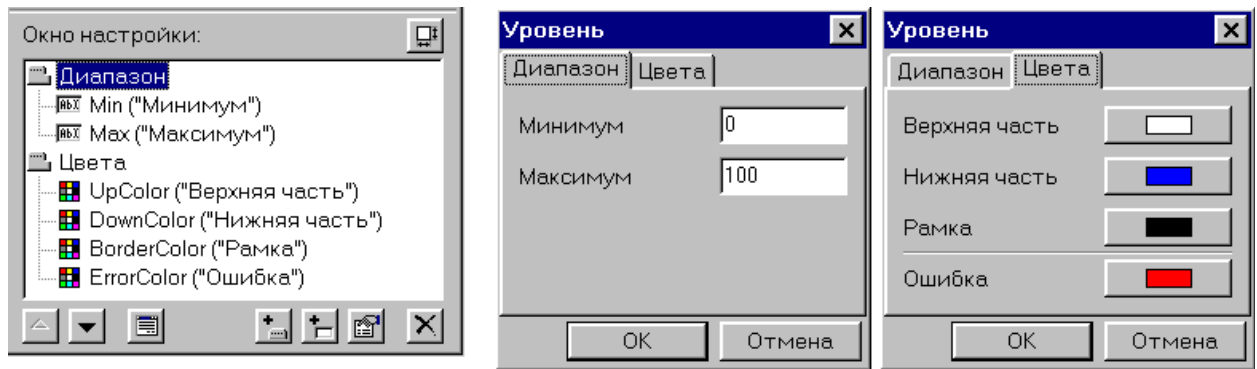


Рис. 429. Список полей ввода (слева) и две вкладки созданного окна настроек (в центре и справа)

Мы добавили в блок настроечные параметры и создали окно настройки для них. Теперь нужно ввести эти параметры в модель. Текст на вкладке редактора “рисование” (это единственный введенный фрагмент программы в модели индикатора уровня) нужно изменить, заменив в нем константы на настроечные параметры. В тексте ниже изменения выделены двойным подчеркиванием. Комментарии оставлены без изменения, хотя в них вместо слов “красный”, “белый”, “синий” и “черный” теперь следовало бы тоже использовать имена параметров.

```
// Вспомогательные переменные
int right,bottom,h;

// Правый нижний угол области блока
right=DrawData->Left+DrawData->Width;
bottom=DrawData->Top+DrawData->Height;

// Проверка наличия ошибки на входе
if(x==rdsbcppHugeDouble_||_Max<=_Min) // Заливаем красным
{ // Красный фон
 rdsXGSetBrushStyle(0,RDS_GFS_SOLID,ErrorColor);
 // Черная линия
 rdsXGSetPenStyle(0,PS_SOLID,
 DrawData->DoubleZoom,BorderColor,R2_COPYPEN);
 // Прямоугольник - черная рамка и красный фон
 rdsXGRectangle(DrawData->Left,DrawData->Top,right,bottom);
 // Завершаем реакцию, больше ничего не рисуем
 return;
}

// Высота синего столбика в точках экрана
h=(x-Min)*DrawData->Height/(Max-Min);
// Ограничение
if(h<0)
 h=0;
else if(h>DrawData->Height)
 h=DrawData->Height;

// Отключаем рамку
rdsXGSetPenStyle(RDS_GFSTYLE,PS_NULL,0,0,0);

// Рисуем верхнюю (белую) часть
if(h!=DrawData->Height) // Есть белое
```

```

{ // Белый фон
 rdsXGSetBrushStyle(0,RDS_GFS_SOLID,UpColor);
 // Верхняя часть - на h меньше высоты блока
 rdsXGRectangle(DrawData->Left,
 DrawData->Top,
 right,
 bottom-h+1);
}
// Рисуем нижнюю (синюю) часть
if (h!=0)
{ // Синий фон
 rdsXGSetBrushStyle(0,RDS_GFS_SOLID,DownColor);
 // Нижняя часть - от h до высоты блока
 rdsXGRectangle(DrawData->Left,
 bottom-h,
 right,
 bottom);
}

// Рисуем рамку
// Черная линия, толщина - с учетом масштаба
rdsXGSetPenStyle(0,PS_SOLID,DrawData->DoubleZoom,
 BorderColor,R2_COPYPEN);
// Отключаем заливку
rdsXGSetBrushStyle(RDS_GFSTYLE,RDS_GFS_EMPTY,0);
// Прямоугольник рамки
rdsXGRectangle(DrawData->Left,DrawData->Top,right,bottom);

```

В тексте модели, кроме замены цветовых констант на настроечные параметры, изменена также формула вычисления высоты столбика  $h$  (теперь там учитываются параметры границ диапазона  $x$ , которые раньше были жестко заданы) и проверка возможности рисования. Если раньше рисование было невозможно только тогда, когда на вход блока приходило значение-индикатор ошибки `rdsbcppHugeDouble`, то теперь ошибкой следует считать и неправильное задание диапазона индикатора: если пользователь по ошибке сделает верхнюю границу диапазона `Max` меньше или равной нижней `Min`, рисование индикатора становится бессмысленным. На самом деле, следовало бы также проверить и `Min`, и `Max` на равенство `rdsbcppHugeDouble`, но, для упрощения примера, будем считать пользователя достаточно разумным, чтобы не вводить в качестве одной из границ вопросительный знак, обозначающий в РДС ошибочное значение.

Теперь цвета и границы диапазона нашего блока могут быть настроены пользователем. Для его удобства можно сделать так, чтобы окно настройки открывалось по двойному щелчку вместо редактора модели (см. стр. 197). Индикатор, у которого изменен входной диапазон и цвета верхней и нижней части, изображен на рис. 430.

К модели может быть добавлено любое число настроечных параметров и они могут быть распределены по вкладкам окна так, как хочет разработчик. Как уже упоминалось в конце §3.6.6 (стр. 68), стандартный блок автокомпиляции не позволяет создавать модели, в которых параметры могут как подаваться на входы по связям, так и вводиться в окне настройки. Тем не менее, предоставляемых модулем возможностей достаточно для настройки большинства блоков.

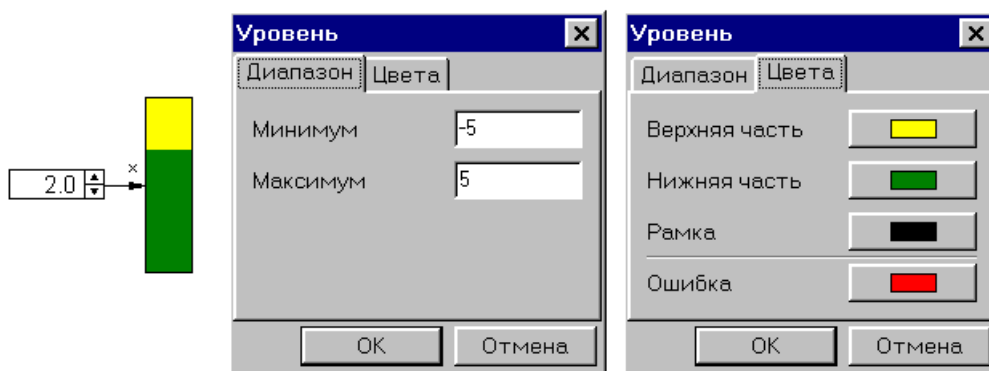


Рис. 430. Измененный индикатор и вкладки его окна настроек

### §3.7.7. Задание пользователем имен динамических переменных

Рассматривается способ сделать имена динамических переменных, при помощи которых блоки обмениваются информацией друг с другом без явного проведения связей, настройчными параметрами блока.

В §3.7.3.2 (стр. 136) были приведены примеры моделей блоков, передающих друг другу вещественное значение через динамическую переменную: одна модель записывала в переменную данные, другая – считывала их. Такие блоки можно использовать для того, чтобы связать между собой далеко отстоящие друг от друга части схемы без явного проведения между ними связи: блок в одной части схемы будет записывать данные в переменную, а блок в другой части – считывать их. Модели в приведенном примере обладают одним существенным недостатком: имя переменной, через которую блоки передают данные, жестко встроено в сами модели. Если потребуется организовать передачу другого вещественного значения, придется создавать пару точно таких же моделей, в которых будет задано другое имя переменной.

Гораздо удобнее сделать имя переменной, через которую идет обмен данными, параметром блока, и дать пользователю возможность вводить ее имя в окне настроек этого блока. В модуле автокомпиляции есть такая возможность: при создании динамической переменной можно вместо указания для переменной фиксированного имени связать это имя с настройчным параметром типа “rdsbcppString” (специальный тип для работы со строками, поддерживаемый модулем автокомпиляции). Если создать для этого параметра поле ввода, пользователь сможет изменять имя динамической переменной в настройках блока.

Создадим две модели для обмена вещественным значением через динамическую переменную: как и в §3.7.3.2, первая модель будет записывать в эту переменную значение своего входа, а вторая – выдавать это значение на свой выход. Однако, в этом примере мы сделаем имя динамической переменной настраиваемым.

Начнем с блока-передатчика, на вход которого будет подаваться значение для записи в динамическую переменную. Структура статических переменных этого блока будет такой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 1                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| x          | double     | Вход              | ✓           | 0                         |

Его модель будет запускаться при самом первом запуске расчета (поэтому у сигнала Start единичное начальное значение), а также при срабатывании связи, подключенной к входу x (поэтому у этого входа установлен флажок “пуск”). Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95), нажмем в

верхней части вкладки “переменные” редактора модели кнопку “изменить” и введем указанную выше структуру переменных.

Теперь добавим в модель объект для работы с динамической переменной, имя которой будет считываться из настроечного параметра. Для этого выполним следующие действия:

- на вкладке “переменные” левой панели редактора модели в нижней ее части (см. рис. 334 на стр. 42) нажмем кнопку со знаком “+”;
- в открывшемся окне добавления динамической переменной (рис. 431) установим флажок “произвольная переменная”;
- в “имя переменной в RDS” выберем вариант “из настроечного параметра” и введем в поле справа от этого варианта имя “VarName” – параметр с таким именем будет хранить имя нашей динамической переменной;
- в поле “имя переменной в программе” введем “DynVar” – по этому имени мы будем обращаться к переменной в программе модели;
- в выпадающем списке “блок-владелец” выберем вариант “подсистема (RDS\_DVPARENT)” – переменная будет создаваться в родительской подсистеме блока;
- в выпадающем списке “действие” выберем “создать”;
- в выпадающем списке “тип” оставим проставленный там по умолчанию вариант “double” – наша переменная будет вещественной;
- закроем окно добавления переменной кнопкой “OK”.

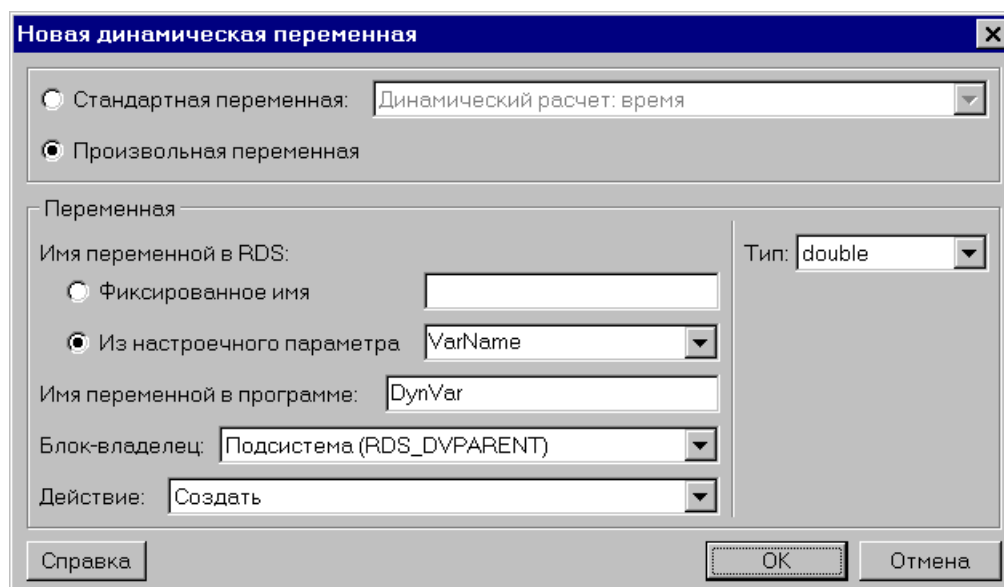


Рис. 431. Добавление объекта для динамической переменной с привязкой имени к настроечному параметру

После нажатия кнопки “OK” появится предупреждение об отсутствии в модели настроечного параметра с именем “VarName” с предложением создать его, на которое следует ответить “да”. После этого откроется окно добавления настроечного параметра (см. рис. 418), в котором поля “имя” и “тип” заблокированы: в поле “имя” уже проставлено “VarName”, а в поле “тип” – “rdsbcppString” (типом параметра для хранения имени переменной обязательно должна быть строка). В этом окне следует установить флажок “добавить для этой переменной поле ввода” и, под этим флажком, ввести в поле “заголовок” текст “имя переменной”, а в поле “ширина” – “100” (остальные поля ввода окна, включая пустое значение параметра по умолчанию, оставим без изменения), а затем закрыть окно кнопкой “OK”.

Класс `rdsbcppString`, используемый в автокомпилируемых моделях для хранения текстовых строк, подробно рассматривался в §3.7.2.5 (стр. 112), поэтому здесь мы не будем на нем останавливаться.

После закрытия окна добавления параметра в списке динамических переменных редактора модели появится созданный нами объект с именем `DynVar` (рис. 432 слева), а на вкладке “настройки” – параметр `VarName` и поле ввода для него (рис. 432 справа). В списке динамических переменных имя “`VarName`” отображается без кавычек, поскольку это не имя переменной в РДС, а имя настроечного параметра, т. е. имя переменной программы модели.

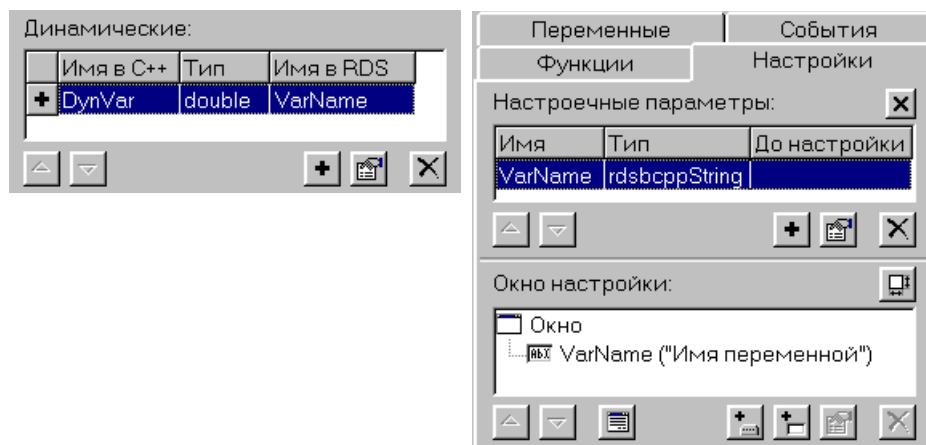


Рис. 432. Созданный объект в списке динамических переменных модели (слева) и вкладка “настройки” после добавления параметра (справа)

На вкладке “настройки” следует нажать кнопку с размерными стрелками справа от текста “окно настройки” и ввести в поле ввода “заголовок” текст “передатчик” – это будет заголовком нашего окна настройки. Так пользователю будет понятнее, с каким именно блоком он работает.

Все действия по слежению за тем, чтобы имя создаваемой переменной совпадало с текстом, который пользователь ввел в окне настроек, будут добавлены в модель автоматически. При изменении имени в настройках модель сама будет удалять переменную с прежним именем и создавать переменную с новым. Переменная также будет создаваться сразу после загрузки параметров блока из файла или буфера обмена. Осталось только ввести в модель реакцию на такт расчета, в которой значение входа блока будет копироваться в динамическую переменную. Раскроем раздел “моделирование и режимы” вкладки “события” и дважды щелкнем на его подразделе “модель”. На открывшейся одноименной вкладке введем текст, аналогичный тексту модели из §3.7.3.2:

```
DynVar=x;
DynVar.NotifySubscribers();
```

Займемся теперь моделью приемника, который будет читать вещественное число из динамической переменной, имя которой задается пользователем, и выдавать его на свой выход. Структура переменных приемника будет такой:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| y          | double     | Выход             |             | 0                         |

Создадим новый блок с автокомпилируемой моделью, зададим для него запуск по сигналу (см. стр. 95) и введем в его редакторе эту структуру переменных. В эту модель необходимо добавить объект `DynVar` с таким же настроечным параметром `VarName`, как и

в предыдущей модели. Повторим все действия по добавлению этих объектов и созданию окна настройки, описанные выше, с двумя исключениями: при добавлении динамической переменной (см. рис. 431) вместо действия “создать” выберем действие “найти и подписаться”, а в качестве заголовка окна настройки укажем не “передатчик”, а “приемник”.

Введем в модель реакцию на изменение динамической переменной (раздел “моделирование и режимы”, подраздел “изменение динамической переменной”), в которой мы будем считывать число из этой переменной и выдавать его на выход. Текст реакции тоже будет аналогичен реакции модели из §3.7.3.2:

```
y=DynVar;
Ready=1; // Выходные связи должны работать
```

Обе модели готовы. Блок-передатчик будет записывать значение своего входа в динамическую переменную, а блок-приемник будет читать ее и передавать на свой выход. Поскольку передатчик создает переменную в своей родительской подсистеме, приемник должен размещаться либо в одной подсистеме с передатчиком, либо в подсистеме любого уровня вложенности внутри этой подсистемы. Можно сделать так, чтобы передатчик создавал переменную в корневой подсистеме схемы, тогда она будет видна отовсюду, и приемник можно будет размещать в любом месте. Для этого достаточно будет заменить в параметрах динамических переменных обеих моделей блок-владелец с “подсистема (RDS\_DVPARENT)” на “система (RDS\_DVROOT)”.

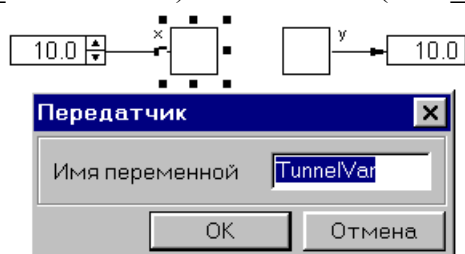


Рис. 433. Схема для проверки приемника и передатчика и окно настройки передатчика

Проверим работу созданных блоков. Соберем схему, изображенную на рис. 433: разместим приемник и передатчик в одной подсистеме, к входу передатчика (на рисунке он слева) присоединим поле ввода, а к выходу приемника – индикатор. Вызовем окно настройки передатчика (пункт “настройка” в контекстном меню) и введем там какое-нибудь имя переменной. В настройках приемника введем то же самое имя. Теперь,

если запустить расчет, значение на выходе приемника будет повторять значение на входе передатчика. Имена переменных в настройках блоков можно менять как угодно – пока имя в приемнике будет совпадать с именем в передатчике, блоки будут связаны. Можно, при желании, добавить в схему еще несколько передатчиков, задав им другие имена переменных, и еще несколько приемников. Каждый приемник будут получать значение от передатчика, в настройках которого введено то же самое имя переменной.

Следует отметить, что сейчас по двойному щелчку на блоке-приемнике и блоке-передатчике открывается не окно настройки, а окно редактора модели – это удобно для отладки, но неудобно для пользователя. Как сделать так, чтобы по двойному щелчку открывалось окно настройки, описано в §3.7.6 (стр. 194, см. также рис. 422 на стр. 197).

### §3.7.8. Программное управление динамическими переменными

Описывается программное выполнение всех действий по созданию, удалению и подписке на динамические переменные.

В предыдущем параграфе рассматривались модели блоков, в которых имена динамических переменных хранились в настроечных параметрах блоков. Модуль автокомпиляции при этом автоматически добавлял в программу модели вызовы, необходимые для удаления и прекращения подписки на переменную со старым именем и создания и подписки на переменную с новым именем при изменении этого имени. В некоторых, довольно редких, случаях может потребоваться задавать имя динамической переменной программно – например, при изменении состояния блока или значений других

его переменных. Это осуществимо, но при этом придется отказаться от автоматического создания переменной и автоматического получения к ней доступа – эти функции, добавляемые в модель модулем автокомпиляции, нужно будет отключить. Вместо этого придется создавать переменную и получать к ней доступ вручную при помощи специальных функций-членов объекта для работы с переменной. В каждом таком объекте для этой цели предусмотрено четыре функции:

- `BOOL Create(int Block, char *Name, BOOL Fixed)` – создание динамической переменной с именем `Name` в блоке, указываемом целым параметром `Block`, который может принимать одно из трех значений: `RDS_DVSELF` (создать переменную в вызвавшем блоке), `RDS_DVPARENT` (создать переменную в родительской подсистеме) или `RDS_DVROOT` (создать переменную в корневой подсистеме). Чаще всего используются варианты `RDS_DVPARENT` и `RDS_DVROOT`, поскольку переменная, созданная в простом блоке, не будет видна никому, кроме этого блока. Значение `TRUE` в параметре `Fixed` разрешает удалять переменную только создавшему ее блоку, значение `FALSE` – любому из блоков схемы. Функция возвращает успешность создания переменной. Если вызвать эту функцию-член у объекта, который уже связан с созданной динамической переменной, ранее созданная переменная будет удалена, после чего будет создана новая, согласно параметрам функции.
- `BOOL Delete(void)` – удалить переменную, ранее созданную вызовом `Create`. Возвращается успешность удаления.
- `BOOL Subscribe(int Block, char *Name, BOOL Search)` – получить доступ к переменной с именем `Name` в блоке, указываемом целым параметром `Block`, который принимает те же три значения, что и одноименный параметр функции `Create`: `RDS_DVSELF` (искать переменную в вызвавшем блоке), `RDS_DVPARENT` (искать переменную в родительской подсистеме) или `RDS_DVROOT` (искать переменную в корневой подсистеме). Значение `TRUE` в параметре `Search` заставит РДС искать переменную с указанным в параметре `Name` именем и заложенным в сам объект типом начиная с указанного блока вверх по всей иерархии подсистем вплоть до корневой. Функция возвращает `FALSE` только в случае какой-либо серьезной ошибки – например, если вместо имени переменной передана пустая строка. `TRUE` возвращается даже тогда, когда переменная с указанным именем не найдена: РДС запоминает факт обращения к этой переменной и, как только она появится, предоставит блоку доступ к ней. Для проверки фактического существования переменной используется функция-член `Exists` (см. стр. 133). Если вызвать функцию `Subscribe` у объекта, который уже связан с какой-либо динамической переменной, прежняя связь будет разорвана и будет установлена новая.
- `void Unsubscribe(void)` – разорвать связь с динамической переменной, ранее установленную вызовом `Subscribe`.

Во всех этих функциях тип динамической переменной не указывается, он всегда жестко закладывается в объект для работы с ней при его добавлении в редактор модели (см. §3.6.3 на стр. 42).

В качестве примера использования этих функций изменим пример из предыдущего параграфа: будем вручную выполнять все действия, которые необходимы для изменения имени переменной. Начнем с модели блока-передатчика.

Прежде всего, откроем редактор модели блока, дважды щелкнем на динамической переменной “`DynVar`”, в окне ее параметров в выпадающем списке “действие” выберем вариант “все действия – вручную” (рис. 434) и закроем окно кнопкой “ОК”.



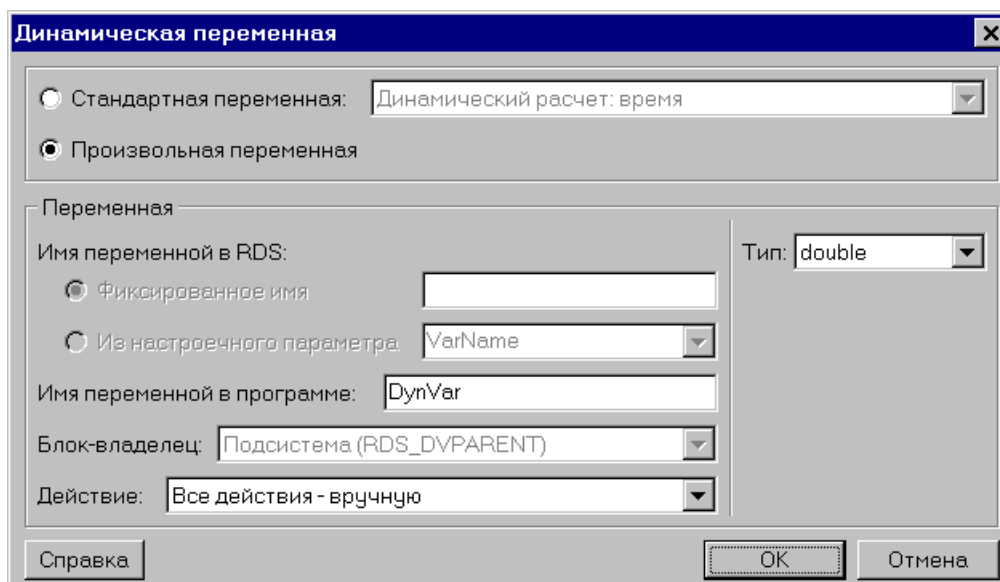


Рис. 434. Параметры объекта динамической переменной без привязки к конкретной переменной

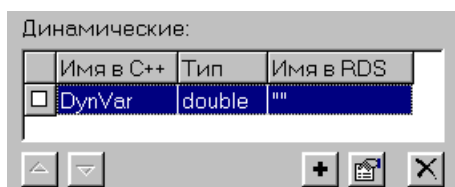


Рис. 435. Измененный объект в списке динамических переменных модели

После закрытия окна в списке динамических переменных редактора модели в левой колонке рядом с именем объекта DynVar будет нарисован пустой белый квадрат (рис. 435) – это указывает на то, что объект автоматически не связывается с какой-либо динамической переменной.

Теперь в нашей модели есть объект DynVar для обращения к динамической переменной типа double, не связанный автоматически ни с какой переменной, и объект VarName, хранящий имя переменной, введенное пользователем в настройках (он остался от прежней версии модели). Необходимо дать указание объекту DynVar создать переменную с таким именем – для этого служит его функция-член Create, описанная выше в этом параграфе. Причем вызывать эту функцию нужно после любого изменения значения настроечного параметра VarName.

В автокомпилируемой модели есть всего два места, в которых значение настроечных параметров может меняться. Во-первых, их значения могут быть изменены пользователем – это происходит в момент закрытия окна настройки блока кнопкой “OK”. Во-вторых, значения параметров изменяются в момент загрузки данных блока – это происходит при загрузке в память схемы с этим блоком, при отмене пользователем сделанного им изменения в схеме, при вставке блока в схему из библиотеки или буфера обмена, и т.п. Нам нужно создавать динамическую переменную с новым именем вызовом функции Create в обоих этих случаях. Эта функция автоматически удаляет переменную, которая была ранее создана этим же объектом, поэтому при изменении имени переменной в параметре нам не придется заботиться об удалении переменной со старым именем перед созданием новой.

Начнем с создания переменной при изменении пользователем ее имени в окне настроек блока. Если в модель добавлено окно настроек, при его закрытии кнопкой “OK” вызывается реакция на событие “вызов настройки”. Добавим в нашу модель эту реакцию: на левой панели окна редактора выберем вкладку “события” (см. §3.6.4 на стр. 46), раскроем на ней раздел “разное” и дважды щелкнем на его подразделе “вызов настройки (RDS\_BFM\_SETUP)”. При этом значок подраздела станет желтым, а в правой части окна появится новая пустая вкладка “настройка”. Введем на ней следующий текст:



```
DynVar.Create(RDS_DVPARENT, VarName.c_str(), TRUE);
```

Здесь мы вызываем у объекта DynVar функцию-член Create, передавая ей параметры RDS\_DVPARENT (создать переменную в родительской подсистеме блока), VarName.c\_str() (строка с именем переменной из настроечного параметра VarName) и TRUE (удалить созданную переменную сможет только этот блок). Переменную мы создаем в родительской подсистеме блока-передатчика, чтобы к ней могли получить доступ блоки в одной с ним подсистеме и в подсистемах, вложенных в нее. Следует обратить внимание на то, что во втором параметре функции мы вызываем у параметра VarName функцию-член c\_str: дело в том, что VarName имеет тип rdsbcppString (это объект специального класса для хранения строк), а функция Create требует имени переменной типа char\*. Функция c\_str возвращает указатель на строку char\*, хранящуюся внутри объекта.

Теперь, если пользователь откроет окно настроек блока и закроет его кнопкой “ОК”, мы стираем ранее созданную блоком динамическую переменную, и создаем новую, с именем, взятым из параметра VarName. Все это делает вызов функции Create.

Добавим в нашу модель реакцию на загрузку параметров блока. На вкладке “события” раскроем раздел “загрузка и запись данных” (важно не перепутать его с похожим разделом “загрузка и запись состояния”) и дважды щелкнем на его подразделе “загрузка данных блока (RDS\_BFM\_LOADTXT)”. Значок подраздела станет желтым, а в правой части окна появится новая вкладка “загрузка/текст”, на которой нужно ввести точно такой же вызов функции Create, какой мы ввели на вкладке “настройка” (можно скопировать его оттуда через буфер обмена). Теперь при загрузке данных блока, когда сохраненное ранее значение параметра VarName будет восстановлено в момент загрузки отдельного блока или всей схемы, мы тоже создаем динамическую переменную с новым именем.

Уничтожение созданных динамических переменных при уничтожении данных всего блока (например, при закрытии РДС или перед загрузкой в память другой схемы) производится автоматически, но, чтобы проиллюстрировать возможность программного удаления переменной, сделаем это вручную, в реакции на событие очистки данных блока. На вкладке “события” раскроем раздел “создание и уничтожение” и дважды щелкнем на его подразделе “очистка блока (RDS\_BFM\_CLEANUP)”. Введем на открывшейся вкладке следующий текст:

```
DynVar.Delete();
```

Здесь мы просто удаляем ранее созданную переменную – у функции Delete нет параметров.

Осталось изменить реакцию на такт расчета, в которой значение входа блока копируется в динамическую переменную. Раскроем раздел “моделирование и режимы” вкладки “события” и дважды щелкнем на его подразделе “модель”. На открывшейся одноименной вкладке введем следующий текст:

```
if(DynVar.Exists())
{ DynVar=x;
 DynVar.NotifySubscribers();
}
```

В отличие от модели из §3.7.7, здесь мы обязательно должны убедиться в существовании динамической переменной, поскольку в параметрах объекта DynVar мы указали, что все действия с ним мы будем выполнять вручную. Если функция-член Exists этого объекта вернет TRUE, значит, переменная существует, и мы можем присвоить ей значение с входа блока x и уведомить об этом всех ее подписчиков вызовом NotifySubscribers.

Модель передатчика готова, отредактируем теперь модель приемника. В ней нужно изменить параметры объекта DynVar точно так же, как они были изменены для передатчика: в выпадающем списке “действие” следует выбрать вариант “все действия – вручную” (см. рис. 434).

Добавим в модель команды для получения доступа к переменной, имя которой содержится в параметре `VarName`. Как и в предыдущей модели, их нужно выполнять при изменении значения настроечного параметра, то есть в реакциях на вызов окна настроек (вкладка “события”, раздел “разное”, подраздел “вызов настройки”) и на загрузку данных блока (раздел “загрузка и запись данных”, подраздел “загрузка данных блока”). В обеих реакциях введем один и тот же текст:

```
DynVar.Subscribe(RDS_DVPARENT, VarName.c_str(), TRUE);
```

У объекта `DynVar` мы вызываем функцию-член `Subscribe`, передавая ей параметры `RDS_DVPARENT` (искать переменную в родительской подсистеме блока), `VarName.c_str()` (строка с именем переменной из настроечного параметра `VarName`) и `TRUE` (если переменной не будет в родительской подсистеме, искать ее далее вверх по иерархии). Эта функция автоматически разорвет старую связь объекта `DynVar` с переменной, если она была установлена ранее, и даст ему доступ к переменной с новым именем.

При уничтожении данных всего блока разрыв связей с динамическими переменными, доступ к которым он затребовал, производится автоматически, поэтому мы могли бы не предпринимать для этого никаких дополнительных действий. Чтобы проиллюстрировать возможность программного разрыва связи с переменной, все-таки сделаем это вручную в реакции на событие очистки данных блока (раздел “создание и уничтожение”, подраздел “очистка блока” на вкладке “события”). Введем в эту реакцию такой текст:

```
DynVar.Unsubscribe();
```

Функция `Unsubscribe` без параметров разрывает ранее созданную функцией `Subscribe` связь с динамической переменной.

Теперь отредактируем в модели реакцию на изменение динамической переменной (раздел “моделирование и режимы”, подраздел “изменение динамической переменной”), в которой мы считываем число из этой переменной и выдаем его на выход. Новый текст реакции будет таким:

```
if (DynVar.Exists())
{
 y=DynVar;
 Ready=1;
}
```

Прежде чем обращаться к динамической переменной через объект `DynVar`, мы должны проверить, существует ли эта переменная. Даже если в параметрах модели включен запрет ее вызова при отсутствии динамических переменных, на объект `DynVar` этот запрет не распространяется – при создании объекта мы указали, что все действия с ним мы будем выполнять вручную. Поэтому сначала мы вызываем его функцию-член `Exists`. Только если она вернет `TRUE`, что будет означать существование переменной, мы скопируем ее значение в выход блока `y` и взведем сигнал готовности `Ready` (в отличие от реакции на такт расчета, в реакции на изменение динамической переменной сигнал готовности не взводится автоматически).

Измененные модели приемника и передатчика будут работать точно так же, как и модели, описанные в предыдущем параграфе, пользователь не заметит между ними разницы. В данном случае для решения поставленной задачи вызов функций для программного управления переменными не требуется, они используются только для демонстрации работы с ними.

### §3.7.9. Всплывающие подсказки

Описывается создание всплывающих подсказок к блокам, сообщающих пользователю различную информацию о блоке и значения его основных параметров.

Если пользователь задержит курсор над изображением блока, модель этого блока может вывести всплывающую подсказку – окно с текстом, которое на некоторое время появится поверх изображения, и, через некоторое время, исчезнет. Всплывающие подсказки часто используются в Windows для вывода различных пояснений к кнопкам и полям ввода, когда такие пояснения заняли бы в самом окне слишком много места. В подсказках к блокам часто выводятся значения параметров этих блоков, сообщения об ошибках или комментарии. Модель может вывести подсказку ко всему блоку или к отдельному элементу его изображения, причем она может задать не только текст, который увидит пользователь, но и некоторые другие параметры подсказки – например, время ее нахождения на экране, или координаты зоны окна, при выходе курсора из которой подсказка автоматически исчезнет. Для того, чтобы при задержке курсора над блоком подсказка появилась на экране, должны быть одновременно выполнены два условия: во-первых, в параметрах блока должен быть разрешен вывод всплывающих подсказок и, во-вторых, модель блока, среагировав на событие вывода подсказки (ему соответствует константа `RDS_BFM_POPUPHINT`, см. §A.2.6.12 приложения к руководству программиста [2]), должна передать текст подсказки в РДС функцией `rdsSetHintText`. Ниже будут рассмотрены два примера моделей, выводивших подсказки пользователю.

Начнем с простого примера, в котором подсказка выводится ко всему блоку целиком. В §3.7.6 (стр. 194) была создана модель блока, выдающего на выход  $y$  произведение входа  $x$  и настроенного параметра  $K$ . Поскольку параметр  $K$  скрыт внутри блока, пользователь может увидеть его, только вызвав окно настройки в режиме редактирования. Сделаем так, чтобы при задержке курсора над блоком значение  $K$  выводилось во всплывающей подсказке – так пользователю будет удобнее работать.

Прежде всего, добавим в модель реакцию на событие вывода всплывающей подсказки. Для этого на левой панели окна редактора выберем вкладку “события” (см. §3.6.4 на стр. 46), раскроем на ней раздел “разное” и дважды щелкнем на его подразделе “всплывающая подсказка (`RDS_BFM_POPUPHINT`)” (рис. 436). При этом значок подраздела станет желтым, а в правой части окна появится новая пустая вкладка “подсказка”. Программа на этой вкладке должна сформировать текст подсказки и передать его в РДС.

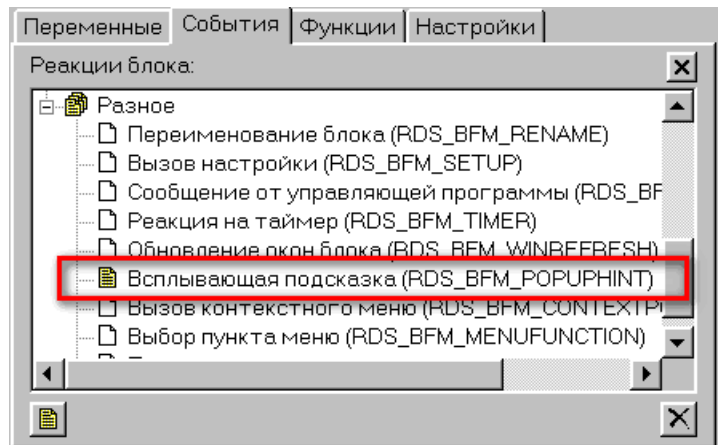


Рис. 436. Вывод подсказки в списке событий

Введем на вкладке “подсказка” следующий текст:

```
char buf[100]; // Здесь будет формироваться текст
// Вывод текста в массив buf
sprintf(buf, "K: %lf", K);
// Передача сформированного текста в РДС
rdsSetHintText(buf);
```

Здесь мы для создания текста со значением параметра используем `sprintf` – стандартную библиотечную функцию языка C, выполняющую форматированный вывод в строку. Она записывает текст подсказки, в котором после буквы “K” и двоеточия выведено значение

параметра K, во вспомогательный массив buf (его размера в сто символов хватит для выполняемого нами вывода текста). Затем вызывается функция rdsSetHintText, которая передает содержимое массива buf в РДС. Теперь, как только реакция модели на событие закончится, на экране возле курсора мыши должна будет появиться подсказка с переданным нами текстом.

Во введенном фрагменте программы все написано верно, однако, если попытаться скомпилировать модель, будет выведено сообщение об ошибке с текстом, подобным “call to undefined function 'sprintf' in...” (“вызов неизвестной функции 'sprintf'...”, конкретный текст сообщения зависит от используемого компилятора). Дело в том, что функция sprintf, которую мы используем, описана в файле заголовков “stdio.h”, а этот файл не включается в программу модели по умолчанию. Мы уже сталкивались с этим в примере из §3.7.2.5 (см. стр. 114). Чтобы пользоваться функцией sprintf, необходимо включить этот файл вручную. Для этого следует на вкладке “события” левой панели редактора (см. §3.6.4 на стр. 46) раскрыть раздел “описания” (это самый первый раздел в списке) и дважды щелкнуть на его подразделе “глобальные описания”, после чего на открывшейся пустой вкладке “описания” в правой части окна ввести единственную строчку: “#include <stdio.h>”. После этого ошибки при компиляции исчезнут.

Теперь в нашей модели есть реакция на вывод всплывающей подсказки, однако, если задержать курсор над блоком, никакой подсказки не появится. Необходимо разрешить блоку выводить подсказку – без этого созданная нами реакция не будет вызвана. Разрешить вывод подсказки можно двумя способами: через окно параметров блока и через окно групповой установки. В нашем случае, поскольку у нас только один блок с такой моделью, первый вариант удобнее: нужно открыть окно параметров блока (пункт “параметры” в его контекстном меню), и на вкладке “DLL” включить флажок “блок выводит всплывающую подсказку” (рис. 437).

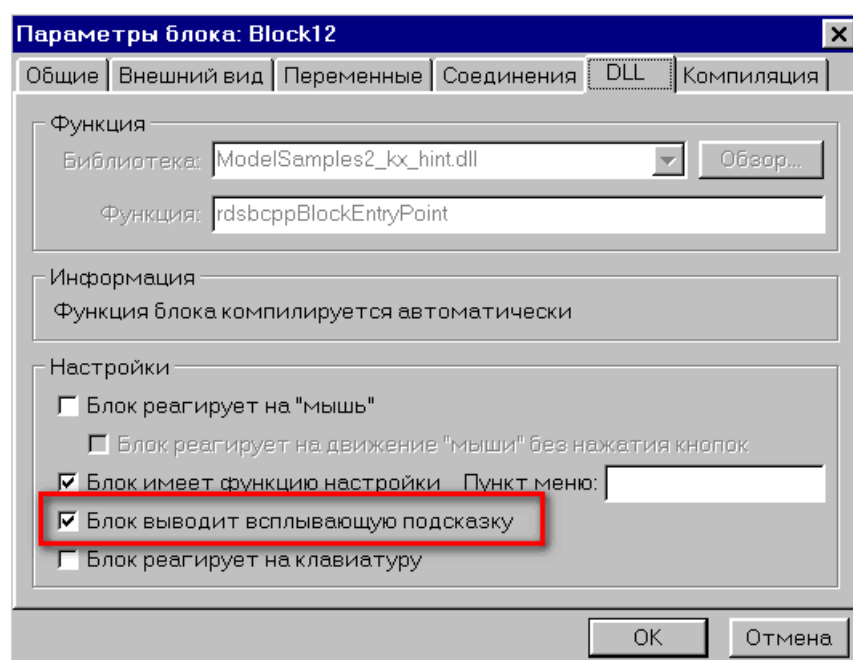


Рис. 437. Разрешение всплывающей подсказки в окне параметров блока

Если бы наша модель была подключена к нескольким блокам, разрешить вывод подсказки удобнее было бы для всех них одновременно через окно групповой установки. Для этого нужно было бы открыть редактор модели и выбрать пункт меню “модель | установка параметров блоков” (см. §3.6.8 на стр. 76). В открывшемся окне следовало бы выбрать

вкладку “DLL”, включить на ней флажок “всплывающая подсказка” и выбрать в выпадающем списке справа от него вариант “есть” (рис. 438).

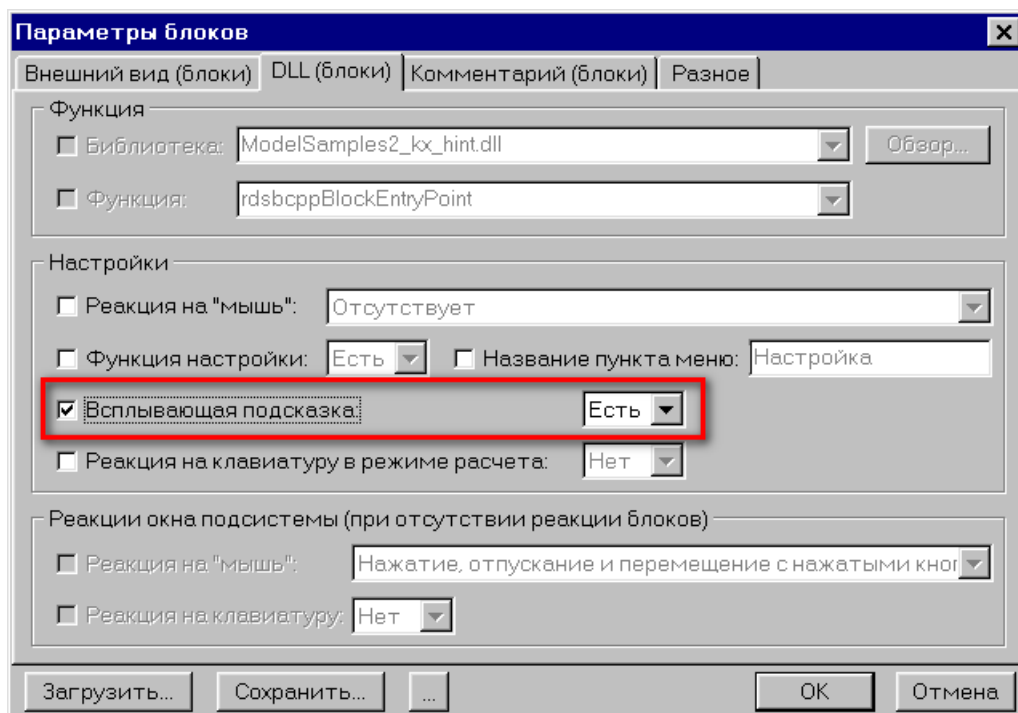


Рис. 438. Разрешение всплывающей подсказки в окне групповой установки

Теперь вывод подсказки разрешен и, задержав курсор над изображением блока, можно увидеть значение множителя  $K$ , введенного в настройках этого блока (рис. 439).

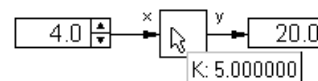


Рис. 439. Всплывающая подсказка к блоку

В нашей подсказке в функции `sprintf` мы использовали формат “%lf”, поэтому в значении  $K$  выведено слишком много незначащих нулей. Мы не можем знать заранее, в каком формате пользователь введет значение в настройках, поэтому не можем заложить в подсказку фиксированное число знаков в дробной части. Однако, мы можем воспользоваться функцией РДС `rdsDtoA`, которая способна сама отбросить незначащие нули – мы уже использовали ее в круглом индикаторе со шкалой из §3.7.5 (см. стр. 190).

Изменим текст программы на вкладке “подсказка” так, чтобы формат вывода  $K$  подбирался автоматически. Старый текст мы полностью сотрем, и введем вместо него следующий:

```
// Преобразование K в строку с подбором формата
char *val=rdsDtoA(K,-1,NULL);
// Сложение строк "K:" и значения K
char *hint=rdsDynStrCat("K: ",val,FALSE);
// Передача текста подсказки в РДС
rdsSetHintText(hint);
// Освобождение динамических строк
rdsFree(val);
rdsFree(hint);
```

В первой строке этой программы мы присваиваем вспомогательной переменной `val` результат возврата функции `rdsDtoA`, в которую мы передаем вещественное число  $K$  и значение `-1` в качестве числа знаков дробной части (функция `rdsDtoA` подробно описана в пункте А.5.4.5 приложения к руководству программиста) – отрицательное число во втором параметре заставит функцию автоматически подобрать для числа формат. `NULL` в третьем параметре передается вместо указателя на целую переменную, куда `rdsDtoA` могла бы

записать длину сформированной строки. Сама строка, указатель на которую возвращается функцией, формируется в динамической памяти, и ее нужно будет потом обязательно освободить вызовом `rdsFree`.

Во второй строке мы, при помощи функции РДС `rdsDynStrCat`, объединяем строку с буквой “К” и двоеточием со строкой значения параметра, которую ранее вернула `rdsDtoA`, и присваиваем ее результат вспомогательной переменной `hint` – это и будет текст нашей подсказки. Функция `rdsDynStrCat`, подробно описанная в пункте А.5.4.6 приложения к руководству программиста, тоже формирует строку-результат в динамической памяти, поэтому эту строку тоже нужно будет освобождать при помощи `rdsFree`. Помимо объединяемых строк в первом и втором параметрах, в третьем параметре `rdsDynStrCat` передается логическое значение, разрешающее или запрещающее функции возвращать `NULL` если обе объединяемые строки пусты. В нашем случае первая строка гарантированно не пуста, поэтому не важно, что передается в третьем параметре (мы передаем `FALSE`).

Далее сформированный текст подсказки передается в РДС уже знакомым нам вызовом `rdsSetHintText`, после чего обе динамических строки (`hint` и `val`) освобождаются вызовами `rdsFree` – они нам больше не нужны.

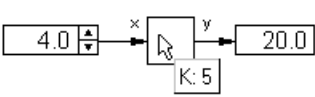


Рис. 440. Всплывающая подсказка к блоку – второй вариант

Новая версия текста подсказки не содержит лишних нулей в дробной части числа (рис. 440). При этом, если пользователь введет дробное значение `K`, в подсказке отобразится столько знаков после десятичной точки, сколько он ввел.

В рассмотренном примере выводилась одна и та же подсказка при попадании курсора мыши в любую точку прямоугольной области окна, занимаемой блоком. Если требуется выводить разные подсказки к разным элементам изображения одного и того же блока, необходимо, во-первых, определять, какой именно элемент находится под курсором, и, во-вторых, ограничить зону действия подсказки размером этого элемента, чтобы при выходе курсора из нее подсказка могла снова появиться уже для другого элемента. Чтобы сделать это возможным, в параметре `HintData` функции реакции на всплывающую подсказку передается указатель на структуру `RDS_POPUPHINTDATA`, при помощи которой модель считывает и устанавливает параметры подсказки. Эта структура подробно описана в §А.2.6.12 приложения к руководству программиста. Многие из ее полей по смыслу совпадают с полями структуры `RDS_DRAWDATA` (см. стр. 175), используемой при рисовании. Кратко перечислим поля структуры и их назначение:

| Поле структуры                                   | Назначение                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>int</b> <code>x, y</code>                     | Координаты курсора мыши на рабочем поле окна подсистемы на момент вывода подсказки (уже с учетом масштаба подсистемы).                                                                                                                                                                                                       |
| <b>int</b> <code>BlockX, BlockY</code>           | Координаты точки привязки блока на рабочем поле с учетом масштаба и возможной связи положения этого блока с переменными. Для блоков с векторной картинкой точка привязки – это положение начала координат этой картинки, для всех остальных – левый верхний угол прямоугольной области.                                      |
| <b>int</b> <code>Left, Top, Width, Height</code> | Координаты левого верхнего угла ( <code>Left, Top</code> ) прямоугольной области, занимаемой блоком, ее ширина ( <code>Width</code> ) и высота ( <code>Height</code> ) в текущем масштабе с учетом возможной связи положения блока с его переменными. Соответствуют одноименным полям структуры <code>RDS_PDRAWDATA</code> . |

| <i>Поле структуры</i>                             | <i>Назначение</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>int</b> HZLeft,<br>HZTop, HZWidth,<br>HZHeight | Возвращаемая моделью зона действия подсказки: горизонтальная (HZLeft) и вертикальная (HZTop) координаты верхнего левого угла зоны, ее ширина (HZWidth) и высота (HZHeight). При выходе курсора мыши за пределы этой прямоугольной зоны будет запрошен повторный вывод подсказки. По умолчанию в этих полях записаны координаты и размер всей прямоугольной области блока, то есть они совпадают с полями Left, Top, Width и Height соответственно, поэтому подсказка для любой точки его изображения будет одной и той же. Модель может записать в эти поля другие значения, чтобы при выходе курсора из указанной зоны можно было вывести другую подсказку. |
| <b>int</b><br>ReshowTimeout                       | Возвращаемое моделью время в миллисекундах после гашения подсказки, по истечении которого подсказку необходимо вывести снова. По умолчанию в этом поле записан ноль – подсказка не будет выведена повторно до тех пор, пока курсор мыши не покинет зону ее действия.                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>int</b> HideTimeout                            | Возвращаемое моделью время в миллисекундах после вывода подсказки, по истечении которого ее необходимо убрать с экрана. По умолчанию в этом поле записано стандартное для Windows значение.                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>int</b> IntZoom                                | Текущий масштаб окна родительской подсистемы блока в процентах (используется крайне редко).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>double</b><br>DoubleZoom                       | Текущий масштаб окна родительской подсистемы блока в долях единицы: 1 – 100%, 0.5 – 50%, 2 – 200% и т.п. Соответствует одноименному полю структуры RDS_PDRAWDATA.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

Помимо указателя HintData, в функцию реакции на вывод подсказки передается также ссылка на целую переменную Show, присвоение ей константы RDS\_BFR\_NOTPROCESSED отменит вывод подсказки. Обычно эта переменная в модели никак не используется, поскольку для того, чтобы отменить вывод подсказки, достаточно просто ничего не передавать в РДС функцией rdsSetHintText.

Рассмотрим пример модели, которая будет выводить разные подсказки к разным частям изображения своего блока. Создадим блок, на входе которого будет матрица вещественных чисел с произвольным количеством строк и двумя столбцами. Каждая строка этой матрицы содержит пару координат точки на плоскости (горизонтальная координата в нулевом столбце и вертикальная в первом), и блок будет рисовать эти точки на своем изображении в виде небольших окружностей, соединяя эти окружности линиями в порядке следования строк матрицы (рис. 441). При наведении курсора мыши на любую из этих точек во всплывающей подсказке должны отображаться номер точки и ее координаты.

Чтобы не усложнять пример, мы не будем закладывать в этот блок настройку отображаемого диапазона координат, цветов и размера окружности точек. Окружности будем рисовать радиусом в три точки для масштаба 100% (как всегда, увеличивая и уменьшая этот радиус вместе с масштабом), а диапазон определим по максимальным и минимальным значениям координат в матрице. Диапазону горизонтальных вещественных координат (то есть минимуму и максимуму значений в нулевом столбце матрицы) будет соответствовать вся ширина блока за вычетом двух радиусов окружности точки (иначе окружности крайних точек выйдут за пределы изображения блока), а диапазону вертикальных (минимуму и максимуму значений в первом столбце матрицы) – вся высота

блока за вычетом тех же двух радиусов. На рис. 441 отображаемый диапазон изображен пунктирной линией внутри блока, на настоящем блоке эта линия рисоваться не будет.

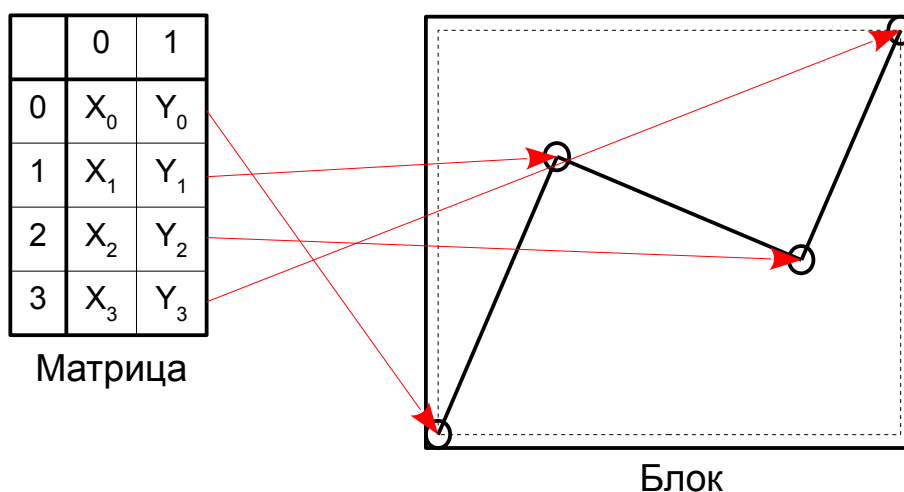


Рис. 441. Предполагаемый внешний вид блока и его связь с входной матрицей

У нашего блока, помимо двух обязательных сигналов, будет единственный вход: матрица вещественных чисел А. Создадим в схеме новый блок, запускающийся по сигналу (см. стр. 95), и зададим ему следующую структуру статических переменных:

| <i>Имя</i> | <i>Тип</i>     | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|----------------|-------------------|-------------|---------------------------|
| Start      | Сигнал         | Вход              | ✓           | 1                         |
| Ready      | Сигнал         | Выход             |             | 0                         |
| A          | Матрица double | Вход              | ✓           | [ ] 0                     |

В модели этого блока будет две реакции: реакция на рисование блока, в которой мы будем рисовать точки и линии между ними, и реакция на всплывающую подсказку, в которой мы будем проверять попадание курсора мыши в одну из точек и выводить ее координаты. Очевидно, в этих реакциях будет много общего: в обеих нужно вычислять координаты точек, для чего нужно будет определять границы отображаемого диапазона, то есть минимальные и максимальные значения координат в матрице. Чтобы не писать в этих реакциях одни и те же вычисления два раза, мы введем в нашу модель дополнительную функцию расчета вспомогательных параметров, которая будет вызываться из обеих реакций. Эту функцию необходимо сделать членом класса блока, поскольку ей необходим доступ к переменной блока А. Параметры, которые она вычислит, тоже сделаем полями класса – так к ним проще будет обращаться из реакций. Таким образом, и функцию, и все вспомогательные параметры мы будем вводить на вкладке описаний внутри класса. Откроем ее: на левой панели окна редактора выберем вкладку “события” (см. §3.6.4 на стр. 46), раскроем на ней раздел “описания” и дважды щелкнем на его подразделе “описания внутри класса блока” (рис. 442). При этом значок подраздела станет желтым, а в правой части окна появится новая пустая вкладка “описания в классе”.



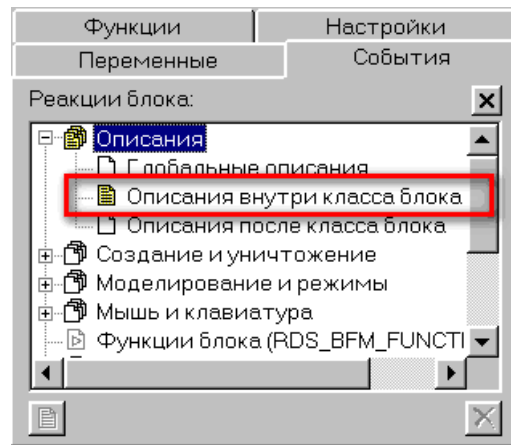


Рис. 442. Описания внутри класса в списке событий

Введем на вкладке следующий текст:

```
// Вспомогательные параметры,
// используемые и при рисовании,
// и при выводе подсказки

// Радиус круга точки с учетом масштаба
int point_r;
// Границы области рисования внутри блока
int draw_x1,draw_y1,draw_x2,draw_y2;
// Диапазон координат в матрице
double xmin,xmax,ymin,ymax;

// Функция вычисления перечисленных выше параметров
void CalcParams(int left,int top, // левый верхний угол блока
 int width,int height, // ширина и высота
 double zoom) // масштаб
{
 // Радиус круга точки
 point_r=zoom*3; // 3 точки с учетом масштаба

 // Зона рисования внутри блока (на радиус круга внутрь от рамки)
 draw_x1=left+point_r+1;
 draw_y1=top+point_r+1;
 draw_x2=left+width-point_r-1;
 draw_y2=top+height-point_r-1;

 // Определение минимума и максимума координат в матрице
 xmin=xmax=A[0][0];
 ymin=ymax=A[0][1];
 for(int i=1;i<A.Rows();i++)
 { if(xmin>A[i][0]) xmin=A[i][0];
 if(xmax<A[i][0]) xmax=A[i][0];
 if(ymin>A[i][1]) ymin=A[i][1];
 if(ymax<A[i][1]) ymax=A[i][1];
 }
 if(xmin==xmax) // Нет горизонтального диапазона
 { // Принудительно создаем его
 xmin-=1; xmax+=1;
 }
}
```

```

 if (ymin==ymax) // Нет вертикального диапазона
 { // Принудительно создаем его
 ymin-=1; ymax+=1;
 }
}

```

В начале этого текста описываются добавляемые нами поля класса для вспомогательных параметров (эти описания станут именно полями класса, а не глобальными переменными, поскольку весь введенный нами текст будет вставлен внутрь описания класса блока). В целом поле `point_r` будет вычисляться радиус окружности точки с учетом текущего масштаба подсистемы. В целых полях `draw_x1` и `draw_y1` будут вычисляться координаты левого верхнего, а в `draw_x2` и `draw_y2` – правого нижнего углов области рисования внутри прямоугольника блока (то есть пунктирной области на рис. 442). Вещественные поля `xmin`, `xmax`, `ymin` и `ymax` будут содержать минимальные и максимальные значения координат из поступившей на вход блока матрицы.

Сразу за описаниями полей записана функция `CalcParams`, которая будет вычислять значения этих полей. В функцию передаются координаты левого верхнего угла (`left`, `top`) и размеры (`width`, `height`) прямоугольной области блока, а также масштабный коэффициент подсистемы `zoom`: в зависимости от того, из какой именно реакции будет вызываться эта функция, эти значения будут браться либо из структуры `RDS_DRAWDATA` (при рисовании), либо из структуры `RDS_POPUPHINTDATA` (при выводе подсказки).

Внутри функции мы сначала вычисляем действительный размер окружности точки `point_r`, умножая 3 (радиус окружности в масштабе 100% мы решили сделать равным трем точкам экрана) на масштабный коэффициент `zoom`. Затем вычисляются размеры области рисования точек внутри блока: из общих размеров блока, переданных в параметрах функции, со всех сторон вычитается по радиусу `point_r` с дополнительным запасом в одну точку. После этого, в цикле перебирая все точки матрицы на входе блока `A`, мы вычисляем диапазоны содержащихся в ней координат `xmin`, `xmax`, `ymin` и `ymax`. Горизонтальные координаты содержатся в нулевом столбце матрицы, то есть в `A[...] [0]`, а вертикальные – в первом столбце, то есть в `A[...] [1]`. Проверка наличия в матрице двух столбцов и, по крайней мере, одной строки будет выполняться внутри реакций на события до вызова функции `CalcParams`, поэтому в самой функции нет никаких проверок – мы просто обращаемся к нулевому и первому столбцу, считая, что они в матрице есть.

После того, как диапазоны координат точек в матрице определены, нужно проверить, не совпадают ли минимальное и максимальное значения по каждой из координат. Для преобразования вещественных координат точки в ее целые координаты на экране нам придется делить значение координаты на размер соответствующего ей диапазона, и, если этот размер будет нулевым, мы не сможем выполнить преобразование. Например, чтобы вычислить горизонтальную координату точки на экране, нужно разделить ее вещественную координату на  $(xmax - xmin)$  и умножить результат на  $(draw\_x2 - draw\_x1)$ , а затем добавить координату левой границы блока – это обычное линейное преобразование диапазонов. При этом, очевидно, значение `xmax` не должно быть равным `xmin`.

Выйти из этого затруднения очень просто: поскольку мы не рисуем на блоке никаких шкал или координатных сеток, при совпадении максимального и минимального значений какой-либо координаты (то есть если данная координата совпадает у всех точек матрицы – они расположены на строго горизонтальной или вертикальной линии) можно принудительно расширить этот диапазон, добавив какую-либо константу к максимальному значению и вычтя ее же из минимального. При этом все точки матрицы окажутся в середине этого диапазона и будут нарисованы в середине изображения блока. Константу можно взять любой – мы будем добавлять и вычитать единицу. Два оператора `if`, расположенных после цикла перебора точек матрицы, выполняют именно эту функцию. Таким образом, на момент

завершения функции CalcParams, диапазоны xmin...xmax и ymin...ymax гарантированно не будут иметь нулевой размер.

Теперь введем в нашу модель реакцию на событие рисования блока. На вкладке “события” левой панели окна редактора раскроем раздел “внешний вид блока” и дважды щелкнем на подразделе “рисование блока (RDS\_BFM\_DRAW)” (см. рис. 412 на стр. 177) – в правой части окна появится новая пустая вкладка “рисование”. Запишем на ней следующую программу:

```
// Вспомогательные переменные
int right,bottom,ix,iy;

// Правый нижний угол области блока
right=DrawData->Left+DrawData->Width;
bottom=DrawData->Top+DrawData->Height;

// Черная линия
rdsXGSetPenStyle(0,PS_SOLID,DrawData->DoubleZoom,0,R2_COPYPEN);
// Белый фон
rdsXGSetBrushStyle(0,RDS_GFS_SOLID,0xffffffff);
// Прямоугольник – черная рамка и белый фон
rdsXGRectangle(DrawData->Left,DrawData->Top,right,bottom);

// Отключаем заливку
rdsXGSetBrushStyle(RDS_GFSTYLE,RDS_GFS_EMPTY,0);

if(A.Cols()<2 || A.Rows()<1) // Меньше двух столбцов или нет строк
 return;

// Вычисляем вспомогательные параметры
CalcParams(DrawData->Left,DrawData->Top,
 DrawData->Width,DrawData->Height,
 DrawData->DoubleZoom);

// Рисуем точки и линии
for(int i=0;i<A.Rows();i++)
{ // Вычисляем оконные координаты точки i
 ix=draw_x1+(A[i][0]-xmin)*(draw_x2-draw_x1)/(xmax-xmin);
 iy=draw_y2-(A[i][1]-ymin)*(draw_y2-draw_y1)/(ymax-ymin);
 // Рисуем окружность радиусом point_r
 rdsXGEllipse(ix-point_r,iy-point_r,ix+point_r,iy+point_r);
 // Соединяем линией с предыдущей точкой
 if(i) rdsXGLineTo(ix,iy);
 else rdsXGMoveTo(ix,iy);
}
```

В этой программе для рисования используются графические функции РДС, уже знакомые нам по §3.7.5 (стр. 173). В ее начале описано несколько вспомогательных переменных, затем располагаются команды установки стиля линии, заливки и рисования прямоугольника размером во весь блок. После их выполнения блок будет выглядеть как белый прямоугольник с черной рамкой, поверх этого прямоугольника мы и будем рисовать точки из матрицы.

Прежде чем приступить к рисованию точек, необходимо проверить, соответствует ли матрица нашим ожиданиям. Чтобы она описывала хотя бы одну точку, в ней должна быть по крайней мере одна строка и два столбца. Если число столбцов в матрице A (A.Cols()), см. §3.7.2.2 на стр. 99) меньше двух, или число строк (A.Rows()) меньше одной, мы немедленно завершаем реакцию – с такой матрицей ничего нельзя нарисовать. В противном случае мы вызываем написанную ранее функцию расчета вспомогательных параметров

CalcParams, передавая ей координаты блока, его размеры и масштаб подсистемы, взятые из структуры RDS\_DRAWDATA по указателю DrawData.

Теперь мы можем пользоваться полями класса, которые заполнила функция CalcParams. В цикле по всем точкам матрицы для каждой точки *i* мы вычисляем экранные координаты (*ix*,*iy*) по ее вещественным координатам (*A[i][0]*,*A[i][1]*), рисуем круг радиусом *point\_r* с центром в точке (*ix*,*iy*), и, если это не самая первая точка матрицы (то есть если *i* не равно нулю), соединяем эту точку с предыдущей вызовом *rdsXGLineTo*. Для самой первой точки вместо *rdsXGLineTo* вызывается функция *rdsXGMoveTo*, которая просто устанавливает координаты точки для последующих вызовов *rdsXGLineTo* (графические функции РДС подробно рассматриваются в §A.5.18 приложения к руководству программиста).

Программа рисования написана – теперь введем в модель реакцию на вывод подсказки. Для этого раскроем на вкладке “события” раздел “разное” и дважды щелкнем на подразделе “всплывающая подсказка (RDS\_BFM\_POPUPHINT)” (см. рис. 436 на стр. 211). В правой части окна появится пустая вкладка “подсказка”, на которой мы введем следующий текст:

```
// Индекс найденной точки
int index=-1;
// Экранные координаты найденной точки
int ix,iy;

if(A.Cols()<2 || A.Rows()<1) // Меньше двух столбцов или нет строк
 return;

// Вычисляем вспомогательные параметры
CalcParams(HintData->Left,HintData->Top,
 HintData->Width,HintData->Height,
 HintData->DoubleZoom);

// Ищем попадание в круг точки
for(int i=0;i<A.Rows();i++)
{ // Вычисляем оконные координаты точки i
 ix=draw_x1+(A[i][0]-xmin)*(draw_x2-draw_x1)/(xmax-xmin);
 iy=draw_y2-(A[i][1]-ymin)*(draw_y2-draw_y1)/(ymax-ymin);
 if(fabs(HintData->x-ix)<=point_r &&
 fabs(HintData->y-iy)<=point_r) // Попали
 { index=i;
 break;
 }
}

if(index>=0) // Нашли точку
{ char *hint,*val_x,*val_y,*val_n;
 // Левый верхний угол зоны подсказки
 HintData->HZLeft=ix-point_r;
 HintData->HZTop=iy-point_r;
 // Размеры зоны подсказки
 HintData->HZWidth=HintData->HZHeight=point_r*2;
 // Задержка гашения подсказки - одна минута
 HintData->HideTimeout=60000;
 // Формирование текста подсказки
 val_n=rdsItoA(index,10,0); // номер точки
 val_x=rdsDtoA(A[index][0],-1,FALSE); // x
 val_y=rdsDtoA(A[index][1],-1,FALSE); // y
 hint=rdsDynStrCat("Точка: ",val_n,FALSE);
```

```

rdsAddToDynStr(&hint, "\nX: ", FALSE);
rdsAddToDynStr(&hint, val_x, FALSE);
rdsAddToDynStr(&hint, "\nY: ", FALSE);
rdsAddToDynStr(&hint, val_y, FALSE);
// Передача текста в РДС
rdsSetHintText(hint);
// Освобождение динамических строк
rdsFree(hint);
rdsFree(val_x);
rdsFree(val_y);
rdsFree(val_n);
}

```

В начале этого текста описаны вспомогательные целые переменные `index`, `ix` и `iy` – в них будут занесены номер точки (строки матрицы) под курсором мыши и ее координаты в окне подсистемы соответственно. Переменная `index` инициализирована значением `-1`: если в процессе перебора точек матрицы ни одна из них не окажется близко к курсору мыши, значение `index` останется отрицательным, и это будет признаком того, что курсор не над точкой матрицы и подсказку выводить не нужно.

Далее мы, как и при рисовании, проверяем размер матрицы на входе блока: если в ней нет хотя бы двух столбцов и одной строки, мы немедленно завершаем реакцию. Если же числа строк и столбцов достаточно для хранения данных об одной точке, мы вычисляем вспомогательные параметры вызовом `CalcParams`. В нее мы передаем координаты блока, его размеры и масштаб подсистемы, на этот раз взятые из структуры `RDS_POPUPHINTDATA` по указателю `HintData`.

Теперь, когда диапазоны координат матрицы и границы области рисования внутри блока вычислены, мы можем проверить, попал ли курсор мыши в одну из точек матрицы `A`. Для этого мы перебираем все ее точки и вычисляем для каждой экранные координаты `(ix, iy)` по вещественным координатам `(A[i][0], A[i][1])` при помощи тех же самых формул, которые использовались в программе рисования. Если расстояния между координатами курсора мыши (`HintData->x`, `HintData->y`) и точкой матрицы `(ix, iy)` и по горизонтали, и по вертикали не больше радиуса круга точки `point_r`, значит, курсор попал внутрь этого круга (точнее, внутрь квадрата, в который вписан этот круг, но для проверки это не принципиально). В этом случае мы присваиваем переменной `index` номер найденной точки и прерываем цикл перебора точек оператором `break`. В переменных `ix` и `iy` при этом останутся координаты этой точки на экране.

После выполнения цикла перебора в переменной `index` останется начальное значение `-1`, если курсор мыши не оказался в пределах круга ни одной из точек. Если же курсор попал в один из кругов, значение `index` будет большим или равным нулю, и нам нужно вывести подсказку, сформировав текст из номера точки (`index`) и двух ее координат (`A[index][0]` и `A[index][1]`). Кроме того, нам нужно ограничить зону действия подсказки данной точкой, чтобы при выходе курсора за ее круг и попадании его в круг другой точки подсказка вывелась бы снова, уже с другими координатами. Этим мы и займемся.

Сначала зададим левый верхний угол зоны действия подсказки – он будет находиться левее и выше самой точки `(ix, iy)` на радиус ее круга `point_r`. В поля `HintData->HZLeft` и `HintData->HZTop` мы записываем `ix-point_r` и `iy-point_r` соответственно. Ширина и высота зоны должны быть равны диаметру, то есть двум радиусам, круга, поэтому в поля `HintData->HZWidth` и `HintData->HZHeight` мы записываем значение `point_r*2`. Теперь зона действия подсказки установлена. В нашей подсказке будет три числа, и, чтобы пользователь гарантированно успел ее прочесть, увеличим время ее показа до одной минуты: в поле `HintData->HideTimeout` мы записываем `60000` (шестьдесят тысяч миллисекунд – это одна минута).

Теперь нужно сформировать текст. Можно воспользоваться для этого стандартной функцией `sprintf`, заранее отведя под формируемую строку буфер заведомо большего размера, но мы будем использовать функции РДС, работающие с динамическими строками. Сначала мы формируем динамические строки из целого номера точки при помощи функции `rdsItoA` (она описана в §A.5.4.11 приложения к руководству программиста). У этой функции три параметра: преобразуемое число, система счисления (мы передаем 10, то есть используем десятичную систему) и минимальное число разрядов в числе (нам не нужно дополнять число ведущими нулями до заданного количества разрядов, поэтому мы передаем ноль). Указатель на динамическую строку, который возвращает функция, мы записываем во вспомогательную переменную `val_n`, потом эту строку обязательно нужно будет освободить вызовом `rdsFree`. Координаты точки мы преобразуем в строки при помощи функции `rdsDtoA`, которая уже встречалась нам ранее (см. стр. 190, подробно эта функция описана в §A.5.4.5 приложения к руководству программиста). Динамические строки с символьным представлением этих координат записываются во вспомогательные переменные `val_x` и `val_y`, их тоже нужно будет освободить функцией `rdsFree`. Далее начинается формирование самого текста подсказки: прежде всего мы объединяем строку “Точка:” и строку `val_n` с номером точки в новую динамическую строку `hint` при помощи функции `rdsDynStrCat` (мы уже использовали ее в этом параграфе выше, на стр. 213). Затем мы по очереди добавляем в конец к этой динамической строке строки “\nX:”, `val_x`, “\nY:” и `val_y` (символы “\n”, как всегда в С, означают перевод строки – текст всплывающей подсказки может состоять из нескольких строк). Добавление осуществляется при помощи функции `rdsAddToDynStr`, описанной в §A.5.4.1 приложения к руководству программиста:

```
rdsAddToDynStr(&hint, // Куда добавляем
 "\nX: ", // Что добавляем
 FALSE); // Заменять пустую строку на NULL?
```

В первом параметре этой функции передается указатель на переменную, в которой хранится указатель на динамическую строку, к которой дописывается указанный текст. В данном случае мы передаем `&hint`, то есть, после последнего вызова `rdsAddToDynStr`, в `hint` будет находиться указатель на полностью сформированный текст подсказки. Этот текст мы передаем в РДС вызовом `rdsSetHintText`, после чего освобождаем более не нужные нам динамические строки `hint`, `val_n`, `val_x` и `val_y` при помощи `rdsFree`.

Теперь в нашей модели есть все необходимые реакции, но нужно еще переключить сам блок на программное рисование и разрешить ему вывод всплывающих подсказок. Проще всего сделать это в окне параметров блока (см. рис. 413 на стр. 182 и рис. 437 на стр. 212), но можно также вызвать групповую установку параметров блоков из редактора модели и включить рисование и подсказки там.

Для проверки работы созданного блока можно собрать схему, изображенную на рис. 443. В ней к входу блока подключен выход стандартного блока ввода матрицы, в который введена матрица из восьми строк и двух столбцов. После запуска расчета при наведении курсора на одну из нарисованных точек в подсказке отображается номер соответствующей этой точке строки матрицы и координаты этой точки.

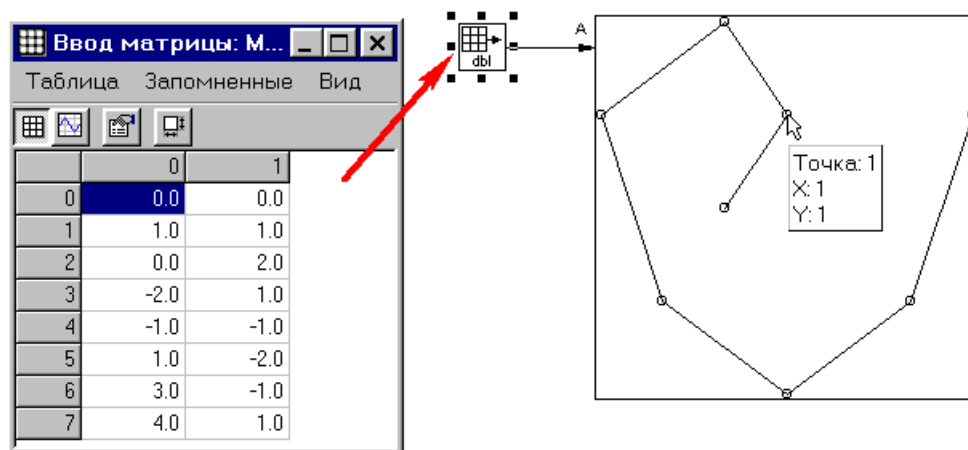


Рис. 443. Тестирование блока с разными подсказками к разным точкам изображения

### §3.7.10. Пометки на блоках

Рассматривается вывод дополнительных изображений поверх блоков, которые могут привлекать внимание пользователя к ошибкам или к неправильно настроенным параметрам.

Независимо от того, каким именно способом задается внешний вид блока в подсистеме (прямоугольником с текстом, векторной картинкой или программно), его модель имеет возможность нарисовать что-либо дополнительное поверх этого изображения. Обычно эта возможность используется для привлечения внимания пользователя в тех случаях, когда блок не может правильно работать – например, из-за недопустимых значений на входе или из-за отсутствия необходимой ему динамической переменной. Если внешний вид блока рисуется программно, в таких дополнительных пометках обычно нет необходимости, поскольку их можно вывести в основной реакции рисования блока. Именно так мы поступали в примерах из §3.7.5 (стр. 173): при отсутствии значений на входах индикатор уровня и стрелочный индикатор рисовались красным цветом. Если же внешний вид блока задается векторной картинкой или прямоугольником с текстом, возможности сообщения пользователю об ошибках сильно ограничены: векторная картинка не может отражать состояние блока в режиме редактирования, а внешний вид прямоугольника с текстом вообще не может быть изменен простыми средствами ни в одном из режимов РДС.

Для рисования дополнительных изображений поверх блока используется специальная реакция его модели: “дополнительное рисование блока”, которая на вкладке “события” находится в разделе “внешний вид блока” вместе с основной реакцией рисования (рис. 444). В отличие от последней, реакция на дополнительное рисование вызывается у всех блоков, независимо от их параметров, поэтому как-либо включать ее не нужно: достаточно ввести реакцию в модель. В остальном эти две реакции очень похожи: в реакции на дополнительное рисование тоже можно использовать как графические функции РДС, так и стандартные функции Windows API, и

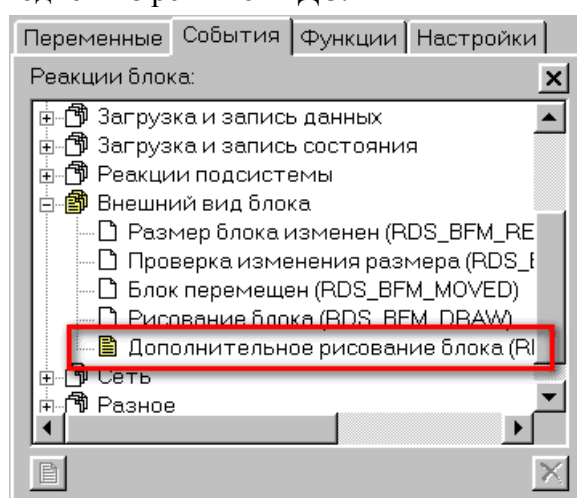


Рис. 444. Дополнительное рисование в списке событий

в нее передается указатель на структуру RDS\_DRAWDATA, уже знакомую нам по §3.7.5 (см. стр. 175).

В качестве примера дополнительного рисования изменим модель одного из двух блоков, рассматривавшихся в §3.7.3.2 (стр. 136). Эти блоки обменивались данными через вещественную динамическую переменную с именем “AmbientTemperature”: блок-передатчик создавал переменную и записывал в нее значение своего входа, а блок-приемник (их могло быть несколько) считывал это значение и передавал его на свой выход. При отсутствии блока-передатчика в схеме блок-приемник никак не сообщал пользователю о невозможности работы – он просто ничего не передавал на выход. Сделаем так, чтобы, обнаружив отсутствие переменной “AmbientTemperature”, блок-приемник рисовал поверх себя желтую иконку с красным восклицательным знаком (так поступают многие стандартные блоки).

По умолчанию в параметрах всех моделей включен запрет работы при отсутствии необходимых динамических переменных (см. рис. 388 на стр. 134), и мы не сможем ничего нарисовать поверх блока: если переменная “AmbientTemperature” не будет существовать, не будет вызвана ни одна из реакций нашей модели, в том числе и реакция дополнительного рисования. Нужно выключить этот запрет и, поскольку теперь мы уже не можем быть уверены в существовании динамической переменной на момент вызова модели, включить в уже имеющуюся реакцию блока необходимые проверки.

Чтобы разрешить модели работать независимо от наличия или отсутствия заданных на вкладке редактора “переменные” динамических переменных, необходимо вызвать окно параметров модели пунктом главного меню окна редактора “модель | параметры модели” и выключить флажок “не реагировать на события без динамических переменных” (рис. 445).

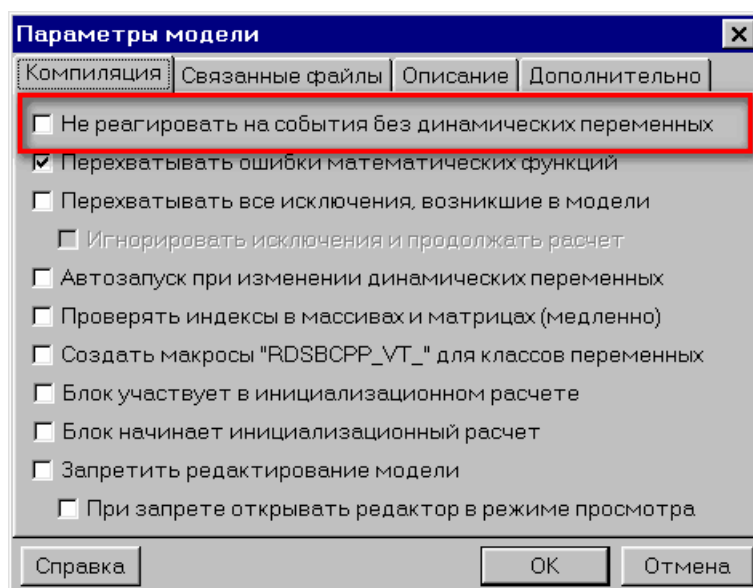


Рис. 445. Выключение запрета работы модели без динамических переменных

Теперь наша модель будет запускаться и при отсутствии “AmbientTemperature”, поэтому в реакцию на изменение динамической переменной необходимо вставить проверку ее существования. Если этого не сделать, при удалении этой переменной (в РДС удаление динамической переменной тоже считается ее изменением) модель попытается обратиться к ней, что вызовет ошибку. Текст на вкладке “изменение динамической” нужно исправить следующим образом (добавленные строки выделены двойным подчеркиванием):

```
if(!AmbientTemperature.Exists()) // Нет переменной
return; // Завершаем реакцию
t=AmbientTemperature;
Ready=1; // Выходные связи должны сработать
```



Теперь в самом начале реакции мы вызываем у объекта `AmbientTemperature`, работающего с одноименной динамической переменной, функцию-член `Exists`, которая возвращает “истину”, только если переменная существует. Если эта функция вернет “ложь”, мы немедленно прерываем выполнение реакции оператором `return` – теперь оператор присвоения выходу `t` значения `AmbientTemperature` выполнится, только если динамическая переменная будет существовать.

Добавим в нашу модель реакцию на дополнительное рисование: на вкладке “события” левой панели окна редактора раскроем раздел “внешний вид блока” и дважды щелкнем на подразделе “дополнительное рисование блока (`RDS_BFM_DRAWADDITIONAL`)” (см. рис. 444 на стр. 223) – в правой части окна появится новая пустая вкладка “доп. рисование”. В ней необходимо ввести следующий текст:

```
if(!AmbientTemperature.Exists()) // Нет переменной
{
 int w,h;
 if(rdsXGGetStdIconSize(RDS_STDICON_YELCIRCEXCLAM,&w,&h))
 rdsXGDrawStdIcon(DrawData->Left+(DrawData->Width-w)/2,
 DrawData->Top+(DrawData->Height-h)/2,
 RDS_STDICON_YELCIRCEXCLAM);
}
```

Как и в реакции на изменение переменной, здесь мы сначала проверяем существование переменной `AmbientTemperature` при помощи функции-члена `Exists`: если переменная существует, ничего рисовать не нужно. В противном случае мы будем рисовать иконку с красным восклицательным знаком.

Иконка, которую мы собираемся рисовать, входит в состав РДС, поэтому проще всего обратиться к ней при помощи встроенных графических функций. Красному восклицательному знаку в желтом круге соответствует константа `RDS_STDICON_YELCIRCEXCLAM`. Чтобы нарисовать эту иконку по центру изображения блока, нужно знать ее размеры – их мы получаем при помощи функции `rdsXGGetStdIconSize` (см. §A.5.18.9 приложения к руководству программиста [2]), которая определяет ширину и высоту иконки, идентификатор которой передан в первом параметре, и помещает их в целые переменные, указатели на которые переданы во втором и третьем – в данном случае, это `w` и `h`. Зная размеры иконки и координаты прямоугольника блока, которые можно взять из структуры `RDS_DRAWDATA`, указатель на которую передается в функцию реакции в параметре `DrawData`, можно вывести эту иконку в центр блока вызовом функции `rdsXGDrawStdIcon` (см. §A.5.18.5 приложения к руководству программиста). Первые два параметра функции – вычисляемые нами горизонтальная и вертикальная координаты верхнего левого угла выводимой иконки, третий – идентификатор стандартной иконки `RDS_STDICON_YELCIRCEXCLAM`.

Теперь, независимо от того, какой внешний вид мы дадим блоку и в каком режиме будет находиться РДС, при отсутствии в схеме динамической переменной “`AmbientTemperature`” поверх нашего блока будет рисоваться восклицательный знак (рис. 446). В этом можно убедиться, удалив из схемы блок, создающий эту переменную – переменная удалится вместе с блоком, и на нашем блоке появится предупреждающая пометка.

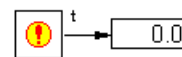


Рис. 446. Иконка поверх блока

Подобную индикацию ошибок поверх блоков имеет смысл сочетать с всплывающими подсказками (см. §3.7.9 на стр. 211), чтобы пользователь, наведя курсор на блок, смог прочесть, какая именно ошибка произошла. Здесь мы не будем добавлять подсказку к блоку, чтобы не усложнять пример.

Следует учитывать, что при включенном избирательном обновлении окон подсистем (см. §2.11.4 части I и §2.10.2 руководства программиста) блоки, внешний вид которых задан как прямоугольник с текстом, автоматически не перерисовываются, и событие

дополнительного рисования у них тоже не возникает. В рассмотренном примере это не важно, поскольку динамическая переменная “AmbientTemperature”, на отсутствие которой блок указывает рисованием иконки, может появиться или исчезнуть только в режиме редактирования из-за вставки или удаления создающего ее блока, а в режиме редактирования избирательное обновление не используется, и все блоки всегда перерисовываются. Когда РДС перейдет в режим расчета, где используется избирательное обновление, иконка уже будет нарисована и ее состояние не изменится.

Если же дополнительные пометки на блоке могут появляться и исчезать в режиме расчета, для нормальной работы избирательного обновления модель блока должна сообщать РДС об изменении этих пометок вызовом функции `rdsForceBlockRedraw`.

### §3.7.11. Реакция блока на мышшь

Рассматривается добавление в модели блоков реакции на нажатие кнопок мыши и перемещение ее курсора, позволяющей создавать различные интерактивные кнопки и рукоятки.

Введение в модель блока реакции на нажатие и отпускание кнопок мыши и перемещение ее курсора – основной способ создания интерактивных блоков: кнопок, рукояток и т.п. Даже блоки, выполняющие ввод с клавиатуры (например, стандартное поле ввода), как правило, активируются щелчком на их изображении, то есть реагируют на нажатие левой кнопки мыши. Следует помнить, что модель блока может реагировать на мышшь только в режимах моделирования и расчета – если разработчику необходимо, чтобы блок взаимодействовал с пользователем еще и в режиме редактирования, необходимо либо вводить какие-либо поля и флаги в окно настройки (см. §3.7.6 на стр. 194), либо дополнять контекстное меню блока (см. §3.7.12 на стр. 238), либо программно создавать собственные

окна и панели (см. §1.8, §2.7.5 и §2.10.4 руководства программиста [1]).

Для реакции на мышшь в РДС предусмотрены события нажатия кнопки, отпускания кнопки, двойного щелчка и перемещения курсора. В редакторе модели все они находятся в группе “мышшь и клавиатура” (рис. 447). Для того, чтобы в ответ на соответствующее действия пользователя вызвалась одна из этих реакций, должны одновременно выполняться следующие условия:

- должен быть включен режим моделирования или расчета;
- курсор мыши должен находиться в пределах изображения блока (то есть внутри его описывающего прямоугольника);
- блок должен находиться на слое, для которого разрешено редактирование;
- в параметрах блока должна быть разрешена реакция на мышшь.

Если эти условия выполнены, при нажатии или отпускании кнопки или при перемещении курсора мыши вызовется реакция модели, в которую через параметр `MouseData` будет передан указатель на структуру `RDS_MOUSEDATA` (см. §A.2.6.8 приложения к руководству программиста [2]), описывающую произошедшее событие. Ниже кратко перечислены поля этой структуры и их назначение:

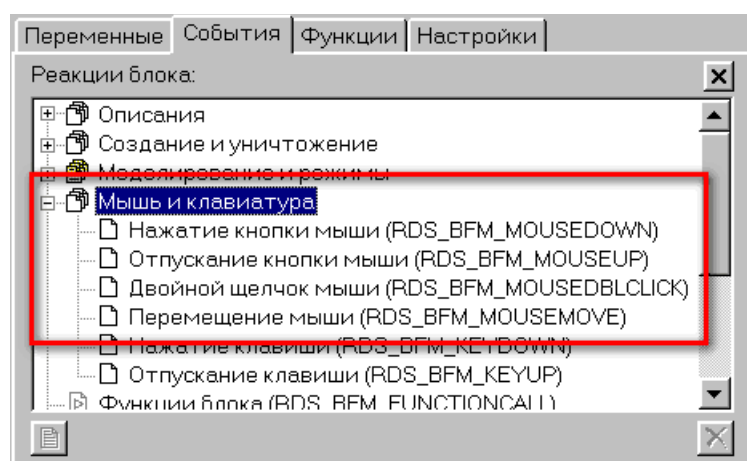


Рис. 447. Реакции на мышшь в списке событий

| <i>Поле структуры</i>               | <i>Назначение</i>                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>int</b> x, y                     | Координаты курсора мыши на рабочем поле окна подсистемы на момент возникновения события (уже с учетом текущего масштаба подсистемы).                                                                                                                                                                                                      |
| <b>int</b> BlockX, BlockY           | Координаты точки привязки блока на рабочем поле с учетом масштаба и возможной связи положения этого блока с переменными. Для блоков с векторной картинкой точка привязки – это положение начала координат этой картинке, для всех остальных – левый верхний угол прямоугольной области.                                                   |
| <b>int</b> Left, Top, Width, Height | Координаты левого верхнего угла (Left, Top) прямоугольной области, занимаемой блоком, ее ширина (Width) и высота (Height) в текущем масштабе с учетом возможной связи положения блока с его переменными.                                                                                                                                  |
| <b>int</b> IntZoom                  | Текущий масштаб окна родительской подсистемы блока в процентах (используется крайне редко).                                                                                                                                                                                                                                               |
| DWORD Button                        | Кнопка мыши, нажатие или отпускание которой вызвало событие: RDS_MLEFTBUTTON – левая кнопка, RDS_MRIGHTBUTTON – правая кнопка, RDS_MMIDDLEBUTTON – средняя кнопка.                                                                                                                                                                        |
| DWORD Shift                         | Набор битовых флагов, описывающие нажатые специальные клавиши клавиатуры и кнопки мыши в момент возникновения события: RDS_MLEFTBUTTON, RDS_MRIGHTBUTTON, RDS_MMIDDLEBUTTON – нажатые кнопки мыши (см. выше), RDS_KSHIFT – нажата клавиша “Shift”, RDS_KALT – нажата клавиша “Alt”, RDS_KCTRL – нажата клавиша “Ctrl”.                    |
| <b>double</b> DoubleZoom            | Текущий масштаб окна родительской подсистемы блока в долях единицы: 1 – 100%, 0.5 – 50%, 2 – 200% и т.п.                                                                                                                                                                                                                                  |
| <b>int</b> MouseEvent               | Константа, указывающая на произошедшее событие (нажатие, отпускание, перемещение курсора и т.п.) Поскольку в автокомпилируемых моделях для каждого события создается независимая функция, в них это поле структуры используется крайне редко. Подробнее о его возможных значениях можно прочесть в приложении к руководству программиста. |
| <b>int</b> Viewport                 | Номер порта вывода (см. §3.6 руководства программиста), из которого пришла информация о событии (нажатии, отпускании кнопки, перемещении курсора и т.п.), или –1, если событие произошло в обычном окне подсистемы.                                                                                                                       |

Параметр `MouseData`, указывающий на структуру описания события – не единственный параметр функций реакции на нажатие и отпускание кнопок мыши и перемещение курсора. Помимо него, в эти функции передается ссылка на целую переменную `Result`, изменяя которую можно повлиять на действия, которые РДС выполнит после того, как реакция сработает. В эту переменную можно записать одну из трех стандартных констант РДС:

- `RDS_BFR_DONE` – блок среагировал на мышшь, никаких дальнейших действий не требуется;
- `RDS_BFR_NOTPROCESSED` – блок отказался реагировать на мышшь, необходимо вызвать реакцию блока, находящегося под ним, если такой есть, или реакцию всей подсистемы;

- RDS\_BFR\_SHOWMENU (только при нажатии правой кнопки) – блок среагировал на мышшь, но, несмотря на это, РДС необходимо вывести обычное контекстное меню блока, появляющееся по правой кнопке.

По умолчанию в переменной Result записана константа RDS\_BFR\_DONE, поэтому, если модель не предпримет никаких специальных действий, блок будет перехватывать все щелчки, пришедшие на его изображение. Чтобы блок, которому разрешена реакция на мышшь, мог становиться “прозрачным” для щелчков и пропускать их к блокам, изображения которых он перекрывает, модель этого блока должна записать в Result константу RDS\_BFR\_NOTPROCESSED. В реакции на нажатие правой кнопки модель может также записать в Result константу RDS\_BFR\_SHOWMENU, информируя РДС о том, что, хотя нажатие обработано, необходимо все равно вывести контекстное меню блока. Это необходимо делать, если блок не обрабатывает щелчки правой кнопкой, потому что в противном случае модель перехватит нажатие любой кнопки и заблокирует меню (пример использования RDS\_BFR\_SHOWMENU будет приведен далее).

Теперь перейдем к примерам моделей блоков, реагирующих на мышшь. Сначала рассмотрим самый простой пример: создадим блок, который по щелчку левой кнопки мыши будет увеличивать свой целый выход на единицу, а по щелчку правой – уменьшать его, тоже на единицу. Создадим в схеме новый блок, запускающийся по сигналу (см. стр. 95), и зададим ему следующую структуру статических переменных:

| Имя   | Тип    | Вход/выход | Пуск | Начальное значение |
|-------|--------|------------|------|--------------------|
| Start | Сигнал | Вход       | ✓    | 0                  |
| Ready | Сигнал | Выход      |      | 0                  |
| y     | int    | Выход      |      | 0                  |

В модели этого блока будет единственная реакция – реакция на нажатие кнопки мыши, в которой мы будем увеличивать или уменьшать переменную y в зависимости от нажатой кнопки. Добавим в модель эту реакцию: на вкладке “события” левой панели окна редактора раскроем раздел “мышшь и клавиатура” (см. рис. 447 на стр. 226) и дважды щелкнем на подразделе “нажатие кнопки мыши (RDS\_BFM\_MOUSEDOWN)”. В правой части окна появится новая пустая вкладка “нажатие кнопки мыши”, в которой необходимо ввести следующий текст:

```
switch(MouseData->Button) // Какая кнопка нажата
{
 case RDS_MLEFTBUTTON: y++; break; // Левая
 case RDS_MRIGHTBUTTON: y--; break; // Правая
}
// Вводим сигнал готовности для передачи выхода по связям
Ready=1;
```

Здесь мы, в зависимости от того, какая именно кнопка мыши нажата, либо увеличиваем, либо уменьшаем y. Идентификатор кнопки мы считываем из поля Button структуры RDS\_MOUSEDATA, переданной через указатель MouseData. После изменения y мы принудительно присваиваем сигналу готовности Ready единицу: у нашего блока нет реакции на такт расчета, в котором готовность взводится автоматически, поэтому, если мы хотим, чтобы изменившееся значение y было передано по связям на входы других блоков после выполнения реакции на нажатие кнопки мыши, необходимо взвести сигнал готовности вручную.

Если запустить расчет прямо сейчас, можно будет увидеть, что при щелчках на изображении блока его выход не изменяется. Дело в том, что, несмотря на то, что мы ввели в модель реакцию на нажатие кнопки мыши, мы не разрешили самому блоку реагировать на мышшь, поэтому наша реакция не вызывается. Поскольку наш блок пока существует в единственном числе, проще всего разрешить реакцию на мышшь в окне его параметров (пункт

“параметры” в его контекстном меню): на вкладке “DLL” этого окна следует включить флажок “блок реагирует на мышь” (рис. 448).

Если бы наша модель была подключена к нескольким блокам, разрешить реакцию на мышь проще было бы через окно групповой установки (см. §3.6.8 на стр. 76), в котором на вкладке “DLL” нужно было бы включить флажок “реакция на мышь” и выбрать в выпадающем списке справа от него вариант “нажатие, отпускание и перемещение с нажатыми кнопками” (рис. 449).

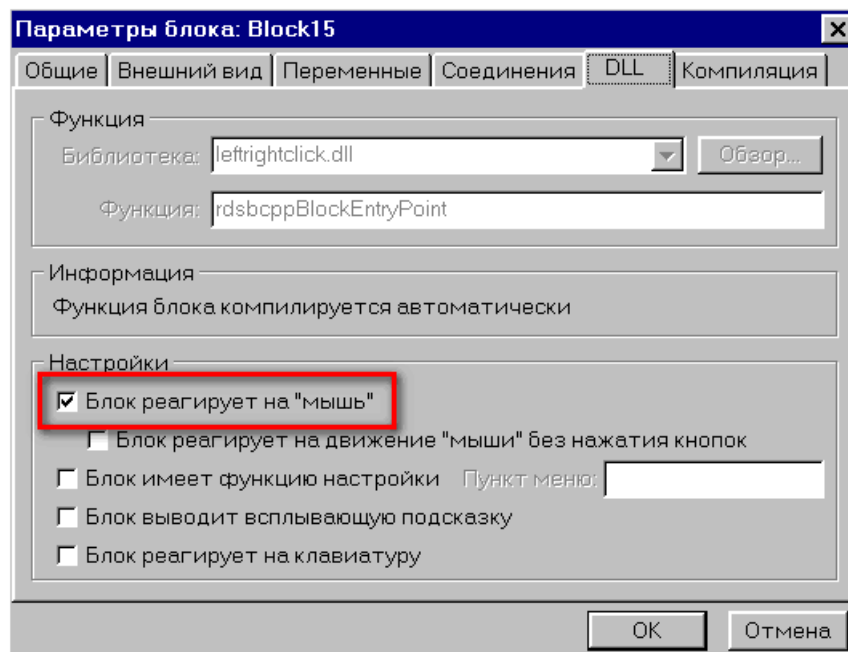


Рис. 448. Разрешение реакции на мышь в окне параметров блока

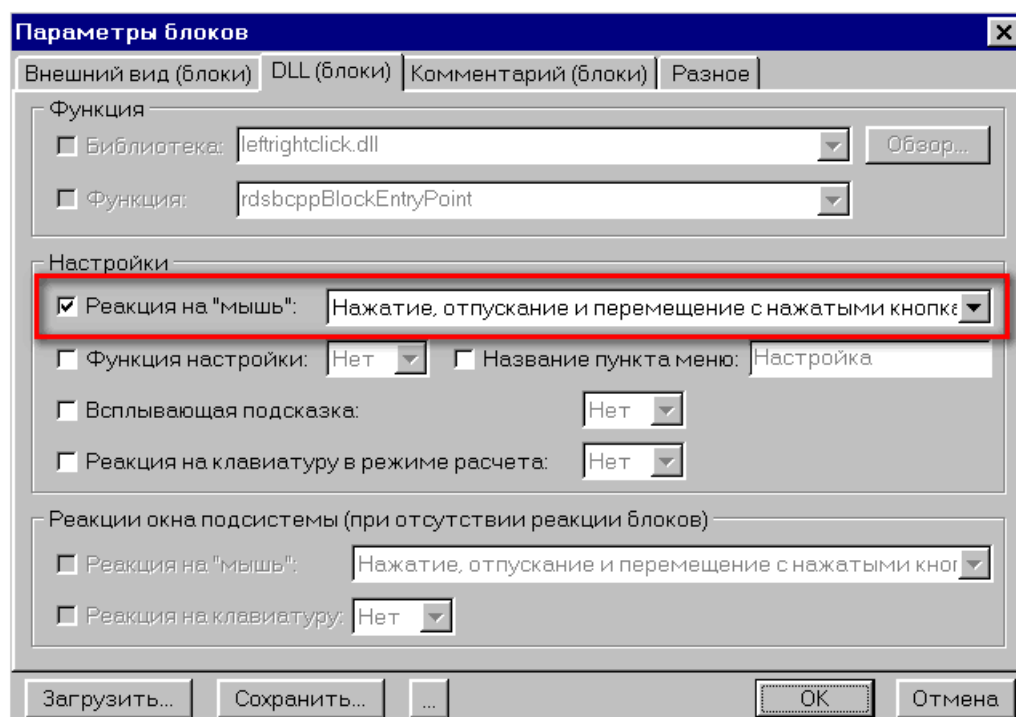


Рис. 449. Разрешение реакции на мышь в окне групповой установки

Для тестирования созданной модели подключим к выходу нашего блока индикатор (рис. 450). Если запустить расчет, щелчки левой кнопки мыши на изображении блока будут

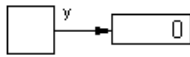


Рис. 450. Тестирование блока, реагирующего на левую и правую кнопки мыши

увеличивать число на индикаторе, а щелчки правой – уменьшать его.

Созданный нами блок очень примитивен – выполняемые им действия жестко привязаны к кнопкам мыши. Если блок выполняет несколько действий, чаще всего их привязывают к различным частям изображения

блока: например, на блоке может быть нарисовано несколько кнопок, и модель может по-разному реагировать на их нажатие. Чтобы определить, в какой именно части изображения блока находится курсор мыши в момент нажатия кнопки, модель может сравнивать координаты курсора из полей *x* и *y* структуры структуры *RDS\_MOUSEDATA*, передаваемой ей через указатель *MouseData* (см. стр. 226), с заранее вычисленными координатами элементов изображения (координаты всей прямоугольной области блока можно считать из полей *Left*, *Top*, *Width* и *Height* той же структуры). Если блок изображается векторной картинкой, можно поступить проще: присвоить различным элементам картинки уникальные целые идентификаторы, а в момент нажатия кнопки мыши запрашивать у РДС идентификатор элемента картинки под курсором. Так можно достаточно легко создавать на изображении блока области, чувствительные к нажатию.

Создадим новый блок, который также будет увеличивать и уменьшать свой целый выход *y*, но уже в зависимости от того, на какой части картинки пользователь щелкнул мышью. Кроме уменьшения и увеличения предусмотрим также обнуление выхода. Создадим в схеме новый блок, запускающийся по сигналу (см. стр. 95), и зададим ему точно такую же, как и у предыдущего блока, структуру статических переменных:

| Имя   | Тип    | Вход/выход | Пуск | Начальное значение |
|-------|--------|------------|------|--------------------|
| Start | Сигнал | Вход       | ✓    | 0                  |
| Ready | Сигнал | Выход      |      | 0                  |
| y     | int    | Выход      |      | 0                  |

Теперь временно закроем редактор модели, сохранив при этом сделанные в структуре переменных изменения, откроем окно параметров блока и вызовем редактор векторной картинки (см. §2.10 части I). Добавим в картинку три квадрата: красный – слева, белый – в центре и зеленый – справа. Красному квадрату дадим идентификатор 1, зеленому – 2, а белому – 3 (рис. 451).

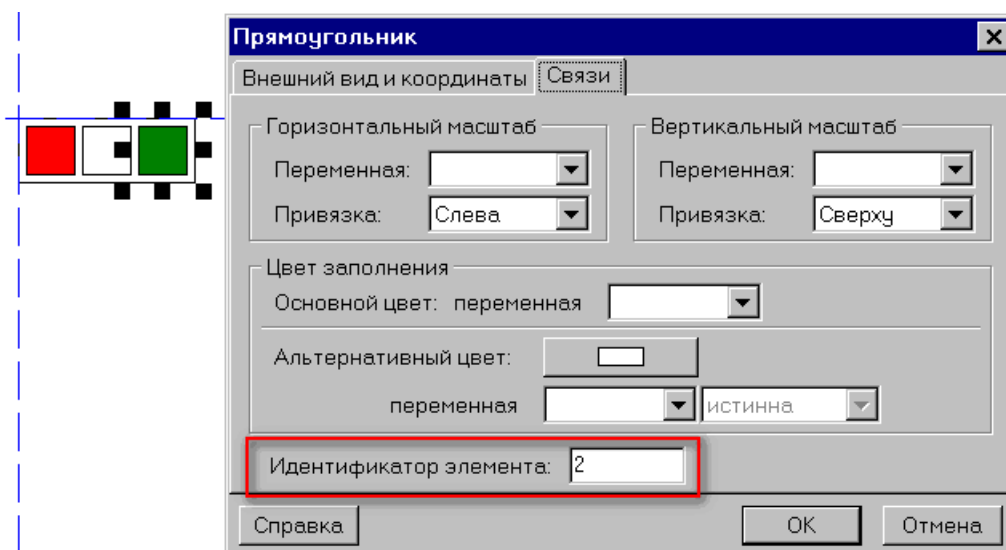


Рис. 451. Идентификатор одного из элементов в редакторе картинки

Разрешим новому блоку реакцию на мышь (см. рис. 448 на стр. 229) и введем в модель реакцию на нажатие кнопки мыши, в которой мы будем выяснять, на какой из квадратов нажал пользователь. Снова откроем редактор модели, на вкладке “события” левой панели окна редактора раскроем раздел “мышь и клавиатура” (см. рис. 447 на стр. 226), дважды щелкнем на подразделе “нажатие кнопки мыши”, и на появившейся справа одноименной вкладке введем следующий текст:

```
if(MouseData->Button==RDS_MLEFTBUTTON) // Нажата левая
{ switch(rdsGetMouseObjectId(MouseData)) // Идентификатор
{ case 1: y--; break; // Красный квадрат
case 2: y++; break; // Зеленый квадрат
case 3: y=0; break; // Белый квадрат
}
// Вводим сигнал готовности для передачи выхода по связям
Ready=1;
}
else // Нажата не левая
Result=RDS_BFR_SHOWMENU; // Разрешаем контекстное меню
```

Сначала мы выясняем, левая ли кнопка нажата, сравнивая поле `Button` структуры описания события с константой `RDS_MLEFTBUTTON`. Если это так, мы запрашиваем у РДС идентификатор элемента картинки под курсором мыши при помощи функции `rdsGetMouseObjectId` (см. §A.5.6.31 приложения к руководству программиста), в которую передается указатель `MouseData` на структуру описания события `RDS_MOUSEDATA`. Функция вернет идентификатор, присвоенный нами элементу под курсором в редакторе векторной картинки, и, в зависимости от его значения, мы либо увеличим значение `y`, либо уменьшим его, либо присвоим ему ноль. Затем, как и в прошлом примере, мы вручную взводим сигнал готовности блока `Ready`, чтобы новое значение `y` передалось по связям. Если курсор будет находиться в пределах изображения блока, но не над одним из тех элементов, которым мы присвоили идентификаторы, `rdsGetMouseObjectId` вернет нулевое значение, и мы не выполним в реакции никаких действий над `y`.

Если же была нажата не левая кнопка, мы присваиваем переменной `Result` значение `RDS_BFR_SHOWMENU`, чтобы не блокировать вывод контекстного меню по правой кнопке мыши. Если мы не сделаем этого, РДС будет считать, что щелчок правой кнопкой мыши перехвачен моделью блока, и меню выведено не будет.

Для тестирования созданной модели, как и в прошлом примере, подключим к выходу блока индикатор (рис. 452). Если запустить расчет, щелчки левой кнопки мыши на левом квадрате картинки будут увеличивать число на индикаторе, на правом – уменьшать его, а на центральном – сбрасывать его в ноль.



Рис. 452. Тестирование блока, запрашивающего идентификатор элемента картинки под курсором

При создании различных блоков-рукояток необходимо отслеживать не только нажатие кнопки мыши, как в уже рассмотренных примерах, но и ее отпускание и перемещение курсора. При этом следует учитывать, что по умолчанию на перемещение курсора будет реагировать только тот блок, над изображением которого этот курсор находится. При создании рукояток это не очень удобно: если пользователь, нажав кнопку мыши на блоке-рукоятке и перемещая эту рукоятку, случайно выведет курсор за пределы блока, рукоятка перестанет перемещаться. Или, что еще хуже, курсор может попасть на изображение другого блока, тоже реагирующего на перемещения курсора. Если пользователь при этом следит за индикатором, связанным с выбранной им рукояткой, он может не заметить, что в системе начал изменяться какой-то другой параметр, связанный с блоком, в который случайно попал курсор.



Чтобы избежать описанных проблем, обычно при создании различных рукояток используют так называемый “захват мыши”. Реагируя на нажатие кнопки мыши, модель включает этот захват, после чего, независимо от того, где окажется курсор, реакция на его перемещение будет вызываться только у модели захватившего блока. При отпускании кнопки мыши модель выключает захват, и реакция на перемещение, как обычно, снова начинает вызываться только у блоков под курсором.

Рассмотрим пример модели блока, захватывающего мышью и реагирующего на перемещение курсора. Создадим блок, изображающий двухкоординатную рукоятку (рис. 453) – этот же пример рассматривается в §2.12.2 руководства программиста.

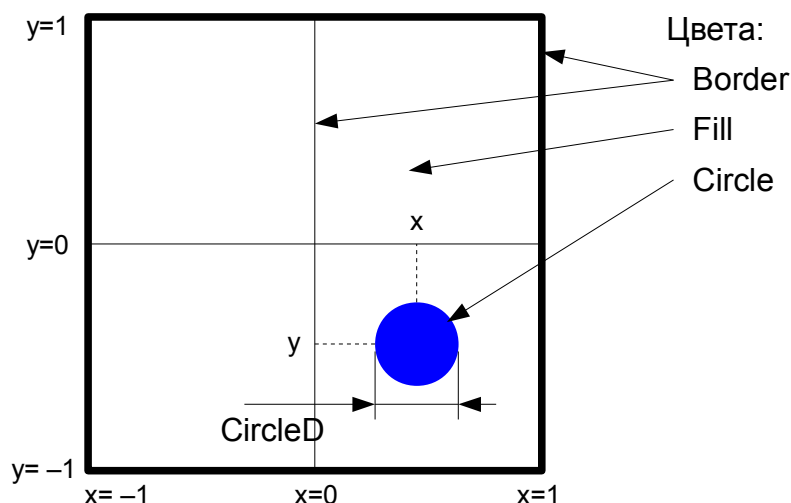


Рис. 453. Предполагаемый внешний вид блока-рукоятки и его параметры

Блок будет представлять собой прямоугольник с перекрестием по центру, внутри которого пользователь сможет двигать мышью закрашенный круг, нажав на этом круге левую кнопку и перемещая его в желаемое положение. Выходами блока будут вещественные переменные  $x$  и  $y$ , соответствующие положению центра круга внутри прямоугольника. Перемещение центра круга по горизонтали от левой границы прямоугольника до правой меняет  $x$  от  $-1$  до  $1$ , перемещение круга по вертикали от нижней границы до верхней меняет  $y$  от  $-1$  до  $1$ . Совпадение центра круга с перекрестием соответствует точке  $x=0$ ,  $y=0$ . Рисовать круг в прямоугольнике мы будем программно, причем нужно будет следить за тем, чтобы при приближении круга к границе его часть, выходящая за прямоугольник, не рисовалась. Цвета блока и размер круга мы сделаем настроечными параметрами блока: диаметр круга в точках экрана будет храниться в целом параметре `CircleD`, цвет рамки и перекрестия – в параметре `Border`, цвет фона прямоугольника – в параметре `Fill`, цвет круга – в параметре `Circle` (все цвета будут иметь стандартный для Windows тип `COLORREF`). Для лучшей визуальной обратной связи сделаем так, чтобы в процессе перетаскивания круг изображался другим цветом – будем хранить его в параметре `CircleMoving`.

Создадим в схеме новый блок, запускающийся по сигналу (см. стр. 95) – нам не нужно, чтобы он работал каждый такт – и зададим ему следующую структуру статических переменных:



| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| x          | double     | Выход             |             | 0                         |
| y          | double     | Выход             |             | 0                         |

В редакторе модели на вкладке “настройки” добавим следующие настроечные параметры, одновременно создавая для них поля ввода (см. §3.7.6 на стр. 194):

| <i>Имя</i>   | <i>Тип</i> | <i>По умолчанию</i> | <i>Заголовок поля ввода</i> | <i>Тип поля ввода</i>                        |
|--------------|------------|---------------------|-----------------------------|----------------------------------------------|
| Border       | COLORREF   | 0                   | Рамка                       | Выбор цвета                                  |
| Fill         | COLORREF   | 0xffffffff          | Заливка                     | Выбор цвета                                  |
| Circle       | COLORREF   | 0xff0000            | Круг                        | Выбор цвета                                  |
| CircleMoving | COLORREF   | 0xff                | Круг в движении             | Выбор цвета                                  |
| CircleD      | int        | 20                  | Диаметр круга               | Ввод (+ / –), шаг 1, минимум 0, максимум 100 |

В качестве заголовка окна настройки введем “рукоятка”. В результате вкладка “настройки” окна редактора и окно настройки блока (его можно увидеть, нажав кнопку “тест” под списком полей ввода) будут выглядеть так, как на рис. 454.

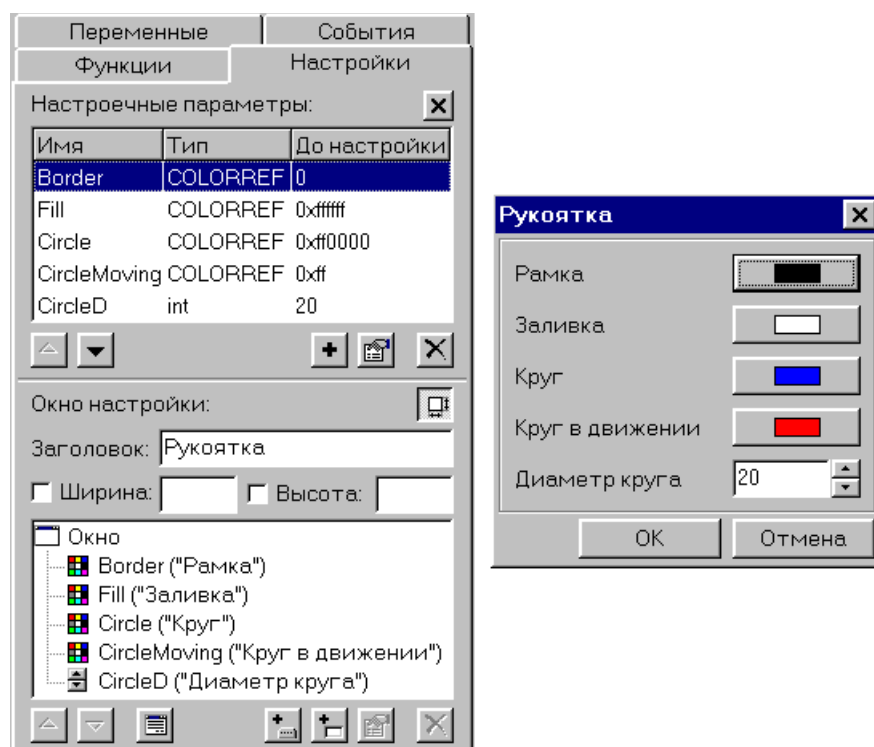


Рис. 454. Настроечные параметры и тест окна настройки рукоятки

Теперь добавим в модель функцию рисования. На левой панели окна редактора следует выбрать вкладку “события”, раскрыть на ней раздел “внешний вид блока” и дважды щелкнуть на его подразделе “рисование блока (RDS\_BFM\_DRAW)” (см. рис. 412 на

стр. 177). При этом значок подраздела станет желтым, а в правой части окна появится новая пустая вкладка “рисование”. На этой вкладке необходимо ввести следующий текст:

```
// Вспомогательные переменные
int hx,hy,cx,cy; RECT r;
int hR=CircleD*DrawData->DoubleZoom/2; // Радиус круга-рукоятки
// Мышь захвачена блоком?
BOOL captured=rdsbcppBlockData->Flags & RDS_MOUSECAPTURE;

// Если размер блока - нулевой, рисовать негде
if(DrawData->Height==0 || DrawData->Width==0)
 return;

// Рисование поля блока
rdsXGSetPenStyle(0,PS_SOLID,1,Border,R2_COPYPEN);
rdsXGSetBrushStyle(0,RDS_GFS_SOLID,Fill);
rdsXGRectangle(DrawData->Left,DrawData->Top,
 DrawData->Left+DrawData->Width,
 DrawData->Top+DrawData->Height);

// Вычисление центра прямоугольника блока
cx=DrawData->Left+DrawData->Width/2;
cy=DrawData->Top+DrawData->Height/2;

// Вычисление координат центра круга-рукоятки
hx=cx+x*DrawData->Width/2;
hy=cy-y*DrawData->Height/2;

// Установка области отсечения
r.left=DrawData->Left+1;
r.top=DrawData->Top+1;
r.right=DrawData->Left+DrawData->Width-1;
r.bottom=DrawData->Top+DrawData->Height-1;
rdsXGSetClipRect(&r);

// Рисование перекрестия
rdsXGMoveTo(cx,DrawData->Top);
rdsXGLineTo(cx,DrawData->Top+DrawData->Height);
rdsXGMoveTo(DrawData->Left,cy);
rdsXGLineTo(DrawData->Left+DrawData->Width,cy);

// Рисование круга (цвет зависит от признака захвата мыши)
rdsXGSetPenStyle(RDS_GFSTYLE,PS_NULL,0,0,0);
rdsXGSetBrushStyle(0,RDS_GFS_SOLID,
 captured?CircleMoving:Circle);
rdsXGEllipse(hx-hR,hy-hR,hx+hR+1,hy+hR+1);

// Отмена отсечения
rdsXGSetClipRect(NULL);
```

Чтобы круг, который перетаскивает пользователь, увеличивался и уменьшался вместе с масштабом подсистемы, мы вычисляем радиус рисуемого круга `hR` как половину произведения настроечного параметра `CircleD` и масштабного множителя подсистемы `DrawData->DoubleZoom`. На время перетаскивания круга мы будем захватывать мышь, поэтому признаком движения круга можно считать наличие захвата мыши. Для захвата мыши блок взводит в поле `Flags` структуры данных блока `RDS_BLOCKDATA` (см. стр. 88) битовый флаг `RDS_MOUSECAPTURE`. Структура данных блока доступна во всех реакциях модели через указатель `rdsbcppBlockData`. Таким образом, если флаг взведен, то есть

если выражение `rdsbcppBlockData->Flags & RDS_MOUSECAPTURE` не равно нулю, то блок в данный момент захватил мышь. В процедуре рисования это будет означать, что круг нужно рисовать цветом `CircleMoving` вместо `Circle`. Полученный таким образом признак захвата записывается во временную логическую переменную `captured`.

Далее размеры блока сравниваются с нулем (на блоке нулевой ширины или высоты бессмысленно что-либо рисовать), устанавливается цвет рамки `Border` и цвет заливки `Fill` и рисуется прямоугольник размером во весь блок (программное рисование подробно рассматривается в §3.7.5 на стр. 173). Затем вычисляется центр изображения блока (`cx`, `cy`) и координаты центра круга (`hx`, `hy`), соответствующие текущим значениям вещественных выходов `x` и `y`.

Теперь необходимо ограничить рисование внутренней областью прямоугольника, чтобы нарисованный нами круг не вышел за габариты блока, даже если пользователь подтащит центр этого круга к самой границе. Для ограничения рисования заданной прямоугольной областью используется функция `rdsXGSetClipRect` (см. §A.5.18.2 приложения к руководству программиста). Сначала мы заносим во вспомогательную структуру `r` стандартного для Windows типа `RECT` координаты прямоугольной области, лежащей внутри прямоугольника блока с отступом в одну точку экрана, а затем передаем указатель на эту структуру в `rdsXGSetClipRect`. Начиная с этого момента Windows будет автоматически отсекал части нарисованных изображений, выходящие за пределы прямоугольника `r`.

Теперь можно нарисовать линии перекрестия с центром в (`cx`, `cy`) и круг радиуса `hR` с центром в (`hx`, `hy`). Цвет круга зависит от вспомогательной переменной `captured`: если мышь захвачена блоком (то есть круг в данный момент перетаскивается пользователем), он будет нарисован цветом `CircleMoving`, в противном случае – цветом `Circle`. Нарисовав все, что нужно, мы отменяем ранее установленное ограничение рисования, вызвав функцию `rdsXGSetClipRect` с параметром `NULL`.

Программное рисование блока написано – теперь нужно заняться реализацией перетаскивания круга. Перетаскивание будет устроено следующим образом:

- в момент нажатия левой кнопки мыши на изображении круга модель будет захватывать мышь и запоминать текущие координаты центра нарисованного круга и координаты курсора мыши на момент нажатия;
- при каждом перемещении курсора модель будет вычислять новые координаты центра круга (для этого нам будут нужны запомненные при нажатии кнопки координаты), а по ним – новые вещественные значения `x` и `y`;
- при отпускании кнопки модель снимет захват мыши.

Начнем с того, что добавим в класс блока поля, в которых мы будем запоминать координаты курсора и координаты центра круга на момент начала перетаскивания. На левой панели окна редактора выберем вкладку “события”, раскроем на ней раздел “описания” и дважды щелкнем на его подразделе “описания внутри класса блока” (см. рис. 442 на стр. 217). В правой части окна появится новая пустая вкладка “описания в классе”, на которой нужно ввести следующий текст:

```
// Центр круга (рукоятки) до начала перетаскивания
int OldCircleX, OldCircleY;
// Координаты курсора на момент начала перетаскивания
int OldMouseX, OldMouseY;
```

Перед перетаскиванием круга в полях `OldCircleX` и `OldCircleY` мы будем сохранять координаты его центра, а в полях `OldMouseX` и `OldMouseY` – координаты курсора мыши. Начиная перетаскивание, пользователь, вероятнее всего, нажмет не точно в центр круга, поэтому нам нужно запомнить две пары координат. В процессе перетаскивания положения центра круга относительно курсора мы будем сохранять постоянным.

Теперь, как в предыдущих примерах, добавим в модель реакцию на нажатие кнопки мыши. На вкладке “события” раскроем раздел “мышь и клавиатура” (см. рис. 447 на стр. 226), дважды щелкнем на подразделе “нажатие кнопки мыши” и введем на открывшейся вкладке следующий текст:

```
// Вспомогательные переменные
int hx,hy,cx,cy,hR;
hR=CircleD*MouseData->DoubleZoom/2; // Радиус круга

// Если размер - нулевой, реакция не имеет смысла
if (MouseData->Height==0 || MouseData->Width==0)
 return;
// Если нажата не левая кнопка, перетаскивать не надо
// Разрешаем в этом случае вызов контекстного меню блока
if (MouseData->Button!=RDS_MLEFTBUTTON)
{
 Result=RDS_BFR_SHOWMENU;
 return;
}

// Координаты центра блока
cx=MouseData->Left+MouseData->Width/2;
cy=MouseData->Top+MouseData->Height/2;
// Координаты центра круга-рукоятки
hx=cx+x*MouseData->Width/2;
hy=cy-y*MouseData->Height/2;

// Проверка попадания курсора в круг
if (abs(MouseData->x-hx)<=hR && abs(MouseData->y-hy)<=hR)
{
 // Курсор попал в круг
 // Запоминаем координаты центра круга на момент
 // начала перетаскивания
 OldCircleX=hx;
 OldCircleY=hy;
 // Координаты курсора на начало перетаскивания
 OldMouseX=MouseData->x;
 OldMouseY=MouseData->y;
 // Вводим флаг захвата мыши
 rdsbcppBlockData->Flags|=RDS_MOUSECAPTURE;
}
```

Здесь, как и в функции рисования, мы сначала вычисляем радиус круга (он нам нужен для проверки попадания курсора мыши в круг) и прерываем выполнение реакции, если один из размеров блока – нулевой. Затем, как в предыдущем примере, мы выясняем, левая ли клавиша мыши нажата. Если нажата не левая клавиша, мы записываем в переменную Result константу RDS\_BFR\_SHOWMENU, чтобы разрешить РДС вывод контекстного меню, и завершаем реакцию. В противном случае, опять, как в функции рисования, вычисляются координаты центра изображения блока (cx, cy) и координаты центра круга (hx, hy), соответствующие текущим значениям x и y.

Далее мы проверяем попадание курсора в круг – точнее, в прямоугольник, внутрь которого вписан этот круг. Если по обеим координатам расстояние между курсором (поля x и y структуры MouseData) и текущим центром круга (hx, hy) не больше радиуса круга hR, значит, курсор попал внутрь круга. При этом мы запоминаем координаты курсора в полях OldMouseX и OldMouseY, которые мы уже добавили в класс блока, а координаты центра круга – в полях OldCircleX и OldCircleY. Затем мы взводим флаг захвата мыши RDS\_MOUSECAPTURE в поле Flags структуры данных блока rdsbcppBlockData при помощи оператора присваивания с битовым “ИЛИ”:

```
rdsbcppBlockData->Flags|=RDS_MOUSECAPTURE;
```

Вся основная работа будет выполняться в реакции на перемещение курсора мыши, но сначала добавим в модель реакцию на отпускание кнопки – в ней нам нужно только снять захват мыши. На вкладке “события” в разделе “мышь и клавиатура” (см. рис. 447 на стр. 226) дважды щелкнем на подразделе “отпускание кнопки мыши” и введем на открывшейся одноименной вкладке единственную строчку:

```
RDS_SETFLAG(rdsbcppBlockData->Flags,RDS_MOUSECAPTURE,FALSE);
```

Здесь мы используем стандартный макрос РДС RDS\_SETFLAG (см. §A.5.2.4 приложения к руководству программиста) для очистки флага RDS\_MOUSECAPTURE в поле Flags структуры данных блока. Можно было бы, не пользуясь макросом, записать

```
rdsbcppBlockData->Flags=
(DWORD)(rdsbcppBlockData->Flags & (~RDS_MOUSECAPTURE));
```

однако, запись с макросом несколько компактнее.

Теперь займемся реакцией на перемещение курсора мыши. На вкладке “события” в разделе “мышь и клавиатура” дважды щелкнем на подразделе “перемещение мыши” и на открывшейся одноименной вкладке введем следующий текст:

```
// Вспомогательные переменные
```

```
int hx,hy,cx,cy;
```

```
// Если размер - нулевой, реакция не имеет смысла
```

```
if(MouseData->Height==0 || MouseData->Width==0)
```

```
{ x=y=0.0;
```

```
return;
```

```
}
```

```
if(!(rdsbcppBlockData->Flags & RDS_MOUSECAPTURE))
```

```
{ // Мышь не захвачена - ничего не нужно делать
```

```
return;
```

```
}
```

```
// Новые координаты центра рукоятки
```

```
hx=OldCircleX+(MouseData->x-OldMouseX);
```

```
hy=OldCircleY+(MouseData->y-OldMouseY);
```

```
// Координаты центра блока
```

```
cx=MouseData->Left+MouseData->Width/2;
```

```
cy=MouseData->Top+MouseData->Height/2;
```

```
// По новым координатам центра рукоятки вычисляем соответствующие
```

```
// им вещественные значения выходов, ограничивая их
```

```
// диапазоном [-1...1]
```

```
x=2.0*(hx-cx)/MouseData->Width;
```

```
if(x>1.0) x=1.0;
```

```
else if(x<-1.0) x=-1.0;
```

```
y=-2.0*(hy-cy)/MouseData->Height;
```

```
if(y>1.0) y=1.0;
```

```
else if(y<-1.0) y=-1.0;
```

```
// Вводим сигнал готовности для передачи выхода по связям
```

```
Ready=1;
```

Мы снова проверяем размеры блока и завершаем реакцию, обнуляя выходы, если его ширина или высота равны нулю. Затем мы проверяем флаг захвата мыши: если он не взведен, значит, была нажата не левая кнопка, или при нажатии кнопки курсор был за пределами круга – в общем, перетаскивание не было начато. В этом случае мы немедленно завершаем реакцию.

Если мышь захвачена, то есть перетаскивание круга выполняется, мы вычисляем новое положение центра круга с учетом перемещения курсора: курсор с момента нажатия

кнопки мыши переместился из (OldMouseX, OldMouseY) в (MouseData->x, MouseData->y), значит, центр круга сместился на ту же величину от (OldCircleX, OldCircleY). Зная новое положение центра круга внутри прямоугольника блока, мы можем вычислить новые значения вещественных выходов  $x$  и  $y$ . Эти значения мы ограничиваем диапазоном  $[-1...1]$ , то есть границами блока. Вычислив  $x$  и  $y$ , мы, как и в прошлых примерах, взводим сигнал готовности блока Ready, чтобы значения выходов передались по связям.

Модель блока-рукоятки полностью написана. Теперь, чтобы он смог работать, необходимо правильно настроить его параметры:

- чтобы включить программное рисование, необходимо в окне параметров блока на вкладке “внешний вид” включить флажок “внешний вид – определяется функцией DLL” (см. рис. 411 на стр. 174);
- чтобы пользователь мог задать размеры блока, растягивая мышью маркеры его выделения, на этой же вкладке следует включить флажок “разрешить масштабирование”;
- чтобы модель блока вызывалась при нажатии и отпускании кнопок мыши и перемещении курсора, на вкладке “DLL” окна параметров следует включить флажок “блок реагирует на мышь” (см. рис. 448 на стр. 229).

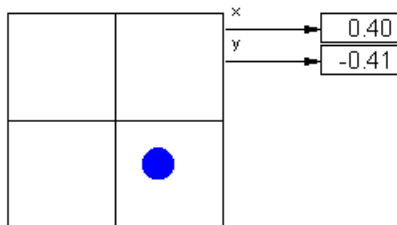


Рис. 455. Тестирование рукоятки

Теперь наш блок полностью готов к работе. Сделаем его достаточно большим, чтобы было удобно перетаскивать круг внутри него, и подключим к его выходам  $x$  и  $y$  числовые индикаторы (рис. 455). Если настроечные параметры блока не изменены, то его прямоугольник будет белым с черной рамкой и перекрестием, а круг будет синим. Запустив расчет и перетаскивая круг внутри прямоугольника (круг при этом будет становиться красным), на индикаторах можно будет наблюдать изменение значений выходов: когда центр круга будет

находиться в центре блока, выходы будут близки к нулю, у границ выходы будут близки к  $\pm 1$ . При выведении курсора за пределы прямоугольника блока центр круга останется на его границе (при этом выходящий за прямоугольник сегмент не будет нарисован), а при возврате курсора обратно в прямоугольник круг снова будет следовать за ним. При желании, можно войти в окно настроек блока через контекстное меню и изменить его цвета и диаметр круга.

### §3.7.12. Добавление пунктов в контекстное и главное меню

Рассматривается программное добавление моделью блока новых пунктов в контекстное (вызываемое по правой кнопке) и главное меню РДС. При помощи этих пунктов блок может принимать от пользователя команды в режиме редактирования, в котором большинство остальных способов взаимодействия моделей блоков с пользователем отключено.

Модели блоков могут добавлять свои собственные пункты как в контекстное меню (меню, которое вызывается при нажатии правой кнопки мыши на изображении блока), так и в главное меню РДС. Добавление пунктов в контекстное меню обычно используется для предоставления пользователю быстрого доступа к каким-либо функциям, выполняемым блоком, или для переключения режимов работы этого блока. Добавление пунктов в главное меню используется гораздо реже: в отличие от контекстного, индивидуального для каждого блока, системное меню – общее для всей схемы, поэтому туда имеет смысл добавлять пункты для действий, уникальных для данной схемы. Например, в главное меню можно добавить пункт для открытия какого-либо очень важного окна подсистемы, чтобы пользователь не искал эту подсистему по всей схеме.

Пункты контекстного меню всегда добавляются в его конец, а пункты главного – в специальный подпункт “система | дополнительно”. Доступность этих пунктов пользователю

определяется не режимом РДС, а разработчиком модели: он может разрешать или запрещать различные пункты в зависимости от текущего состояния блока по своему желанию. В этом параграфе будут приведены только самые простые примеры добавления пунктов в меню, более подробно эти вопросы рассмотрены в §2.12.6 и 2.12.7 руководства программиста [1].

Ранее, в §3.7.11, рассматривался блок-рукоятка, с помощью которого пользователь мог задавать значения двух координат, перемещая мышью круг внутри прямоугольника (см. стр. 232). Добавим в контекстное меню этого блока пункты, позволяющие обнулять одну из задаваемых координат или обе координаты сразу: это позволит не только точно приводить к нулю выходы блока (точно переместить мышью круг может оказаться не такой простой задачей), но и даст пользователю возможность сбрасывать значения блока не только в режимах моделирования и расчета, но и в режиме редактирования, в котором реакции блока на мышшь не вызываются.

Чтобы добавить пункты в контекстное меню блока, в модель этого блока необходимо ввести сразу две реакции: реакцию на вызов контекстного меню, в которой и будут добавлены новые пункты, и реакцию на выбор пункта меню, в которой модель блока будет реагировать на выбор пользователем одного из этих добавленных пунктов. В списке событий редактора модели обе эти реакции находятся в разделе “разное” (рис. 456).

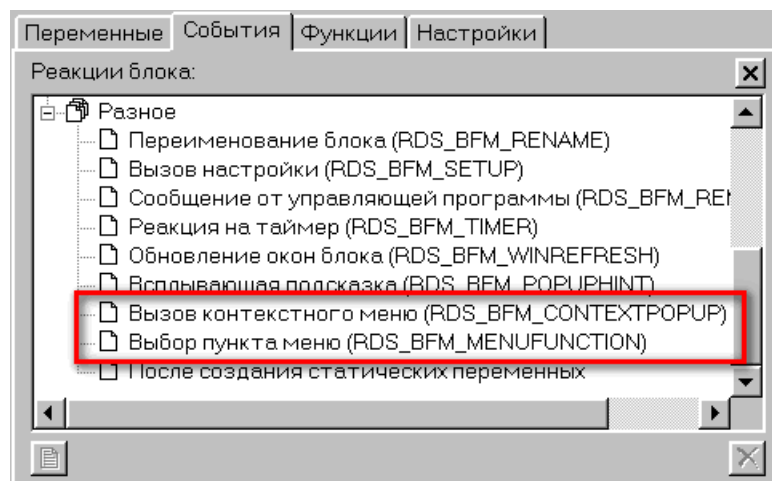


Рис. 456. Реакции на вызов меню и выбор пункта в списке событий

Введем в ранее созданный блок-рукоятку реакцию на открытие контекстного меню, в которой мы добавим в меню три новых пункта. Для этого откроем редактор модели этого блока, на вкладке “события” в разделе “разное” дважды щелкнем на подразделе “вызов контекстного меню” и введем на открывшейся вкладке “вызов меню” следующий текст:

```
// Разделитель
rdsAdditionalContextMenuItemEx(NULL, RDS_MENU_DIVIDER, 0, 0);

// Исполняемые пункты
rdsAdditionalContextMenuItemEx("Обнулить X",
 x==0.0?RDS_MENU_DISABLED:0,1,0);
rdsAdditionalContextMenuItemEx("Обнулить Y",
 y==0.0?RDS_MENU_DISABLED:0,2,0);
rdsAdditionalContextMenuItemEx("Обнулить все",
 x==0.0&y==0.0?RDS_MENU_DISABLED:0,3,0);
```

Для добавления пунктов в контекстное меню из реакции на вызов этого меню мы используем функцию `rdsAdditionalContextMenuItemEx` (см. §A.5.17.2 приложения к руководству программиста [2]). В первом параметре функции передается текст добавляемого пункта, во втором – набор битовых флагов, определяющих его внешний вид, в третьем и четвертом – два целых числа, которые без изменения будут переданы в реакцию модели на выбор этого

пункта (можно считать эти числа идентификатором пункта, по которому модель сможет понять, какой именно из добавленных пунктов был выбран пользователем).

В самом первом вызове функции вместо текста пункта передается NULL, а в битовых флагах – RDS\_MENU\_DIVIDER. Это приведет к тому, что вместо нормального пункта в контекстное меню будет добавлена горизонтальная линия, которая визуально отделит добавляемые нами пункты от остальной части меню, за которую отвечает РДС. На самом деле, можно было бы и не передавать флаг RDS\_MENU\_DIVIDER, поскольку NULL вместо текста пункта уже дает РДС понять, что нужно добавить не пункт меню, а горизонтальный разделитель. Два последних параметра rdsAdditionalContextMenuItemEx при таком вызове игнорируются (разделитель не может быть выбран пользователем), поэтому в них можно передать что угодно – мы передаем нули.

Следующие три вызова rdsAdditionalContextMenuItemEx добавляют в меню пункты “обнулить X”, “обнулить Y” и “обнулить все”, с которыми мы связываем пары целых чисел (1,0), (2,0) и (3,0) соответственно. В реакции на выбор пункта меню нам достаточно будет проверять только первое из этих чисел: значение 1 будет означать, что выбран пункт “обнулить X”, значение 2 – “обнулить Y”, значение 3 – “обнулить все”. Во втором параметре каждого вызова стоит условное выражение, имеющее значение RDS\_MENU\_DISABLED, если соответствующие этому пункту меню переменные блока уже равны нулю, и ноль в противном случае. Битовый флаг RDS\_MENU\_DISABLED указывает РДС на то, что пункт меню должен быть запрещенным и что пользователь не может его выбрать. Таким образом, пункты “обнулить” будут разрешенными, только если соответствующие им координаты еще не обнулены – это даст пользователю некоторую визуальную обратную связь.

Добавление пунктов мы реализовали, теперь нужно ввести в модель реакцию на их выбор. На вкладке “события” в разделе “разное” дважды щелкнем на подразделе “выбор пункта меню” и введем на открывшейся вкладке “пункт меню” следующий текст:

```
// Какой пункт выбран
switch (MenuData->Function)
{
 case 1: x=0; break; // Обнулить X
 case 2: y=0; break; // Обнулить Y
 case 3: x=y=0; break; // Обнулить все
}
// Вводим сигнал готовности для передачи выходов по связям
Ready=1;
```

В эту реакцию в параметре MenuData передается указатель на структуру RDS\_MENUFUNCDATA, состоящую всего из двух целых полей: Function и MenuData (см. §A.2.6.7 приложения к руководству программиста). В поле Function записано первое из двух целых чисел, которые мы связали с выбранным пользователем пунктом меню в вызове rdsAdditionalContextMenuItemEx, в поле MenuData – второе из них. Таким образом, для того, чтобы выяснить, какой из трех добавленных нами пунктов выбрал пользователь, нам нужно сравнить MenuData->Function с числами 1, 2 и 3, что мы и делаем в операторе switch.

После того, как, в зависимости от выбранного пользователем пункта, мы обнулили нужный выход блока, мы, как всегда, вводим сигнал готовности блока Ready, чтобы новое значение выхода передалось по связям.

Теперь, если скомпилировать модель и щелкнуть на блоке правой кнопкой мыши, в конце его контекстного меню появятся разделитель и три новых пункта (рис. 457). Меню будет выглядеть по-разному, в зависимости от режима РДС, но добавленные нами пункты будут в нем присутствовать всегда. При этом пункт обнуления выхода будет разрешенным только в том случае, если этот выход не равен нулю в точности (на рисунке выход x в точности равен нулю, и пункт “обнулить X” заблокирован).



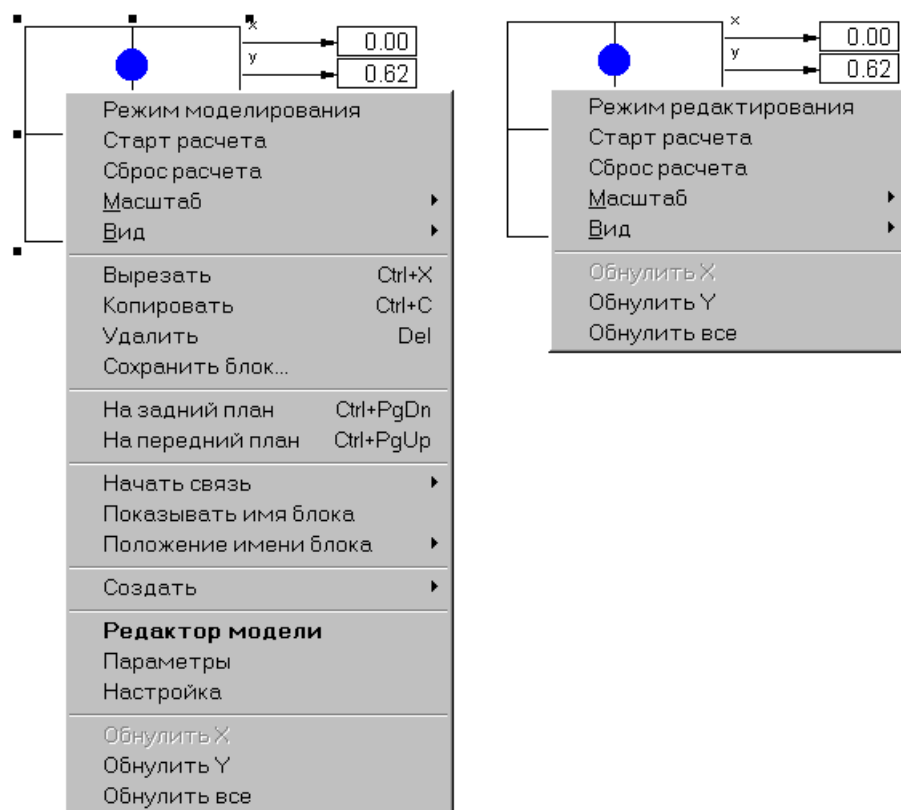


Рис. 457. Контекстное меню блока в режиме редактирования (слева) и моделирования (справа)

Теперь рассмотрим добавление пунктов меню в главное меню РДС. Это несколько сложнее, чем добавлять пункты в контекстное. В рассмотренном выше примере мы добавляли в контекстное меню временные пункты, которые автоматически уничтожались РДС при закрытии этого меню (подробнее об этом – в §2.12.6 руководства программиста). Главное меню, в отличие от контекстного, не связано с каким-либо конкретным блоком, поэтому у блока не может быть реакции на открытие главного меню. Если бы такая реакция была предусмотрена в РДС, ее пришлось бы вызывать для всех блоков загруженной схемы, спрашивая у модели каждого из них, не желает ли она добавить что-либо в главное меню. Такое усложнение не было бы оправданным, поэтому добавление пунктов в главное меню устроено иначе. Модель блока в любой момент своего существования может создать в главном меню свой собственный пункт при помощи вызова `rdsRegisterMenuItem` (см. §A.5.17.8 приложения к руководству программиста), РДС запомнит этот факт и свяжет этот пункт с создавшим его блоком, пока модель не уничтожит этот пункт или пока блок не будет удален.

Чтобы проиллюстрировать возможность добавления пунктов в главное меню, создадим блок, который будет вычислять скорость расчета РДС в тактах в секунду. Значение этой скорости будет показываться пользователю при выборе пункта меню “система | дополнительно | статистика” или при нажатии сочетания клавиш `Ctrl+Alt+S`. Наш блок будет работать следующим образом: при запуске расчета он запомнит время с момента загрузки Windows, возвращаемое стандартной функцией API `GetTickCount`. В каждом такте расчета он будет увеличивать на единицу внутреннюю переменную – счетчик тактов. При запросе пользователем статистики модель разделит число тактов на прошедший интервал времени и покажет результат пользователю. Чтобы статистику можно было вызвать и при остановленном расчете, в момент остановки блок тоже будет запоминать результат `GetTickCount` в специальной переменной, и, при запросе статистики, показывать скорость

последнего выполненного расчета. Внутренний счетчик тактов блока мы назовем Num, переменную для хранения времени начала расчета – StartTick, переменную, для хранения времени его конца – EndTick. Таким образом, наш блок будет иметь следующую структуру статических переменных:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| Num        | int        | Внутренняя        |             | 0                         |
| StartTick  | double     | Внутренняя        |             | 0                         |
| EndTick    | double     | Внутренняя        |             | 0                         |

Несмотря на то, что функция GetTickCount возвращает целое число миллисекунд с момента загрузки Windows, переменные для хранения результата ее возврата мы сделали вещественными – для вычисления скорости нам потребуются вещественные числа, поскольку нам придется делить число миллисекунд на тысячу, чтобы получить скорость в тактах в секунду.

Создадим в схеме новый блок (см. стр. 95) и зададим ему указанную выше структуру статических переменных. Нам нужно, чтобы блок запускался каждый такт (он должен считать эти такты) – можно просто при его создании установить флаг запуска каждый такт (см. рис. 363 на стр. 94), а можно написать модель таким образом, чтобы блок запускался каждый такт независимо от состояния этого флага. Так мы и поступим.

Прежде всего, добавим в нашу модель стандартный файл заголовков “stdio.h” – сообщение пользователю мы будем формировать функцией printf, описанной в этом файле. Раскроем на вкладке “события” левой панели редактора раздел “описания” и дважды щелкнем на открывшемся подразделе “глобальные описания” (см. стр. 114). На появившейся в правой части окна пустой вкладке введем команду включения нужного файла:

```
#include <stdio.h>
```

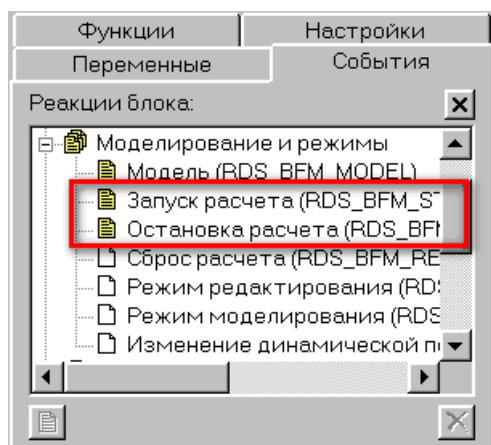


Рис. 458. Запуск и остановка расчета в списке событий

Теперь добавим в модель реакцию на запуск расчета: при запуске мы должны сбросить счетчик тактов и запомнить время начала расчета. На вкладке “события” раскроем раздел “моделирование и режимы” и дважды щелкнем на подразделе “запуск расчета” (рис. 458). Откроется вкладка “запуск”, на которой нужно ввести следующие команды:

```
// Сброс счетчика тактов
Num=0;
// Запоминание времени старта
StartTick=GetTickCount();
// Принудительный запуск модели
Start=1;
```

В счетчик тактов Num мы записываем ноль, в переменную времени начала расчета – результат возврата GetTickCount (то есть текущее время с момента загрузки Windows в миллисекундах), а в сигнал запуска блока Start – единицу, что приведет к принудительному запуску модели в ближайшем такте расчета. Если для блока установлен запуск каждый такт, то модель запустится независимо от значения Start. Если же установлен запуск по сигналу, то модель запустится потому, что мы только что взвели этот сигнал.

При остановке расчета нам нужно запомнить время этой остановки в переменной EndTick. Добавим в модель реакцию на остановку: в том же самом разделе “моделирование и режимы” вкладки “события” дважды щелкнем на подразделе “остановка расчета” (см. рис. 458) и на открывшейся вкладке “остановка” введем оператор присваивания:

```
// Запоминание времени остановки
EndTick=GetTickCount();
```

В такте расчета мы должны, во-первых, увеличить на единицу счетчик тактов Num, и, во-вторых, взвести сигнал Start для принудительного запуска модели в следующем такте независимо от настроек блока. В разделе “моделирование и режимы” дважды щелкнем на подразделе “модель” (см. рис. 458) и введем на открывшейся вкладке следующий текст:

```
// Увеличиваем счетчик тактов
Num++;
// Принудительный запуск модели
Start=1;
```

Теперь перейдем собственно к добавлению пункта в главное меню. Будем делать это при инициализации блока, то есть в момент подключения к нему модели. На вкладке “события” раскроем раздел “создание и уничтожение” и дважды щелкнем в нем на подразделе “инициализация блока” (рис. 459). Откроется вкладка “инициализация”, на которой мы введем вызов для добавления пункта в главное меню:

```
rdsRegisterMenuItem("Статистика",
 RDS_MENU_SHORTCUT|
 RDS_MENU_UNIQUECAPTION,
 'S',
 RDS_KALT|RDS_KCTRL,
 0,0);
```

В первом параметре функции rdsRegisterMenuItem передается текст добавляемого пункта меню – в нашем случае, это “статистика”. Второй параметр содержит битовые флаги, управляющие пунктом. У нас это объединение флага RDS\_MENU\_SHORTCUT, указывающего на то, что к пункту будет привязано сочетание клавиш, и флага RDS\_MENU\_UNIQUECAPTION, блокирующего добавление пункта, если другой пункт с таким названием в меню уже есть. В третьем и четвертом параметрах передаются код и флаги сочетания клавиш соответственно – у нас это клавиша “S” и объединение флагов RDS\_KALT и RDS\_KCTRL, то есть сочетанием клавиш для нашего пункта будет Ctrl+Alt+S. Наконец, в двух последних параметрах передается пара целых чисел, по которым модель в реакции на вызов пользовательского пункта меню сможет опознать этот пункт. Точно так же модель в предыдущем примере опознавала пункты контекстного меню – реакция на выбор пункта контекстного и главного меню в модели общая. В нашем случае мы передаем два нуля: наш блок имеет единственной пользовательский пункт главного меню и не добавляет ничего в контекстное, поэтому в реакции на выбор пункта меню можно вообще ничего не проверять: если модель реагирует на вызов какого-то созданного ей пункта, значит, это пункт “статистика”.

Теперь можно добавить в модель реакцию на выбор пункта, точно так же, как мы делали это для контекстного меню ранее. На вкладке “события” в разделе “разное” дважды щелкнем на подразделе “выбор пункта меню” (см. рис. 456 на стр. 239) и введем на открывшейся вкладке “пункт меню” следующий текст:

```
// У нас – единственный пункт меню
double speed,msec; // Вспомогательные переменные
char buffer[100]; // Буфер для сообщения пользователю
```

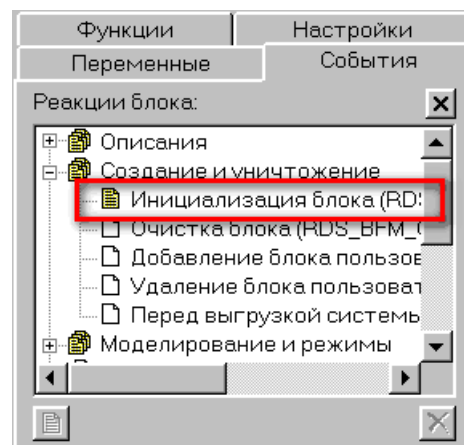


Рис. 459. Инициализация блока в списке событий

```

if(rdsCalcProcessIsRunning()) // Расчет работает
 EndTick=GetTickCount(); // Текущее время
else if(rdsCalcProcessNeverStarted())
 { // Расчет остановлен и ни разу не запускался
 rdsMessageBox("Расчет еще не запускался","Статистика",
 MB_OK|MB_ICONWARNING);
 return;
 }

// Расчет работает или остановлен, в EndTick – конечное время
msec=EndTick-StartTick; // Прошло миллисекунд
if(msec==0) // Невозможно вычислить скорость
 return;

speed=Num/(msec/1000.0); // Скорость в тактах/сек

// Формирование и вывод сообщения
sprintf(buffer,"Скорость расчета: %d тактов/сек",(int)speed);
rdsMessageBox(buffer,"Статистика",MB_OK|MB_ICONWARNING);

```

Сначала мы вызываем функцию `rdsCalcProcessIsRunning` (см. §A.5.2.10 приложения к руководству программиста), чтобы узнать, работает ли сейчас расчет. Если он работает, мы записываем текущее время в переменную `EndTick`. В противном случае, то есть если расчет не работает, мы вызываем функцию `rdsCalcProcessNeverStarted` (см. §A.5.2.11 приложения к руководству программиста), чтобы узнать, запускался ли расчет хотя бы один раз. Если расчет ни разу не запускался (функция вернула `TRUE`), мы не можем вычислить статистику, о чем функцией `rdsMessageBox` (см. §A.5.5.6 приложения к руководству программиста) выводится сообщение пользователю, и реакция завершается. Если же расчет запускался, значит, он был остановлен, и в реакции на остановку в переменную `EndTick` уже было записано время конца расчета. Таким образом, в переменной `EndTick` будет записано либо текущее время, если расчет сейчас работает, либо время остановки, если он был остановлен ранее. В обоих случаях мы можем вычислить скорость расчета.

Чтобы вычислить скорость, сначала мы определяем общее время расчета в миллисекундах – это разность между `EndTick` и `StartTick`. Если эта разность равна нулю, мы не можем вычислить скорость, и реакция завершается (впрочем, такая ситуация маловероятна: для этого пользователь должен успеть вызвать пункт меню в той же миллисекунде, в которой запущен расчет). Далее мы вычисляем скорость, разделив число выполненных тактов на время расчета, при помощи функции `sprintf` формируем текст сообщения и показываем его пользователю вызовом `rdsMessageBox`.

Теперь можно скомпилировать модель, и в главном меню появится новый пункт “система | дополнительно | статистика”, рядом с которым будет указано вызывающее его сочетание клавиш (рис. 460). Если запустить расчет и выбрать этот пункт или нажать `Ctrl+Alt+S`, на экране должно появиться сообщение с вычисленной скоростью расчета.

В этом примере мы добавляем пункт в главное меню, но не удаляем его: он будет удален автоматически при отключении модели от блока (например, при удалении блока). Кроме того, при создании пункта мы нигде не запомнили уникальный идентификатор, который возвращает функция `rdsRegisterMenuItem`. Если бы мы хотели как-то изменять пункт меню (например, запрещать его, или включать возле него галочку), нужно было бы запомнить этот идентификатор и передавать его в соответствующие функции РДС, описанные в §A.5.17 приложения к руководству программиста. Этот идентификатор потребовался бы и в том случае, если бы мы захотели удалить созданный пункт вручную.

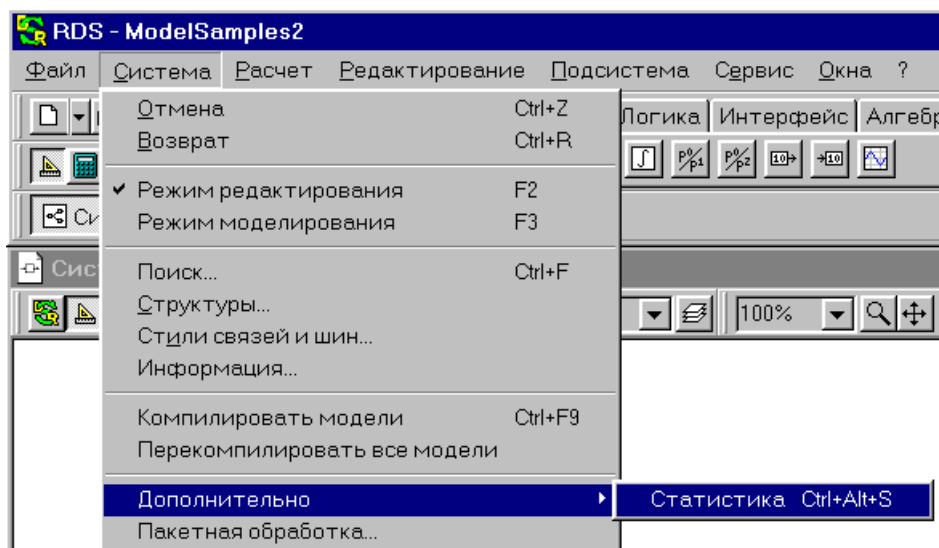


Рис. 460. Добавленный пункт главного меню

### §3.7.13. Вызов функций блоков

Описывается добавление в модели блоков возможностей непосредственного вызова моделей других блоков и реакций на такие вызовы. Такие вызовы, называемые функциями блоков, позволяют блокам быстро передавать друг другу информацию любого типа без участия пользователя.

#### §3.7.13.1. Общие принципы работы с функциями блока

Рассматривается реализация вызовов функций блоков и реакций на их вызовы в модуле автокомпиляции. Описывается устройство автоматически создаваемых объектов, используемых в автокомпилируемых моделях для этих целей.

Модуль автокомпиляции облегчает разработчику реализацию в моделях блоков функций, которые могут непосредственно вызываться другими блоками. Каждой такой функции блока в РДС присваивается произвольное текстовое имя, по которому ее опознают другие блоки. Эти блоки могут находиться в разных DLL, написанных на разных языках программирования, поэтому всю передачу данных между ними берет на себя РДС. Чтобы добавить в модель функцию блока (не важно – будет она вызываться у этого блока или, наоборот, этот блок будет вызывать ее у других), необходимо добавить эту функцию на вкладку “функции” левой панели редактора модели (см. §3.6.5 на стр. 57), указав для нее текстовое имя, тип ее параметра и имя внутреннего объекта, который будет создан в модели для работы с ней. Для вызова функции у других блоков нужно будет вызывать функции-члены этого объекта, передавая в них параметры функции и идентификаторы вызываемых блоков. После того, как функция будет добавлена в модель, на вкладке “события” редактора появится новое событие – вызов этой функции у данного блока. В реакции на это событие записываются все действия, которые блок должен выполнить, если у него вызвана функция с таким именем.

Функции блоков создаются программистами их моделей, и пользователь не может как-то повлиять на их работу. Как правило, все начинается с того, что создатель модели решает включить в свой блок не только обработку данных на входах и выдачу результатов на выходы, но и возможность непосредственного выполнения команд от других блоков – то есть добавляет в модель реакцию на вызов функции. Прежде всего необходимо придумать для этой функции уникальное текстовое имя и набор данных, который будет передаваться при ее вызове (у функции может быть только один параметр, поэтому обычно такие данные оформляют в виде структуры и передают указатель на нее). Для обеспечения уникальности

имени функции это имя обычно делают длинным и каким-либо образом отражающим смысл выполняемых ей действий. Чтобы избежать возможных проблем с кодировкой строк, в именах функций обычно используют только латинские буквы, цифры и знаки препинания, хотя в РДС и нет жестких требований к этим именам. Общие рекомендации по выбору имен для новых функций даются в §2.13.1 руководства программиста [1], примеры имен стандартных функций приведены в §1.6 части I.

Придумав имя функции, необходимо решить, будут ли у нее параметры, и если будут, то какие именно. Формально у функции блока есть только один параметр типа `void*`, то есть “указатель на что-либо”. Если функции не нужны параметры, она просто игнорирует этот указатель (при этом в нем чаще всего передают значение `NULL`). Если же параметры нужны, этот указатель обычно ссылается на какую-либо область памяти, содержащую эти параметры. РДС передает указатель на область параметров от вызвавшей модели к вызванной без каких-либо проверок, поэтому разработчику модели вызываемого блока следует самостоятельно проверять, соответствуют ли переданные параметры вызванной функции. Такая проверка очень важна, поскольку модель вызывающего блока может быть разработана другим программистом, и нельзя гарантировать того, что он вызовет функцию, правильно передав ей параметры. Передача неправильных параметров может привести к серьезным ошибкам. Допустим, например, что функция блока принимает один параметр типа `double`. Можно сделать параметром функции указатель на это число, при этом реакция на вызов этой функции будет преобразовывать полученный указатель типа `void*` к типу `double*` и обращаться через него к переданному числу. Однако, разработчик вызывающей модели может по ошибке (или из-за недостаточно ясно написанного описания функции) передать в нее не указатель на `double`, а указатель на четырехбайтовое целое число `int`. РДС приведет этот указатель к типу `void*` и передаст его модели вызываемого блока. Эта модель, в свою очередь, приведет этот указатель к типу `double*` и попытается считать по нему вещественное восьмибайтовое число. В лучшем случае она при этом считает неправильное значение, в худшем – обращение к восьми байтам по адресу, по которому отведено только четыре, вызовет ошибку общей защиты приложения (GPF).

Чтобы уменьшить вероятность возникновения подобных проблем, параметры функции, какими бы они ни были, обычно оформляют в виде структуры, в первое поле которой записывают ее же размер. Например, для функции, принимающей вещественное число `double`, можно описать такую структуру параметров:

```
typedef struct
{ DWORD servSize; // Размер структуры параметров
 // ... параметры функции ...
 double Value; // Передаваемое вещественное число
} TMyFuncParam;
```

При вызове функции в поле `servSize` этой структуры нужно записать ее размер, полученный стандартным оператором языка C `sizeof`, а в поле `Value` – передаваемое при вызове функции вещественное значение:

```
TMyFuncParam param; // Структура параметров
param.servSize=sizeof(param); // Размер структуры
param.Value=12.34; // Передаваемое число
// При вызове функции в качестве параметра
// будет передаваться ¶m
```

В реакции на вызов функции переданный указатель будет приводиться к типу `TMyFuncParam*` (“указатель на `TMyFuncParam`”) – модуль автокомпиляции вставляет операцию приведения типа автоматически. Вызванная модель при этом может сравнить значение поля `servSize` этой структуры с ее размером. Если значение поля не меньше размера структуры, значит, передано достаточно данных для работы. В реакциях на вызов

функции их параметр всегда имеет имя Param, поэтому текст реакции на вызов функции, вводимый пользователем, будет выглядеть следующим образом:

```
// Param имеет тип TMyFuncParam*
if(Param!=NULL && Param->servSize>=sizeof(TMyFuncParam))
{ // Можно выполнять функцию
 ...
}
```

Здесь не проверяется точное равенство значения поля (то есть размера структуры у вызвавшей модели) размеру структуры у вызываемой, вместо этого проверяется *достаточность* размера переданной структуры. Это делается для того, чтобы, если в будущем к структуре параметров функции будут добавлены дополнительные поля для расширения ее возможностей, все ранее написанные модели, не пользующиеся этим новыми возможностями, продолжили бы нормально работать. Если размер структуры, переданной в качестве параметра функции, окажется больше ожидаемого, старая модель все равно может выполнить функцию, поскольку все нужные ей старые поля в этой структуре присутствуют. Дополнительные поля в этом случае можно добавлять только в конец структуры, не изменяя ту ее часть, с которой будут работать старые модели. При этом новая модель, выполняющая функцию, должна по переданному размеру структуры определить, есть ли в структуре новые поля, и, если их нет, но размер структуры достаточен для старой версии функции, выполнить ее. Например, можно написать реакцию так:

```
// Param имеет тип TMyFuncParam*
if(Param!=NULL)
{ if(Param->servSize>=sizeof(TMyFuncParam))
 { // Можно выполнять новую версию функции
 ...
 }
 else if(Param->servSize>=размер_для_старой_версии)
 { // Можно выполнять старую версию функции
 ...
 }
}
```

Для того, чтобы вызвать функцию блока, необходимо знать ее имя, структуру ее параметров и идентификатор блока, у которого она будет вызываться. Имя и структуру параметров обычно описывает разработчик, придумавший эту функцию (описание структуры параметров часто выносят в отдельный файл заголовков). Идентификатор блока, функция которого вызывается, нужно каким-либо образом получить у РДС. Чаще всего используются следующие варианты:

- одновременный вызов функции у всех блоков схемы;
- одновременный вызов функции у всех блоков родительской подсистемы вызывающего блока (идентификатор этой подсистемы содержится в поле Parent структуры данных блока, доступной по указателю rdsbcppBlockData, см. стр. 88);
- вызов функции некоторого блока в ответ на вызов этим блоком функции данного блока (идентификатор вызвавшего функцию блока может быть считан из структуры RDS\_FUNCTIONCALldata, указатель на которую доступен из реакции модели на вызов функции);
- вызов функции у блока, идентификатор которого получен путем анализа схемы при помощи сервисных функций РДС (например, можно перебрать все блоки, соединенные связями с данным, и вызвать у них какую-либо функцию, см. стр. 260);
- вызов функции у блока, зарегистрировавшего себя в качестве исполнителя этой функции (см. §3.7.13.4 на стр. 268).

Можно получить идентификатор блока и другим способом (например, если известно полное имя блока, можно получить его идентификатор при помощи функции



rdsBlockByFullName, см. §A.5.6.3 приложения к руководству программиста [2]), но перечисленные выше способы используются чаще всего. В любом случае, вызов функции блока, как правило, используется для передачи этому блоку каких-либо данных, и при этом, обычно, известно, какому именно блоку передаются данные. Если адресат данных неизвестен (например, нужно выполнить какую-то команду, и не важно, кто именно ее выполнит), функцию вызывают сразу у всех блоков схемы или у блока, который заявил РДС о том, что он выполняет данную функцию.

Для каждой добавленной в модель блока функции модуль автокомпиляции создает описание специального класса, различные функции-члены которого отвечают за вызов функций блоков, поиск в схеме блоков-исполнителей функций и т. п. Для работы с функцией модуль автоматически добавляет в программу объект такого класса. Например, если для какой-либо функции, принимающей параметр типа TMyFuncParam\* (“указатель на структуру TMyFuncParam”, эта структура описывалась выше в этом параграфе), был создан объект с именем MyFunc, и необходимо вызвать эту функцию у всех блоков родительской подсистемы вызывающего блока, этот вызов в модели будет записан так:

```
TMyFuncParam param; // Структура параметров
param.servSize=sizeof(param); // Размер структуры
param.Value=12.34; // Передаваемое число
MyFunc.Broadcast(
 rdsbcppBlockData->Parent, // Родительская подсистема
 0, // Флаги вызова (нет)
 ¶m); // Параметр функции
```

Перечислим коротко основные функции-члены класса объектов, создаваемых для каждой функции:

- `int Id(void)` – уникальный целый идентификатор функции, присвоенный ей в РДС. Идентификатор функции может потребоваться в тех случаях, когда вызов функции производится не при помощи объекта, созданного модулем автокомпиляции, а напрямую, при помощи сервисных функций РДС. Пример использования функции:

```
rdsBroadcastFuncCallsDelayed(// Отложенный вызов
 rdsbcppBlockData->Parent, // Родительская подсистема
 MyFunc.Id(), // Идентификатор функции
 ¶m, // Область параметров функции
 sizeof(param), // Размер области
 0); // Флаги вызова (нет)
```

- `int Call(RDS_BHANDLE Block, структура_параметров *param)` – вызов функции у блока с идентификатором Block, если у функции есть параметр. В качестве параметра передается указатель param, имеющий тип “указатель на данные типа структура\_параметров”. Функция возвращает целое число, которое возвратила модель блока, среагировавшего на вызов функции. Пример использования функции:

```
// Вызов функции у своего собственного блока
MyFunc.Call(// Вызов
 rdsbcppBlockData->Block, // Этот блок
 ¶m); // Область параметров функции
```

- `int Call(RDS_BHANDLE Block)` – вызов функции у блока с идентификатором Block, если у функции нет параметра (вместо указателя на параметр в модель вызываемого блока передается NULL). Используется точно так же, как и предыдущая функция. Пример использования этой функции-члена приведен в §3.7.13.3 (стр. 266).
- `int Call(структура_параметров *param)` – вызов функции у блока, зарегистрированного как исполнитель данной функции, если у функции есть параметр. От двух предыдущих версий функции-члена Call эта версия отличается тем, что в ней не указывается идентификатор вызываемого блока – РДС определяет его самостоятельно, находя блок, объявивший себя исполнителем этой функции. В качестве параметра



передается указатель `param`, имеющий тип “указатель на данные типа *структура\_параметров*”. Функция возвращает целое число, которое возвратила модель блока, среагировавшего на вызов функции. Эту функцию-член можно вызывать **только из модели блока**, то есть из реакций блока на различные события и из функций, объявленных членами класса блока. Пример использования этой функции-члена приведен в §3.7.13.4 (стр. 268).

- `int Call(void)` – вызов функции у блока, зарегистрированного как исполнитель данной функции, если у функции нет параметра (вместо указателя на параметр в модель вызываемого блока передается `NULL`). Для этой функции-члена справедливы те же ограничения в использовании, что и для предыдущей – ее нельзя использовать вне модели блока, то есть в функциях из глобальных описаний (не являющихся членами класса блока).
- `int Broadcast(RDS_BHANDLE Sys, DWORD Flags, структура_параметров *param)` – вызов функции у всех блоков подсистемы `Sys`, если у функции есть параметр. В качестве параметра передается указатель `param`, имеющий тип “указатель на данные типа *структура\_параметров*”. `Flags` – флаги, управляющие вызовом функций (см. §A.5.13.4 приложения к руководству программиста). Функция возвращает общее число блоков, модели которых были вызваны. Пример использования функции:  

```
// Вызов функции у всех блоков в родительской подсистеме
MyFunc.Broadcast(// Вызов у блоков подсистемы
 rdsbcppBlockData->Parent, // Родительская подсистема
 0, // Флаги вызова (нет)
 ¶m); // Область параметров функции
```
- `int Broadcast(RDS_BHANDLE Sys, DWORD Flags)` – вызов функции у всех блоков подсистемы `Sys`, если у функции нет параметра (вместо указателя на параметр в модель вызываемого блока передается `NULL`). Используется точно так же, как и предыдущая функция.
- `void RegisterProvider(void)` – зарегистрировать данный блок в РДС в качестве исполнителя функции. Обычно такая регистрация делается автоматически при добавлении функции в редактор модели при помощи флажка “объявить блок исполнителем функции” на панели “дополнительные действия” окна параметров функции (см. рис. 339 на стр. 59), поэтому в вызове `RegisterProvider` нет необходимости. Этот вызов нужен только в том случае, если регистрация блока производится вручную – например, если разработчик хочет разрешать или запрещать ее в настройках блока. Вызывать `RegisterProvider` можно только из модели блока, то есть из реакций на события и из функций, объявленных членами класса блока.
- `void UnregisterProvider(void)` – отменить регистрацию блока в качестве исполнителя функции. Эта функция-член нужна только при ручной регистрации блока вызовом `RegisterProvider`. Если блок зарегистрирован установкой флажка при добавлении функции в редактор модели, при удалении блока регистрация будет отменена автоматически.
- `void SubscribeToProvider(void)` – найти и запомнить блок, объявивший себя исполнителем этой функции. После такого поиска для вызова функции можно пользоваться версией функции-члена `Call`, в которую не передается идентификатор вызываемого блока. При изменении блока-исполнителя (например, при его стирании и появлении нового) запомненная информация будет обновляться автоматически. Для поиска исполнителя достаточно установить флажок “найти в схеме исполнителя функции” на панели “дополнительные действия” окна параметров функции (см. рис. 339 на стр. 59), поэтому в ручном вызове `SubscribeToProvider` обычно нет необходимости. Вызывать эту функцию-член можно только из модели блока, то есть из реакций на события и из функций, объявленных членами класса блока.

- `void UnsubscribeFromProvider(void)` – прекратить слежение за блоками-исполнителями данной функции. Как правило, в ручном вызове этой функции нет необходимости. Как и предыдущую, эту функцию-член можно вызывать только непосредственно из модели блока.
- `BOOL Subscribed(void)` – успешность поиска исполнителя функции (`TRUE`, если исполнитель функции существует в схеме, `FALSE` в противном случае). Вызывается только непосредственно из модели блока.
- `RDS_BHANDLE Provider(void)` – получить идентификатор найденного блока-исполнителя (если его нет, возвращается `NULL`). Изнутри модели блока вызвать функцию у исполнителя можно и не получая в явном виде его идентификатор – для этого достаточно использовать вариант функции-члена `Call`, в который не передается идентификатор блока. Однако, если вызов функции блока производится не непосредственно из какой-либо реакции модели (то есть не из функции-члена класса блока), этот вариант `Call` будет недоступен, и нужно будет пользоваться другим вариантом этой функции, в котором первым параметром будет идентификатор вызываемого блока. Для этого нужно будет как-то передать в то место программы, которое будет вызывать функцию, результат возврата `Provider`. Эта функция тоже вызывается только непосредственно из модели блока (т. е. из реакций на события и из функций, объявленных членами класса блока), поэтому снаружи модели получить идентификатор блока-исполнителя из самого объекта функции невозможно.
- `BOOL IsProvider(void)` – возвращает `TRUE`, если данный блок зарегистрирован как исполнитель функции, и `FALSE` в противном случае. Вызывается только непосредственно из модели блока.

Примеры различных способов вызова функций при помощи объектов, создаваемых модулем автокомпиляции, будут рассмотрены далее. Следует учитывать, что эти объекты поддерживают только *прямой*, то есть немедленный, вызов функций блоков – *отложенный* вызов, если это потребуется, необходимо выполнять при помощи сервисных функций РДС (назначение и особенности отложенного вызова функций блоков рассматриваются в §2.13.5 руководства программиста).

### §3.7.13.2. Вызов функции у всех блоков подсистемы

Рассматривается способ вызова заданной функции у всех блоков какой-либо подсистемы и, при необходимости, у блоков всех подсистем, вложенных в нее.

Ранее, в §3.7.11 (стр. 230) мы создали модель блока, который по щелчкам мыши на элементах своей картинке увеличивал, уменьшал или сбрасывал в ноль значение своего выхода. Сделаем так, чтобы одним щелчком мыши можно было увеличить, уменьшить или сбросить выходы всех таких блоков в подсистеме. Проще всего сделать это при помощи вызова функций блоков: мы добавим в подсистему новый управляющий блок, который, при щелчках мыши на нем, будет вызывать у всех блоков в одной с ним подсистеме функции для увеличения, уменьшения или сброса выхода. В наш старый блок мы добавим реакции на вызов таких функций, в которых выход блока будет изменяться согласно вызванной функции. Модели всех остальных блоков в подсистеме, включая, например, стандартные индикаторы, не будут иметь реакций на вызов наших функций, и, поэтому, будут просто игнорировать их. Это стандартная практика в РДС: если модель блока не рассчитана на выполнение какой-либо функции, она немедленно завершается, не выполняя никаких действий. Таким образом, вызов у блока функции, которую он не поддерживает, не приводит к каким-либо неприятным последствиям.

Решить нашу задачу с использованием функций блоков можно двумя путями. Во-первых, можно создать три разных функции: одну для уменьшения, другую для увеличения, третью для сброса выхода. В этом случае у функций не будет параметров, сам факт их вызова

будет говорить блоку, что именно нужно сделать с его выходом. Во-вторых, можно, вместо трех, создать единственную функцию с параметром, в котором будет содержаться команда на увеличение, уменьшение или сброс выхода. Первый путь проще, поэтому, сначала мы пойдем по нему.

Сначала нужно придумать уникальные имена для наших функций. Создаваемая модель является частью описания пользователя, поэтому включим в имена функций текст “UserManual” (“описание пользователя”). Функции эти будут выполняться блоками, которые в §3.7.11 иллюстрировали щелчки мыши на картинке, поэтому добавим в имена функций текст “PictureClick” (“щелчок на картинке”). И, наконец, функции будут отвечать за увеличение, уменьшение и сброс выхода блока, поэтому их имена будут содержать “Inc”, “Dec” и “Reset” (сокращения от “увеличить”, “уменьшить” и “сбросить”). Перечислим все эти слова через точки, и получим имена для наших новых функций:

| <i>Имя функции</i>            | <i>Параметр</i> | <i>Назначение</i> |
|-------------------------------|-----------------|-------------------|
| UserManual.PictureClick.Inc   | нет             | увеличение выхода |
| UserManual.PictureClick.Dec   | нет             | уменьшение выхода |
| UserManual.PictureClick.Reset | нет             | сброс выхода      |

Мы уже решили, что в первом варианте моделей у функций параметров не будет.

Начнем с создания управляющего блока, который будет вызывать функции у всех блоков своей подсистемы. Создадим в схеме новый блок с автокомпилируемой моделью (см. стр. 95) и зададим для него векторную картинку, похожую на картинку нашего старого блока (стр. 230). Чтобы не путать эти блоки, заменим в картинке простой белый прямоугольник на прямоугольник со скругленными углами, а квадраты заменим кругами (рис. 461, редактор векторной картинки блока и работа с ним описывается в §2.10 части

I). Идентификаторы кругов сделаем такими же, какие были у квадратов: красному (левому) кругу дадим идентификатор 1, зеленому (правому) – 2, а белому (центральному) – 3 (см. также рис. 451 на стр. 230). Чтобы модель блока отзывалась на щелчки мыши, разрешим в окне параметров блока реакцию на мышь (см. рис. 448 на стр. 229).



Рис. 461. Картинка управляющего блока

Теперь нужно добавить в модель три придуманных нами функции блока. Откроем редактор модели и выберем на его левой панели вкладку “функции” (см. рис. 338 на стр. 58) – она пока пуста. Добавим новую функцию: нажмем в нижней части панели кнопку “+” и заполним открывшееся окно описанием функции “UserManual.PictureClick.Inc” (рис. 462).

В верхней панели окна включим флажок “произвольная функция”: мы добавляем функцию, которую придумали сами, она не относится к стандартным. В поле “имя функции в RDS” введем “UserManual.PictureClick.Inc” – это имя нашей функции. После того, как мы выйдем из этого поля, поля “объект для функции в программе” и “имя функции реакции в классе блока” заполнятся автоматически – они станут похожими на введенное имя функции, и при этом такими, чтобы удовлетворять правилам синтаксиса языка С. В поле “объект для функции” появится “rdsfuncUserManual\_PictureClick\_Inc”, в поле “имя функции реакции” – “rdsfuncUserManual\_PictureClick\_IncEvent”. Чтобы вызывать эту функцию из модели блока, нам придется обращаться к созданному для функции объекту по его имени, и длинное имя, предлагаемое для объекта по умолчанию, использовать не очень удобно. Поскольку это имя будет использоваться только внутри нашей модели, мы можем изменить его как угодно – главное, чтобы оно не совпало с именем какой-либо другой переменной в программе. Исправим его на “rdsfuncUMPC\_Inc”, так будет гораздо короче. При этом имя функции реакции тоже изменится (оно преобразуется в “rdsfuncUMPC\_IncEvent”), но нам это не важно: с именем функции реакции мы напрямую не работаем, модуль автокомпиляции самостоятельно добавляет ее вызов там, где необходимо.

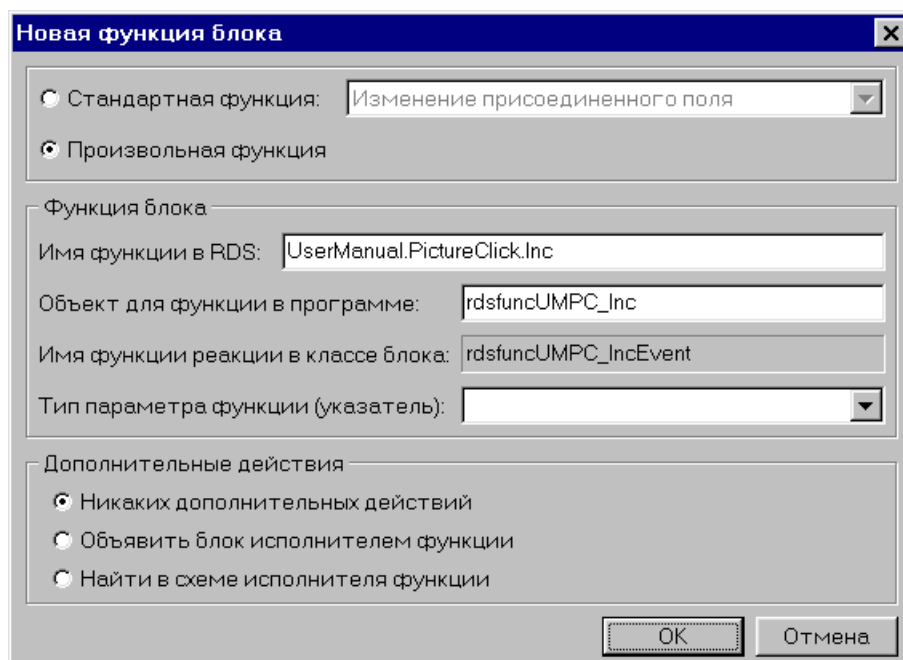


Рис. 462. Добавление функции “UserManual.PictureClick.Inc” в модель блока

Параметра у нашей функции нет, поэтому поле “тип параметра функции (указатель)” мы оставляем пустым. В нижней части окна на панели “дополнительные действия” мы оставляем включенный по умолчанию флажок “никаких дополнительных действий” – мы не собираемся регистрировать наш блок в РДС в качестве выделенного исполнителя этой функции или искать такого исполнителя (см. §3.7.13.4 на стр. 268). Нажмем кнопку “ОК”, чтобы функция добавилась в редактор модели – она появится в списке на вкладке “функции”.

Точно так же добавим в модель функции “UserManual.PictureClick.Dec” и “UserManual.PictureClick.Reset”, изменяя предлагаемые по умолчанию имена их объектов на “rdsfuncUMPC\_Dec” и “rdsfuncUMPC\_Reset” соответственно. Если все сделано правильно, список на вкладке “функции” примет вид, изображенный на рис. 463: в колонке “имя C++” будут содержаться имена объектов, созданных для функций, в колонке “имя в RDS” – имена функций, колонка “параметр” будет пустой. Все иконки в левой колонке будут пустыми белыми листами, поскольку в нашем блоке нет реакций на вызов этих функций. Их и не будет: управляющий блок, который мы создаем, будет только вызывать эти функции у других блоков.

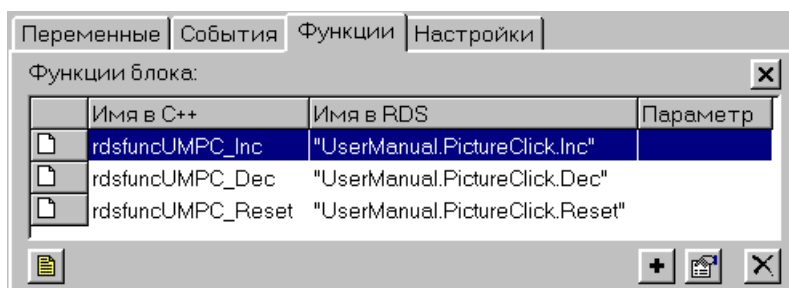


Рис. 463. Вкладка “функции” редактора модели после добавления функций (размеры панели и колонок увеличены для наглядности)

Теперь нужно добавить в модель блока реакцию на нажатие кнопки мыши: в зависимости от того, на каком из цветных кругов картинки блока щелкнет пользователь,

блок должен будет вызывать ту или иную функцию у всех блоков в одной с ним подсистеме. Как и в модели блока, которым мы собираемся управлять (см. стр. 231), в реакции на нажатие кнопки мыши мы будем запрашивать у РДС идентификатор элемента векторной картинки под курсором. Разным цветным кругам мы присвоили разные идентификаторы, и это позволит нам понять, где пользователь щелкнул мышью.

На вкладке “события” левой панели окна редактора модели раскроем раздел “мышь и клавиатура” (см. рис. 447 на стр. 226), дважды щелкнем на подразделе “нажатие кнопки мыши” и на появившейся справа одноименной вкладке введем следующий текст:

```
if (MouseData->Button==RDS_MLEFTBUTTON) // Нажата левая
{ switch (rdsGetMouseObjectId(MouseData)) // Идентификатор
 { case 1: // Красный круг
 rdsfuncUMPC_Dec.Broadcast(rdsbcppBlockData->Parent,0);
 break;
 case 2: // Зеленый круг
 rdsfuncUMPC_Inc.Broadcast(rdsbcppBlockData->Parent,0);
 break;
 case 3: // Белый круг
 rdsfuncUMPC_Reset.Broadcast(rdsbcppBlockData->Parent,0);
 break;
 }
}
else // Нажата не левая
 Result=RDS_BFR_SHOWMENU; // Разрешаем контекстное меню
```

Если нажата левая кнопка мыши, поле Button структуры описания события, указатель MouseData на которую передается в функцию реакции, будет содержать константу RDS\_MLEFTBUTTON. В этом случае мы запрашиваем у РДС идентификатор элемента картинки под курсором мыши при помощи функции rdsGetMouseObjectId (см. §A.5.6.31 приложения к руководству программиста [2]), которая вернет идентификатор, присвоенный нами элементу в редакторе векторной картинки. В зависимости от его значения, мы вызовем функцию, соответствующую объектам rdsfuncUMPC\_Dec, rdsfuncUMPC\_Inc или rdsfuncUMPC\_Reset, для всех блоков родительской подсистемы нашего блока.

Для вызова функции всех блоков какой-либо подсистемы через созданный для этой функции объект используется функция-член этого объекта Broadcast, принимающая два или три параметра. Если, как в нашем случае, у функции блока, для которой создан объект, нет параметра, в Broadcast передается идентификатор подсистемы и флаги вызова функции. Если бы у наших функций был параметр, он стал бы третьим параметром функции Broadcast. В общем виде вызов функции у всех блоков подсистемы выглядит так:

```
имя_объекта_функции.Broadcast(
 идентификатор_подсистемы,
 битовые_флаги_вызова,
 параметр_функции); // Отсутствует для функций без параметра
```

В качестве идентификатора подсистемы мы во всех трех случаях передаем идентификатор родительской подсистемы нашего блока. Мы берем его из поля Parent структуры данных блока (см. стр. 88), доступной в модели по указателю rdsbcppBlockData. Битовые флаги управляют вызовом функции у блоков. Мы передаем вместо них ноль – это значит, что функция будет вызвана у всех блоков указанной подсистемы независимо от их действий. Указание флага RDS\_BCALL\_SUBSYSTEMS привело бы к тому, что функция была бы вызвана и у всех блоков всех подсистем, вложенных в указанную, а указание флага RDS\_BCALL\_ALLOWSTOP позволило бы какому-нибудь из блоков прекратить все дальнейшие вызовы (эти флаги описаны в §A.5.13.4 приложения к руководству

программиста). Нам это не нужно, поэтому флагов мы не указываем. Параметра у наших функций нет, третий параметр во всех трех вызовах Broadcast отсутствует.

Если была нажата не левая кнопка мыши (то есть поле MouseData->Button не равно RDS\_MLEFTBUTTON), мы присваиваем переменной Result значение RDS\_BFR\_SHOWMENU, чтобы не блокировать вывод контекстного меню по правой кнопке мыши.

Модель управляющего блока готова – теперь в режимах моделирования и расчета по щелчкам мыши он будет вызывать одну из трех созданных нами функций у блоков своей подсистемы. Только никто пока не будет отзываться на эти функции. Добавим в ранее созданную модель блока, которым мы собираемся управлять (см. §3.7.11 на стр. 230), реакцию на них.

Откроем редактор модели управляемого блока и добавим в него три наших функции – точно так же, как мы делали для управляющего. Крайне важно не ошибиться в именах функций, они должны в точности совпадать с именами, которые мы использовали для управляющего блока. Если они будут отличаться хотя бы регистром символов, с точки зрения РДС это будут разные функции, и управляемый блок не будет реагировать на вызов, сделанный управляющим. Объекты, создаваемые для функций в управляемом блоке, можно не переименовывать: хотя у них и будут длинные, неудобные в использовании имена, в модели управляемого блока мы не будем обращаться к этим объектам. Здесь мы только реагируем на вызов функций, а не вызываем их – следовательно, имя объекта, через который вызывается функция, для нас не важно.

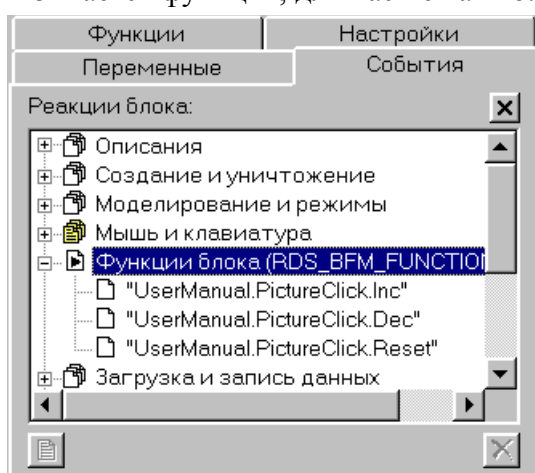


Рис. 464. События реакций на вызовы функций

После добавления всех трех функций вкладка “функции” левой панели редактора будет выглядеть примерно так же, как и у управляющего блока (см. рис. 463 на стр. 252). Нам нужно добавить в модель реакции на вызов каждой из этих функций. Реакцию на вызов функции можно добавить как с вкладки “функции”, так и, как обычно, с вкладки “события”. На вкладке “функции” можно выбрать функцию в списке и нажать кнопку с листком в левой нижней части вкладки. На вкладке “события” нужно раскрыть группу “функции блока” и дважды щелкнуть на ее подразделе с именем нужной функции (рис. 464).

Сначала любым из этих способов добавим реакцию на вызов функции блока “UserManual.PictureClick.Inc” – реагируя на нее, блок

должен увеличить свой выход на единицу. На открывшейся пустой вкладке с названием этой функции нужно ввести следующий текст:

```
y++; // Увеличиваем выход
// Вводим сигнал готовности для передачи выхода по связям
Ready=1;
```

Реакция содержит всего два оператора: первый увеличивает выход *y* на единицу, второй – взводит сигнал готовности блока *Ready*, чтобы по окончании очередного такта расчета новое значение выхода передалось по связям в соединенные с ним блоки. Сигнал готовности блока взводится автоматически только при вызове модели этого блока в такте расчета, вызов функции блока не взводит его, поэтому нам, как и в реакции на щелчок мыши в этом же блоке, нужно сделать это вручную.

Реакция на вызов функции “UserManual.PictureClick.Dec”, в которой блок должен уменьшать выход, будет аналогичной:

```

y--; // Уменьшаем выход
Ready=1; // Вводим сигнал готовности для
 // передачи выхода по связям

```

Реакция на функцию сброса выхода “UserManual.PictureClick.Reset” тоже будет похожей:

```

y=0; // Сбрасываем выход
Ready=1; // Вводим сигнал готовности для
 // передачи выхода по связям

```

Модель блока готова. Теперь, если поместить один управляющий и несколько управляемых блоков в одну и ту же подсистему (рис. 465) и запустить расчет, щелчки на зеленом круге управляющего блока будут одновременно увеличивать выходы всех управляемых, щелчки на красном – уменьшать их, а щелчки на белом – сбрасывать. При этом в подсистему можно добавлять сколько угодно управляемых блоков, все они будут получать команды от управляющего, и для этого пользователю не потребуется проводить новые связи.

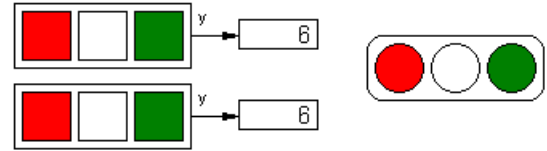


Рис. 465. Управляющий (справа) и управляемые (слева) блоки

В этом примере для увеличения, уменьшения и сброса выхода управляемого блока мы использовали три разных функции. Можно было обойтись одной, сделав команду блоку (уменьшить, увеличить или сбросить выход) параметром этой функции. По аналогии с ранее созданными, дадим этой новой функции имя “UserManual.PictureClick.Cmd” и придумаем для нее структуру параметров. Чтобы вызываемая модель могла проверить правильность переданных ей параметров, оформим параметры функции в виде структуры, первым полем которой будет размер этой структуры (см. стр. 246). Поскольку эта структура нам будет нужна и в модели управляющего, и в модели управляемого блоков, запишем ее в файл “pictureclick.h”, который разместим в одной папке с файлами моделей этих блоков – в этом случае в обеих моделях нам нужно будет просто добавить в глобальные описания команду “#include” для этого файла. В “pictureclick.h” нужно ввести следующее описание:

```

struct TPictureClickFuncParam
{
 DWORD servSize; // Размер структуры
 int Command; // Команда блоку
};

```

Первое поле структуры (servSize) будет содержать ее собственный размер, а в целое поле Command мы будем записывать команду управляемому блоку. Для упрощения модели сделаем так, чтобы команды совпадали с идентификаторами элементов векторной картинки, щелчки по которым должны изменять выходы блоков (эти идентификаторы мы сделали одинаковыми и в управляющем, и в управляемом блоках): 1 – уменьшить выход, 2 – увеличить, 3 – сбросить. Таким образом, в управляющем блоке в качестве посылаемой через функцию команды можно будет использовать идентификатор элемента картинки, на котором щелкнул пользователь.

Как и в прошлый раз, начнем с модели управляющего блока. Откроем ее редактор, сотрем три добавленных ранее функции (см. §3.6.5 на стр. 57) и добавим новую с именем “UserManual.PictureClick.Cmd” (рис. 466).

Длинное имя объекта для функции в программе, предлагаемое по умолчанию, заменим на rdsfuncUMPC\_Cmd (с коротким именем удобнее работать), а поле “тип параметра” на этот раз не будем оставлять пустым: введем в него “TPictureClickFuncParam\*”, то есть “указатель на TPictureClickFuncParam”. После нажатия “ОК” в списке функций блока будет единственная функция – “UserManual.PictureClick.Cmd” (рис. 467).



Рис. 466. Добавление функции “UserManual.PictureClick.Cmd” в модель блока

| Имя в C++       | Имя в RDS                   | Параметр                |
|-----------------|-----------------------------|-------------------------|
| rdsfuncUMPC_Cmd | UserManual.PictureClick.Cmd | TPictureClickFuncParam* |

Рис. 467. Вкладка “функции” редактора модели после добавления функции с параметром (размеры панели и колонок увеличены для наглядности)

Реакцию управляющего блока на нажатие кнопки мыши мы изменим следующим образом:

```

if (MouseData->Button==RDS_MLEFTBUTTON) // Нажата левая
{ TPictureClickFuncParam param; // Структура параметров
 param.servSize=sizeof(param); // Размер
 // Команда – это идентификатор элемента под курсором
 param.Command=rdsGetMouseObjectId(MouseData);
 rdsfuncUMPC_Cmd.Broadcast(// Вызов функции всех блоков
 rdsbcppBlockData->Parent,0,¶m);
}
else // Нажата не левая
 Result=RDS_BFR_SHOWMENU; // Разрешаем контекстное меню

```

Если нажата левая кнопка мыши, мы заполняем вспомогательную структуру param созданного нами типа TPictureClickFuncParam (команду включения файла “pictureclick.h”, в котором он описан, мы добавим позже) параметрами функции, которую мы будем вызывать. В поле servSize мы записываем размер самой структуры, полученный оператором sizeof, а в поле Command – идентификатор элемента картинки под курсором, как и раньше, полученный при помощи вызова rdsGetMouseObjectId. Затем мы вызываем уже знакомую нам функцию-член Broadcast у объекта функции rdsfuncUMPC\_Cmd – это приведет к вызову “UserManual.PictureClick.Cmd” у всех блоков



подсистемы. Поскольку у нашей новой функции есть параметр, в третьем параметре Broadcast мы передаем указатель на структуру param.

Если нажата не левая кнопка, мы, как и во всех предыдущих моделях, присваиваем переменной Result значение RDS\_BFR\_SHOWMENU, иначе контекстное меню блока не будет показано.

Можно обратить внимание на то, что теперь мы не проверяем, что вернула функция rdsGetMouseObjectId, то есть какой именно элемент картинки оказался под курсором. Если пользователь щелкнет мышью в пределах картинки блока, но не попадет ни в один из цветных кругов, rdsGetMouseObjectId вернет ноль, и этот ноль будет передан в качестве команды в параметре функции. Однако, такой команды у нас нет, поэтому управляемые блоки ничего не выполнят, и никаких последствий от такого вызова не будет.

Нам осталось только добавить в модель команду включения файла “pictureclick.h”, в котором описан тип параметра нашей функции. Для этого на вкладке “события” левой панели редактора (см. §3.6.4 на стр. 46) нужно раскрыть раздел “описания”, дважды щелкнуть на его подразделе “глобальные описания”, и на открывшейся пустой вкладке “описания” в правой части окна ввести единственную строчку:

```
#include "pictureclick.h"
```

Модель управляющего блока готова. Изменим теперь модель управляемого, чтобы она реагировала на новую функцию с параметром. Прежде всего, сотрем в ней три имеющихся на данный момент функций. Поскольку для этих функций введены реакции, редактор модели будет предупреждать об их существовании. Нужно согласиться на удаление функции вместе с текстом реакции – эти реакции в новой модели нам не понадобятся. Затем нужно добавить в список функцию “UserManual.PictureClick.Cmd” точно так же, как мы только что сделали в модели управляющего блока. И, как и в модели управляющего блока, добавить в глобальные описания команду для включения файла “pictureclick.h”.

Теперь нужно ввести реакцию на вызов функции “UserManual.PictureClick.Cmd”. Проще всего выбрать эту функцию на вкладке “функции” левой панели редактора (она там единственная) и нажать кнопку с листком в левой нижней части этой вкладки (см. рис. 467 на стр. 256). На открывшейся вкладке с названием функции нужно ввести следующий текст:

```
if (Param==NULL || Param->servSize<sizeof(TPictureClickFuncParam))
 return; // Нет параметра или недостаточный размер структуры
// Параметр в порядке
switch (Param->Command)
{
 case 1: y--; break;
 case 2: y++; break;
 case 3: y=0; break;
}
// Вводим сигнал готовности для передачи выхода по связям
Ready=1;
```

Прежде, чем разбирать текст этой реакции, следует обратить внимание на заголовок создаваемой для нее функции, который редактор модели показывает к верхней части вкладки. Он выглядит так:

```
void rdsbcppBlockClass::rdsfuncUMPC_CmdEvent (
 TPictureClickFuncParam* Param,
 RDS_PFUNCTIONCALLDATA FData,
 int &Result)
```

У функции реакции, текст которой мы пишем, три параметра. Параметр Param типа TPictureClickFuncParam\* – это тот самый параметр функции, который мы описывали при ее создании, и который передается от вызывающего блока в последнем параметре функции-члена Broadcast. Именно к Param мы будем обращаться, чтобы считать команду блока. Параметр FData представляет собой указатель на структуру описания функции (она описывается в §A.2.4.6 приложения к руководству программиста), из которой можно узнать,

какой именно блок вызвал функцию, сколько блоков уже вызвано и т.п. В нашей модели эта информация нам не потребуется, как и целый параметр `Result`, передаваемый по ссылке, через который функция может вернуть вызвавшему блоку одно целое число (наша функция ничего не возвращает).

Вернемся теперь к реакции на вызов функции. Сначала мы проверяем допустимость переданного параметра. Если `Param` равен `NULL`, значит, при вызове нашей функции не был передан параметр. Если `Param` не равен `NULL`, но первое поле структуры по этому указателю меньше размера структуры `TPictureClickFuncParam`, значит, вызвавшая модель передала указатель на неправильную структуру. В обоих случаях мы немедленно завершаем реакцию, поскольку считать команду из параметра функции невозможно. Если же параметр прошел обе проверки, мы читаем из переданной структуры поле `Command` и, в зависимости от его значения, увеличиваем, уменьшаем или сбрасываем выход блока, после чего принудительно взводим сигнал готовности `Ready` для передачи выхода по связям.

Если все было сделано правильно, схема с новыми моделями управляющего и управляемого блоков будет работать точно так же, как и со старыми: щелчки на зеленом круге управляющего блока будут одновременно увеличивать выходы всех управляемых, щелчки на красном – уменьшать их, а щелчки на белом – сбрасывать.

### §3.7.13.3. Вызов функции у одного блока

Описывается вызов функции у блока с известным идентификатором, рассматриваются способы определения этого идентификатора.

В предыдущем параграфе мы рассмотрели вызов функции у всех блоков подсистемы. Если идентификатор блока известен, можно вызвать функцию только у этого конкретного блока. Основной вопрос здесь – как получить идентификатор этого блока. Как правило, мы хотим вызвать функцию у блока, обладающего какой-то особенностью. Например, нам может быть известно полное имя этого блока, или этот блок может быть соединен связью с вызывающим. Рассмотрим оба этих варианта, начав с самого простого – будем вызывать функцию у блока, имя которого нам известно.

Созданная в предыдущем параграфе модель управляющего блока позволяла увеличивать, уменьшать и сбрасывать выходы у всех управляемых блоков в одной подсистеме. Сделаем еще одну модель, которая будет сбрасывать выход только у одного управляемого блока, полное имя которого мы будем задавать в настройках. Напомним, что 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.

Создадим в схеме новый блок с автокомпилируемой моделью (см. стр. 95), выберем на левой панели редактора вкладку “настройки”, нажмем в ее верхней части кнопку “+” (см. рис. 341 на стр. 61) и заполним окно добавления параметра согласно рис. 468. В поля этого окна введены следующие значения:

- на панели “переменная”:
  - ♦ “имя” – `“RemoteName”` (так будет называться наш настроечный параметр);
  - ♦ “тип” – `“rdsbcppString”` (это специальный класс для хранения строк, создаваемый модулем автокомпиляции);
  - ♦ “по умолчанию” – оставлено пустым;
- на панели “поле ввода”:
  - ♦ “добавить для этой переменной поле ввода” – установлен флажок;
  - ♦ “заголовок” – “имя связанного блока”;
  - ♦ “ширина” – 100;

- ♦ “ТИП” – “ВВОД”.

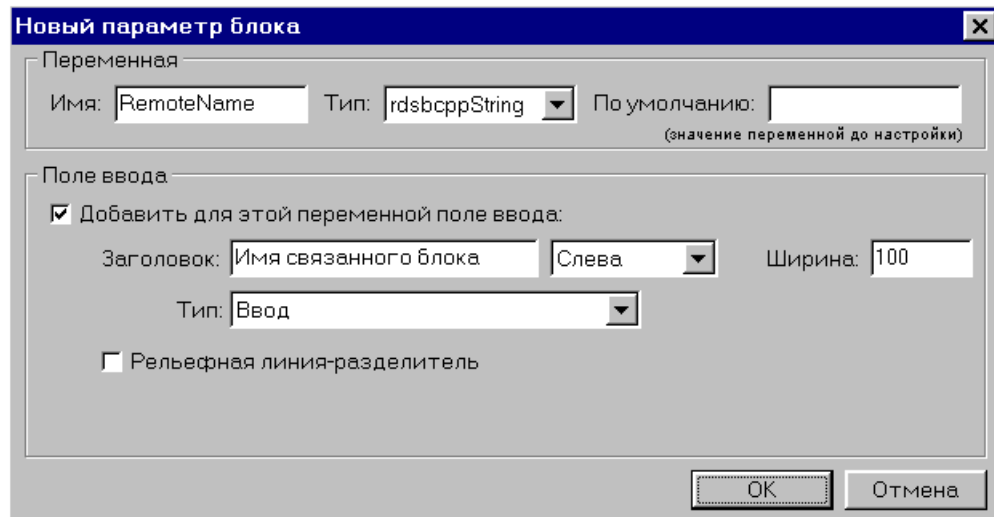


Рис. 468. Добавление в модель блока строкового параметра

После нажатия кнопки “ОК” в модель блока будет добавлен параметр с именем RemoteName, в который пользователь сможет вводить любую строку. Окно для ввода параметра и процедуры его загрузки и сохранения вместе со схемой будут добавлены в модель автоматически (см. §3.6.6 на стр. 60).

Наш новый блок будет искать в схеме другой блок, имя которого содержится в параметре RemoteName, и вызывать у него функцию “UserManual.PictureClick.Cmd” с параметром 3, которая сбросит его выход (эту функцию мы придумали и создали в конце предыдущего параграфа). Структура TPictureClickFuncParam, в которую записывается параметр функции, описана в файле “pictureclick.h”, поэтому, как и в предыдущих моделях, нам нужно добавить в модель команду для его включения. Раскроем на вкладке “события” левой панели редактора (см. §3.6.4 на стр. 46) раздел “описания”, дважды щелкнем на его подразделе “глобальные описания”, и, в открывшейся справа вкладке “описания”, введем эту команду:

```
#include "pictureclick.h"
```

Теперь нужно добавить в модель блока функцию “UserManual.PictureClick.Cmd” – сделаем это точно так же, как делали в предыдущем параграфе (стр. 255), и, точно так же, переименуем для краткости создаваемый для функции объект в “rdsfuncUMPC\_Cmd”.

Сбрасывать значение выхода блока, имя которого записано в параметре RemoteName (то есть вызывать его функцию), наш блок будет по щелчку пользователя. Добавим в нашу модель реакцию на нажатие кнопки мыши. Раскроем раздел “мышь и клавиатура” на вкладке “события” (см. рис. 447 на стр. 226), дважды щелкнем на подразделе “нажатие кнопки мыши”, и на появившейся вкладке введем следующий текст:

```
if(MouseData->Button==RDS_MLEFTBUTTON) // Нажата левая
{ RDS_BHANDLE b;
 // Получаем идентификатор блока по полному имени
 b=rdsBlockByFullName(RemoteName.c_str(),NULL);
 if(b!=NULL) // Такой блок есть
 { TPictureClickFuncParam param; // Структура параметров
 param.servSize=sizeof(param); // Размер
 param.Command=3; // Команда (сброс)
 rdsfuncUMPC_Cmd.Call(b,¶m); // Вызов функции
 }
}
```

```

else // Нажата не левая
 Result=RDS_BFR_SHOWMENU; // Разрешаем контекстное меню

```

Как и в предыдущих моделях, все действия мы выполняем только тогда, когда нажата левая кнопка мыши, то есть если поле `Button` структуры `MouseData` равно константе `RDS_MLEFTBUTTON`. В этом случае мы вызываем функцию `rdsBlockByFullName` (см. §A.5.6.3 приложения к руководству программиста [2]), которая ищет в схеме блок по его полному имени. Функция возвращает уникальный идентификатор блока, который имеет принятый в РДС тип `RDS_BHANDLE`: его мы записываем во вспомогательную переменную `b`. В первом параметре функции `rdsBlockByFullName` передается строка (`char*`) с полным именем блока. Поскольку `RemoteName` – это объект класса `rdsbcppString`, хранящего эту строку, для доступа к самой строке внутри этого объекта мы используем его функцион-член `c_str` без параметров. Во втором параметре `rdsBlockByFullName` может передаваться указатель на структуру описания, которую эта функция заполняет параметрами найденного блока. Нам это не нужно, поэтому мы передаем `NULL`.

Если блок с именем, записанным в `RemoteName`, существует в схеме, `rdsBlockByFullName` вернет ненулевой идентификатор, то есть значение `b` не будет равно `NULL`. В этом случае мы заполняем вспомогательную структуру `param` параметрами вызываемой функции (мы уже добавили в глобальные описания модели команду включения файла “pictureclick.h”, в котором описан тип этой структуры `TPictureClickFuncParam`, поэтому мы имеем право использовать ее в модели). В поле `servSize` мы, как обычно, записываем размер самой структуры, а в поле команды `Command` – число 3, поскольку для созданных нами блоков это число считается командой сброса. Заполнив структуру параметров, мы вызываем функцию “`UserManual.PictureClick.Cmd`” у блока `b`, передавая в качестве ее параметра указатель на структуру `param`. Выглядит этот вызов так:

```

rdsfuncUMPC_Cmd.Call(b, ¶m);

```

Здесь `rdsfuncUMPC_Cmd` – это объект, созданный модулем автокомпиляции для функции “`UserManual.PictureClick.Cmd`” (мы указывали имя объекта при добавлении функции в модель), а `Call` – одна из функций-членов этого объекта.

Если нажата не левая кнопка мыши, мы, как и раньше, присваиваем переменной `Result` значение `RDS_BFR_SHOWMENU`, чтобы не блокировать контекстное меню.

Разрешим для созданного нами блока реакцию на мышшь (см. рис. 448 на стр. 229), и введем в окне его настройки, вызываемом пунктом “настройка” контекстного меню, полное имя какого-нибудь уже имеющегося в схеме блока, который увеличивает, уменьшает или сбрасывает свой выход по щелчкам мыши (рис. 469). Если отображение имен блоков запрещено, имя любого блока можно прочесть в строке состояния подсистемы, выделив его.

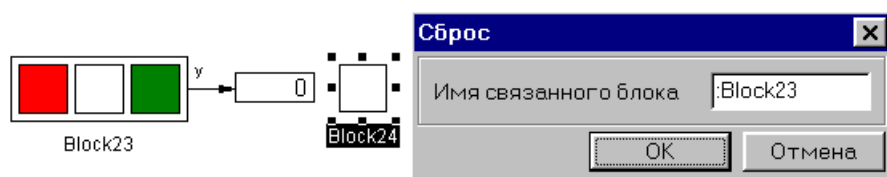


Рис. 469. Управляемый блок “Block23” (слева), сбрасывающий блок (в центре, выделен) и окно его настройки (справа)

Теперь, если запустить расчет, щелчок на созданном нами блоке будет сбрасывать выход блока, имя которого введено в окне настроек. При этом, поскольку в настройках мы задаем полное имя связанного блока, наш новый блок можно разместить в любой подсистеме. В отличие от управляющего блока, созданного в §3.7.13.2 (стр. 250), он не обязан находиться в одной подсистеме с управляемым, как на рис. 469.

Достаточно часто бывает нужно вызвать какую-либо функцию у блоков, соединенных связями с данным блоком. Часто такой вызов используют для организации передачи данных

в режимах моделирования и редактирования, в которых связи не работают (такой пример будет рассмотрен ниже). В некоторых, более редких, случаях, связи, соединяющие блоки, вообще не используются для передачи данных – вместо этого они просто показывают блокам их соседей, а передача данных между ними производится только за счет вызовов функций. В §2.13.4 руководства программиста [1], например, рассматривается использование вызовов функций для поиска путей в графе, составленном из блоков и связей. Здесь мы не будем рассматривать подобные примеры из-за их сложности, вместо этого добавим в один из ранее созданных нами блоков возможность передачи данных по связям в режиме моделирования.

С точки зрения пользователя, у созданного ранее блока, который по щелчкам мыши на элементах картинки увеличивает, уменьшает или сбрасывает в ноль значение своего выхода (см. §3.7.11, стр. 230), и который мы использовали в качестве примера начиная с §3.7.13.2 (стр. 250), есть один, достаточно серьезный, недостаток: он не позволяет непосредственно ввести значение выхода с клавиатуры. Можно, конечно, организовать ввод числа с клавиатуры в модели этого блока, однако, это потребует слишком больших усилий. Гораздо проще будет добавить в переменные блока целый вход, значение которого без изменений передается на выход, и соединить его связью со стандартным полем ввода, в которое пользователь будет вводить число (рис. 470). Чтобы увеличение, уменьшение и сброс, выполняемые нашим блоком, отражались и на поле ввода, вход поля ввода нужно будет соединить с выходом нашего блока, замкнув их в кольцо. Таким образом, ввод числа в поле ввода будет приводить к тому, что значение с выхода этого поля будет по связи передаваться на вход нашего блока и далее, без изменения, на его выход. Щелчки на нашем блоке будут приводить к изменению значения на его выходе, которое будет по связи попадать в поле ввода и появляться в нем. В результате, выход поля ввода и нашего блока будут все время синхронизированы.

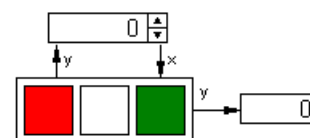


Рис. 470. Кольцо из блока и поля ввода

Однако, все это будет работать только в режиме расчета, то есть в единственном режиме РДС, в котором значения передаются по связям. Если остановить расчет, щелчки на нашем блоке будут изменять значение его выхода, но в поле ввода они попадать не будут – связь между ними не будет работать до запуска расчета. Чтобы заставить кольцо из нашего блока и поля ввода синхронно работать и в режиме моделирования, нужно использовать функцию “Common.ControlValueChanged”, которую поддерживают все библиотечные блоки пользовательского интерфейса: поля ввода, ручки и т.п. Работает она так: при изменении значения в одном из таких блоков модель этого блока *принудительно* передает данные выходов по связям независимо от режима РДС (для этого используется специальная сервисная функция `rdsActivateOutputConnections`, см. §A.5.6.1 приложения к руководству программиста), а затем вызывает у всех соединенных с этими выходами блоков функцию “Common.ControlValueChanged”. У этой функции нет параметров, блоки должны считать измененное пользователем значение со своих входов. Таким образом, чтобы синхронизировать наш блок с полем ввода в любом режиме, нам нужно, во-первых, при любом изменении выхода блока *принудительно* передавать новое значение по связям и вызывать у всех блоков, соединенных с этим выходом, функцию “Common.ControlValueChanged” (так выход нашего блока попадет в поле ввода), и, во-вторых, реагировать на вызов этой функции, передавая свой вход на выход (так значение из поля ввода попадет на выход нашего блока).

Здесь очень важно вовремя прервать цепочку вызовов. Если не предпринять никаких специальных мер, при вводе нового значения в поле ввода это поле вызовет наш блок, наш блок установит это значение на своем выходе и вызовет поле ввода, поле ввода примет это значение и снова вызовет наш блок, наш блок снова вызовет поле ввода и т.д. Это будет продолжаться до тех пор, пока стек приложения не переполнится и не произойдет аварийное

завершение РДС. Чтобы избежать этого, проще всего ввести в данные блока специальный флаг, который мы будем взводить перед вызовом “Common.ControlValueChanged” у соединенных блоков и сбрасывать после него. Реагируя на вызов функции, модель будет проверять этот флаг: если он взведен, значит, данный блок сам уже вызвал у кого-то “Common.ControlValueChanged” – то есть, он уже принимает участие в цепочке вызовов, и снова вызывать эту функцию у соседей не нужно (подробнее о применении такого флага можно прочесть в §2.13.2 руководства программиста).

Таким образом, чтобы дать пользователю возможность объединять наш блок с полем ввода или какими-либо другими интерфейсными блоками, нам нужно внести в его модель следующие изменения:

- добавить в модель целый вход для подключения выходов других блоков и логический флаг для блокировки повторного вызова функции;
- для передачи нового значения выхода в соединенные блоки при любых изменениях выхода принудительно активировать выходные связи и вызывать у подключенных к ним блоков функцию “Common.ControlValueChanged”;
- для приема значения, измененного другими блоками, добавить реакцию на вызов “Common.ControlValueChanged”, в которой значение входа будет передаваться на выход (при этом необходимо принудительно активировать выходные связи и выполнить все действия, описанные в предыдущем пункте);
- в реакции на такт расчета просто передавать значение входа на выход, чтобы блок работал в режиме расчета так же, как и все обычные блоки.

Самое сложное среди этих действий – поиск идентификаторов блоков, подключенных к выходу данного, для вызова у них “Common.ControlValueChanged”. Это можно выполнить двумя способами:

1. В цикле перебрать все связи, подключенные к выходу блока, при помощи функции `rdsGetBlockLink` (см. §A.5.6.20 приложения к руководству программиста). Для каждой из этих связей перебрать все ее точки и, при помощи функции `rdsGetPointDescription` (см. §A.5.6.36 там же), получить идентификатор блока, соединенного с точкой.
2. Вызвать функцию `rdsEnumConnectedBlocks` (см. §A.5.6.12 там же), которая переберет все блоки, соединенные с данным, и для каждого из них вызовет функцию специального вида, указатель на которую мы передадим в параметрах.

Мы выберем второй путь. Хотя программа модели при этом может показаться более запутанной, она будет значительно короче, поскольку не нужно будет организовывать два вложенных цикла и анализировать точки связей и их параметры – это за нас сделает РДС.

Начнем изменять модель все того же блока с картинкой, по которой может щелкать пользователь – эту модель мы рассматриваем, начиная с §3.7.11. Прежде всего, добавим в блок две новых переменных: целый вход `x`, к которому будет подключаться связь от поля ввода, и внутреннюю логическую переменную `NoCall`, которая будет использоваться как флаг блокировки лишних вызовов “Common.ControlValueChanged”, предохраняющий от бесконечной рекурсии этих вызовов. Новая структура переменных нашего блока будет выглядеть так:

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| Start      | Сигнал     | Вход              | ✓           | 0                         |
| Ready      | Сигнал     | Выход             |             | 0                         |
| y          | int        | Выход             |             | 0                         |
| x          | int        | Вход              | ✓           | 0                         |
| NoCall     | Логический | Внутренняя        |             | 0                         |

Теперь добавим в модель функцию “Common.ControlValueChanged”. Выберем на левой панели редактора вкладку “функции” – после последнего изменения модели, сделанного в §3.7.13.2, в списке на этой панели должна быть единственная функция “UserManual.PictureClick.Cmd” (см. рис. 467 на стр. 256). Нажмем на вкладке кнопку “+” и, в открывшемся окне, установим флажок “стандартная функция” и выберем в выпадающем списке справа от него “изменение присоединенного поля” (рис. 471). Поскольку мы добавляем стандартную функцию, поля “имя функции в RDS” и “тип параметра функции” будут заблокированы – для стандартной функции их изменить нельзя. Можно только изменить имя объекта, с помощью которого мы будем вызывать функцию в программе. Сделаем это: для краткости переименуем объект в “rdsfuncCVC”.

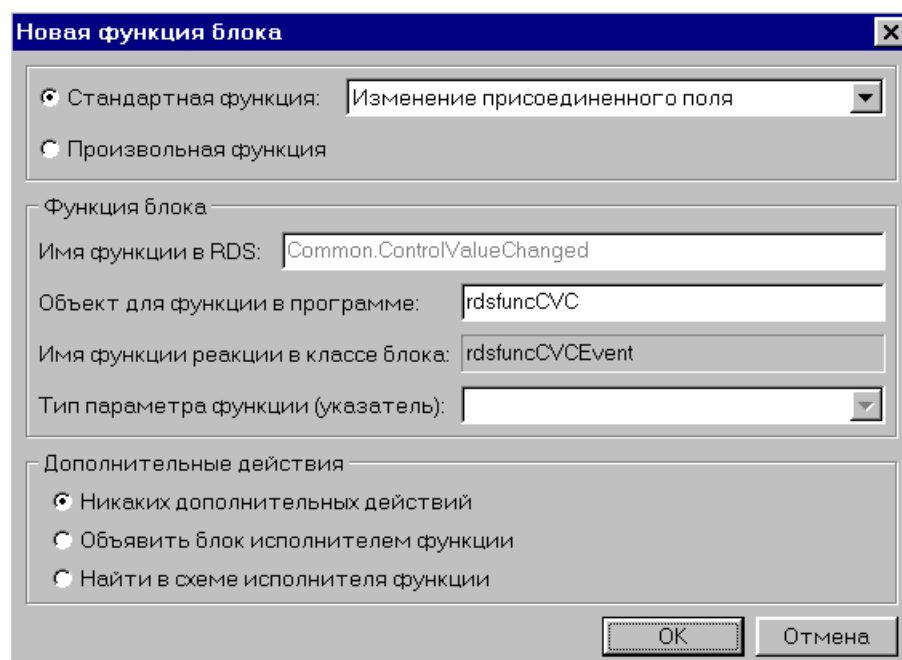


Рис. 471. Добавление в модель блока стандартной функции “Common.ControlValueChanged”

Вызывать “Common.ControlValueChanged” у присоединенных блоков нам предстоит в нескольких местах модели, поэтому этот вызов мы оформим в виде отдельной функции. Причем, поскольку нам нужен доступ к флагу NoCall, который будет блокировать лишние вызовы, нам нужно сделать эту функцию членом класса блока – только функции-члены класса блока имеют доступ к статическим переменным этого блока. Выберем на левой панели редактора вкладку “события”, раскроем на ней раздел “описания”, а затем дважды щелкнем на его пункте “описания внутри класса блока”. На открывшейся справа пустой вкладке введем следующий текст:

```
// Вызов функции у всех присоединенных блоков
void InformNeighbours(void)
{ if (NoCall) // Вызов запрещен
 return;

 // Принудительно активируем выходные связи
 rdsActivateOutputConnections(NULL, TRUE);

 NoCall=1; // Вводим флаг блокировки
 // Перебираем все соединенные блоки
 rdsEnumConnectedBlocks(
 NULL,
 RDS_BEN_OUTPUTS | RDS_BEN_TRACELINKS,
```



```

 ControlValChanged_Callback, // Функция обратного вызова
 NULL);
 NoCall=0; // Сбрасываем флаг блокировки
};

```

Здесь мы описываем функцию с именем `InformNeighbours`. Поскольку мы описали ее внутри класса блока, она станет членом этого класса. В этой функции мы, прежде всего, проверяем, не взведен ли флаг `NoCall`. Если он взведен, значит, этот блок уже вызвал “`Common.ControlValueChanged`” у своих соседей, и повторять этот вызов не нужно – мы немедленно завершаем работу функции (так мы прерываем рекурсию вызовов в кольце блоков). В противном случае мы принудительно передаем по связям данные всех выходов нашего блока (в данном случае, у нас единственный выход `y`) при помощи функции `rdsActivateOutputConnections` (см. §A.5.6.1 приложения к руководству программиста). Эта функция принимает два параметра: идентификатор блока, выходы которого нужно передать по связям (мы передаем `NULL`, что означает, что передаются выходы того блока, из модели которого вызвана функция), и логическое значение, разрешающее использование обычной логики передачи данных РДС (мы передаем `TRUE`, то есть разрешаем использование логики – нам нужна самая обычная передача данных по связям, аналогичная автоматически работающей в режиме расчета).

После принудительной передачи данных выхода на входы соединенных блоков мы взводим флаг `NoCall` (теперь повторный вызов `InformNeighbours` для данного блока будет временно запрещен) и вызываем функцию `rdsEnumConnectedBlocks` (см. §A.5.6.12 приложения к руководству программиста) чтобы перебрать все блоки, соединенные связями с данным. Эта функция принимает четыре параметра:

- идентификатор блока, соседи по связям которого перебираются (мы передаем `NULL` – будут перебраны соседи того блока, из модели которого вызвана функция);
- флаги, управляющие перебором блоков (мы передаем объединенные битовым ИЛИ `RDS_BEN_OUTPUTS` и `RDS_BEN_TRACELINKS` – будут перебираться только блоки, соединенные с выходом данного, и связи будут прослеживаться внутрь и наружу подсистем);
- указатель на функцию обратного вызова (обычную функцию языка C, не “функцию блока” РДС), которая будет вызвана для каждого найденного блока (мы передаем `ControlValChanged_Callback`, нам еще предстоит написать функцию с таким именем);
- указатель на дополнительные данные, передаваемые в функцию обратного вызова (мы передаем `NULL` – у нас нет таких данных).

После того, как все соседи блока перебраны, мы сбрасываем флаг `NoCall` (теперь уведомление соседей блока об изменении его выхода снова разрешено). На этом работа `InformNeighbours` завершается.

Хотя мы создали функцию `InformNeighbours` для того, чтобы информировать блоки, соединенные связями с данным, об изменении его выхода при помощи вызова “`Common.ControlValueChanged`”, можно заметить, что сам этот вызов внутри созданной функции отсутствует. Мы должны вставить его в функцию обратного вызова с именем `ControlValChanged_Callback`: функция `InformNeighbours` через сервисную функцию `rdsEnumConnectedBlocks` будет вызывать эту функцию обратного вызова для каждого блока, соединенного с выходом данного, а функция обратного вызова уже должна информировать конкретный переданный ей блок об изменении значения на его входе.

Функция обратного вызова, используемая в `rdsEnumConnectedBlocks`, должна иметь следующий вид:

```

BOOL RDSCALL имя_функции(
 RDS_PPOINTDESCRIPTION nearpoint, // Точка связи данного блока

```



```
RDS_PPOINTDESCRIPTION farpoint, // Точка связи "соседа"
LPVOID ptr); // Дополнительный параметр
```

В первом параметре этой функции (nearpoint) передается указатель на структуру RDS\_POINTDESCRIPTION (см. §A.4.14 приложения к руководству программиста), описывающую точку связи, соединенную с выходом данного блока. Во втором параметре (farpoint) передается указатель на такую же структуру, описывающую точку, соединенную со входом другого блока. Анализируя поля этих структур, можно узнать, какая именно переменная данного блока соединена с другим блоком, имя соединенной переменной в другом блоке и идентификатор этого другого блока. Нам нужен будет только идентификатор блока из структуры farpoint – у него мы будем вызывать “Common.ControlValueChanged”. В третьем параметре (ptr) передается указатель на дополнительные данные из последнего параметра rdsEnumConnectedBlocks – мы не передаем никаких дополнительных данных, поэтому этот параметр нас не волнует. Функция должна вернуть TRUE, если перебор блоков нужно продолжить, и FALSE, если его нужно прервать – мы будем перебирать все блоки и всегда возвращать TRUE.

Теперь нам нужно написать функцию такого вида с именем ControlValChanged\_Callback, но прежде нужно разобраться с одной проблемой, возникающей при этом. По правилам языка С любой идентификатор, будь то переменная или функция, должен быть описан до места своего использования. Наша функция ControlValChanged\_Callback используется в параметре rdsEnumConnectedBlocks внутри функции InformNeighbours, которая вставлена в описания внутри класса блока. Функция обратного вызова не может быть членом класса блока (она должна быть обычной функцией C), поэтому вставить ее в те же описания в классе перед InformNeighbours мы не можем. Таким образом, описать функцию обратного вызова мы должны до класса блока, а единственные доступные пользователю описания, которые можно вставить до этого класса – это глобальные описания. Однако, если мы запишем ControlValChanged\_Callback в глобальных описаниях, она окажется перед тем местом в программе, в котором описываются объекты для вызова функций (они описываются там же, где и класс блока), поэтому мы не сможем использовать в ней объект rdsfuncCVC, необходимый для вызова “Common.ControlValueChanged”.

Решить проблему просто: в глобальных описаниях мы запишем только прототип функции ControlValChanged\_Callback, а ее тело, в котором будет использоваться объект rdsfuncCVC, разместим в описаниях после класса блока. Откроем раздел глобальных описаний (вкладка “события” на левой панели редактора модели – раскрыть раздел “описания” – дважды щелкнуть на пункте “глобальные описания”) и добавим туда к уже имеющейся там команде включения файла “pictureclick.h” прототип функции:

```
#include "pictureclick.h"
```

```
// Прототип функции обратного вызова
BOOL RDSCALL ControlValChanged_Callback(
 RDS_PPOINTDESCRIPTION, RDS_PPOINTDESCRIPTION, LPVOID);
```

Теперь добавим тело этой функции в описания после класса блока. На вкладке “события” в разделе “описания” дважды щелкнем на пункте “описания после класса блока” и на открывшейся вкладке введем текст:

```
// Функция обратного вызова для "Common.ControlValueChanged"
BOOL RDSCALL ControlValChanged_Callback(
 RDS_PPOINTDESCRIPTION /*nearpoint*/,
 RDS_PPOINTDESCRIPTION farpoint,
 LPVOID /*ptr*/)
{ // Вызов функции "Common.ControlValueChanged" у блока на другом
 // конце связи
 rdsfuncCVC.Call(farpoint->Block);
```

```

 // Возвращаем TRUE – не останавливаем перебор блоков
 return TRUE;
 }

```

В параметре `farpoint` этой функции будет передан указатель на структуру, описывающую точку связи, соединенную с найденным блоком на другом конце этой связи от нашего. В поле `Block` этой структуры записан идентификатор этого блока – он нам и нужен для вызова. Функция “`Common.ControlValueChanged`” не имеет параметров, поэтому функция-член `Call` объекта `rdsfuncCVC`, созданного для вызова функции, будет иметь единственный параметр – идентификатор вызываемого блока, то есть `farpoint->Block`. Фактически, наша функция обратного вызова состоит только из вызова “`Common.ControlValueChanged`” через ее объект `rdsfuncCVC` для блока `farpoint->Block`. После этого вызова мы возвращаем `TRUE`, чтобы перебор блоков, соединенных связями с данным, продолжался.

Теперь нам нужно добавить вызов функции `InformNeighbours` при каждом изменении выхода блока. Сначала добавим его в уже имеющуюся реакцию на нажатие кнопки мыши (вкладка “события” на левой панели редактора модели – раскрыть раздел “мышь и клавиатура” – дважды щелкнуть на пункте “нажатие кнопки мыши”). Добавленный текст выделен двойным подчеркиванием:

```

if(MouseData->Button==RDS_MLEFTBUTTON) // Нажата левая
{ switch(rdsGetMouseObjectId(MouseData)) // Идентификатор
 { case 1: y--; break; // Красный квадрат
 case 2: y++; break; // Зеленый квадрат
 case 3: y=0; break; // Белый квадрат
 }
 // Вводим сигнал готовности для передачи выхода по связям
 Ready=1;
 // Информировать соседние блоки
 InformNeighbours();
}
else // Нажата не левая
 Result=RDS_BFR_SHOWMENU; // Разрешаем контекстное меню

```

Теперь при щелчках на картинке блока в режиме моделирования будет вызываться функция `InformNeighbours`, принудительно передающая выход блока на входы блоков, соединенных с ним, и сообщаящая им об этом вызовом “`Common.ControlValueChanged`”.

В реакцию на вызов у блока функции “`UserManual.PictureClick.Cmd`”, введенную в модель в конце §3.7.13.2 (стр. 255), тоже вставим вызов `InformNeighbours`. На вкладке “события” раскроем раздел “функции блока” и дважды щелкнем на его пункте “`UserManual.PictureClick.Cmd`”. Изменим текст на открывшейся справа вкладке следующим образом (добавленные строки выделены двойным подчеркиванием):

```

if(Param==NULL || Param->servSize<sizeof(TPictureClickFuncParam))
 return; // Нет параметра или недостаточный размер структуры
// Параметр в порядке
switch(Param->Command)
{ case 1: y--; break;
 case 2: y++; break;
 case 3: y=0; break;
}
// Вводим сигнал готовности для передачи выхода по связям
Ready=1;
// Информировать соседние блоки
InformNeighbours();

```

Теперь и при внешних командах, отданных блоку через вызов “UserManual.PictureClick.Cmd”, он будет принудительно передавать данные выхода в соседние блоки.

Добавим в модель реакцию на такт расчета (вкладка “события” – раскрыть раздел “моделирование и режимы” – дважды щелкнуть на пункте “модель”) – раньше, до появления у блока входа, эта реакция не была нужна:

```
if(y==x) // Выход равен входу – ничего не делаем
{ Ready=0;
 return;
}
// Копируем вход в выход и уведомляем соседей
y=x;
InformNeighbours();
```

Здесь, если поступившее на вход  $x$  значение не отличается от  $y$ , мы сбрасываем сигнал готовности блока Ready, чтобы данные его выхода не передавались по связям, и завершаем модель. Это экономит процессорное время – на вход блока могут часто поступать одинаковые значения (например, если блок, передающий их, по каким-то причинам срабатывает каждый такт), а передавать их на выход повторно нет никакой необходимости. Если же выход отличается от входа, мы присваиваем выходу новое значение и вызываем InformNeighbours. Может возникнуть вопрос: зачем вызывать InformNeighbours для принудительной передачи данных соседним блокам, если в режиме расчета данные с выхода блока и так передаются по связям? Конечно, можно и не делать этого, но, поскольку мы предполагаем, что наш блок будет частью кольца соединенных блоков, лучше передать данные его выхода как можно быстрее – это позволит убрать колебания, которые могут возникнуть при подключении по кольцу. Представим себе, что на выходе нашего блока и на выходе соединенного с ним поля ввода почему-то оказались разные значения, и у обоих блоков взведен сигнал готовности. В этом случае в каждом такте расчета поле ввода будет передавать свое значение нашему блоку, а наш блок – другое значение полю ввода: блоки будут обмениваться значениями выходов и взводить сигнал готовности, и эти колебания будут продолжаться все время расчета. Принудительная передача данных по связям “выравнивает” значения всех выходов: на момент конца такта, когда происходит обычная передача, значения на выходах блоков уже будут одинаковыми, а сигналы готовности – сброшенными (их сбрасывает rdsActivateOutputConnections после передачи).

Теперь нам нужно ввести в наш блок реакцию на вызов “Common.ControlValueChanged”, чтобы он, как и стандартные блоки, мог получать значение по связи от другого блока (например, от поля ввода) вне режима расчета. На вкладке “события” раскроем раздел “функции блока”, дважды щелкнем на его пункте “Common.ControlValueChanged”, и введем на открывшейся вкладке следующий текст:

```
if(y==x) // Выход равен входу
 return;
if(NoCall) // Вызов блокирован
 return;
y=x; // Копируем с входа
Ready=1; // Взводим готовность
// Информировать соседей
InformNeighbours();
```

Здесь мы проверяем, поступило ли на вход действительно новое значение и не заблокирован ли временно вызов функций. В обоих случаях мы немедленно завершаем реакцию – если вход не отличается от выхода, или если данный блок уже находится в цепочке последовательных вызовов функции в кольце блоков, модель не выполняет никаких действий. В противном случае мы копируем значение входа на выход, взводим сигнал

готовности, разрешая тем самым передачу данных выхода по связям, и принудительно передаем данные связанным блокам, вызывая `InformNeighbours`.

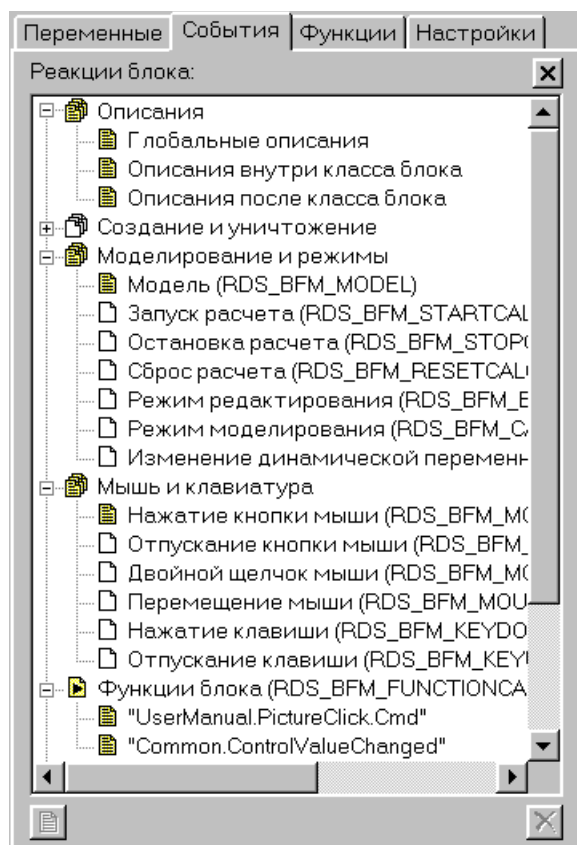


Рис. 472. Список событий и описаний блока

Для добавления в наш блок поддержки функции `“Common.ControlValueChanged”` нам потребовалось ввести в него достаточно много изменений и добавить в него несколько новых реакций и описаний (полный список фрагментов программы блока после всех изменений изображен на рис. 472). Зато теперь, если собрать схему, изображенную на рис. 470 (стр. 261), то в режиме моделирования щелчки на нашем (цветном) блоке будут отражаться на значении поля ввода, а изменения, вносимые в поле, будут отражаться на выходе нашего блока (на индикаторе, подключенном к нему). Это стало возможным, поскольку все блоки на рисунке, включая индикатор, поддерживают функцию `“Common.ControlValueChanged”` и могут передавать и принимать данные с ее помощью, минуя стандартные механизмы связей РДС.

В нашем примере у блока был единственный выход. Если у блока несколько выходов, и мы хотим добавить в него столько же входов и ввести поддержку `“Common.ControlValueChanged”`, следует обязательно каким-то образом узнавать, из-за изменения какого именно входа вызвана функция. Если не делать этого, можно потерять

измененное значение одного или нескольких входов – причины и механизм этого явления подробно рассматриваются в §2.13.2 руководства программиста на примере модели блока, аналогичного двухкоординатной рукоятке из §3.7.11 (стр. 232). Здесь мы не будем подробно рассматривать способы организации работы интерфейсных блоков с несколькими входами – скажем только, что проще всего в таких случаях привязать к каждому входу управляющий сигнал, автоматически взводящийся при срабатывании связи, подключенной к этому входу (см. §3.7.2.7 на стр. 123), и игнорировать значения входов, сигналы которых не взведены.

#### §3.7.13.4. Регистрация и поиск исполнителя функции

Описывается способ регистрации блока как исполнителя какой-либо функции, что позволит другим блокам легко находить его. Также описывается поиск блока-исполнителя для известной функции.

Регистрация блока в качестве исполнителя функции позволяет другим блокам, расположенным в одной с ним подсистеме или в подсистемах внутри нее, находить этот блок в схеме и вызывать нужную им функцию непосредственно у него. Таких исполнителей может быть в схеме несколько – каждый из них будет обслуживать свою ветвь иерархии схемы, начиная с собственной подсистемы. Поиском исполнителей функции и слежением за их появлением и исчезновением занимается РДС. Любому блоку, модель которого запросила информацию об исполнителе, постоянно предоставляется информация о ближайшем исполнителе в иерархии – после первичного запроса, модели не нужно выполнять каких-либо действий для поддержания информации об исполнителе в актуальном состоянии. Так можно довольно просто добавлять в схему новые функции, доступные блокам независимо от

их размещения: можно организовать ведение журналов событий, связь с внешними приложениями и т.п.

Работа с блоками-исполнителями функций состоит из двух частей: регистрации, выполняемой блоком, у которого будет вызываться функция, и подписки на исполнителя, выполняемой блоком, который будет вызывать функцию. Модуль автокомпиляции автоматизирует обе части: для того, чтобы объявить блок исполнителем функции, достаточно просто включить одноименный флажок при добавлении функции в модель. Точно так же, флажком при добавлении функции в модель, включается и подписка блока на исполнителя функции. Для вызова функции у исполнителя, на который подписан блок, у объекта, созданного для этой функции, вызывается функция-член `Call` без указания идентификатора блока. Следует, однако, помнить, что такой вызов возможен только непосредственно из модели блока, то есть из функции, являющейся членом класса блока. Если вызов производится из функции, не принадлежащей классу блока (например, из функции обратного вызова, подобной функции `ControlValChanged_Callback` из §3.7.13.3), `Call` без идентификатора блока будет недоступна. В этом случае нужно как-то передать в эту функцию идентификатор исполнителя, полученный через функцию-член `Provider` (подробнее о технической реализации вызовов функций через объекты можно прочесть в §3.7.13.5 на стр. 279).

Рассмотрим простой пример: создадим блок, который, при вызове у него функции, будет выводить сообщение пользователю. В параметрах функции будет содержаться текст сообщения и его тип: информационное сообщение, предупреждение или сообщение об ошибке. Назовем эту функцию “`UserManual.Message`”, а параметры ее оформим в виде структуры, первым полем которой будет размер этой структуры (см. стр. 246). Структуру мы запишем в файл “`UserMessage.h`”, который разместим в одной папке с файлом модели блока. Создадим этот файл “`UserMessage.h`” (например, в “блокноте” Windows) и введем в него следующий текст:

```
struct TUserMessageFuncParam
{ DWORD servSize; // Размер структуры
 int Type; // Тип сообщения
 char *Message; // Текст сообщения
};
// Типы сообщений
#define UMFP_TYPE_INF 0 // Информационное
#define UMFP_TYPE_WARN 1 // Предупреждение
#define UMFP_TYPE_ERR 2 // Ошибка
```

Первое поле структуры (`servSize`) будет содержать ее собственный размер, чтобы вызванный блок мог сравнить его с действительным размером структуры и понять, правильно ли переданы параметры. В поле `Type` будет храниться тип сообщения: 0 – информационное, 1 – предупреждение, 2 – сообщение об ошибке. В поле `Message` должен быть записан указатель на строку с текстом сообщения. Чтобы не запоминать, какое целое число соответствует какому типу сообщения, после описания структуры мы вводим `define`-константы для каждого из перечисленных выше типов.

Создадим в схеме новый блок с автокомпилируемой моделью (см. стр. 95) и сразу введем в эту модель команду для включения файла с описанной нами структурой: на вкладке “события” левой панели редактора раскроем раздел “описания”, дважды щелкнем на его пункте “глобальные описания” и на открывшейся справа пустой вкладке “описания” введем единственную строчку:

```
#include "UserMessage.h"
```

Теперь добавим в модель функцию “`UserManual.Message`”. На вкладке “функции” левой панели нажмем кнопку “+” и заполним открывшееся окно согласно рис. 473.

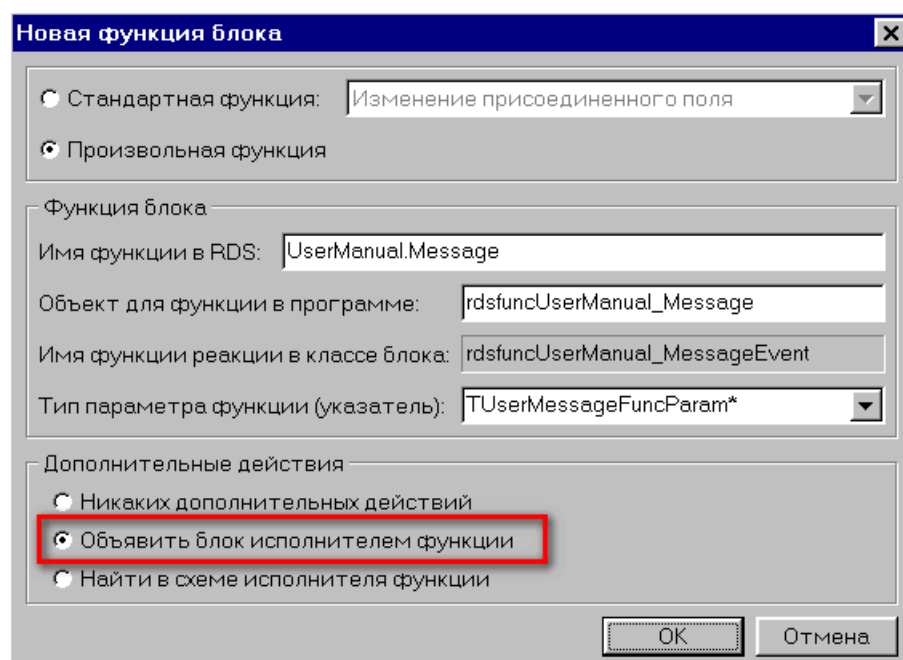


Рис. 473. Добавление в модель блока функции “UserManual.Message”  
(рамкой выделен флажок, регистрирующий блок как исполнителя функции)

На верхней панели окна выбран флажок “произвольная функция” – мы добавляем функцию, которую придумали сами. На панели “имя функции в RDS” введено название нашей функции. Поле “объект для функции в программе” можно оставить без изменений: хотя модуль автокомпиляции и предлагает, как обычно, слишком длинное имя для объекта, в этой модели мы не будем обращаться к этому объекту вручную, поэтому не важно, как он называется. В поле “тип параметра функции” введен тип указателя на описанную нами структуру TUserMessageFuncParam, то есть “TUserMessageFuncParam\*”. Наконец, на панели “дополнительные действия”, в отличие от предыдущих примеров, выбран флажок “объявить блок исполнителем функции”.

После нажатия “ОК” новая функция появится в списке функций на вкладке (рис. 474). Можно заметить, что, кроме иконки с пустым листом (указывающей, что мы еще не ввели текст реакции на эту функцию), рядом с ее именем будет находиться знак “+” – так помечаются функции, исполнителем которых является блок. Теперь с момента подключения модели к блоку и до ее отключения (например, из-за удаления блока из схемы) наш блок будет считаться исполнителем функции “UserManual.Message”.

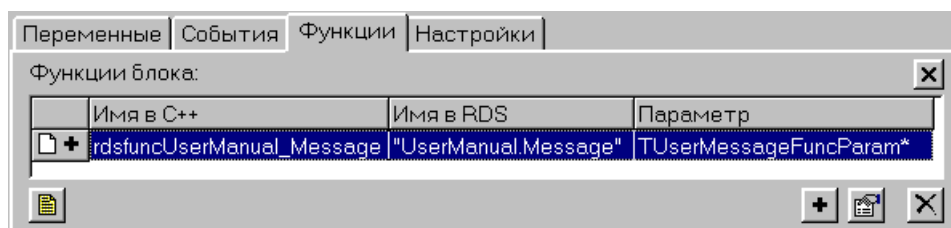


Рис. 474. Список функций в редакторе модели после добавления  
“UserManual.Message” и регистрации блока как ее исполнителя

Функция добавлена в модель и блок зарегистрирован как ее исполнитель – теперь нужно ввести реакцию на эту функцию. Проще всего сделать это, выбрав функцию в списке на панели “функции” (она там единственная) и нажав кнопку с желтым листком в левой нижней части вкладки. На открывшейся справа вкладке с именем функции (“UserManual.Message”) введем следующий текст:

```

if (Param!=NULL && // Параметр передан
 Param->servSize>=sizeof(TUserMessageFuncParam)) // Размер верен
{ int icon;
 char *msg;
 // В зависимости от типа сообщения выбираем иконку для него
 switch (Param->Type)
 { case UMFP_TYPE_INF: // Информация
 icon=MB_ICONINFORMATION; break;
 case UMFP_TYPE_WARN: // Предупреждение
 icon=MB_ICONWARNING; break;
 default: // Ошибка
 icon=MB_ICONERROR;
 }
 if (Param->Message) // Передан текст сообщения
 msg=Param->Message;
 else // Текст не передан
 msg="Сообщение";
 // Вывод сообщения пользователю
 rdsMessageBox(msg, rdsbcppBlockData->BlockName, icon|MB_OK);
}

```

Сначала, как и в предыдущих примерах, мы проверяем, переданы ли для функции правильные параметры. Параметр Param должен указывать на структуру TUserMessageFuncParam: если он равен NULL, или если поле servSize (оно в структуре первое) меньше размера этой структуры, значит, параметры переданы неверно, и блок не может ничего вывести. Если обе проверки прошли, то, в зависимости от поля Type переданной структуры, мы присваиваем целой переменной icon одну из стандартных констант Windows API: MB\_ICONINFORMATION, MB\_ICONWARNING или MB\_ICONERROR. Эти константы при выводе сообщения пользователю указывают иконку, изображаемую рядом с текстом сообщения. Затем в переменную msg записывается либо указатель на текст сообщения, полученный из поля Message структуры параметров функции, либо, если вместо текста почему-то передали NULL, указатель на строку “сообщение”. После этого вызывается сервисная функция РДС rdsMessageBox (см. §A.5.5.6 приложения к руководству программиста [2]), выводящая пользователю окно с сообщением msg, иконкой icon и единственной кнопкой “ОК”. От стандартной функции Windows API MessageBox она отличается только тем, что не блокирует выполнение расчета.

После компиляции модели созданного блока все блоки в одной с ним подсистеме и подсистемах, вложенных в нее, получают возможность вызывать у него функцию “UserManual.Message”. Создадим еще один блок, который будет выводить сообщение пользователю при превышении значением входа некоторого заданного уровня. У блока будет три вещественных входа: x, L и delta. Как только x превысит L на delta, блок выведет пользователю сообщение об этом, и будет “молчать”, пока x не станет меньше L минус delta. Такой гистерезис мы вводим для того, чтобы небольшие колебания x вокруг L не приводили к каскаду сообщений, каждое из которых пользователь должен будет закрывать кнопкой “ОК”. Для того, чтобы блок помнил, можно ли ему сейчас выводить сообщение или нет, мы введем в него внутреннюю логическую переменную out: она будет получать значение 1, если x превысит L+delta и сообщение будет выведено, и сбрасываться в 0, как только x станет меньше L-delta. Таким образом, наш блок будет иметь следующую структуру переменных:

| <i><b>Имя</b></i> | <i><b>Тип</b></i> | <i><b>Вход/выход</b></i> | <i><b>Пуск</b></i> | <i><b>Начальное значение</b></i> |
|-------------------|-------------------|--------------------------|--------------------|----------------------------------|
| Start             | Сигнал            | Вход                     | ✓                  | 0                                |
| Ready             | Сигнал            | Выход                    |                    | 0                                |

| <i>Имя</i> | <i>Тип</i> | <i>Вход/выход</i> | <i>Пуск</i> | <i>Начальное значение</i> |
|------------|------------|-------------------|-------------|---------------------------|
| x          | double     | Вход              | ✓           | 0                         |
| L          | double     | Вход              | ✓           | 0                         |
| out        | Логический | Внутренняя        |             | 0                         |
| delta      | double     | Вход              | ✓           | 0.1                       |

Создадим в схеме новый блок с автокомпилируемой моделью (см. стр. 95) и зададим для него указанную структуру переменных. Этот блок будет искать исполнителя функции “UserManual.Message” и вызывать эту функцию для сообщения, поэтому в его модели нам потребуется описание структуры TUserMessageFuncParam. Откроем вкладку глобальных описаний модели (вкладка “события” – раздел “описания” – двойной щелчок на пункте “глобальные описания”) и введем на ней команду включения файла “UserMessage.h”, в котором описана эта структура:

```
#include "UserMessage.h"
```

Теперь добавим в модель функцию “UserManual.Message” и укажем, что блок должен найти исполнителя этой функции. На вкладке “функции” левой панели редактора нажмем кнопку “+” и заполним открывшееся окно согласно рис. 475.

Рис. 475. Добавление в модель блока функции “UserManual.Message” (рамкой выделен флажок, включающий поиск исполнителя функции)

В целом, окно заполнено почти так же, как и у предыдущего блока (рис. 473 на стр. 270), за несколькими исключениями. На верхней панели окна опять выбран флажок “произвольная функция” – добавляемая функция не является стандартной. На панели “имя функции в RDS” снова введено название нашей функции. Поле “объект для функции в программе”, в отличие от предыдущей модели, мы меняем на “rdsfuncUM\_Message”: этот блок вызывает функцию, поэтому имя объекта, который мы будем использовать для этого, сделаем покороче. В поле “тип параметра функции” введен уже знакомый нам тип указателя на структуру “TUserMessageFuncParam\*”. А на панели “дополнительные действия” теперь выбран флажок “найти в схеме исполнителя функции”.

Рядом с именем добавленной нами функции в списке функций (рис. 476), помимо иконки с пустым листом (текст реакции на эту функцию не введен), будет находиться



изображение лупы: оно указывает на то, что блок будет запрашивать у РДС поиск исполнителя этой функции, и, поэтому, ее можно вызывать без указания идентификатора вызываемого блока. Теперь, с момента подключения модели к блоку и до ее отключения, наш блок будет знать ближайшего по иерархии исполнителя функции “UserManual.Message”, если, конечно, такой исполнитель существует в схеме.

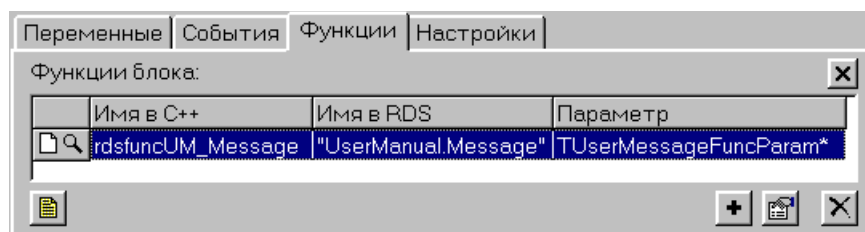


Рис. 476. Список функций в редакторе модели после добавления “UserManual.Message” и включения поиска ее исполнителя

Теперь введем реакцию блока на такт расчета, из которой мы и будем вызывать функцию. На вкладке “события” левой панели редактора раскроем раздел “моделирование и режимы”, дважды щелкнем на пункте “модель” и введем на открывшейся одноименной вкладке следующий текст:

```
if(x>L+delta) // Вход выше уровня с запасом
{
 if(!out) // Сообщение об этом еще не выводилось
 {
 TUserMessageFuncParam param; // Параметры функции
 param.servSize=sizeof(param); // Размер структуры
 param.Type=UMFP_TYPE_WARN; // Предупреждение
 // Формируем динамическую строку с текстом сообщения
 param.Message=
 rdsDynStrCat(rdsbcppBlockData->BlockName,
 " - превышение уровня", FALSE);
 // Вызываем функцию у исполнителя, кем бы он ни был
 rdsfuncUM_Message.Call(¶m);
 // Освобождаем динамическую строку
 rdsFree(param.Message);
 out=1; // Запоминаем факт вывода сообщения
 }
}
else if(x<L-delta) // Вход ниже уровня с запасом
 out=0; // Теперь снова разрешаем вывод сообщения
```

Сначала мы проверяем, не превысило ли значение  $x$  значение  $L$  с запасом  $\delta$ . Если это так, и сообщение о превышении уровня еще не выводилось (то есть логический флаг `out` не взведен), мы должны вывести пользователю сообщение через исполнителя функции “UserManual.Message”. Для вызова функции нам потребуется структура типа `TUserMessageFuncParam` – назовем объект этой структуры `param`. В поле `servSize` этой структуры мы записываем ее размер, полученный при помощи оператора `sizeof`, а в поле `Type` – константу `UMFP_TYPE_WARN` из файла “UserMessage.h” (он уже включен в глобальных описаниях), означающую предупреждающее сообщение. В поле `Message` при помощи функции `rdsDynStrCat` (см. §A.5.4.6 приложения к руководству программиста) мы формируем динамическую строку, состоящую из имени данного блока и слов “превышение уровня”. Эту строку потом нужно будет освободить вызовом `rdsFree`.

Теперь все поля структуры параметров заполнены – можно вызывать функцию. Для этого мы используем функцию-член `Call` объекта `rdsfuncUM_Message`, созданного для “UserManual.Message”. В эту функцию-член мы передаем только указатель на структуру `param`, идентификатор вызываемого блока не передается. Именно отсутствие

идентификатора блока в вызове Call говорит о том, что функция вызывается у ее найденного исполнителя, а не у какого-то другого блока. Вызвав функцию, мы освобождаем ранее созданную строку param.Message при помощи rdsFree (см. §A.5.4.8 приложения к руководству программиста) и взводим флаг out, чтобы не выводить сообщение повторно, пока x не снизится.

Если же значение x оказалось ниже значения L с запасом delta, мы сбрасываем out, чтобы при следующем превышении снова вывести сообщение.

Для проверки работы созданных моделей можно собрать схему, изображенную на рис. 477. Если запустить расчет и дать x значение, большее суммы L и delta, на экране появится сообщение о превышении уровня, выведенное блоком-исполнителем, с указанием имени блока, в котором это превышение возникло. Если создать подсистему и поместить туда блоки, проверяющие уровень (копии блока Block22 на рисунке), эти блоки будут выводить такие же сообщения, поскольку РДС будет предоставлять им доступ к блоку-исполнителю.

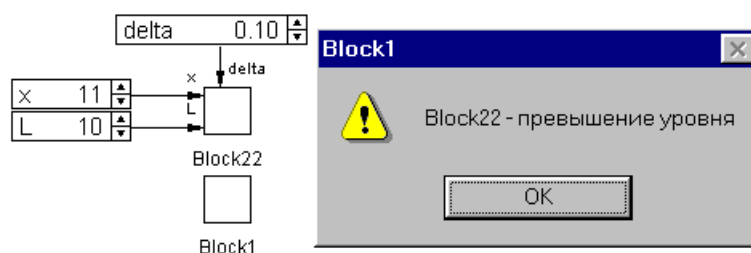


Рис. 477. Схема для тестирования исполнителя функции и выводимое сообщение:  
здесь Block1 – исполнитель, Block22 – блок проверки уровня

Может показаться, что вывод сообщения было бы гораздо проще включить непосредственно в модель блока, проверяющего уровень. Это действительно так, но если в схеме будет много блоков с разными моделями, выводящими сообщения, и мы захотим что-то изменить в процедуре вывода, нам придется изменять все эти модели блоков. В данном же случае нам достаточно будет изменить модель блока-исполнителя или просто заменить его на другой.

Сделаем новую модель блока-исполнителя той же самой функции “UserManual.Message”, которая, вместо вывода сообщения на экран, будет записывать его в текстовый файл с указанием времени его поступления. Проще всего будет не создавать новый блок с нуля, а сделать копию старого блока и его модели, а затем изменить эту модель. Заодно мы продемонстрируем влияние иерархии подсистем на доступность исполнителя – поместим копию блока в отдельной подсистеме, и посмотрим, какие блоки свяжутся с каким исполнителем.

Создадим в той же подсистеме, в которой мы тестировали два созданных в этом параграфе блока (см. рис. 477), новую подсистему. Скопируем старый блок-исполнитель в буфер обмена (в режиме редактирования выделим его и нажмем Ctrl+C), а затем перейдем в созданную подсистему, нажмем в том месте ее рабочего поля, куда будет вставляться копия блока, правую кнопку мыши, и выберем в меню пункт “вставить”. Модуль автокомпиляции при этом спросит, хотим ли мы оставить у копии блока старую модель или нужно копировать и модель тоже (рис. 478, см. также §3.5 на стр. 28). Мы хотим сделать копию модели, поэтому следует выбрать варианты “создать копию файла модели” или “создать копию файла модели, введя имя вручную”.

Откроем редактор модели скопированного блока и начнем исправлять ее. Прежде всего, для того, чтобы записывать сообщения в файл, нам потребуется как-то задавать имя этого файла. Кроме того, нужно предусмотреть возможность стирания этого файла в момент загрузки схемы (новый сеанс работы со схемой – новый журнал), поэтому мы введем в новую

модель два настроечных параметра: строку FileName с именем файла и логический параметр ClearOnLoad, который будет управлять стиранием файла.

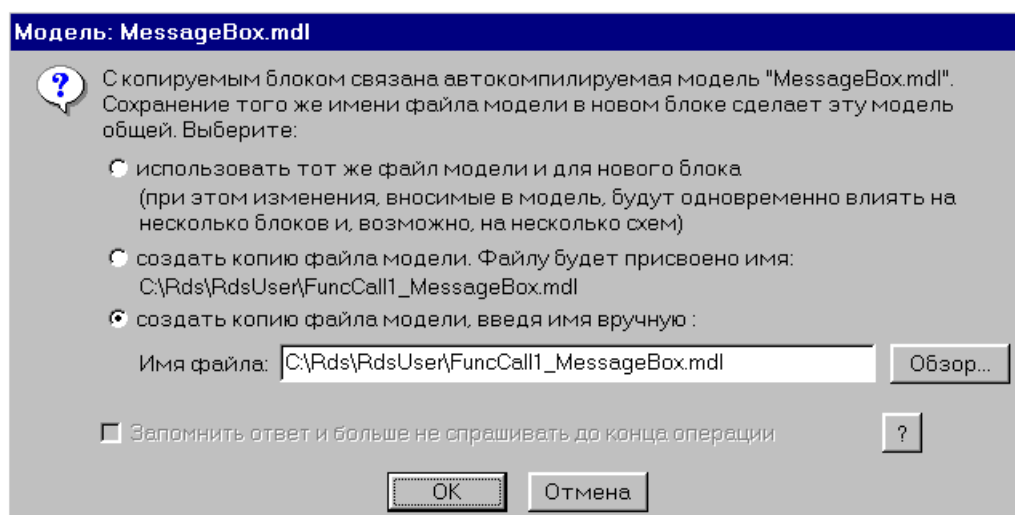


Рис. 478. Запрос модуля автокомпиляции на создание копии модели

Выберем на левой панели редактора вкладку “настройки”, нажмем в ее верхней части кнопку “+” (см. рис. 341 на стр. 61) и заполним окно добавления настроечного параметра следующим образом:

- на панели “переменная”:
  - ♦ “имя” – “FileName”;
  - ♦ “тип” – “rdsbcpString” (это специальный класс для хранения строк, создаваемый модулем автокомпиляции);
  - ♦ “по умолчанию” – оставлено пустым;
- на панели “поле ввода”:
  - ♦ “добавить для этой переменной поле ввода” – установлен флажок;
  - ♦ “заголовок” – “имя файла”;
  - ♦ “ширина” – 200;
  - ♦ “тип” – “сохранение файла”.

После нажатия кнопки “OK” в модели блока появится настроечный параметр с именем FileName, для которого, по желанию пользователя, будет открываться стандартный диалог сохранения файла Windows. Точно так же, еще раз нажав “+”, добавим второй настроечный параметр:

- на панели “переменная”:
  - ♦ “имя” – “ClearOnLoad”;
  - ♦ “тип” – “BOOL” (стандартный логический тип Windows);
  - ♦ “по умолчанию” – “TRUE”;
- на панели “поле ввода”:
  - ♦ “добавить для этой переменной поле ввода” – установлен флажок;
  - ♦ “заголовок” – “очищать при загрузке”;
  - ♦ “ширина” – не важно;
  - ♦ “тип” – “флаг”.

Окно для ввода параметров и процедуры их загрузки и сохранения вместе со схемой будут добавлены в модель автоматически (см. §3.6.6 на стр. 60).

Дадим окну настройки заголовок “журнал” (для этого нужно вызвать панель параметров окна кнопкой “размеры и параметры окна” в правой верхней части нижней половины вкладки “настройки” (см. рис. 347 на стр. 65). Теперь вкладка “настройки” должна выглядеть так, как на рис. 479.

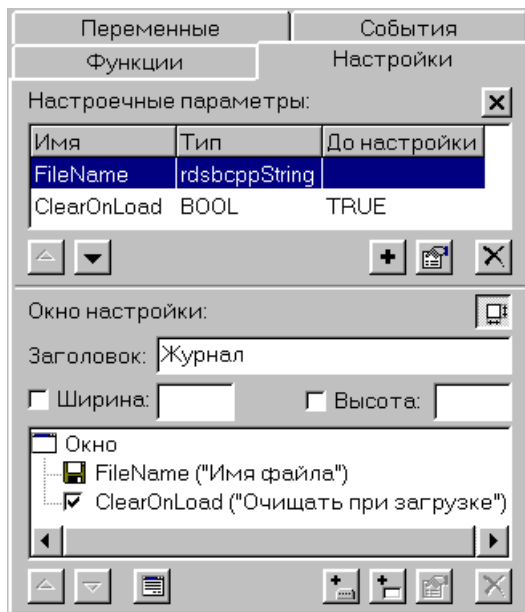


Рис. 479. Настроечные параметры нового блока-исполнителя

дважды щелкнем на его пункте “после загрузки схемы”. Введем на этой вкладке следующий текст:

```
char *path; // Переменная для пути к файлу

if(!ClearOnLoad) // Не нужно стирать файл
 return;
// Файл должен быть стерт

// Добавляем к имени файла путь, если его там нет
path=rdsGetFullFilePath(FileName.c_str(),NULL,NULL);
if(path!=NULL) // Удачно
{ DeleteFile(path); // Удаляем файл
 rdsFree(path); // Освобождаем память строки
}
```

Здесь мы сначала проверяем, нужно ли стирать файл. Если настроечный параметр `ClearOnLoad` не равен `TRUE`, стирать файл не нужно, мы немедленно завершаем реакцию. В противном случае мы вызываем функцию `rdsGetFullFilePath` (см. §A.5.4.9 приложения к руководству программиста), которая преобразует сокращенный путь к файлу из настроечного параметра `FileName` в полный (например, добавит к нему путь к файлу схемы, если в самом параметре путь отсутствует). Поскольку `FileName` – это объект класса `rdsbcppString`, а функция `rdsGetFullFilePath` требует строки типа `char*`, мы вызываем у `FileName` функцию-член `c_str()`, которая возвращает указатель на строку, хранящуюся в этом классе (см. §3.7.2.5 на стр.112). Путь, который возвращает `rdsGetFullFilePath`, это динамическая строка, поэтому нам нужно будет потом освободить ее при помощи `rdsFree`.

Если переменная `path`, которой мы присвоили результат вызова `rdsGetFullFilePath`, не равна `NULL` (то есть строку из `FileName` удалось преобразовать в полный путь к файлу), мы удаляем файл по этому пути функцией Windows API `DeleteFile` и освобождаем строку `path` вызовом `rdsFree`.

Теперь изменим реакцию на вызов функции “`UserManual.Message`” – фактически, ее нужно полностью переписать. Откроем вкладку этой реакции (вкладка “события” – раздел

Для того, чтобы добавлять к сообщению текущее время, нам потребуется стандартная функция `sprintf`, а это значит, что нам нужно включить в модель файл “`stdio.h`”. Откроем вкладку глобальных описаний (вкладка “события” – раздел “описания” – двойной щелчок на пункте “глобальные описания”) и добавим к уже имеющейся там команде включения файла “`UserMessage.h`” новую (изменения выделены двойным подчеркиванием):

```
#include "UserMessage.h"
#include <stdio.h>
```

Мы решили дать пользователю возможность стирать текстовый файл с сообщениями в момент загрузки схемы, чтобы он не разрастался до бесконечности. В этом нам поможет реакция на событие `RDS_BFM_AFTERLOAD`, возникающее у всех блоков немедленно после загрузки схемы из файла. Откроем ее вкладку: на вкладке “события” раскроем раздел “загрузка и запись данных” и

“функции блока” – двойной щелчок на пункте “UserManual.Message”), и заменим текст, который там находится, на следующий:

```
// Локальные переменные
char *path,*msg,*stype,buf[100];
HANDLE h; // Дескриптор файла
DWORD temp;
SYSTEMTIME time; // Дата и время из Windows

if(Param==NULL) // Параметр не передан
 return;
if(Param->servSize<sizeof(TUserMessageFuncParam))
 return; // Размер параметра неверен

// Тип сообщения
switch(Param->Type)
{
 case UMFP_TYPE_INF: stype="Информация"; break;
 case UMFP_TYPE_WARN: stype="Предупреждение"; break;
 default: stype="Ошибка";
}
// Текст сообщения
if(Param->Message)
 msg=Param->Message;
else
 msg="Сообщение";

// Получение полного пути к файлу
path=rdsGetFullFilePath(FileName.c_str(),NULL,NULL);
if(path==NULL) // Не удалось сформировать путь
 return;
// Открываем файл path на запись
h=CreateFile(path,GENERIC_WRITE,0,NULL,
 OPEN_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);
rdsFree(path); // Строка path больше не нужна
if(h==INVALID_HANDLE_VALUE) // Ошибка открытия файла
 return;
// 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 ",
 time.wDay,time.wMonth,time.wYear,
 time.wHour,time.wMinute,time.wSecond);
// Записываем дату и время в файл
WriteFile(h,buf,strlen(buf),&temp,NULL);
// Записываем тип сообщения
WriteFile(h,stype,strlen(stype),&temp,NULL);
// Переводим строку и добавляем два пробела
WriteFile(h,"\r\n ",4,&temp,NULL);
// Выводим текст сообщения
WriteFile(h,msg,strlen(msg),&temp,NULL);
// Переводим строку
WriteFile(h,"\r\n",2,&temp,NULL);
// Закрываем файл
CloseHandle(h);
```

Реакция получилась довольно длинной, но большая ее часть состоит из команд записи в файл. В самом ее начале описано несколько вспомогательных переменных, которые будут использоваться для работы с файлом и получения текущего времени.

Прежде всего мы проверяем, был ли передан параметр при вызове функции (`Param==NULL` – не был передан) и достаточен ли размер переданной структуры для работы функции (`Param->servSize` не меньше размера структуры `TUserMessageFuncParam`). Если это не так, работа блока невозможна, и реакция немедленно завершается. В противном случае вспомогательной переменной `stype`, в зависимости от константы из `Param->Type`, присваивается указатель на строку “информация”, “предупреждение” или “ошибка”, а переменной `msg` – указатель на текст сообщения из `Param->Message` или на строку “сообщение”, если в `Param->Message` находится `NULL`.

Теперь нужно открыть текстовый файл на запись. Сначала, как и в реакции на загрузку всей схемы, рассмотренной ранее, вызовом `rdsGetFullFilePath` мы добавляем к имени файла из настроечного параметра `FileName` путь к схеме, если в этом параметре отсутствует путь. Возвращенный указатель на созданную динамическую строку записывается в переменную `path`, потом эту строку нужно будет удалить вызовом `rdsFree`. Затем мы открываем файл на запись при помощи функции Windows API `CreateFile` с константой `GENERIC_WRITE` и присваиваем дескриптор открытого файла переменной `h`. Сразу после этого строка `path` освобождается при помощи вызова `rdsFree` – она больше не нужна. Если файл открыть не получилось, `CreateFile` вернет вместо дескриптора константу `INVALID_HANDLE_VALUE`, и, в этом случае, мы завершаем выполнение реакции.

Если файл открыт, мы перемещаем указатель записи в его конец вызовом функции Windows API `SetFilePointer` – теперь мы должны дописать в него сообщение с текущей датой и временем. Дату и время мы записываем в структуру `time` стандартного типа `SYSTEMTIME` вызовом `GetLocalTime`, а потом формируем во вспомогательном массиве `buf` ее символьное представление в виде “день-месяц-год час:минута:секунда” функцией `sprintf` (“stdio.h”, в котором она описана, мы уже включили в модель). После этого, вызывая функцию Windows API `WriteFile`, в открытый файл с дескриптором `h` мы последовательно записываем:

- дату и время из `buf`;
- тип сообщения из `stype`;
- перевод строки с возвратом каретки (“\r\n”) и два пробела;
- текст сообщения из `msg`;
- перевод строки с возвратом каретки.

Закончив запись, мы закрываем файл функцией Windows API `CloseHandle`.

Может показаться, что открытие файла перед записью каждого сообщения и закрытие после нее – не самый лучший способ ведения журнала сообщений. Можно было бы открыть файл в момент поступления первого сообщения и не закрывать его до конца работы, однако, это привело бы к некоторым проблемам. Пока файл открыт, пользователь не смог бы получить к нему доступ – например, удалить его. Кроме того, если мы закрываем файл после записи сообщения, мы можем использовать этот файл в нескольких блоках одновременно, и они смогут дописывать свои сообщения в его конец, не мешая друг другу. Мы предполагаем, что сообщения в схеме будут возникать не очень часто (в противном случае стоит пересмотреть способ ведения журнала), и постоянные открытия-закрытия файла не будут существенно замедлять ее работу.

Модель закончена – перейдем к ее тестированию. Скомпилируем ее, закроем редактор, вызовем настройки блока (щелчок правой кнопкой мыши на нем – пункт меню “настройка”) и введем в поле имени файла “Log.txt” (рис. 480). Поскольку мы не указали путь, файл “Log.txt” будет размещен в той же папке, что и схема (а также и модели, если мы не указывали пути к файлам моделей при их создании), так что его будет легко найти. Теперь

скопируем в созданную для этого блока подсистему еще и блок, проверяющий превышение уровня, со всеми подключенными к нему полями ввода (см. рис. 477 на стр. 274). Для этого выделим блок вместе с полями ввода, нажмем Ctrl+C, перейдем в подсистему, нажмем в том месте ее рабочего поля, куда будет вставляться копия блока, правую кнопку мыши, после чего выберем в меню пункт “вставить”. На запрос модуля автокомпиляции о том, нужно ли создать копию модели (см. рис. 478 на стр. 275), нужно ответить “использовать тот же файл модели”, чтобы обе копии блока имели одну модель. В результате всех этих действий схема станет похожа на рис. 481. На нем Block22 и Block2 – блоки проверки уровня, Sys1 – подсистема, в которую мы поместили новый блок-исполнитель, Block1 – старый исполнитель, Block11 – новый исполнитель.

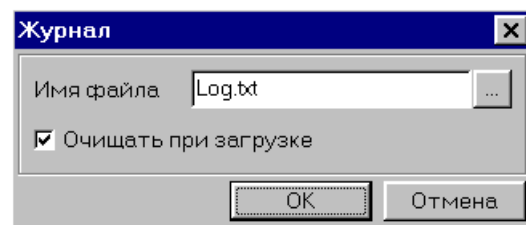


Рис. 480. Настройка блока записи сообщений в файл

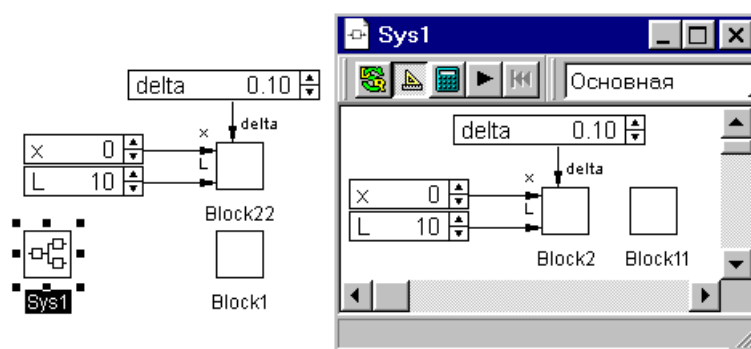


Рис. 481. Тестирование второго блока-исполнителя

Если теперь запустить расчет, превышения уровня в блоке Block22 будут, как и прежде, приводить к выводу сообщений на экран. Но если уровень будет превышен в блоке Block2, сообщение об этом будет записано в файл “Log.txt” в виде следующих двух строчек:

18-08-2013 15:11:44 Предупреждение  
Block2 – превышение уровня

Модели блоков Block2 и Block22 одинаковы, каждая из них для вывода сообщения обращается к блоку-исполнителю функции “UserManual.Message”. Но для Block22 этим исполнителем будет Block1, а для Block2 – Block11, поскольку он находится ближе в иерархии подсистем. Если удалить из схемы Block11, превышение уровня в любом из блоков будет выводить сообщение на экран: для Block22 ничего не изменится, а для Block2 ближайшим в иерархии исполнителем функции станет Block1. Если же удалить Block1, Block22 вообще не сможет вывести сообщение, поскольку ни в его подсистеме, ни выше по иерархии исполнителей не осталось, а Block2 будет продолжать выводить сообщения в файл.

### §3.7.13.5. Объекты функций в автокомпилируемых моделях

Рассматриваются технические особенности классов и объектов, автоматически создаваемых в программе модели для работы с функциями блоков.

В §3.7.13.1 (стр. 245) объяснялось, что для каждой функции блока модуль автокомпиляции создает объект специально сформированного для нее класса, и перечислялись основные функции-члены такого класса. При этом для некоторых из этих функций было указано, что их можно вызывать только из модели блока, то есть только из функций, являющихся членами класса самого блока (например, из любой функции реакции на событие). Может возникнуть вопрос: почему доступность публичной функции-члена объекта C++ может зависеть от места ее вызова?



Дело в том, что, на самом деле, для каждой функции создается не один класс и объект этого класса, а два разных класса и два объекта этих классов: один такой объект является глобальным, второй принадлежит классу блока. Функции-члены, относящиеся к функции блока в целом (например, вызов ее у конкретного блока или всех блоков конкретной подсистемы) описаны как в классе для глобального объекта, так и в классе объекта, принадлежащего блоку. Функции-члены, относящиеся к действиям, связанным с конкретным блоком (например, регистрация этого блока как исполнителя функции), описаны только в классе, объект которого добавляется в класс блока, и их нельзя вызвать из глобальных функций, поскольку оттуда этот объект недоступен. Имя глобального объекта и объекта в классе блока совпадают, поэтому разработчик модели везде использует одно и то же имя для обращения к объекту функции. Объект в классе блока перекрывает видимость одноименного глобального объекта, поэтому в функциях класса блока можно использовать расширенный набор функций-членов этого объекта. Из глобальных же функций будет виден только глобальный объект, класс которого не содержит функций-членов, оперирующих данными конкретного блока.

Рассмотрим, например, описания, автоматически добавляемые модулем автокомпиляции в программу модели блока проверки уровня, рассмотренного в §3.7.13.4 (стр. 268) для функции “UserManual.Message” (см. рис. 475 на стр. 272). Их можно увидеть, выбрав в окне редактора модели пункт меню “модель | показать текст C++” и прокрутив текст вниз до комментария “объекты для функций блока”:

```
//-----
// Объекты для функций блока
//-----
// Функция "UserManual.Message"
class rdsbcppFunction0G : public rdsbcppFunction // Глобальный
{ public:
 // Вызов у блока
 int Call(RDS_BHANDLE Block, TUserMessageFuncParam* param)
 {return rdsCallBlockFunction(Block, _Id, param);};
 // Вызов у блоков подсистемы
 int Broadcast(RDS_BHANDLE Parent,
 DWORD Flags, TUserMessageFuncParam* param)
 {return rdsBroadcastFunctionCallsEx(Parent,
 _Id, param, Flags);};
 rdsbcppFunction0G(void):rdsbcppFunction(){};
 ~rdsbcppFunction0G(){};
};
rdsbcppFunction0G rdsfuncUM_Message;
class rdsbcppFunction0L : public rdsbcppFunction // Локальный
{ public:
 // Вызов у блока
 int Call(RDS_BHANDLE Block, TUserMessageFuncParam* param)
 {return rdsCallBlockFunction(Block, _Id, param);};
 // Вызов у блоков подсистемы
 int Broadcast(RDS_BHANDLE Parent,
 DWORD Flags, TUserMessageFuncParam* param)
 {return rdsBroadcastFunctionCallsEx(Parent,
 _Id, param, Flags);};
 // Вызов у исполнителя функции
 int Call(TUserMessageFuncParam* param)
 {return _Link?
 rdsCallBlockFunction(_Link->Block, _Id, param):0;};
 // Перевод функций подписки/регистрации в public
 inline void RegisterProvider(void){_RegisterProvider();};
```



```

inline void UnregisterProvider(void) {_UnregisterProvider();};
inline void SubscribeToProvider(void) {_SubscribeToProvider();};
inline void UnsubscribeFromProvider(void)
 {_UnsubscribeFromProvider();};
rdsbcppFunction0L(void):rdsbcppFunction(){};
~rdsbcppFunction0L(){};

};
//-----

```

Здесь описаны два класса с именами `rdsbcppFunction0G` и `rdsbcppFunction0L` (эти имена модуль автокомпиляции создает автоматически). Оба класса являются потомками класса `rdsbcppFunction`, описанного в “CommonAC.hpp” и содержащего основные поля и функции, необходимые для работы с функциями блока. Класс `rdsbcppFunction0G` используется для создания глобального объекта: сразу за описанием этого класса следует объявление глобальной переменной `rdsfuncUM_Message` этого типа (такое имя мы выбрали для объекта функции при ее создании – см. стр. 272). Класс `rdsbcppFunction0L`, содержащий дополнительные функции-члены для регистрации и поиска исполнителя функции, будет использован для описания поля класса блока, которое тоже будет иметь имя `rdsfuncUM_Message` (это описание выделено двойным подчеркиванием):

```

//-----
// Класс блока
//-----
class rdsbcppBlockClass
{ public:
 // Структура данных блока
 RDS_PBLOCKDATA rdsbcppBlockData;
 // Статические переменные
 rdsbcstSignal Start;
 rdsbcstSignal Ready;
 rdsbcstDouble x;
 rdsbcstDouble L;
 rdsbcstLogical out;
 rdsbcstDouble delta;

 // Объекты функций блоков (те же имена, что и у глобальных)
 rdsbcppFunction0L rdsfuncUM_Message;

 // Инициализация переменных блока
 void rdsbcppInitVars(void *base)
 {
 ...
 }
}

```

Таким образом, использование имени `rdsfuncUM_Message` внутри реакций на события и любых других функций, принадлежащих классу блока, будет обращением к объекту типа `rdsbcppFunction0L`. Использование этого же имени в функциях, не принадлежащих к классу блока, будет обращением к объекту типа `rdsbcppFunction0G`.

Ситуация, когда из глобальной функции необходимо обратиться к функциям, относящимся к регистрации и поиску исполнителей, возникает крайне редко: любая работа с исполнителем функции связана с каким-либо конкретным блоком, который либо объявляет себя таким исполнителем, либо запрашивает его поиск, то есть приказывает РДС постоянно сообщать данному конкретному блоку о появлении и исчезновении исполнителей заданной функции. Таким образом, регистрация исполнителей обычно производится из модели регистрирующегося блока, а поиск – из модели блока, запрашивающего этот поиск. Разработчик модели может захотеть вынести какие-либо действия из модели блока в отдельную функцию просто для своего удобства, и, если среди этих действий будет вызов

специфических функций-членов рассматриваемых здесь объектов, лучше всего оформить такую функцию как член класса блока, то есть описать ее в разделе “описания внутри класса блока” (см. стр. 217). Есть также возможность, при необходимости, вызвать функцию у блока-исполнителя не через объект внутри класса блока (в примере выше – объект типа `rdsbcppFunction0L`), а через глобальный объект. Для этого нужно просто явно указать идентификатор блока, у которого вызывается функция.

В примере, рассматриваемом в предыдущем параграфе, мы выводили сообщение пользователю, вызывая функцию “`UserManual.Message`” у ее исполнителя следующим образом:

```
rdsfuncUM_Message.Call(¶m);
```

(здесь `param` – структура параметров функции типа `TUserMessageFuncParam`). В эту версию функции `Call` не передается идентификатор блока – он будет автоматически определен, и функция будет вызвана у ближайшего найденного исполнителя. Допустим, мы, по каким-то причинам, хотим вынести этот вызов в отдельную глобальную функцию. В этом случае мы уже не сможем пользоваться `Call` с одним параметром – в глобальном объекте, который будет виден из глобальной функции, такой версии функции-члена `Call` просто нет. Но у него есть обычная версия `Call`, в первом параметре которой явно передается идентификатор вызываемого блока. В нашей глобальной функции мы можем воспользоваться только ей, поэтому в параметрах этой функции нужно будет передать идентификатор блока:

```
// Наша глобальная функция для вызова "UserManual.Message"
void OurGlobalCaller(RDS_BHANDLE block, // вызываемый блок
 TUserMessageFuncParam *param) // параметр
{ rdsfuncUM_Message.Call(block,param); }
```

Этот идентификатор блока-исполнителя можно получить у объекта типа `rdsbcppFunction0L` при помощи функции-члена `Provider` (см. стр. 250). Таким образом, *изнутри* нашей модели вызов `OurGlobalCaller` выглядел бы так:

```
OurGlobalCaller(rdsfuncUM_Message.Provider(), ¶m);
```

В большинстве случаев, разработчику модели можно не задумываться об организации классов для функций блоков: если он обращается к этим функциям из модели, ему всегда доступны все возможности. Принимать во внимание наличие двух разных классов для одной функции блока нужно только при обращении к ним из глобальных функций – например, из функций обратного вызова, подобных рассматриваемой на стр. 264: такие функции не могут быть членами какого-либо класса по требованиям РДС. В этом случае нужно каким-либо образом передавать в глобальные функции идентификатор блока – например, описанным выше способом. К счастью, все функции обратного вызова в РДС имеют дополнительный параметр типа `void*`, через который в них можно передать указатель на любые необходимые данные (подробнее это рассматривается в руководстве программиста [1]).

### §3.8. Краткий перечень вводимых в модель описаний и реакций на события

Перечисляются все возможные реакции на события, которые пользователь может ввести в автокомпилируемую модель блока, и кратко описываются их параметры. Рассматривается общая структура формируемой модулем автокомпиляции программы и место пользовательских описаний в ней.

#### §3.8.1. Дополнительные описания, вводимые в модель

Рассматриваются три группы описаний, которые пользователь может вставить внутрь формируемой модулем автокомпиляции программы. Они могут содержать описания типов, функций, глобальных переменных и констант, дополнительных полей класса блока, команды включения файлов заголовков и т.п.

Помимо реакций на события, происходящие с блоком в схеме, в модель могут быть вставлены различные описания пользователя: описания типов и классов, служебные функции, команды включения различных файлов заголовков и т.п. Модуль автокомпиляции

предоставляет три точки для вставки таких описаний внутрь автоматически формируемой программы модели: глобальные описания, описания внутри класса блока и описания после класса блока. Тексты всех этих описаний вводятся на вкладке “события” в соответствующих подразделах раздела “описания”.

Внутри формируемого текста эти три точки вставки описаний размещены следующим образом (ниже они выделены двойной рамкой):

*Глобальные описания из параметров модуля автокомпиляции*

```
#include <windows.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
```

*Автоматически сформированные служебные описания и команды включения стандартных заголовков РДС*

```
#include <RdsDef.h>
#define RDS_SERV_FUNC_BODY rdsbcppGetService
#include <RdsFunc.h>
#include <CommonBl.h>
#include <CommonAC.hpp>
...
```

*Глобальные описания пользователя*

*Автоматически сформированные описания классов для функций блока*

```
class rdsbcppFunction0G : public rdsbcppFunction // Глобальный
{ ... }
class rdsbcppFunction0L : public rdsbcppFunction // Локальный
{ ... }
...
```

*Автоматически сформированная главная функция DLL*

```
int WINAPI DllEntryPoint(HINSTANCE hinst,
 unsigned long reason,void *lpReserved)
{ ... }
```

*Макросы для автоматического формирования описаний классов переменных блока*

```
// Переменная char ("S")
RDSBCPP_STATICPLAINCLASS(rdsbcstSignal,char);
```

```
// Динамическая переменная double ("D")
RDSBCPP_DYNAMICPLAINCLASS(rdsbcdtDouble, double, "D");
...
```

```
//-----
// Класс блока
//-----
class rdsbcppBlockClass
{ public:
 // Структура данных блока
 RDS_PBLOCKDATA rdsbcppBlockData;
```

Автоматически сформированные описания динамических и статических переменных блока, настроечных параметров и объектов для доступа к функциям

```
// Динамические переменные
rdsbcdtDouble DynTime;

// Статические переменные
rdsbcstSignal Start;
rdsbcstSignal Ready;

// Объекты функций блоков (те же имена, что и у глобальных)
rdsbcppFunction0L rdsfuncControlValueChanged;
...
```

Автоматически сформированные служебные функции класса блока и заголовки функций реакции на события

```
// Инициализация переменных блока
void rdsbcppInitVars(void *base)
{ ... };
// Проверка существования динамических переменных
BOOL rdsbcppDynVarsOk(void) { return DynTime.Exists(); };
// Загрузка настроечных параметров
char *rdsbcppLoadParameters(char *rdsbcpp_Text);
// Запись настроечных параметров
void rdsbcppSaveParameters(void);
// Вызов окна настройки
BOOL rdsbcppShowSetupWindow(void);

// Функции реакции на события
void rdsbcppModel(void);
...
```

Описания пользователя внутри класса блока

```
}; // class rdsbcppBlockClass
//-----
```

*Автоматически сформированные служебные функции класса блока, тела которых вынесены за пределы самого класса*

```
// Загрузка настроечных параметров
char *rdsbcppLoadParameters(char *rdsbcpp_Text)
{ ... }
// Запись настроечных параметров
void rdsbcppSaveParameters(void)
{ ... }
...
```

*Описания пользователя после класса блока*

*Автоматически сформированная функция модели блока*

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

*Автоматически сформированные функции реакций на события, внутри каждой из которых вставлен введенный пользователем текст*

```
// Один такт моделирования
void rdsbcppBlockClass::rdsbcppModel(...)
{ ... }
...
```

Глобальные описания пользователя размещаются после команд включения стандартных файлов заголовков и специализированных заголовков РДС, поэтому в них можно использовать все стандартные типы, константы и структуры Windows API (LPVOID, HANDLE, HWND и т.п.) Можно также использовать любые стандартные типы, константы и структуры РДС (RDS\_BHANDLE, RDS\_BLOCKDATA и т.п.) В глобальных описаниях **нельзя** ссылаться на объекты, создаваемые для работы с функциями блока (см. §3.7.13 на стр. 245) и на класс блока rdsbcppBlockClass – все эти описания располагаются после глобальных описаний пользователя. Чаще всего в глобальные описания включают:

- команды включения файлов заголовков – как стандартных, так и пользовательских;
- глобальные переменные пользователя;
- описания типов и структур, используемых только в одной модели (если они используются в нескольких, имеет смысл записать их в отдельный файл и включать его в разных моделях, как в примере из §3.7.13.2 на стр. 250);
- пользовательские функции общего назначения, которым не нужен доступ к переменным блока, его настроечным параметрам и к объектам для вызова функций у других блоков.

Описания пользователя в классе блока размещаются в самом конце класса, объект которого будет создаваться для каждого блока с данной моделью. Все описания типов и

объявления переменных и стандартных полей класса на этот момент уже сделаны, поэтому в описаниях внутри класса можно ссылаться на любые объекты и типы. Любые функции, объявленные в этих описаниях, становятся функциями-членами класса блока, поэтому по имени их можно будет вызывать только из других функций-членов и из реакций на события (чтобы вызвать такую функцию из глобальной функции, необходимо как-то передать в нее указатель на объект класса блока, доступный внутри функций-членов через ключевое слово `this`). Чаще всего в описания внутри класса включают:

- пользовательские поля класса блока, если они нужны (нужно следить за тем, чтобы их имена не совпадали с именами переменных блока, настроечных параметров и прочих автоматически добавляемых в класс объектов);
- пользовательские функции, которым нужен доступ к переменным или настроечным параметрам блока, и которые будут вызываться из функций реакции на события;
- пользовательские функции, изнутри которых будут вызываться функции других блоков (см. §3.7.13.5 на стр. 279).

Описания пользователя после класса блока, как следует из их названия, размещаются после класса блока и нескольких его служебных функций. К этому моменту все служебные объекты модели уже описаны, и в описаниях можно на них ссылаться. Чаще всего здесь размещают:

- пользовательские функции общего назначения, которым нужен доступ к объектам для вызова функций других блоков (см. пример на стр. 264);
- тела функций-членов класса, которые разработчик по какой-либо причине решил вынести за пределы самого класса (при этом в описаниях внутри класса все равно должны находиться их заголовки).

Все три точки вставки пользовательских описаний располагаются в тексте программы до функций реакции на события, поэтому в функциях реакции можно пользоваться любыми объектами из этих описаний.

### §3.8.2. Создание и уничтожение блока

Рассматриваются реакции на события, связанные с созданием и уничтожением блока.

#### §3.8.2.1. Инициализация блока

Событие инициализации – это самое первое событие в “жизни” блока. Оно возникает только один раз за все время работы модели в момент подключения этой модели к блоку: при добавлении блока в схему из библиотеки или из буфера обмена, при загрузке схемы из файла, при ручной смене модели блока пользователем через окно параметров блока и т.п. На момент возникновения события параметры блока еще не загружены и переменные еще не созданы, поэтому в реакции на него **нельзя** обращаться к переменным блока. Если попытаться обратиться к переменным блока в реакции на событие инициализации, будет выведено сообщение об ошибке (рис. 482).

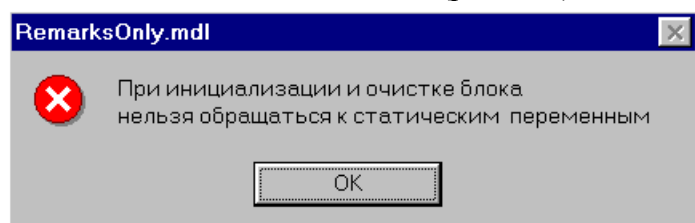


Рис. 482. Сообщение о недопустимости обращения к переменным блока

Не следует вводить в реакцию на инициализацию блока какие-либо команды, задающие начальные параметры расчета, выполняемого блоком. Событие инициализации возникает только один раз, а начальные параметры расчета необходимо задавать после каждого его сброса. Для этого можно использовать специальные

флаги, обрабатываемые в реакции на такт расчета (см. пример на стр. 156) или реакцию на сброс расчета (см. §3.8.3.4 на стр. 292).

Реакция на событие инициализации вводится на вкладке “события” левой панели редактора модели: раздел “создание и уничтожение”, подраздел “инициализация блока” (см. рис. 459 на стр. 243). Чаще всего эта реакция используется для настройки каких-либо объектов блока, параметры которых не настраиваются пользователем и не изменяются в процессе работы блока. Например, в примере из §3.7.12 (стр. 238) в реакции инициализации модель блока добавляет свой собственный пункт в главное меню РДС.

Для реакции на инициализацию модуль автоматически создает в классе блока функцию без параметров с именем `rdsbcppInit`, внутрь которой вставляется текст программы, введенной пользователем на соответствующей вкладке редактора. Функция имеет следующий вид:

```
// Инициализация данных блока
void rdsbcppBlockClass::rdsbcppInit(void)
{
```

*Пользовательский текст реакции*

```
}
```

При написании моделей блоков без использования модуля автокомпиляции событию инициализации соответствует константа РДС `RDS_BFM_INIT` (см. §A.2.4.7 приложения к руководству программиста [2]).

### §3.8.2.2. Очистка данных блока

Очистка данных блока – это самое последнее событие в “жизни” блока, возникающее непосредственно перед отключением модели от этого блока из-за его удаления, выгрузки схемы из памяти (например, при закрытии РДС) или подключения к блоку другой модели. В реакции на это событие, как и в реакции на событие инициализации, *нельзя* обращаться к переменным блока – если сделать это, будет выведено сообщение об ошибке (см. рис. 482). В основном, реакция на очистку данных блока используется для удаления динамических объектов: если в реакции на инициализацию блока разработчик модели вручную отвел память под какие-либо служебные объекты, в реакции на очистку эту память нужно освободить.

Реакция на событие очистки данных вводится на вкладке “события” левой панели редактора модели: раздел “создание и уничтожение”, подраздел “очистка блока”. Для этой реакции модуль автокомпиляции создает в классе блока функцию без параметров с именем `rdsbcppCleanup`, внутрь которой вставляется текст программы, введенной пользователем. Функция имеет следующий вид:

```
// Очистка данных блока перед отключением модели
void rdsbcppBlockClass::rdsbcppCleanup(void)
{
```

*Пользовательский текст реакции*

```
}
```

При написании моделей блоков без использования модуля автокомпиляции событию очистки данных соответствует константа РДС `RDS_BFM_CLEANUP` (см. §A.2.4.3 приложения к руководству программиста [2]).

### §3.8.2.3. Добавление блока пользователем

Это событие возникает после добавления блока в подсистему пользователем из библиотеки, из файла или из буфера обмена. При добавлении блока в схему не из-за

действий пользователя (например, если блок добавлен по команде от другого блока, или при загрузке блока в составе вставляемой из буфера обмена подсистемы) событие не возникает. В реакции на это событие можно, например, спросить что-нибудь у пользователя и, в зависимости от его ответа, изменить параметры блока.

Реакция на событие добавления блока вводится на вкладке “события” левой панели редактора модели: раздел “создание и уничтожение”, подраздел “добавление блока пользователем”. Модуль автокомпиляции создает для этой реакции в классе блока функцию с именем `rdsbcppManualInsert`, внутрь которой вставляется текст программы, введенной пользователем. Функция имеет следующий вид:

```
// Действия после добавления блока пользователем
void rdsbcppBlockClass::rdsbcppManualInsert(
 RDS_PMANUALINSERTDATA InsertData)
{
```

*Пользовательский текст реакции*

```
}
```

В функцию передается единственный параметр `InsertData`, представляющий собой указатель на структуру описания события `RDS_MANUALINSERTDATA`. В файлах заголовков РДС эта структура описана следующим образом:

```
typedef struct {
 int Reason; // Способ добавления (константа RDS_LS_*)
 BOOL Single; // Добавлен только один блок
} RDS_MANUALINSERTDATA;
typedef RDS_MANUALINSERTDATA *RDS_PMANUALINSERTDATA;
```

Поля структуры имеют следующий смысл:

- `Reason` – одна из двух стандартных констант, указывающих на способ добавления блока в схему: `RDS_LS_LOADCLIPBRD`, если блок вставлен из буфера обмена, и `RDS_LS_LOADFROMFILE`, если он добавлен из библиотеки или загружен из файла.
- `Single` – `TRUE`, если в схему вставлен только один блок, и `FALSE`, если вставлено сразу несколько блоков (например, группа блоков из буфера обмена).

При написании моделей блоков без использования модуля автокомпиляции событию добавления блока пользователем соответствует константа РДС `RDS_BFM_MANUALINSERT` (см. §A.2.7.2 приложения к руководству программиста [2] и пример в §2.12.8 руководства программиста [1]).

#### §3.8.2.4. Удаление блока пользователем

Это событие возникает непосредственно перед удалением блока из схемы в результате действий пользователя: если пользователь в режиме редактирования удаляет данный блок, группу выделенных блоков, в которую входит данный, или подсистему, внутри которой данный блок находится. При удалении блока не из-за действий пользователя (например, если память очищается перед загрузкой новой схемы) событие не возникает. Реакция на это событие может использоваться, например, для информирования пользователя о возможных последствиях удаления важных для работы блоков. Отменить удаление блока при помощи этой реакции нельзя.

Реакция на событие удаления блока вводится на вкладке “события” левой панели редактора модели: раздел “создание и уничтожение”, подраздел “удаление блока пользователем”. В классе блока для нее создается функция с именем `rdsbcppManualDelete` с введенным пользователем текстом программы внутри:



```
// Действия перед удалением блока пользователем
void rdsbcppBlockClass::rdsbcppManualDelete(
 RDS_PMANUALDELETEDATA DeleteData)
{
 Пользовательский текст реакции
}
}
```

В функцию передается параметр DeleteData, представляющий собой указатель на структуру описания события RDS\_MANUALDELETEDATA:

```
typedef struct {
 BOOL Single; // Удаляется один блок
 BOOL WithSys; // Удаляется блок внутри удаляемой подсистемы
} RDS_MANUALDELETEDATA;
typedef RDS_MANUALDELETEDATA *RDS_PMANUALDELETEDATA;
```

Поля структуры имеют следующий смысл:

- Single – TRUE, если данный блок – единственный удаляемый, и FALSE, если удаляется группа выделенных блоков, в которую входит данный.
- WithSys – TRUE, если пользователь удаляет подсистему, внутри которой (на любом уровне вложенности) находится этот блок, и FALSE, если этот блок непосредственно входит в удаляемую группу выделенных блоков.

При написании моделей блоков без использования модуля автокомпиляции событию удаления блока пользователем соответствует константа РДС RDS\_BFM\_MANUALDELETE (см. §A.2.7.1 приложения к руководству программиста [2]).

### §3.8.2.5. Перед выгрузкой схемы

Это событие возникает во всех блоках схемы непосредственно перед тем, как текущая загруженная схема будет удалена из памяти из-за загрузки другой схемы, создания новой, или из-за завершения РДС. Реакция на него чаще всего используется для очистки вспомогательных данных, связанных со всей схемой в целом – например, для стирания какого-либо временного файла.

Реакция на это событие вводится на вкладке “события” левой панели редактора модели: раздел “создание и уничтожение”, подраздел “перед выгрузкой системы”. В классе блока для нее создается функция с именем rdsbcppSystemUnload без параметров с введенным пользователем текстом программы внутри:

```
// Действия перед выгрузкой всей системы
void rdsbcppBlockClass::rdsbcppSystemUnload(void)
{
 Пользовательский текст реакции
}
}
```

При написании моделей блоков без использования модуля автокомпиляции событию, возникающему перед выгрузкой схемы, соответствует константа РДС RDS\_BFM\_UNLOADSYSTEM (см. §A.2.4.17 приложения к руководству программиста [2]).

### §3.8.3. Моделирование и переключение режимов

Рассматриваются реакции на события, связанные с переключением режимов РДС, работой в режиме расчета и изменением динамических переменных (см. §3.6.3 на стр. 42).

#### §3.8.3.1. Выполнение такта расчета

Выполнение такта расчета – это самое часто используемое в моделях событие. В режиме расчета оно в каждом такте возникает у всех простых блоков, у которых установлен флаг запуска в каждом такте (см. §2.9.1) или взведен входной сигнал готовности (обычно он называется *Start*, см. пример в §3.7.2.1 на стр. 92). Многие простейшие модели автокомпилируемых блоков состоят только из реакции на это событие. Как правило, в реакции на него модель блока вычисляет значения выходов по значениям входов, настроечных параметров, динамических переменных и т.п. Примеры использования этого события во множестве содержатся в §3.7.

Реакция на выполнение такта расчета вводится на вкладке “события” левой панели редактора модели: раздел “моделирование и режимы”, подраздел “модель”. При создании новой пустой модели вкладка для ввода этой реакции открывается автоматически. В классе блока для нее создается функция с именем *rdsbcppModel*:

```
// Один такт моделирования
void rdsbcppBlockClass::rdsbcppModel(
 RDS_PINITIALCALCDATA InitialCalc)
{

Пользовательский текст реакции

}
```

Реакция на выполнение такта расчета может также вызываться и при так называемом инициализационном, или предварительном, расчете. Предварительный расчет выполняется в момент выхода из режима редактирования по запросу некоторых блоков – например, полей ввода. Он позволяет уменьшить число тактов переходного процесса в алгебраических цепочках, подключенных к таким блокам. Если, например, к полю ввода подключена цепочка из трех последовательно соединенных сумматоров или блоков умножения на константу, правильное значение на выходе третьего блока цепочки установится только через три такта после запуска расчета (см. рис. 10 в §1.4 части I). Если же эти сумматоры или блоки умножения на константу поддерживают предварительный расчет, правильное значение установится на выходе непосредственно перед запуском расчета, поскольку все блоки цепочки будут последовательно вызваны для реакции на такт расчета (хотя расчет еще не идет) и их данные будут переданы на вход следующих блоков в момент выхода из режима редактирования. Для того, чтобы блок вызывался при предварительном расчете, в параметрах его модели должен быть включен флажок “блок участвует в инициализационном расчете” (см. стр. 72). Чтобы блок начинал предварительный расчет, необходимо также включить флажок “блок начинает инициализационный расчет”.

В большинстве случаев разработчику модели можно не задумываться над тем, вызвана данная реакция, как и положено, в нормальном режиме расчета, или при предварительном расчете. При необходимости, о причине вызова можно узнать по передаваемому в функцию указателю *InitCalc*. В режиме расчета этот параметр будет равен *NULL*, при предварительном расчете он будет указывать на структуру *RDS\_INITIALCALCDATA*, содержащую единственное поле *FirstInChain* типа *BOOL*. В этом поле будет находиться значение *TRUE*, если блок был вызван из-за того, что он сам начинает предварительный расчет, и значение *FALSE*, если он вызван из-за прихода на вход данных от другого блока в процессе предварительного расчета.

Здесь не приводятся примеры блоков, поддерживающих инициализационный расчет – в простых моделях вполне можно обойтись и без него, задержка данных на несколько тактов после запуска расчета, как правило, не влияет на работоспособность схемы.

При написании моделей блоков без использования модуля автокомпиляции событию такта расчета соответствует константа РДС RDS\_BFM\_MODEL (см. §A.2.4.9 приложения к руководству программиста [2]).

### §3.8.3.2. Запуск расчета

Это событие возникает при переходе РДС в режим расчета из режима моделирования (при запуске расчета из режима редактирования РДС сначала переходит в режим моделирования, а уже потом – в режим расчета). Реакция на него вводится на вкладке “события” левой панели редактора модели: раздел “моделирование и режимы”, подраздел “запуск расчета” (см. рис. 458 на стр. 242). В классе блока для нее создается функция с именем rdsbcppStartCalc с введенным пользователем текстом программы внутри:

```
// Запуск расчета
void rdsbcppBlockClass::rdsbcppStartCalc(
 RDS_PSTARTSTOPDATA StartStopData)
{
```

*Пользовательский текст реакции*

```
}
```

В функцию передается параметр StartStopData, представляющий собой указатель на структуру описания события RDS\_STARTSTOPDATA:

```
typedef struct {
 BOOL FirstStart; // Расчет запущен с самого начала
 BOOL Loop; // Расчет будет работать непрерывно
} RDS_STARTSTOPDATA;
typedef RDS_STARTSTOPDATA *RDS_PSTARTSTOPDATA;
```

Поля структуры имеют следующий смысл:

- FirstStart – TRUE, если расчет запущен с самого начала (сразу после загрузки схемы или сброса расчета), или FALSE, если расчет повторно запущен после остановки.
- Loop – TRUE, если расчет запущен в нормальном, циклическом, режиме, или FALSE, если после выполнения одного такта он будет автоматически остановлен (пользователь нажал кнопку “выполнить один такт”).

Следует помнить, что это событие возникает не только при первом запуске расчета, но и при его запуске после остановки, поэтому использовать реакцию на него для инициализации расчета без анализа поля FirstStart структуры описания события нельзя.

Пример реакции на это событие приведен в §3.7.12 (стр. 238). При написании моделей блоков без использования модуля автокомпиляции событию запуска расчета соответствует константа РДС RDS\_BFM\_STARTCALC (см. §A.2.4.14 приложения к руководству программиста [2]).

### §3.8.3.3. Остановка расчета

Это событие возникает при остановке расчета, то есть при переходе РДС из режима расчета в режим моделирования. Реакция на него вводится на вкладке “события” левой панели редактора модели: раздел “моделирование и режимы”, подраздел “остановка расчета” (см. рис. 458 на стр. 242). В классе блока для нее создается функция с именем rdsbcppStopCalc:

```
// Остановка расчета
void rdsbcppBlockClass::rdsbcppStopCalc(
 RDS_PSTARTSTOPDATA StartStopData)
{

Пользовательский текст реакции

}
```

Параметр StartStopData – это указатель на структуру описания события RDS\_STARTSTOPDATA, такую же, как и в событии запуска расчета (см. §3.8.3.2 выше). Пример реакции на это событие приведен в §3.7.12 (стр. 238). При написании моделей блоков без использования модуля автокомпиляции остановке расчета соответствует константа РДС RDS\_BFM\_STOPCALC (см. §A.2.4.15 приложения к руководству программиста [2]).

### §3.8.3.4. Сброс расчета

Событие сброса расчета возникает при сбросе расчета пользователем во всей схеме или программном сбросе какой-либо отдельной ее подсистемы функцией rdsResetSystemState (см. §A.5.7.3 приложения к руководству программиста [2]). Реакция на него вводится на вкладке “события” левой панели редактора модели: раздел “моделирование и режимы”, подраздел “сброс расчета”. Реакция на это событие чаще всего служит для возвращения к начальному состоянию каких-либо пользовательских полей класса блока. Введенный пользователем текст программы вставляется в созданную в классе блока функцию без параметров с именем rdsbcppResetCalc:

```
// Сброс расчета
void rdsbcppBlockClass::rdsbcppResetCalc(void)
{

Пользовательский текст реакции

}
```

При написании моделей блоков без использования модуля автокомпиляции сбросу расчета соответствует константа РДС RDS\_BFM\_RESETCALC (см. §A.2.4.12 приложения к руководству программиста).

### §3.8.3.5. Переход в режим редактирования

Это событие возникает при переходе РДС в режим редактирования, в том числе, и сразу после загрузки схемы. Реакция на него вводится на вкладке “события” левой панели редактора модели: раздел “моделирование и режимы”, подраздел “режим редактирования”. Она может использоваться для очистки каких-либо вспомогательных данных, запомненных блоком при переходе в режим моделирования (см. §3.8.3.6 ниже). Введенный пользователем текст программы вставляется в созданную в классе блока функцию без параметров с именем rdsbcppEditMode:

```
// Вход в режим редактирования
void rdsbcppBlockClass::rdsbcppEditMode(void)
{
```

*Пользовательский текст реакции*

}

При написании моделей блоков без использования модуля автокомпиляции событию перехода в режим редактирования соответствует константа РДС RDS\_BFM\_EDITMODE (см. §A.2.4.5 приложения к руководству программиста [2]).

### §3.8.3.6. Переход в режим моделирования

Это событие возникает при переходе РДС в режим моделирования из режимов расчета и редактирования. Реакция на него вводится на вкладке “события” левой панели редактора модели: раздел “моделирование и режимы”, подраздел “режим моделирования”. Эта реакция может использоваться для подготовки и запоминания каких-либо вспомогательных данных, связанных со структурой схемы. Например, блок может составить список идентификаторов других блоков, соединенных с ним связями, и запомнить его: поскольку изменение схемы в режимах моделирования и расчета невозможно, этот список будет актуален до возврата РДС в режим редактирования. Введенный пользователем текст программы вставляется в созданную в классе блока функцию без параметров с именем rdsbcppCalcMode:

```
// Вход в режим моделирования
void rdsbcppBlockClass::rdsbcppCalcMode(void)
{
```

*Пользовательский текст реакции*

}

При написании моделей блоков без использования модуля автокомпиляции событию перехода в режим моделирования соответствует константа РДС RDS\_BFM\_CALCMODE (см. §A.2.4.1 приложения к руководству программиста [2]).

### §3.8.3.7. Изменение динамической переменной

Событие изменения динамической переменной возникает во всех блоках, подписанных на эту переменную, при ее создании, уничтожении, или при вызове для нее функции уведомления подписчиков об изменениях (что такое динамические переменные и как с ними работать объясняется в §3.6.3 на стр. 42 и в §3.7.3 на стр. 129). Реакция на это событие вводится на вкладке “события” левой панели редактора модели: раздел “моделирование и режимы”, подраздел “изменение динамической переменной”. В этой реакции обычно выполняют действия, для которых требуется значение изменившейся переменной. Во многих блоках, получающих данные через динамические переменные, выгоднее производить вычисления не каждый такт, а только при изменении этих переменных.

В классе блока для реакции на событие изменения динамической переменной создается функция с именем rdsbcppDynVarChange, параметр которой указывает на изменившуюся переменную:

```
// Изменение динамической переменной
void rdsbcppBlockClass::rdsbcppDynVarChange(RDS_PDYNVARLINK Link)
{
```

}

Параметр функции Link представляет собой указатель на структуру подписки блока на переменную RDS\_DYNVARLINK:

```
typedef struct {
 LPVOID Data; // Адрес области данных переменной
 LPSTR VarName; // Имя переменной
 LPSTR VarType; // Тип переменной
 RDS_BHANDLE Provider; // Блок-владелец переменной
 LPVOID UID; // Служебный идентификатор переменной
 // (служебное поле, изменять нельзя)
 RDS_VHANDLE Var; // Идентификатор переменной для
 // использования в сервисных функциях
} RDS_DYNVARLINK;
typedef RDS_DYNVARLINK *RDS_PDYNVARLINK;
```

Поля структуры имеют следующий смысл:

- Data – указатель на область данных переменной (в автокомпилируемых моделях не используется).
- VarName – указатель на строку (char\*) с именем динамической переменной. Строка находится во внутренней памяти РДС, ее нельзя изменять.
- VarType – указатель на строку типа динамической переменной (в автокомпилируемых моделях не используется).
- Provider – идентификатор блока, в котором находится данная динамическая переменная (может использоваться, например, для прямого вызова у этого блока какой-либо функции, см. §3.7.13 на стр. 245).
- UID – служебный идентификатор данной переменной, используемый внутри РДС. Значение этого поля нельзя изменять.
- Var – идентификатор переменной (тип RDS\_VHANDLE), используемый в некоторых сервисных функциях для работы с переменными. Эти функции описаны в приложении к руководству программиста [2]. В автокомпилируемых моделях этот идентификатор обычно не используется.

В автокомпилируемых моделях поля этой структуры практически никогда не используются. Чаще всего указатель на эту структуру служит своего рода уникальным идентификатором динамической переменной внутри блока, по которому можно понять, какая именно из переменных изменилась, если в блоке их несколько. Например, если в модель блока добавлена динамическая переменная с именем DynVar1, в реакции на изменение динамической переменной можно сделать следующую проверку:

```
if (DynVar1.CheckLink(Link))
{ // Действия при изменении DynVar1
 ...
}
```

Следует помнить, что событие изменения динамической переменной не возникает автоматически при присваивании ей в модели нового значения: присвоивший значение блок должен явно вызвать у объекта переменной функцию-член NotifySubscribers, чтобы уведомить всех подписчиков на переменную о произведенном изменении. Например, уведомить всех подписчиков на переменную DynVar1 о присвоении ей нового значения можно так:

```
DynVar1=10; // Присвоение значения
DynVar1.NotifySubscribers(); // Уведомление подписчиков
```



Пример реакции на это событие приведен в §3.7.3.2 (стр. 136). При написании моделей блоков без использования модуля автокомпиляции событию изменения динамической переменной соответствует константа РДС RDS\_BFM\_DYNVARCHANGE (см. §A.2.4.4 приложения к руководству программиста).

### §3.8.4. Реакции блока на мышшь и клавиатуру

Рассматриваются реакции на события, связанные с действиями пользователя в режимах моделирования и расчета: движение курсора мыши, нажатие кнопок и т.п. Такие реакции позволяют создавать интерактивные блоки.

#### §3.8.4.1. Нажатие кнопки мыши

Это событие возникает в режимах моделирования и расчета при нажатии пользователем любой кнопки мыши в момент нахождения курсора в пределах изображения блока, если этот блок находится на видимом слое, редактирование которого разрешено (слои подробно описаны в §2.12 части I), и в параметрах этого блока разрешена реакция на мышшь (см. рис. 448 на стр. 229). Реакция на нажатие кнопки мыши вводится на вкладке “события” левой панели редактора модели: раздел “мышшь и клавиатура”, подраздел “нажатие кнопки мыши” (см. рис. 447 на стр. 226). Эта реакция часто используется в моделях блоков пользовательского интерфейса: кнопок, рукояток и т.п.

В классе блока для этого события создается функция с именем rdsbcppMouseDown следующего вида:

```
// Нажатие кнопки мыши
void rdsbcppBlockClass::rdsbcppMouseDown(
 RDS_PMOUSEDATA MouseData, int &Result)
{

Пользовательский текст реакции

}
```

У этой функции два параметра. Параметр MouseData – это указатель на структуру описания события RDS\_MOUSEDATA:

```
typedef struct {
 int x,y; // Координаты курсора мыши
 int BlockX,BlockY; // Координаты точки привязки блока
 int Left,Top; // Верхний левый угол изображения блока
 int Width,Height; // Размеры изображения блока
 int IntZoom; // Масштаб окна подсистемы в %
 DWORD Button; // Кнопка мыши (RDS_M*)
 DWORD Shift; // Флаги мыши и клавиатуры (RDS_M*, RDS_K*)
 double DoubleZoom; // Масштабный к-т окна (в долях единицы)
 int MouseEvent; // Событие
 int Viewport; // Номер порта вывода или -1
} RDS_MOUSEDATA;
typedef RDS_MOUSEDATA *RDS_PMOUSEDATA;
```

Поля структуры имеют следующий смысл (см. также стр. 226):

- x, y – координаты курсора мыши на рабочем поле окна подсистемы на момент возникновения события (уже с учетом масштаба подсистемы).
- BlockX, BlockY – координаты точки привязки блока на рабочем поле с учетом масштаба и возможной связи положения этого блока с переменными. Для блоков с векторной картинкой точка привязки – это положение начала координат этой картинки, для всех остальных – левый верхний угол прямоугольной области.

- `Left, Top, Width, Height` – координаты левого верхнего угла (`Left, Top`) прямоугольной области, занимаемой блоком, ее ширина (`Width`) и высота (`Height`) в текущем масштабе с учетом возможной связи положения и размеров блока с его переменными.
- `IntZoom` – текущий масштаб окна подсистемы в процентах (используется крайне редко).
- `Button` – кнопка мыши, нажатие которой вызвало событие: `RDS_MLEFTBUTTON` – левая кнопка, `RDS_MRIGHTBUTTON` – правая кнопка, `RDS_MMIDDLEBUTTON` – средняя кнопка.
- `Shift` – набор битовых флагов, описывающих клавиши и кнопки, которые были нажатыми в момент возникновения события: `RDS_MLEFTBUTTON`, `RDS_MRIGHTBUTTON`, `RDS_MMIDDLEBUTTON` – нажатые кнопки мыши (см. выше), `RDS_KSHIFT` – нажата клавиша “Shift”, `RDS_KALT` – нажата клавиша “Alt”, `RDS_KCTRL` – нажата клавиша “Ctrl”.
- `DoubleZoom` – текущий масштаб окна родительской подсистемы блока в долях единицы: 1 – 100%, 0.5 – 50%, 2 – 200% и т.п.
- `MouseEvent` – константа, указывающая на произошедшее событие (нажатие, отпускание, перемещение курсора и т.п.) Поскольку в автокомпилируемых моделях для каждого из этих событий создается отдельная функция, это поле структуры используется крайне редко. Подробнее о его возможных значениях можно прочесть в приложении к руководству программиста [2].
- `Viewport` – номер порта вывода (см. §3.6 руководства программиста [1]), из которого пришла информация о событии (нажатии, отпускании кнопки, перемещении курсора и т.п.), или -1, если событие произошло в обычном окне подсистемы.

Из этой структуры можно узнать, какая именно кнопка мыши нажата, где в пределах изображения блока находился курсор в момент нажатия, и т.п.

Второй параметр функции реакции – это ссылка на целую переменную `Result`. Через этот параметр модель может сообщить РДС, обработала ли она нажатие кнопки. Для этого параметру `Result` нужно присвоить одну из трех стандартных констант:

- `RDS_BFR_DONE` – модель успешно обработала нажатие кнопки мыши, дальнейшая обработка нажатия другими блоками или РДС не требуется.
- `RDS_BFR_NOTPROCESSED` – модель отказалась обработать нажатие кнопки. Если под изображением данного блока находится изображение другого, для обработки нажатия кнопки будет вызвана модель этого другого блока.
- `RDS_BFR_SHOWMENU` – модель успешно обработала нажатие кнопки мыши, но, несмотря на это, если нажатая кнопка была правой, необходимо показать стандартное контекстное меню РДС.

Манипулируя значением `Result`, можно, например, сделать блок в некоторых его состояниях “прозрачным” для щелчков мыши, возвращая `RDS_BFR_NOTPROCESSED` (реагировать на нажатие будет блок под данным). Если блок реагирует только на щелчки левой кнопки, при щелчках правой желательно присваивать `Result` значение `RDS_BFR_SHOWMENU`, иначе контекстное меню блока в режимах моделирования и расчета не будет выведено. По умолчанию в `Result` записано значение `RDS_BFR_DONE`, поэтому, если в реакции на событие этому параметру ничего не будет присвоено, блок будет считаться успешно обработавшим нажатие.

Пример реакции на это событие приведен в §3.7.11 (стр. 226). При написании моделей блоков без использования модуля автокомпиляции нажатие кнопки мыши соответствует константа РДС `RDS_BFM_MOUSEDOWN` (см. §A.2.6.9 приложения к руководству программиста).



### §3.8.4.2. Отпускание кнопки мыши

Это событие возникает в режимах моделирования и расчета при отпускании пользователем любой ранее нажатой кнопки мыши в момент нахождения курсора в пределах изображения блока, если этот блок находится на видимом слое, редактирование которого разрешено (слои подробно описаны в §2.12 части I), и в параметрах этого блока разрешена реакция на мышь (см. рис. 448 на стр. 229). Реакция на это событие вводится на вкладке “события” левой панели редактора модели: раздел “мышь и клавиатура”, подраздел “отпускание кнопки мыши” (см. рис. 447 на стр. 226). Чаще всего она используется в моделях блоков пользовательского интерфейса: кнопок, рукояток и т.п.

В классе блока для этого события создается функция с именем `rdsbcppMouseUp` следующего вида:

```
// Отпускание кнопки мыши
void rdsbcppBlockClass::rdsbcppMouseUp(
 RDS_PMOUSEDATA MouseData, int &Result)
{
 Пользовательский текст реакции
}
```

У этой функции два параметра. `MouseData` – это указатель на структуру описания события `RDS_MOUSEDATA`. `Result` – это ссылка на целую переменную, через которую модель может сообщить РДС о результате обработки отпускания кнопки. Оба параметра полностью аналогичны параметрам функции реакции на нажатие кнопки мыши, подробно описанным в §3.8.4.1 выше (стр. 295), за исключением того, что, в данном случае, речь идет об отпускании кнопки, а не об ее нажатии.

Пример реакции на это событие приведен в §3.7.11 (стр. 232). При написании моделей блоков без использования модуля автокомпиляции отпусканию кнопки мыши соответствует константа РДС `RDS_BFM_MOUSEUP` (см. §A.2.6.11 приложения к руководству программиста [2]).

### §3.8.4.3. Двойной щелчок мыши

Это событие возникает в режимах моделирования и расчета при двойном щелчке левой кнопки мыши в момент нахождения курсора в пределах изображения блока, если этот блок находится на видимом слое, редактирование которого разрешено (слои подробно описаны в §2.12 части I), и в параметрах этого блока разрешена реакция на мышь (см. рис. 448 на стр. 229). Реакция на это событие вводится на вкладке “события” левой панели редактора модели: раздел “мышь и клавиатура”, подраздел “двойной щелчок мыши” (см. рис. 447 на стр. 226).

В классе блока для этого события создается функция с именем `rdsbcppMouseDownClick` следующего вида:

```
// Двойной щелчок мыши
void rdsbcppBlockClass::rdsbcppMouseDownClick(
 RDS_PMOUSEDATA MouseData, int &Result)
{
 Пользовательский текст реакции
}
```

Параметры функции полностью аналогичны параметрам функции реакции на нажатие кнопки мыши, подробно описанным в §3.8.4.1 выше (стр. 295), за исключением того, что

здесь речь идет о двойном щелчке, и поле Button структуры описания события можно не анализировать – событие вызывается только левой кнопкой. Следует учитывать, что, поскольку двойной щелчок технически состоит из двух последовательных одиночных щелчков, до события двойного щелчка всегда сначала возникает обычное событие нажатия кнопки.

При написании моделей блоков без использования модуля автокомпиляции двойному щелчку соответствует константа РДС RDS\_BFM\_MOUSEDBLCLICK (см. §A.2.6.8 приложения к руководству программиста [2]).

#### §3.8.4.4. Перемещение курсора мыши

Это событие возникает в режимах моделирования и расчета при перемещении курсора мыши в пределах изображения блока, если этот блок находится на видимом слое, редактирование которого разрешено (слои подробно описаны в §2.12 части I), и в параметрах этого блока разрешена реакция на мышь (см. рис. 448 на стр. 229). По умолчанию событие возникает только тогда, когда в момент перемещения курсора какая-либо из кнопок мыши нажата. Если модель должна обрабатывать перемещения курсора даже если ни одна кнопка не нажата, в параметрах блока должен быть включен дополнительный флажок “блок реагирует на движения мыши без нажатия кнопок” (см. там же на рис. 448).

Реакция на перемещение курсора вводится на вкладке “события” левой панели редактора модели: раздел “мышь и клавиатура”, подраздел “перемещение мыши” (см. рис. 447 на стр. 226). Чаще всего она используется в моделях блоков, имитирующих различные ручки.

В классе блока для этого события создается функция с именем rdsbcppMouseMove следующего вида:

```
// Перемещение мыши
void rdsbcppBlockClass::rdsbcppMouseMove(
 RDS_PMOUSEDATA MouseData, int &Result)
{
 Пользовательский текст реакции
}
```

Параметры функции полностью аналогичны параметрам функции реакции на нажатие кнопки мыши, подробно описанным в §3.8.4.1 выше (стр. 295).

Пример реакции на это событие приведен в §3.7.11 (стр. 232). При написании моделей блоков без использования модуля автокомпиляции перемещению курсора мыши соответствует константа РДС RDS\_BFM\_MOUSEMOVE (см. §A.2.6.10 приложения к руководству программиста [2]).

#### §3.8.4.5. Нажатие клавиши

Это событие возникает в режимах моделирования и расчета при нажатии пользователем какой-либо клавиши на клавиатуре, если окно подсистемы с данным блоком имеет фокус (то есть, это самое верхнее окно, и РДС при этом – активное приложение) и на вкладке “DLL” окна параметров блока включен флажок “блок реагирует на клавиатуру” (рис. 483).

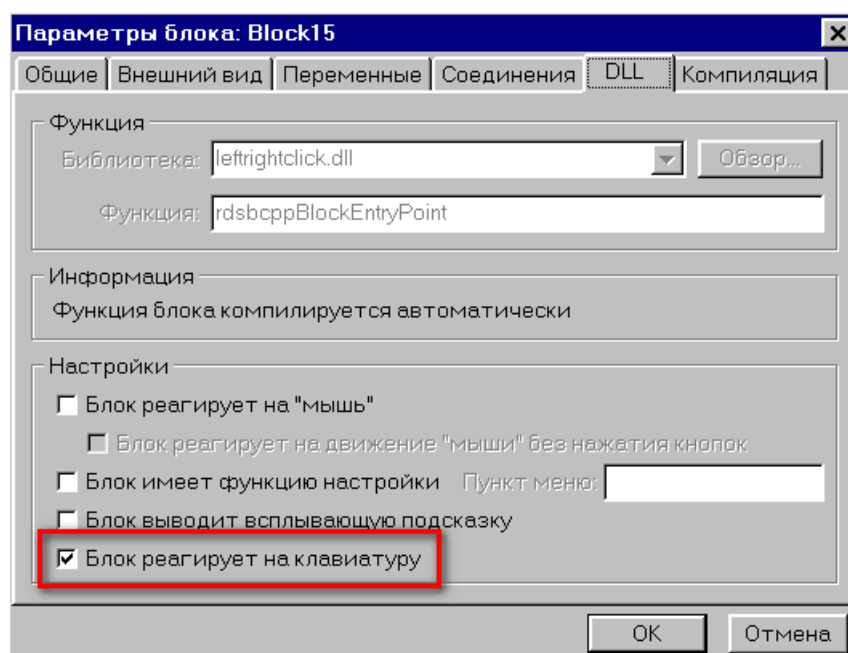


Рис. 483. Разрешение реакции на клавиатуру в окне параметров блока

Реакция на нажатие клавиши вводится на вкладке “события” левой панели редактора модели: раздел “мышь и клавиатура”, подраздел “нажатие клавиши”. В классе блока для нее создается функция с именем `rdsbcppKeyDown` следующего вида:

```
// Нажатие клавиши
void rdsbcppBlockClass::rdsbcppKeyDown(
 RDS_PKEYDATA KeyData, int &Result)
{

Пользовательский текст реакции

}
```

В параметре `KeyData` в функцию передается указатель на структуру описания события `RDS_KEYDATA`:

```
typedef struct {
 int KeyCode; // Виртуальный код клавиши
 BOOL Repeat; // Признак автоповтора
 int RepeatCount; // Число повторений
 DWORD Shift; // Флаги клавиатуры (RDS_M*, RDS_K*)
 int KeyEvent // Событие - RDS_BFM_KEYDOWN или
 // RDS_BFM_KEYUP
 BOOL Handled; // Событие обработано (возврат)
 int Viewport; // Номер порта вывода или -1
} RDS_KEYDATA;
typedef RDS_KEYDATA *RDS_PKEYDATA;
```

Поля структуры имеют следующий смысл:

- `KeyCode` – виртуальный код нажатой клавиши (`VK_*`) согласно описаниям Windows API.
- `Repeat` – `TRUE`, если нажатие клавиши сгенерировано автоповтором клавиатуры, и `FALSE` в противном случае (при одиночных нажатиях клавиш в этом поле всегда находится `FALSE`).
- `RepeatCount` – при автоповторе нажатия клавиши (`Repeat==TRUE`) – число повторов с момента прошлого вызова реакции на событие.

- `Shift` – набор битовых флагов, описывающие клавиши и кнопки, которые были нажатыми в момент возникновения события: `RDS_MLEFTBUTTON`, `RDS_MRIGHTBUTTON`, `RDS_MMIDDLEBUTTON` – нажатые левая, правая и средняя кнопки мыши соответственно, `RDS_KSHIFT` – нажата клавиша “Shift”, `RDS_KALT` – нажата клавиша “Alt”, `RDS_KCTRL` – нажата клавиша “Ctrl”.
- `KeyEvent` – константа, указывающая на произошедшее событие (нажатие или отпускание клавиши). Поскольку в автокомпилируемых моделях для каждого из этих событий создается отдельная функция, это поле структуры используется крайне редко. Подробнее о его возможных значениях можно прочесть в приложении к руководству программиста [2].
- `Handled` – перед вызовом самого первого блока подсистемы (когда подсистема на переднем плане, о нажатии клавиш по очереди информируются все ее блоки) это поле устанавливается в `FALSE`. Модель блока может присвоить ему `TRUE` для того, чтобы сообщить РДС об успешной обработке нажатия клавиши и о том, что дальше перебирать блоки, информируя их о нажатии, не нужно.
- `Viewport` – номер порта вывода (см. §3.6 руководства программиста [1]), из которого пришла информация о нажатии клавиши, или `-1`, если событие произошло в обычном окне подсистемы.

Чаще всего в этой структуре используются поле `KeyCode`, содержащее код нажатой клавиши, и поле `Shift`, содержащее набор битовых флагов, соответствующих одновременно нажатым служебным клавишам “Shift”, “Alt” и “Ctrl”.

Второй параметр функции реакции – это ссылка на целую переменную `Result`. Через этот параметр модель может сообщить РДС, обработала ли она нажатие клавиши. Для этого параметру `Result` нужно присвоить одну из двух стандартных констант:

- `RDS_BFR_DONE` – нажатие клавиши не обработано блоком, РДС будет продолжать перебирать блоки подсистемы, пока один из них не обработает нажатие.
- `RDS_BFR_STOP` – нажатие клавиши обработано, его не нужно передавать в остальные блоки подсистемы и вызывать соответствующий ей пункт главного меню РДС, если такой имеется.

По умолчанию в `Result` записано значение `RDS_BFR_DONE`, поэтому, если в реакции на событие этому параметру ничего не будет присвоено, блок будет считаться успешно обработавшим нажатие.

При написании моделей блоков без использования модуля автокомпиляции нажатие клавиши соответствует константа РДС `RDS_BFM_KEYDOWN` (см. §A.2.6.5 приложения к руководству программиста).

#### §3.8.4.6. Отпускание клавиши

Это событие возникает в режимах моделирования и расчета при отпускании пользователем какой-либо ранее нажатой клавиши на клавиатуре, если окно подсистемы с данным блоком имеет фокус (это самое верхнее окно, и РДС – активное приложение) и на вкладке “DLL” окна параметров блока включен флажок “блок реагирует на клавиатуру” (см. рис. 483 на стр. 299).

Реакция на отпускание клавиши вводится на вкладке “события” левой панели редактора модели: раздел “мышь и клавиатура”, подраздел “отпускание клавиши”. В классе блока для нее создается функция с именем `rdsbcppKeyUp` следующего вида:

```
// Отпускание клавиши
void rdsbcppBlockClass::rdsbcppKeyUp(
 RDS_PKEYDATA KeyData, int &Result)
{
```

*Пользовательский текст реакции*

}

Параметры функции полностью аналогичны параметрам функции реакции на нажатие клавиши, описанным в §3.8.4.5 (стр. 298), за исключением того, что при отпускании клавиши отсутствует автоповтор.

При написании моделей блоков без использования модуля автокомпиляции отпусканию клавиши соответствует константа РДС RDS\_BFM\_KEYUP (см. §A.2.6.6 приложения к руководству программиста [2]).

### §3.8.5. Вызов функции блока

Описывается реакция на вызов функции блока, то есть на непосредственный вызов модели данного блока моделью другого (см. §3.7.13 на стр. 245).

Для каждой функции блока, добавленной в редактор модели (см. §3.6.5 на стр. 57), модуль автокомпиляции автоматически добавляет в список событий реакцию на вызов этой функции. Текст этой реакции вводится на вкладке “события” левой панели редактора модели: раздел “мышь и клавиатура”, подраздел с именем нужной функции (см. рис. 464 на стр. 254). Написание реакций на вызовы функций подробно и с примерами рассматривается в §3.7.13 (стр. 245), здесь же мы рассмотрим техническую сторону этих реакций.

В классе блока для реакции на вызов создается функция-член с именем, автоматически сформированным по имени объекта функции, которое пользователь ввел при ее добавлении в модель. В зависимости от того, есть ли у функции параметр, эта функция-член будет иметь два или три параметра. Если параметр у функции есть, созданная для реакции на ее вызов функция-член будет выглядеть так (двойным подчеркиванием выделены части текста, зависящие от конкретной функции):

```
// Реакция на функцию блока "ИМЯ ФУНКЦИИ"
void rdsbcppBlockClass::ИМЯ РЕАКЦИИ(
 ТИП УКАЗАТЕЛЯ НА ПАРАМЕТР Param,
 RDS_PFUNCTIONCALldata FData,
 int &Result)
{
```

*Пользовательский текст реакции*

}

В параметре Param находится указатель на параметр вызванной функции блока, переданный другой моделью в момент этого вызова. Параметром функции блока, если он есть, всегда должен быть какой-нибудь указатель, и тип этого указателя вводится при добавлении функции в модель (см. рис. 466 на стр. 256, поле ввода “тип параметра функции”). Если, например, в качестве параметра функции было введено “TMyFuncParam\*” (то есть “указатель на некоторую структуру TMyFuncParam”), Param в реакции тоже будет иметь тип TMyFuncParam\*.

Параметр FData – это указатель на общую для всех функций структуру описания события RDS\_FUNCTIONCALldata, из полей которой можно узнать, как и кем именно вызвана функция:

```
typedef struct {
 int Function; // Идентификатор функции в РДС
 LPVOID Data; // Параметр функции
 int Reserved; // Зарезервировано (не используется)
 RDS_BHANDLE Caller; // Вызвавший блок
```

```

 BOOL Broadcast; // Вызов не одного блока, а нескольких
 int BroadcastCnt; // Номер блока среди всех вызванных
 BOOL Stop; // Прекратить вызов блоков (возврат)
 BOOL Delayed; // Отложенный вызов
 DWORD DataBufSize; // Размер буфера при отложенном вызове
} RDS_FUNCTIONCALldata;
typedef RDS_FUNCTIONCALldata *RDS_PFUNCTIONCALldata;

```

Поля этой структуры имеют следующий смысл:

- **Function** – уникальный целый идентификатор вызванной функции в РДС. Этот идентификатор присваивается функции автоматически. Его также можно в любой момент получить из объекта функции, создаваемого модулем автокомпиляции для работы с ней, при помощи функции-члена **Id** этого объекта (см. стр. 248).
- **Data** – указатель, являющийся параметром функции (**NULL**, если параметра нет). Значение этого поля совпадает с описанным выше параметром **Param**, за исключением типа: **Param** уже приведен к нужному типу, а поле **Data** – универсальный указатель типа **void\***. В автокомпилируемых моделях использовать это поле не имеет смысла, вместо него используется **Param**.
- **Caller** – идентификатор (**RDS\_BHANDLE**) вызвавшего функцию блока. Модель вызванного блока может использовать его для того чтобы, в свою очередь, вызвать у него какую-либо функцию в ответ.
- **Broadcast** – значение **TRUE** в этом поле сигнализирует о том, что данная функция была вызвана сразу у нескольких блоков (см. §3.7.13.2 на стр. 250), значение **FALSE** – что она вызвана только у данного блока (см. §3.7.13.3 на стр. 258).
- **BroadcastCnt** – при **Broadcast==TRUE** в этом поле будет находиться порядковый (начинающийся с нуля) номер данного блока среди всех вызванных. Например, если функция вызвана у всех блоков какой-либо подсистемы, и в этой подсистеме находится три блока, при вызове первого из них в **BroadcastCnt** будет передан ноль, при вызове второго – 1, при вызове третьего – 2.
- **Stop** – флаг прекращения вызова функции у группы блоков. Исходно в этом поле записано значение **FALSE**. При **Broadcast==TRUE** модель вызванного блока может записать в **Stop** значение **TRUE**, запретив тем самым вызов функции для оставшихся блоков. Таким образом можно, например, организовать поиск в подсистеме блока, выполняющего какие-либо действия: можно вызвать функцию у всех блоков подсистемы и написать реакцию на вызов этой функции так, чтобы первый же выполняющий ее блок произвел необходимые действия или сообщил вызвавшему блоку свой идентификатор, а затем прервал дальнейший перебор блоков и вызов их функций, присвоив **Stop** значение **TRUE**.
- **Delayed** – значение **FALSE** в этом поле указывает на прямой вызов функции, значение **TRUE** – на отложенный. Отложенные вызовы не поддерживаются модулем автокомпиляции напрямую через объекты функций, но их можно выполнять при помощи обычных функций РДС (см. §2.13.5 руководства программиста [1]).
- **DataBufSize** – при отложенном вызове функции в этом поле передается размер области данных, указатель на которую находится в поле **Data**.

Третий параметр функции реакции – это ссылка на целую переменную **Result**. Значение этой переменной возвращается вызвавшему блоку. Допустим, например, что в модель какого-то блока добавлена некоторая функция без параметров, в реакции на вызов которой записано:

```
Result=123;
```

В модель другого блока добавлена та же самая функция, и объект для работы с ней назван **MyFuncObject**. Если идентификатор первого блока будет записан в переменной **block**

(например, этот блок был найден по какому-то признаку перебором всех блоков подсистемы), то вызов

```
int i=MyFuncObject.Call(block);
```

запишет в переменную *i* значение 123. По умолчанию в *Result* записано значение 0 – если в реакции на вызов функции *Result* ничего не будет присвоено, этот ноль там и останется и будет возвращен вызвавшему блоку.

Если у функции нет параметра, в функции-члене для нее просто будет отсутствовать параметр *Param*:

```
// Реакция на функцию блока "ИМЯ ФУНКЦИИ"
void rdsbcppBlockClass::ИМЯ_РЕАКЦИИ(
 RDS_PFUNCTIONCALldata FData,
 int &Result)
{
```

Пользовательский текст реакции

```
}
```

Смысл оставшихся двух параметров остается тем же.

При написании моделей блоков без использования модуля автокомпиляции вызову любой функции блока соответствует константа РДС *RDS\_BFM\_FUNCTIONCALL* (см. §A.2.4.6 приложения к руководству программиста [2]). При этом, чтобы понять, какая именно функция вызвана, программист должен сам анализировать значение поля *Function* в переданной в модель структуре *RDS\_FUNCTIONCALldata*. Модуль автокомпиляции несколько упрощает работу, вставляя этот анализ в модель автоматически.

### §3.8.6. Загрузка и запись данных блока и всей схемы

Рассматриваются реакции на события, связанные с записью и загрузкой схемы и отдельных ее блоков.

#### §3.8.6.1. Загрузка данных блока

Событие загрузки данных блока возникает при загрузке этого блока из файла в составе схемы, при вставке его в схему из библиотеки или из буфера обмена, а также при отмене изменения параметров блока пользователем (при этом прежние данные блока загружаются из внутреннего буфера отмены операций РДС). Следует учитывать, что реакция на событие вызывается, только если у блока есть какие-либо сохраненные параметры – если у блока нет автоматически сохраняемых настроечных параметров и он ничего не сохранил в реакции на запись данных самостоятельно (см. §3.8.6.2 на стр. 304), событие загрузки данных не возникнет. Реакция на загрузку данных вводится на вкладке “события” левой панели редактора модели: раздел “загрузка и запись данных”, подраздел “загрузка данных блока”. Не следует путать эту реакцию с реакцией на загрузку *состояния* блока (см. §3.8.7.1 на стр. 306).

Каждый блок, помимо общих параметров, которые сохраняет и загружает РДС, может иметь личные параметры, за загрузку и запись которых отвечает модель самого блока. При сохранении схемы или блока на диск или при копировании их в буфер обмена модель вызывается для сохранения этих личных параметров. Если модель что-то сохранила, при последующей загрузке этого блока она будет вызвана для чтения сохраненных параметров – возникнет событие загрузки данных блока. Настроечные параметры, добавленные в модель средствами редактора (см. §3.6.6 на стр. 60 и §3.7.6 на стр. 194), сохраняются и загружаются автоматически, для их загрузки данная реакция не требуется.

В классе блока для реакции на событие загрузки данных создается функция с именем *rdsbcppLoadText*:



```
// Загрузка данных блока в текстовом виде
void rdsbcppBlockClass::rdsbcppLoadText(char *LoadedText)
{
 Пользовательский текст реакции
}

```

В параметре `LoadedText` передается указатель на текст (`char*`), сформированный моделью при записи данных блока (реакция на это событие описана ниже). Следует учитывать, что, если в блоке есть настроечные параметры, добавленные средствами редактора модели, в этот текст их данные не войдут: при загрузке блока сначала данные этих параметров извлекаются из текста, и только потом остаток текста передается в реакцию на событие загрузки. Таким образом, в эту реакцию передается в точности тот же текст, который был создан различными вызовами сервисных функций РДС в реакции на запись данных блока.

Реакция на загрузку данных блока может использоваться двумя способами. Во-первых, если у блока есть какие-то сложные параметры, которые невозможно описать средствами редактора модели (например, массивы, связанные списки и т.п.), модель должна уметь сохранять их в текстовом формате в реакции на запись данных и загружать их в этом же формате в реакции на загрузку. Формат выбирается разработчиком произвольно, все действия по формированию текста со значениями параметров и по последующему разбору этого текста должны выполняться вручную. Сохранение и загрузка данных блока в текстовом формате подробно рассматривается в §2.8.3, §2.8.4 и §2.8.5 руководства программиста [1].

Во-вторых, реакцию на загрузку данных блока можно использовать для того, чтобы выполнить какие-либо действия *после* загрузки стандартных параметров блока. Если в модель блока в ее редакторе были добавлены какие-либо настроечные параметры, реакция на загрузку данных будет вызвана сразу после того, как их значения будут считаны, причем она вызовется даже в том случае, если у модели не будет реакции на запись параметров (в `LoadedText` при этом будет передана пустая строка). В §3.7.7 (стр. 203) был приведен пример модели блока, в которой имя динамической переменной, которую этот блок создает, было настроечным параметром. Поскольку значение параметра, то есть имя переменной, становится известным модели только после загрузки параметров, команда создания переменной в этом примере была записана именно в реакции на загрузку данных.

При написании моделей блоков без использования модуля автокомпиляции событию загрузки данных блока соответствует константа РДС `RDS_BFM_LOADTXT` (см. §A.2.5.5 приложения к руководству программиста [2]).

### §3.8.6.2. Запись данных блока

Событие записи данных блока возникает при сохранении этого блока в файл вместе со всей схемой, при сохранении его в отдельный файл, при копировании его в буфер обмена, а также при записи его во вспомогательный буфер отмены операций перед тем, как пользователь изменит параметры этого блока. Реакция на запись данных вводится на вкладке “события” левой панели редактора модели: раздел “загрузка и запись данных”, подраздел “запись данных блока”. Не следует путать эту реакцию с реакцией на запись *состояния* блока (см. §3.8.7.2 на стр. 307).

Если у блока есть какие-то сложные параметры, которые невозможно описать средствами редактора модели (например, массивы, связанные списки и т.п.), в этой реакции модель должна сформировать текст, описывающий эти параметры, который она сможет потом разобрать в реакции на загрузку данных (см. §3.8.6.1 на стр. 303). Этот текст



передается в РДС при помощи специальных сервисных функций, рассмотренных в §2.8.3, §2.8.4 и §2.8.5 руководства программиста [1].

Настроечные параметры, добавленные в модель средствами редактора (см. §3.6.6 на стр. 60 и §3.7.6 на стр. 194), сохраняются и загружаются автоматически, для их сохранения данная реакция не требуется. Тем не менее, в этой реакции можно сохранить какие-либо дополнительные параметры – сформированный для них текст будет передаваться в реакцию на загрузку данных при загрузке блока.

В классе блока для реакции на событие записи данных блока создается функция с именем `rdsbcppSaveText` без параметров:

```
// Запись данных блока в текстовом виде
void rdsbcppBlockClass::rdsbcppSaveText(void)
{
```

*Пользовательский текст реакции*

```
}
```

При написании моделей блоков без использования модуля автокомпиляции событию записи данных блока соответствует константа РДС `RDS_BFM_SAVETXT` (см. §A.2.5.7 приложения к руководству программиста [2]).

### §3.8.6.3. Перед сохранением схемы

Это событие, как и следует из его названия, возникает у всех блоков непосредственно перед сохранением всей схемы в файл. В реакции на него можно подготовить к сохранению какие-либо параметры, относящиеся ко всей схеме в целом – например, сохранить какие-либо настройки или журналы событий в отдельных файлах. Текст реакции вводится на вкладке “события” левой панели редактора модели: раздел “загрузка и запись данных”, подраздел “перед сохранением схемы”.

В классе блока для реакции на событие “перед сохранением схемы” создается функция с именем `rdsbcppBeforeSave`:

```
// Действия перед сохранением всей схемы
void rdsbcppBlockClass::rdsbcppBeforeSave(void)
{
```

*Пользовательский текст реакции*

```
}
```

При написании моделей блоков без использования модуля автокомпиляции этому событию соответствует константа РДС `RDS_BFM_BEFORESAVE` (см. §A.2.5.3 приложения к руководству программиста [2]).

### §3.8.6.4. После сохранения схемы

Это событие возникает у всех блоков сразу после окончания сохранения схемы в файл. В реакции на него можно сохранить в отдельный файл какие-либо параметры, общие для всей схемы, или завершить действия, начатые в реакции перед сохранением схемы (см. §3.8.6.3 на стр. 305). На момент вызова этой реакции файл схемы уже записан и закрыт, и к нему, при необходимости, можно обращаться. Текст реакции вводится на вкладке “события” левой панели редактора модели: раздел “загрузка и запись данных”, подраздел “после сохранения схемы”.

В классе блока для реакции на событие “после сохранения схемы” создается функция с именем `rdsbcppAfterSave`:

```
// Действия после сохранения всей схемы
void rdsbcppBlockClass::rdsbcppAfterSave(void)
{

Пользовательский текст реакции

}
```

При написании моделей блоков без использования модуля автокомпиляции этому событию соответствует константа РДС RDS\_BFM\_AFTERSAVE (см. §A.2.5.2 приложения к руководству программиста [2]).

### §3.8.6.5. После загрузки схемы

Это событие возникает у всех блоков схемы сразу после того, как схема загружена в память РДС. На этот момент все блоки и связи уже загружены, и модель блока может считывать их параметры, вызывать функции блоков, анализировать граф загруженной схемы при помощи сервисных функций РДС и т.п. Текст реакции на событие вводится на вкладке “события” левой панели редактора модели: раздел “загрузка и запись данных”, подраздел “после загрузки схемы”. В классе блока для этой реакции создается функция с именем rdsbcppAfterLoad:

```
// Действия после загрузки всей схемы
void rdsbcppBlockClass::rdsbcppAfterLoad(void)
{

Пользовательский текст реакции

}
```

При написании моделей блоков без использования модуля автокомпиляции событию “после загрузки схемы” соответствует константа РДС RDS\_BFM\_AFTERLOAD (см. §A.2.5.1 приложения к руководству программиста [2]).

### §3.8.7. Загрузка и запись мгновенного состояния блока

Рассматриваются реакции на события, возникающие при сохранении и восстановлении мгновенного состояния части схемы. Эти события порождаются вызовами специальных сервисных функций РДС и позволяют возвращать схему “назад во времени” по команде от одного из ее блоков.

#### §3.8.7.1. Загрузка состояния блока

Событие загрузки состояния возникает у блока в процессе работы сервисной функции rdsLoadSystemState (см. §A.5.7.2 приложения к руководству программиста [2]), вызвав которую какая-либо модель блока может восстановить ранее сохраненное состояние блоков всей схемы или ее отдельной подсистемы. В реакции на это событие модель должна загрузить все данные, сохраненные ей же при реакции на событие сохранения состояния (см. §3.8.7.2 ниже). Загрузка состояния может использоваться для того, чтобы вернуть схему “назад во времени” и повторить расчет, например, с другими входными данными. В интерфейсе пользователя РДС не предусмотрено сохранения состояния схемы и возврата ее в запомненное состояние – этим должны заниматься модели блоков. Сохранение и загрузка состояний блоков и подсистем подробно описаны в §2.14.3 руководства программиста [1].

Следует помнить, что значения статических переменных блока сохраняются и восстанавливаются автоматически. Модель должна сохранять и восстанавливать только значения динамических переменных, за которые она отвечает, и значения своих внутренних параметров, если таковые имеются.

Текст реакции на событие загрузки состояния вводится на вкладке “события” левой панели редактора модели: раздел “загрузка и запись состояния”, подраздел “загрузка состояния блока”. В классе блока для этой реакции создается функция с именем `rdsbcppLoadState`:

```
// Загрузка параметров состояния блока в двоичном виде
void rdsbcppBlockClass::rdsbcppLoadState(void)
{
```

*Пользовательский текст реакции*

```
}
```

Внутри этой функции данные должны загружаться вызовами функции `rdsReadBlockData` в том же порядке, в котором они сохранялись вызовами `rdsWriteBlockData` при сохранении состояния.

При написании моделей блоков без использования модуля автокомпиляции событию загрузки состояния соответствует константа РДС `RDS_BFM_LOADSTATE` (см. §A.2.4.8 приложения к руководству программиста).

### §3.8.7.2. Запись состояния блока

Событие записи состояния возникает у блока в процессе работы сервисной функции `rdsSaveSystemState` (см. §A.5.7.4 приложения к руководству программиста [2]), вызвав которую модель любого блока может сохранить состояние блоков всей схемы или какой-либо ее отдельной подсистемы, чтобы позже его можно было восстановить вызовом `rdsLoadSystemState` (см. §A.5.7.2 там же). В реакции на это событие модель должна записать все данные, которые она позже будет загружать в реакции на событие загрузки состояния (см. §3.8.7.1 выше). В интерфейсе пользователя РДС не предусмотрено сохранения и восстановления состояния схемы – этим должны заниматься модели блоков. Сохранение и загрузка состояний блоков и подсистем подробно описаны в §2.14.3 руководства программиста [1]. Значения статических переменных блока сохраняются и восстанавливаются автоматически. Модель должна сохранять только значения динамических переменных, за которые она отвечает, и значения своих внутренних параметров, если таковые имеются.

Текст реакции на событие записи состояния вводится на вкладке “события” левой панели редактора модели: раздел “загрузка и запись состояния”, подраздел “запись состояния блока”. В классе блока для нее создается функция с именем `rdsbcppSaveState`:

```
// Запись параметров состояния блока в двоичном виде
void rdsbcppBlockClass::rdsbcppSaveState(void)
{
```

*Пользовательский текст реакции*

```
}
```

Внутри этой функции данные должны записываться вызовами функции `rdsWriteBlockData`, позже они будут загружены вызовами `rdsReadBlockData` в том же порядке.

При написании моделей блоков без использования модуля автокомпиляции событию записи состояния соответствует константа РДС `RDS_BFM_SAVESTATE` (см. §A.2.4.13 приложения к руководству программиста).

### §3.8.8. Реакции окна подсистемы

Рассматриваются реакции на события, связанные с окнами подсистем.

#### §3.8.8.1. Действия с окном подсистемы

Это событие возникает в подсистеме и во всех блоках, находящихся *непосредственно* внутри нее, при открытии и закрытии окна этой подсистемы. Блоки, находящиеся внутри подсистем этой подсистемы (то есть блоки, не имеющие отношения к открытому или закрытому окну), информации об этом событии не получают. Реакцию на него можно использовать, например, для того, чтобы, зная дескриптор только что открытого окна подсистемы, выполнить с ним какие-либо нестандартные действия. Текст этой реакции вводится на вкладке “события” левой панели редактора модели: раздел “реакции подсистемы”, подраздел “действия с окном подсистемы”.

В классе блока для события действий с окном подсистемы создается функция с именем `rdsbcppSysWinOperation` следующего вида:

```
// Действия с окном подсистемы
void rdsbcppBlockClass::rdsbcppSysWinOperation(
 RDS_PWINOPERATIONDATA OperationData)
{

Пользовательский текст реакции

}
```

Параметр функции `OperationData` – это указатель на структуру описания события `RDS_WINOPERATIONDATA`:

```
typedef struct {
 int Operation; // Операция с окном
 HWND Handle; // Дескриптор окна
 BOOL EditMode; // Включен режим редактирования
 BOOL Running; // Идет расчет
 BOOL OwnWindow; // Операция с окном данной подсистемы
} RDS_WINOPERATIONDATA;
typedef RDS_WINOPERATIONDATA *RDS_PWINOPERATIONDATA;
```

Поля структуры имеют следующий смысл:

- `Operation` – константа `RDS_SWO_OPEN`, если окно открылось, и `RDS_SWO_CLOSE`, если оно закрылось.
- `Handle` – дескриптор окна подсистемы (его можно использовать в вызовах Windows API).
- `EditMode` – `TRUE`, если РДС находится в режиме редактирования, и `FALSE` в противном случае (для режимов моделирования и расчета).
- `Running` – `TRUE`, если РДС находится в режиме расчета (то есть работает поток расчета), и `FALSE` в противном случае (для режимов редактирования и моделирования).
- `OwnWindow` – `TRUE`, если было открыто или закрыто окно подсистемы, модель которой вызывается (автокомпилируемые модели подсистем используются крайне редко), и `FALSE`, если открыто или закрыто окно подсистемы, родительской по отношению к вызываемому блоку.

При написании моделей блоков без использования модуля автокомпиляции действиям с окном подсистемы соответствует константа РДС `RDS_BFM_WINDOWOPERATION` (см. §A.2.6.20 приложения к руководству программиста [2]).

### §3.8.8.2. Нажатие кнопки мыши (в окне подсистемы)

Это событие возникает в режимах моделирования и расчета при нажатии пользователем любой кнопки мыши на рабочем поле окна подсистемы, если на это нажатие не среагировал ни один блок в этой подсистеме, и если в параметрах этой подсистемы разрешена реакция ее окна на мышшь (см. §2.11.4 части I). Реакция на него вызывается только в моделях подсистем. Поскольку автокомпилируемые модели чаще всего присоединяют к простым блокам, эта реакция в моделях используется крайне редко.

Реакция на нажатие кнопки мыши в подсистеме вводится на вкладке “события” левой панели редактора модели: раздел “реакции подсистемы”, подраздел “нажатие кнопки мыши” (не следует путать эту реакцию с одноименной реакцией простого блока). В классе блока для нее создается функция с именем `rdsbcppSysWinMouseDown` следующего вида:

```
// Нажатие кнопки мыши
void rdsbcppBlockClass::rdsbcppSysWinMouseDown(
 RDS_PMOUSEDATA MouseData)
{
 Пользовательский текст реакции
}
```

Параметр `MouseData` этой функции в точности соответствуют одноименному параметру функции реакции простого блока на нажатие кнопки мыши (см. §3.8.4.1 на стр. 295).

При написании моделей блоков без использования модуля автокомпиляции нажатию кнопки мыши в окне подсистемы соответствует константа РДС `RDS_BFM_WINDOWMOUSEDOWN` (см. §A.2.6.17 приложения к руководству программиста [2]).

### §3.8.8.3. Отпускание кнопки мыши (в окне подсистемы)

Это событие возникает в режимах моделирования и расчета при отпускании пользователем ранее нажатой кнопки мыши на рабочем поле окна подсистемы, если на это действие не среагировал ни один блок в этой подсистеме, и если в параметрах этой подсистемы разрешена реакция ее окна на мышшь (см. §2.11.4 части I). Реакция на него вызывается только в моделях подсистем, и, поскольку автокомпилируемые модели чаще всего присоединяют к простым блокам, эта реакция используется крайне редко. Текст реакции вводится на вкладке “события” левой панели редактора модели: раздел “реакции подсистемы”, подраздел “отпускание кнопки мыши” (не следует путать эту реакцию с одноименной реакцией простого блока). В классе блока для нее создается функция с именем `rdsbcppSysWinMouseUp` следующего вида:

```
// Отпускание кнопки мыши
void rdsbcppBlockClass::rdsbcppSysWinMouseUp(
 RDS_PMOUSEDATA MouseData)
{
 Пользовательский текст реакции
}
```

Параметр `MouseData` этой функции в точности соответствуют одноименному параметру функции реакции простого блока на отпускание кнопки мыши (см. §3.8.4.2 на стр. 297).

При написании моделей блоков без использования модуля автокомпиляции отпусканию кнопки мыши в окне подсистемы соответствует константа РДС `RDS_BFM_WINDOWMOUSEUP` (см. §A.2.6.19 приложения к руководству программиста [2]).

#### §3.8.8.4. Двойной щелчок (в окне подсистемы)

Это событие возникает в режимах моделирования и расчета при двойном щелчке левой кнопкой мыши на рабочем поле окна подсистемы, если на этот двойной щелчок не среагировал ни один блок в этой подсистеме, и если в параметрах этой подсистемы разрешена реакция ее окна на мышь (см. §2.11.4 части I). Реакция на это событие вызывается только в моделях подсистем, она вводится на вкладке “события” левой панели редактора модели: раздел “реакции подсистемы”, подраздел “двойной щелчок мыши”. В классе блока для нее создается функция с именем `rdsbcppSysWinMouseDblClick` следующего вида:

```
// Двойной щелчок мыши
void rdsbcppBlockClass::rdsbcppSysWinMouseDblClick(
 RDS_PMOUSEDATA MouseData)
{
 Пользовательский текст реакции
}
```

Параметр `MouseData` этой функции в точности соответствует одноименному параметру функции реакции простого блока на двойной щелчок (см. §3.8.4.3 на стр. 297).

При написании моделей блоков без использования модуля автокомпиляции двойному щелчку в окне подсистемы соответствует константа РДС `RDS_BFM_WINDOWMOUSEDCLICK` (см. §A.2.6.16 приложения к руководству программиста [2]).

#### §3.8.8.5. Перемещение курсора (в окне подсистемы)

Это событие возникает в режимах моделирования и расчета при перемещении курсора мыши над рабочим полем окна подсистемы, если на него не среагировал ни один блок в этой подсистеме, и если в параметрах этой подсистемы разрешена реакция ее окна на мышь (см. §2.11.4 части I). По умолчанию событие возникает, только если в момент перемещения курсора какая-либо из кнопок мыши нажата, но в окне параметров подсистемы можно разрешить вызов этой реакции вне зависимости от нажатия кнопок. Реакция на это событие вызывается только в моделях подсистем, она вводится на вкладке “события” левой панели редактора модели: раздел “реакции подсистемы”, подраздел “перемещение мыши”. В классе блока для нее создается функция с именем `rdsbcppSysWinMouseMove` следующего вида:

```
// Перемещение мыши
void rdsbcppBlockClass::rdsbcppSysWinMouseMove(
 RDS_PMOUSEDATA MouseData)
{
 Пользовательский текст реакции
}
```

Параметр `MouseData` этой функции в точности соответствуют одноименному параметру функции реакции простого блока на перемещение курсора мыши (см. §3.8.4.4 на стр. 298).

При написании моделей блоков без использования модуля автокомпиляции перемещению курсора над окном подсистемы соответствует константа РДС `RDS_BFM_WINDOWMOUSEMOVE` (см. §A.2.6.18 приложения к руководству программиста [2]).

#### §3.8.8.6. Нажатие клавиши (в окне подсистемы)

Это событие возникает в подсистемах (в блоках других типов оно не возникает) в режимах моделирования и расчета при нажатии пользователем какой-либо клавиши на



клавиатуре, если окно этой подсистемы с имеет фокус (это самое верхнее окно, и РДС – активное приложение), на вкладке “DLL” окна параметров этой подсистемы включен флажок “окно реагирует на клавиатуру” (см. §2.11.4 части I), и ни один из блоков внутри подсистемы не среагировал на нажатие клавиши.

Реакция на нажатие клавиши в окне вводится на вкладке “события” левой панели редактора модели: раздел “реакции подсистемы”, подраздел “нажатие клавиши” (не следует путать его с одноименным подразделом в разделе “мышь и клавиатура”, задающим реакцию блока, а не окна). Она используется крайне редко, поскольку автокомпилируемые модели чаще всего не подключают к подсистемам. В классе блока для этой реакции создается функция с именем `rdsbcppSysWinKeyDown` следующего вида:

```
// Нажатие клавиши
void rdsbcppBlockClass::rdsbcppSysWinKeyDown(RDS_PKEYDATA KeyData)
{
```

*Пользовательский текст реакции*

```
}
```

Параметр `KeyData` этой функции в точности соответствуют одноименному параметру функции реакции простого блока на нажатие клавиши (см. §3.8.4.5 на стр. 298).

При написании моделей блоков без использования модуля автокомпиляции нажатую клавишу в окне соответствует константа РДС `RDS_BFM_WINDOWKEYDOWN` (см. §A.2.6.14 приложения к руководству программиста [2]).

#### §3.8.8.7. Отпускание клавиши (в окне подсистемы)

Это событие возникает только в подсистемах в режимах моделирования и расчета при отпускании пользователем какой-либо ранее нажатой клавиши на клавиатуре, если окно этой подсистемы с имеет фокус (это самое верхнее окно, и РДС – активное приложение), на вкладке “DLL” окна параметров этой подсистемы включен флажок “окно реагирует на клавиатуру” (см. §2.11.4 части I), и ни один из блоков внутри подсистемы не среагировал на отпускание клавиши.

Реакция на это событие вводится на вкладке “события” левой панели редактора модели: раздел “реакции подсистемы”, подраздел “отпускание клавиши” (не следует путать его с одноименным подразделом в разделе “мышь и клавиатура”, задающим реакцию блока, а не окна). В классе блока для этой реакции создается функция с именем `rdsbcppSysWinKeyUp` следующего вида:

```
// Отпускание клавиши
void rdsbcppBlockClass::rdsbcppSysWinKeyUp(RDS_PKEYDATA KeyData)
{
```

*Пользовательский текст реакции*

```
}
```

Параметр `KeyData` этой функции в точности соответствуют одноименному параметру функции реакции простого блока на отпускание клавиши (см. §3.8.4.6 на стр. 300).

При написании моделей блоков без использования модуля автокомпиляции отпусканию клавиши в окне соответствует константа РДС `RDS_BFM_WINDOWKEYUP` (см. §A.2.6.15 приложения к руководству программиста [2]).

### §3.8.9. Внешний вид блока

Рассматриваются реакции на события, связанные с изменением размеров и положения блока, а также с программным рисованием его изображения в подсистеме.

#### §3.8.9.1. Размер блока изменен

Это событие возникает в блоке, который программно рисует свое изображение (см. §3.7.5 на стр. 173), а также у блоков, изображаемых прямоугольником с текстом, после того, как пользователь тем или иным способом изменил размер блока. В реакции на него можно определить и запомнить размеры каких-либо внутренних элементов изображения блока, вычисляемые относительно его текущего размера, или скорректировать изменения размера, сделанные пользователем, если они не подходят для блока. Например, можно не разрешать пользователю сделать квадратный блок прямоугольным, принудительно устанавливая одинаковую ширину и высоту. От способа изменения размера возникновение события не зависит: оно возникает и после перетаскивания маркеров выделения блока (см. §2.6 части I), и после ввода точных значений размера с клавиатуры при нажатии кнопки “размер для функции DLL” в окне параметров блока (см. §2.9.1 там же).

Текст реакции на это событие вводится на вкладке “события” левой панели редактора модели: раздел “внешний вид блока”, подраздел “размер блока изменен”. В классе блока для нее создается функция с именем `rdsbcppBlockResized` следующего вида:

```
// Размер блока был изменен
void rdsbcppBlockClass::rdsbcppBlockResized(
 RDS_RESIZEDATA ResizeData, int &Result)
{
```

*Пользовательский текст реакции*

```
}
```

У функции два параметра. Параметр `ResizeData` – это указатель на структуру описания события `RDS_RESIZEDATA`:

```
typedef struct {
 BOOL HorzResize; // Изменение горизонтального размера
 BOOL VertResize; // Изменение вертикального размера
 int newWidth, newHeight; // Новые значения ширины и высоты
 int GridDx, GridDy; // Шаги сетки редактора
 BOOL SnapToGrid; // Привязка к сетке
} RDS_RESIZEDATA;
typedef RDS_RESIZEDATA *RDS_RESIZEDATA;
```

Поля структуры имеют следующий смысл:

- `HorzResize` и `VertResize` – логические значения, указывающие на то, каким именно образом пользователь изменил размеры блока:

| <i><b>HorzResize</b></i> | <i><b>VertResize</b></i> | <i><b>Действия пользователя</b></i>                                                                      |
|--------------------------|--------------------------|----------------------------------------------------------------------------------------------------------|
| FALSE                    | FALSE                    | Новые размеры блока заданы числами в окне параметров.                                                    |
| FALSE                    | TRUE                     | Пользователь перетащил верхнюю или нижнюю метку масштабирования вверх или вниз (изменена только высота). |
| TRUE                     | FALSE                    | Пользователь перетащил левую или правую метку масштабирования влево или вправо (изменена только ширина). |



| <i>HorzResize</i> | <i>VertResize</i> | <i>Действия пользователя</i>                                                                             |
|-------------------|-------------------|----------------------------------------------------------------------------------------------------------|
| TRUE              | TRUE              | Пользователь перетащил одну из угловых меток масштабирования (одновременно изменены и ширина, и высота). |

- `newWidth` и `newHeight` – новые значения ширины и высоты блока соответственно для масштаба 100%. Модель может вмешаться в изменение размеров, заменив значения в этих полях. Например, можно записать вместо `newHeight` старое значение высоты блока, тогда изменение высоты, сделанное пользователем, будет проигнорировано. Внутри этой реакции прежние значения ширины и высоты блока можно получить при помощи сервисной функции `rdsGetBlockDimensionsEx` (см. §A.5.6.18 приложения к руководству программиста [2]).
- `GridDx` и `GridDy` – горизонтальный и вертикальный шаги сетки окна родительской подсистемы блока соответственно.
- `SnapToGrid` – TRUE, если в окне родительской подсистемы блока включена привязка к сетке, и FALSE в противном случае.

Второй параметр функции реакции – это ссылка на целую переменную `Result`. Через этот параметр модель может сообщить РДС, следует ли принимать сделанное пользователем изменение размера. Для этого параметру `Result` нужно присвоить одну из двух стандартных констант:

- `RDS_BFR_DONE` – изменение размеров разрешено: блок получит размеры, записанные в полях `newWidth` и `newHeight` переданной через параметр `ResizeData` структуры (при этом в реакции на событие можно изменить эти поля, чтобы скорректировать размеры блока).
- `RDS_BFR_STOP` – изменение размеров отменено, ширина и высота блока останутся неизменными.

При написании моделей блоков без использования модуля автокомпиляции событию изменения размера блока соответствует константа РДС `RDS_BFM_RESIZE` (см. §A.2.7.6 приложения к руководству программиста).

### §3.8.9.2. Проверка изменения размера блока

Это событие возникает в блоке, который программно рисует свое изображение (см. §3.7.5 на стр. 173), а также у блоков, изображаемых прямоугольником с текстом, в процессе изменения размеров блока пользователем, когда последний перетаскивает мышью маркеры выделения блока. Каждое движение курсора мыши в процессе этого перетаскивания порождает одно такое событие. Реакция на него чаще всего используется для вмешательства модели в изменение размера: если модель в реакции на это событие скорректирует поля `newWidth` и `newHeight` структуры `RDS_RESIZEDATA`, которая передается в реакцию, это немедленно отразится на размере прямоугольника, который пользователь видит при перетаскивании маркеров выделения. Пример модели, реагирующей на это событие, приведен на стр. 192.

Текст реакции на событие проверки допустимости изменения размера вводится на вкладке “события” левой панели редактора модели: раздел “внешний вид блока”, подраздел “проверка изменения размера”. В классе блока для нее создается функция с именем `rdsbcppResizing` следующего вида:

```
// Проверка допустимости изменения размера блока
void rdsbcppBlockClass::rdsbcppResizing(
 RDS_PRESIZEData ResizeData, int &Result)
{
```

*Пользовательский текст реакции*

}

У этой функции два параметра, в точности совпадающих с параметрами функции реакции на событие окончательного изменения размера (см. §3.8.9.1 на стр. 312), и имеющих тот же смысл. При написании моделей блоков без использования модуля автокомпиляции событию проверки изменения размера блока соответствует константа РДС RDS\_BFM\_RESIZING (см. §A.2.7.7 приложения к руководству программиста [2]).

### §3.8.9.3. Блок перемещен

Это событие возникает после любого изменения положения блока на рабочем поле подсистемы, то есть после его перетаскивания пользователем (см. §2.6 части I), после задания его координат в окне параметров блока (см. §2.9.1 там же), после вызова одной из сервисных функций для перемещения блока (см., например, §A.5.6.40 приложения к руководству программиста [2]) и т.п. Реакция на него может использоваться для синхронного перемещения группы блоков при перемещении одного из них или для сообщения куда-либо новых координат блока. Отменить перемещение блока, реагируя на это событие, нельзя.

Текст реакции на событие перемещения блока вводится на вкладке “события” левой панели редактора модели: раздел “внешний вид блока”, подраздел “блок перемещен”. В классе блока для нее создается функция с именем `rdsbcppBlockMoved` следующего вида:

```
// Блок перемещен
void rdsbcppBlockClass::rdsbcppBlockMoved(
 RDS_PMOVEDATA MoveData)
{
```

*Пользовательский текст реакции*

}

Параметр функции `MoveData` – это указатель на структуру описания события RDS\_MOVEDATA:

```
typedef struct {
 int MoveReason; // Причина перемещения
 int OldX, OldY; // Старые координаты точки привязки
 int NewX, NewY; // Новые координаты точки привязки
} RDS_MOVEDATA;
typedef RDS_MOVEDATA *RDS_PMOVEDATA;
```

Поля структуры имеют следующий смысл:

- `MoveReason` – одна из стандартных констант, указывающих на причину перемещения, то есть изменения координат, блока:
  - ♦ `RDS_MR_SET` – координаты блока установлены непосредственно, то есть введены пользователем с клавиатуры в окне параметров или установлены одной из сервисных функций;
  - ♦ `RDS_MR_DRAG` – пользователь перетаскивал блок мышью;
  - ♦ `RDS_MR_KEYBOARD` – пользователь переместил выделенный блок нажатиями курсорных клавиш;
  - ♦ `RDS_MR_UNDOREDO` – положение блока изменилось из-за отмены пользователем сделанного перемещения или из-за повтора ранее отмененного перемещения.
- `OldX` и `OldY` – горизонтальная и вертикальная координаты точки привязки блока до перемещения, значения даны для масштаба подсистемы 100% (для блоков с векторной

картинкой точка привязки – это положение начала координат этой картинке, для всех остальных – левый верхний угол прямоугольной области, занимаемой блоком).

- NewX и NewY – координаты точки привязки блока после перемещения для масштаба 100%.

При написании моделей блоков без использования модуля автокомпиляции перемещению блока соответствует константа РДС RDS\_BFM\_MOVED (см. §A.2.7.4 приложения к руководству программиста).

#### §3.8.9.4. Рисование блока

Это событие возникает у всех блоков, для которых в окне параметров установлено программное рисование, при перерисовке окна подсистемы, в котором эти блоки находятся. В реакции на него блок должен нарисовать свой внешний вид при помощи функций Windows API или специальных функций рисования РДС (см. §A.5.18 приложения к руководству программиста [2]). Примеры моделей, реагирующих на это событие, приведены в §3.7.5 (стр. 173).

Текст реакции на событие программного рисования блока вводится на вкладке “события” левой панели редактора модели: раздел “внешний вид блока”, подраздел “рисование блока”. В классе блока для нее создается функция с именем rdsbcppDraw следующего вида:

```
// Рисование внешнего вида блока в схеме
void rdsbcppBlockClass::rdsbcppDraw(
 RDS_PDRAWDATA DrawData)
{
```

*Пользовательский текст реакции*

```
}
```

Параметр функции DrawData – это указатель на структуру описания события RDS\_DRAWDATA:

```
typedef struct {
 HDC dc; // Контекст устройства рисования
 BOOL CalcMode; // Текущий режим РДС
 int BlockX, // Координаты точки привязки
 BlockY; // блока в окне
 double DoubleZoom; // Масштаб окна (в долях единицы)
 BOOL RectValid; // Флаг изменения размеров блока
 int Left,Top; // Верхний левый угол блока
 int Width,Height; // Ширина и высота блока
 RECT *VisibleRect; // Видимая в окне часть рабочего поля
 BOOL FullDraw; // Необходимо перерисовать весь блок
} RDS_DRAWDATA;
typedef RDS_DRAWDATA *RDS_PDRAWDATA;
```

Поля структуры подробно описаны на стр. 175.

При написании моделей блоков без использования модуля автокомпиляции рисованию блока соответствует константа РДС RDS\_BFM\_DRAW (см. §A.2.6.3 приложения к руководству программиста).

#### §3.8.9.5. Дополнительное рисование блока

Это событие возникает у всех блоков при перерисовке окна подсистемы, в котором эти блоки находятся. В отличие от реакции на обычное рисование блока (см. §3.8.9.4 выше), возникновение этого события не зависит от способа рисования, выбранного в окне

параметров: оно возникнет и для блоков, рисующих свое изображение программно, и для блоков, внешний вид которых задается векторной картинкой, и для простых прямоугольников с текстом. Обычно в реакции на это событие поверх изображения блока выводится различная дополнительная информация: например, иконки, сигнализирующие о каких-либо ошибках в блоке. Для рисования в этой реакции используются функции Windows API или специальные функции рисования РДС (см. §A.5.18 приложения к руководству программиста [2]). Пример модели с реакцией на это событие приведен в §3.7.10 (стр. 223).

Текст реакции на событие дополнительного рисования вводится на вкладке “события” левой панели редактора модели: раздел “внешний вид блока”, подраздел “дополнительное рисование блока”. В классе блока для нее создается функция с именем `rdsbcppDrawAdditional` следующего вида:

```
// Дополнительное рисование
void rdsbcppBlockClass::rdsbcppDrawAdditional(
 RDS_PDRAWDATA DrawData)
{
 Пользовательский текст реакции
}
```

Параметр функции `DrawData` – это указатель на структуру описания события `RDS_DRAWDATA`, поля которой подробно описаны на стр. 175.

Следует учитывать, что при включенном избирательном обновлении окон подсистем (см. §2.11.4 части I и §2.10.2 руководства программиста) это событие может не возникать для блоков, изображаемых прямоугольником с текстом, поскольку такие блоки в этом режиме автоматически не перерисовываются. Если изображения, которые выводятся в реакции на это событие, могут изменяться в режиме расчета, при каждом таком изменении следует вызывать функцию `rdsForceBlockRedraw` (см. §A.5.6.15 приложения к руководству программиста) для правильной работы избирательного обновления – в этом случае блок вместе с дополнительными изображениями будет принудительно перерисован при следующем обновлении окна по таймеру.

При написании моделей блоков без использования модуля автокомпиляции дополнительному рисованию блока соответствует константа РДС `RDS_BFM_DRAWADDITIONAL` (см. §A.2.6.4 приложения к руководству программиста).

### §3.8.10. Обмен данными по сети

Рассматриваются реакции на события, связанные с встроенными в РДС процедурами обмена данными по сети.

#### §3.8.10.1. Получены данные по сети

Это событие возникает при обмене данными по сети в блоке, для которого поступили данные с сервера. Стандартный модуль автокомпиляции не автоматизирует передачу данных между блоками по сети, желающие реализовать ее могут изучить §2.15 руководства программиста [1].

Текст реакции на получение данных вводится на вкладке “события” левой панели редактора модели: раздел “сеть”, подраздел “получены данные”. В классе блока для нее создается функция с именем `rdsbcppNetReceive` следующего вида:

```
// Получены данные по сети
void rdsbcppBlockClass::rdsbcppNetReceive(
 RDS_PNETRECEIVEDDATA NetData)
{
}
```

}

Параметр функции NetData – это указатель на структуру RDS\_NETRECEIVEDDATA, описывающую поступившие данные:

```
typedef struct {
 int ConnId; // Идентификатор соединения
 LPSTR Host; // URL или IP-адрес сервера
 int Port; // Порт
 LPSTR Channel; // Имя канала
 int Id; // Принятое целое число
 LPSTR Str; // Принятая строка
 LPVOID Buffer; // Принятый буфер или NULL
 DWORD BufferSize; // Размер принятого буфера
 RDS_NETSTATION SenderStation; // Идентификатор
 // машины-отправителя
 RDS_NETBLOCK SenderBlock; // Идентификатор блока
 // на машине-отправителе
} RDS_NETRECEIVEDDATA;
typedef RDS_NETRECEIVEDDATA *RDS_PNETRECEIVEDDATA;
```

Поля структуры имеют следующий смысл:

- ConnId – уникальный идентификатор сетевого соединения, возвращенный сервисной функцией при его создании.
- Host – имя (URL) или IP-адрес сервера, с которым установлено соединение.
- Port – номер сетевого порта, через который идет обмен данными.
- Channel – имя канала передачи данных, из которого поступили данные.
- Id – целый идентификатор полученных данных (произвольное целое число, указанное передавшим данные блоком в вызове одной из функций передачи).
- Str – указатель на принятую по сети строку во внутренней памяти РДС.
- Buffer – указатель на начало принятых двоичных данных во внутренней памяти РДС.
- BufferSize – размер (в байтах) принятых двоичных данных, на которые указывает поле Buffer.
- SenderStation – идентификатор машины-передатчика, то есть машины, на которой запущена копия РДС, в которую загружена схема, блок которой отправил эти данные.
- SenderBlock – идентификатор блока-отправителя на машине-передатчике.

При написании моделей блоков без использования модуля автокомпиляции получению данных по сети соответствует константа РДС RDS\_BFM\_NETDATA RECEIVED (см. §A.2.8.3 приложения к руководству программиста [2]).

### §3.8.10.2. Сетевое соединение установлено

Это событие возникает в блоке, запросившем сетевое соединение с каким-либо каналом передачи данных сервера, после успешной установки этого соединения. Как правило, модели блоков не отправляют данные другим блокам до наступления этого события. Стандартный модуль автокомпиляции не автоматизирует передачу данных между блоками по сети, желающие реализовать ее могут изучить §2.15 руководства программиста [1].

Текст реакции на установку соединения вводится на вкладке “события” левой панели редактора модели: раздел “сеть”, подраздел “соединение установлено”. В классе блока для нее создается функция с именем rdsbcppNetConnect следующего вида:

```
// Установлено соединение с сервером
void rdsbcppBlockClass::rdsbcppNetConnect (
 RDS_PNETCONNDATA ConnData)
{

Пользовательский текст реакции

}
```

Параметр функции ConnData – это указатель на структуру RDS\_NETCONNDATA, описывающую параметры установленного соединения:

```
typedef struct {
 int ConnId; // Идентификатор соединения
 LPSTR Host; // URL или IP-адрес сервера
 int Port; // Порт
 LPSTR Channel; // Имя канала
 BOOL ByServer; // Соединение разорвано сервером
} RDS_NETCONNDATA;
typedef RDS_NETCONNDATA *RDS_PNETCONNDATA;
```

Поля структуры имеют следующий смысл:

- ConnId – уникальный идентификатор сетевого соединения, возвращенный сервисной функцией при его создании.
- Host – имя (URL) или IP-адрес сервера, с которым установлено соединение.
- Port – номер сетевого порта, через который идет обмен данными.
- Channel – имя канала передачи данных, с которым установлено соединение.
- ByServer (в реакции на установку соединения не используется) – TRUE, если соединение разорвано сервером.

При написании моделей блоков без использования модуля автокомпиляции успешной установке сетевого соединения соответствует константа РДС RDS\_BFM\_NETCONNECT (см. §A.2.8.1 приложения к руководству программиста [2]).

### §3.8.10.3. Сетевое соединение разорвано

Это событие возникает в блоке, ранее запросившем сетевое соединение с каким-либо каналом передачи данных сервера, при разрыве этого соединения. Если соединение разорвано сервером, РДС будет пытаться самостоятельно восстановить его без участия модели. Стандартный модуль автокомпиляции не автоматизирует передачу данных между блоками по сети, желающие реализовать ее могут изучить §2.15 руководства программиста [1].

Текст реакции на разрыв соединения вводится на вкладке “события” левой панели редактора модели: раздел “сеть”, подраздел “соединение разорвано”. В классе блока для нее создается функция с именем rdsbcppNetDisconnect следующего вида:

```
// Разорвано соединение с сервером
void rdsbcppBlockClass::rdsbcppNetDisconnect (
 RDS_PNETCONNDATA ConnData)
{

Пользовательский текст реакции

}
```

Параметр функции ConnData – это указатель на структуру RDS\_NETCONNDATA, описывающую параметры установленного соединения (см. §3.8.10.2 на стр. 317). Поле

ByServer этой структуры будет содержать TRUE, если соединение разорвано сервером, и FALSE, если соединение разорвано моделью самого блока.

При написании моделей блоков без использования модуля автокомпиляции разрыву сетевого соединения соответствует константа РДС RDS\_BFM\_NETDISCONNECT (см. §A.2.8.4 приложения к руководству программиста [2]).

#### §3.8.10.4. Данные приняты сервером

Это событие возникает при обмене данными по сети в блоке, передавшем данные на сервер, после получения сервером этих данных, если блок, передавая данные, запросил такое подтверждение получения. Событие возникает при получении данных именно сервером, а не блоком-получателем данных – серверу еще только предстоит отправить данные получателю. Стандартный модуль автокомпиляции не автоматизирует передачу данных между блоками по сети, желающие реализовать ее могут изучить §2.15 руководства программиста [1].

Текст реакции на подтверждение приема данных вводится на вкладке “события” левой панели редактора модели: раздел “сеть”, подраздел “данные приняты сервером”. В классе блока для нее создается функция с именем rdsbcppNetAccepted следующего вида:

```
// Сервер подтверждает получение данных
void rdsbcppBlockClass::rdsbcppNetAccepted(
 RDS_PNETACCEPTDATA AccData)
{

Пользовательский текст реакции

}
```

Параметр функции AccData – это указатель на структуру RDS\_NETACCEPTDATA, описывающую переданные данные, прием которых подтверждается:

```
typedef struct {
 int ConnId; // Идентификатор соединения
 LPSTR Host; // URL или IP-адрес сервера
 int Port; // Порт
 LPSTR Channel; // Имя канала
 int Id; // Принятое целое число
} RDS_NETACCEPTDATA;
typedef RDS_NETACCEPTDATA *RDS_PNETACCEPTDATA;
```

Поля структуры имеют следующий смысл:

- ConnId – уникальный идентификатор сетевого соединения, возвращенный сервисной функцией при его создании.
- Host – имя (URL) или IP-адрес сервера, с которым установлено соединение.
- Port – номер сетевого порта, через который идет обмен данными.
- Channel – имя канала передачи данных, в который были переданы данные.
- Id – целый идентификатор переданных данных (произвольное целое число, указанное передавшим данные блоком в вызове одной из функций передачи).

При написании моделей блоков без использования модуля автокомпиляции подтверждению получения данных сервером соответствует константа РДС RDS\_BFM\_NETDATAACCEPTED (см. §A.2.8.2 приложения к руководству программиста [2]).

#### §3.8.10.5. Ошибка сети

Это событие возникает в блоке, участвующем в обмене данными по сети, при возникновении различных ошибок в сетевом соединении. Большую часть таких ошибок РДС



обрабатывает самостоятельно, поэтому, как правило, в моделях блоков реакция на это событие не требуется. Стандартный модуль автокомпиляции не автоматизирует передачу данных между блоками по сети, желающие реализовать ее могут изучить §2.15 руководства программиста [1].

Текст реакции на ошибку вводится на вкладке “события” левой панели редактора модели: раздел “сеть”, подраздел “ошибка сети”. В классе блока для нее создается функция с именем `rdsbcppNetError` следующего вида:

```
// Ошибка сети
void rdsbcppBlockClass::rdsbcppNetError(
 RDS_PNETERRORDATA ErrorData)
{
```

*Пользовательский текст реакции*

```
}
```

Параметр функции `ErrorData` – это указатель на структуру `RDS_NETERRORDATA`, описывающую соединение и возникшую ошибку:

```
typedef struct {
 int ConnId; // Идентификатор соединения
 LPSTR Host; // URL или IP-адрес сервера
 int Port; // Порт
 LPSTR Channel; // Имя канала
 int ErrorCode; // Код ошибки (RDS_NETERR_*)
 RDS_NETSTATION Station; // Идентификатор машины
 RDS_NETBLOCK Block; // Идентификатор блока
} RDS_NETERRORDATA;
typedef RDS_NETERRORDATA *RDS_PNETERRORDATA;
```

Поля структуры имеют следующий смысл:

- `ConnId` – уникальный идентификатор сетевого соединения, возвращенный сервисной функцией при его создании.
- `Host` – имя (URL) или IP-адрес сервера, с которым установлено соединение.
- `Port` – номер сетевого порта, через который идет обмен данными.
- `Channel` – имя канала передачи данных.
- `ErrorCode` – один из поддерживаемых кодов ошибки:
  - ◆ `RDS_NETERR_NOBLOCK` – блока, которому переданы данные, не существует;
  - ◆ `RDS_NETERR_SEND` – ошибка при передаче данных;
  - ◆ `RDS_NETERR_RECEIVE` – ошибка при приеме данных;
  - ◆ `RDS_NETERR_DISCONNECT` – ошибка при попытке разорвать связь;
  - ◆ `RDS_NETERR_ACCEPT` – ошибка при установке соединения сервера с клиентом;
  - ◆ `RDS_NETERR_CLIENTCONN` – для блоков схемы, работающей на сервере: ошибка при установке соединения с клиентом;
  - ◆ `RDS_NETERR_GENERAL` – неизвестная ошибка.
- `Station` – идентификатор машины, если он нужен для описания ошибки (при коде ошибки `RDS_NETERR_NOBLOCK`).
- `Block` – идентификатор блока на машине `Station`.

При написании моделей блоков без использования модуля автокомпиляции ошибке сети соответствует константа РДС `RDS_BFM_NETERROR` (см. §A.2.8.5 приложения к руководству программиста [2]).



### §3.8.11. Прочие реакции

Рассматриваются реакции модели на различные события, которые сложно объединить по смыслу в какую-либо группу.

#### §3.8.11.1. Переименование блока

Это событие возникает при изменении имени блока пользователем через окно его параметров, а также при программном изменении имени блока при помощи функции `rdsRenameBlock` (см. §A.5.6.42 приложения к руководству программиста [2]). Реакцию на него можно использовать для изменения имен каких-либо программно созданных объектов (например, динамических переменных), в которых имя блока используется для обеспечения уникальности имени объекта. При этом следует учитывать, что имя блока уникально только в пределах одной подсистемы, в разных подсистемах могут находиться блоки с одинаковыми именами.

Текст реакции на событие изменения имени блока вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “переименование блока”. В классе блока для нее создается функция с именем `rdsbcppBlockRename` следующего вида:

```
// Переименование блока
void rdsbcppBlockClass::rdsbcppBlockRename(LPSTR OldBlockName)
{
 Пользовательский текст реакции
}
```

В параметре `OldBlockName` передается указатель на строку (`char*`) с именем блока до переименования.

При написании моделей блоков без использования модуля автокомпиляции этому событию соответствует константа РДС `RDS_BFM_RENAME` (см. §A.2.7.5 приложения к руководству программиста).

#### §3.8.11.2. Вызов настройки

Это событие возникает при вызове пользователем окна настроек блока. В зависимости от того, добавлены ли в редакторе в модель блока настроечные параметры (см. §3.6.6 на стр. 60 и §3.7.6 на стр. 194), реакция на это событие вызывается по-разному.

- Если в редакторе модели не были созданы настроечные параметры и окно для их ввода, эта реакция будет вызвана в момент выбора пользователем пункта меню, открывающего окно настройки выделенного в подсистеме блока (обычно этот пункт называется “настройка”, но может быть переименован пользователем в окне параметров блока). Всю работу по созданию окна со списком настроек, обеспечению их редактирования пользователем и записи измененных параметров в блок должен при этом вручную выполнять разработчик модели. Многочисленные примеры моделей, в которых настройка параметров осуществляется таким образом, приведены в §2.7 руководства программиста [1].
- Если в редактор модели было добавлено окно настройки параметров, то при выборе соответствующего пункта меню оно откроется автоматически, а эта реакция будет вызвана только после его закрытия кнопкой “ОК”, то есть после того, как пользователь примет внесенные изменения. При закрытии окна кнопкой “отмена” (то есть при отмене изменений) реакция вызвана не будет. В таком варианте реакция на вызов настройки чаще всего нужна для того, чтобы модель как-то отреагировала на измененные пользователем параметры. Пример такого ее использования приведен в §3.7.7 (стр. 203).

Текст реакции на вызов (или изменение) настроек блока вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “вызов настройки”. В классе блока для нее создается функция с именем `rdsbcppSetupFunc` следующего вида:

```
// Вызов функции настройки
```

```
void rdsbcppBlockClass::rdsbcppSetupFunc(int &Result)
{
```

*Пользовательский текст реакции*

```
}
```

Через параметр `Result`, передаваемый по ссылке, функция реакции может вернуть ноль, если схему следует считать не изменившейся, и любое ненулевое значение в противном случае. Если в редакторе модели создано окно параметров, то есть реакция вызвана во втором из перечисленных выше вариантов, можно ничего не присваивать параметру `Result`: в этом случае исходно он будет иметь значение 1, сигнализирующее о наличии изменений в схеме. Поскольку реакция при этом будет вызвана только при закрытии окна кнопкой “ОК”, то есть после изменения параметров блока, это значение будет соответствовать действительности.

При написании моделей блоков без использования модуля автокомпиляции событию вызова настроек блока в первом из перечисленных выше вариантов соответствует константа РДС `RDS_BFM_SETUP` (см. §A.2.6.13 приложения к руководству программиста [2]). Вторым вариантом вызова реакции на это событие без участия модуля автокомпиляции невозможен.

### §3.8.11.3. Сообщение от управляющей программы

Это событие возникает при управлении РДС из внешнего приложения (см. главу 3 руководства программиста [1]) в блоке, который программно вызван из этого приложения при помощи одной из функций специальной библиотеки `RdsCtrl.dll`. Внешнее управление обычно используют для включения РДС в состав более крупных специализированных программных комплексов, в которых пользователь не взаимодействует с РДС и загруженной схемой непосредственно. Пользовательский интерфейс в таких случаях обычно обеспечивается внешним приложением, которое в нужный момент передает блокам схемы какую-либо информацию и получает от них ответ. Модуль автокомпиляции никак не автоматизирует создание блоков, отвечающих на такие внешние вызовы. Он предоставляет разработчику возможность написать реакцию модели на сообщение от внешней программы, но разбор переданных данных и отправку ответа разработчик должен выполнить в этой реакции вручную.

Текст реакции на вызов от внешней программы вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “сообщение от управляющей программы”. В классе блока для нее создается функция с именем `rdsbcppRemoteMessage` следующего вида:

```
// Сообщение от управляющей программы
```

```
void rdsbcppBlockClass::rdsbcppRemoteMessage (
 RDS_PREMOTEMSGDATA MessageData, int &Result)
{
```

*Пользовательский текст реакции*

```
}
```

У функции два параметра. Параметр `MessageData` – это указатель на структуру типа `RDS_REMOTEMSGDATA`, содержащую полученные от внешнего приложения данные и описание вызова:

```
typedef struct {
 LPSTR String; // Переданная приложением строка
 int Value; // Переданное приложением число
 LPSTR ControllerName; // Имя приложения
} RDS_REMOTEMSGDATA;
typedef RDS_REMOTEMSGDATA *RDS_PREMOTEMSGDATA;
```

Поля структуры имеют следующий смысл:

- `String` – указатель на строку (`char*`), переданную вызвавшим приложением (сама строка находится во внутренней памяти РДС и будет удалена после завершения этой реакции).
- `Value` – целое число, переданное вызвавшим приложением.
- `ControllerName` – указатель на строку (`char*`) с названием управляющего приложения, если это приложение сообщило РДС свое название.

Второй параметр функции реакции – это ссылка на целую переменную `Result`, через которую модель блока может вернуть вызвавшему приложению целое число в ответ на переданные им данные. Для возврата в вызвавшее приложение строки используется сервисная функция `rdsRemoteReply` (см. §A.5.21.6 приложения к руководству программиста [2]).

При написании моделей блоков без использования модуля автокомпиляции вызову от внешнего приложения соответствует константа РДС `RDS_BFM_REMOTEMSG` (см. §A.2.4.11 приложения к руководству программиста).

#### §3.8.11.4. Срабатывание таймера

Это событие возникает при срабатывании таймера блока в некоторых режимах работы такого таймера. Модель блока может создавать таймеры, которые будут автоматически вызывать ее через заданный интервал времени или с заданной частотой. У таких таймеров есть несколько режимов работы и несколько способов сообщения блоку о срабатывании (это событие – только один из них). Модуль автокомпиляции никак не автоматизирует работу с таймерами, интересующиеся могут изучить §2.9 руководства программиста [1].

Текст реакции на срабатывание таймера вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “реакция на таймер”. В классе блока для нее создается функция с именем `rdsbcppTimer`, параметром которой является идентификатор сработавшего таймера `TimerId`, используемый во всех функциях РДС (см. §A.5.12 приложения к руководству программиста [2]):

```
// Реакция на срабатывание таймера блока
void rdsbcppBlockClass::rdsbcppTimer(RDS_TIMERID TimerId)
{

Пользовательский текст реакции

}
```

При написании моделей блоков без использования модуля автокомпиляции срабатыванию таймера соответствует константа РДС `RDS_BFM_TIMER` (см. §A.2.4.16 приложения к руководству программиста).

### §3.8.11.5. Обновление окон блока

Это событие возникает в подсистемах и блоках при необходимости обновить принадлежащие им окна по таймеру или по команде от пользователя, другого блока или внешнего управляющего приложения. Обычные окна подсистем обновляются автоматически, для этого не требуется реакция на это событие. Такая реакция нужна только в блоках, которые программно открывают какие-либо окна и самостоятельно следят за их обновлением. Автокомпилируемые блоки крайне редко программно открывают какие-либо отдельные окна – это достаточно сложная процедура, и в простых моделях она, как правило, не нужна. По этой причине и реакция на обновление окон в автокомпилируемых моделях практически никогда не используется. Тем не менее, иногда она может служить альтернативой реакции на срабатывание таймера (см. §3.8.11.4 выше) – таймер можно запрограммировать так, чтобы вместо стандартного события срабатывания он вызывал блок для обновления окон. Пример модели, использующей это событие таким образом и созданной без применения модуля автокомпиляции, приведен в §2.10.4 руководства программиста [1].

Текст реакции на обновление окон вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “обновление окон блока”. В классе блока для нее создается функция с именем `rdsbcppWinRefresh` следующего вида:

```
// Обновление немодальных окон блока
void rdsbcppBlockClass::rdsbcppWinRefresh(
 RDS_PWINREFRESHDATA WinRefreshData)
{
 Пользовательский текст реакции
}
```

Параметр функции `WinRefreshData` – это указатель на структуру `RDS_WINREFRESHDATA`:

```
typedef struct {
 RDS_TIMERID Timer; // Таймер, вызвавший обновление, или NULL
 double Delay; // Возвращаемое время обновления окна, мс
} RDS_WINREFRESHDATA;
typedef RDS_WINREFRESHDATA *RDS_PWINREFRESHDATA;
```

Поля структуры имеют следующий смысл:

- `Timer` – идентификатор сработавшего таймера, созданного с флагом `RDS_TIMERS_WINREF` (такие таймеры служат для обновления окон, см. §2.9 руководства программиста) или `NULL`, если обновление окна вызвано не таймером, а сервисной функцией или командой пользователя.
- `Delay` – возвращаемое моделью полное время обновления окон блока в миллисекундах, если она его вычисляет (исходно в этом поле находится значение `-1` и, если оно там и останется, время обновления будет вычислено РДС автоматически).

При написании моделей блоков без использования модуля автокомпиляции событию обновления окон блока соответствует константа РДС `RDS_BFM_WINREFRESH` (см. §A.2.6.21 приложения к руководству программиста [2]).

### §3.8.11.6. Всплывающая подсказка

Это событие возникает у блока, в параметрах которого разрешен вывод всплывающей подсказки, если курсор мыши задержался в пределах его изображения (время задержки определяется Windows). В реакции на него модель формирует и передает РДС текст

подсказки, которую нужно показать пользователю. Примеры моделей, выводющих всплывающие подсказки к блокам, приведены в §3.7.9 (стр. 211).

Текст реакции на вывод подсказки вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “всплывающая подсказка”. В классе блока для нее создается функция с именем `rdsbcppPopupHint` следующего вида:

```
// Установка текста всплывающей подсказки
void rdsbcppBlockClass::rdsbcppPopupHint(
 RDS_PPUPHINTDATA HintData, int &Show)
{
 Пользовательский текст реакции
}
```

Параметр функции `MessageData` – это указатель на структуру типа `RDS_POPUPHINTDATA`, содержащую текущие параметры изображения блока, через которую модель возвращает параметры показа подсказки, если это необходимо:

```
typedef struct {
 int x,y; // Координаты курсора мыши
 int BlockX,BlockY; // Координаты точки привязки блока
 int Left,Top; // Верхний левый угол изображения блока
 int Width,Height; // Размеры изображения блока
 int HZLeft,HZTop, // Возвращаемый размер зоны
 HZWidth,HZHeight; // действия подсказки
 int ReshowTimeout; // Возвращаемый интервал повторного вывода
 int HideTimeout; // Возвращаемый интервал гашения
 int IntZoom; // Масштаб окна в %
 double DoubleZoom; // Масштабный к-т окна (в долях единицы)
} RDS_POPUPHINTDATA;
typedef RDS_POPUPHINTDATA *RDS_PPUPHINTDATA;
```

Назначение полей этой структуры описано на стр. 214. Для возврата в РДС сформированного текста всплывающей подсказки используется специальная функция `rdsSetHintText`.

Второй параметр функции реакции – это ссылка на целую переменную `Show`, через которую модель блока может запретить вывод подсказки: если записать в эту переменную константу `RDS_BFR_NOTPROCESSED` вместо исходно содержащейся в ней `RDS_BFR_DONE`, подсказка выведена не будет. Если в реакции на событие не будет вызвана функция `rdsSetHintText`, то есть текст подсказки не будет передан в РДС, подсказка тоже не появится на экране.

При написании моделей блоков без использования модуля автокомпиляции выводу всплывающей подсказки соответствует константа РДС `RDS_BFM_POPUPHINT` (см. §A.2.6.12 приложения к руководству программиста [2]).

### §3.8.11.7. Вызов контекстного меню

Это событие возникает непосредственно перед появлением на экране контекстного меню блока. Как правило, реакция на него используется для добавления в это меню дополнительных пунктов, при выборе которых модель блока будет вызываться для выполнения каких-либо действий. Примеры моделей, добавляющих свои пункты в контекстное меню, приведены в §3.7.12 (стр. 238).

Текст реакции на это событие вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “вызов контекстного меню”. В классе блока для нее создается функция с именем `rdsbcppContextPopup` следующего вида:

```

// Действия перед вызовом контекстного меню
void rdsbcppBlockClass::rdsbcppContextPopup(
 RDS_PCONTEXTPOPUPDATA MenuData)
{

```

*Пользовательский текст реакции*

```

}

```

Параметр функции MenuData – это указатель на структуру RDS\_CONTEXTPOPUPDATA:

```

typedef struct {
 BOOL EditMode; // Включен режим редактирования
 BOOL FreeSpace; // Щелчок на свободном месте окна
} RDS_CONTEXTPOPUPDATA;
typedef RDS_CONTEXTPOPUPDATA *RDS_PCONTEXTPOPUPDATA;

```

Поля структуры имеют следующий смысл:

- EditMode – TRUE, если РДС в момент вызова меню находится в режиме редактирования, и FALSE, если РДС находится в режимах моделирования или расчета (пункты меню в режиме редактирования, как правило, отличаются).
- FreeSpace – FALSE, если вызвано контекстное меню самого блока, и TRUE, если контекстное меню вызвано щелчком не на изображении блока, а на свободном месте окна подсистемы. Это поле может принимать значение TRUE только в том случае, если данный блок – подсистема, и ее окно в данный момент открыто. Поскольку автокомпилируемые модели подсистем практически не используются, необходимости в анализе значения этого поля обычно не возникает.

При написании моделей блоков без использования модуля автокомпиляции вызову контекстного меню блока соответствует константа РДС RDS\_BFM\_CONTEXTPOPUP (см. §A.2.6.2 приложения к руководству программиста [2]).

### §3.8.11.8. Выбор пункта меню

Это событие возникает при выборе пользователем пункта, добавленного моделью в контекстное меню блока или системное меню РДС. Примеры моделей, добавляющих свои пункты в контекстное и системное меню, приведены в §3.7.12 (стр. 238).

Текст реакции на это событие вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “выбор пункта меню”. В классе блока для нее создается функция с именем rdsbcppMenuFunc следующего вида:

```

// Реакция на зарегистрированный пункт меню
void rdsbcppBlockClass::rdsbcppMenuFunc(
 RDS_PMENUFUNCDATA MenuData)
{

```

*Пользовательский текст реакции*

```

}

```

Параметр функции MenuData – это указатель на структуру RDS\_MENUFUNCDATA, содержащую идентификаторы, присвоенные моделью выбранному пункту меню при его создании:

```

typedef struct {
 int Function; // Номер функции меню
 int MenuData; // Данные меню
} RDS_MENUFUNCDATA;
typedef RDS_MENUFUNCDATA *RDS_PMENUFUNCDATA;

```

Поля структуры имеют следующий смысл:



- `Function` – идентификатор выбранного пользователем пункта меню, указанный при его создании.
- `MenuData` – дополнительное целое число, указанное при создании выбранного пункта меню.

При написании моделей блоков без использования модуля автокомпиляции выбору пункта меню соответствует константа РДС `RDS_BFM_MENUFUNCTION` (см. §A.2.6.7 приложения к руководству программиста [2]).

### §3.8.11.9. После создания статических переменных

Это событие возникает в простом блоке после любого изменения его статических переменных. Реакция на это событие может использоваться, например, для программного анализа этой структуры или для присвоения переменным начальных значений, отличающихся от заданных в редакторе модели. Эта реакция вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “после создания статических переменных”. В классе блока для нее создается функция с именем `rdsbcppStaticVarsInit` без параметров:

```
// После успешного создания статических переменных
void rdsbcppBlockClass::rdsbcppStaticVarsInit(void)
{
```

*Пользовательский текст реакции*

```
}
```

У этого события нет точного соответствия со стандартными событиями РДС. Функция реакции для него вызывается внутри автоматически добавляемой в модель модулем автокомпиляции реакции на событие проверки переменных `RDS_BFM_VARCHECK` (см. §A.2.4.18 приложения к руководству программиста [2]) при условии успешного прохождения переменными этой проверки.

### §3.8.11.10. Прочие события

Эта реакция вызывается в ответ на все события РДС, для которых в редакторе модели не предусмотрены отдельные функции реакции. Она вводится на вкладке “события” левой панели редактора модели: раздел “разное”, подраздел “прочие события”. В классе блока для нее создается функция с именем `rdsbcppOther` следующего вида:

```
// Прочие события
void rdsbcppBlockClass::rdsbcppOther(
 int CallMode, void *ExtParam, int &Result)
{
```

*Пользовательский текст реакции*

```
}
```

Первый параметр функции – целый идентификатор произошедшего события `CallMode`. В нем передается одна из констант `RDS_BFM_...`, описанных в руководстве программиста и приложении к руководству программиста. Второй параметр – указатель общего вида `ExtParam`. Он указывает на структуру параметров события, его необходимо привести к нужному, соответствующему конкретному событию, типу вручную. Третий параметр – ссылка на целую переменную `Result`, в которую можно записать значение, которое модель должна вернуть РДС в ответ на вызов (по умолчанию в ней записана константа `RDS_BFR_DONE`, указывающая на успешное выполнение реакции).

В автокомпилируемых моделях реакция на прочие события используется редко – для всех основных событий предусмотрены специализированные функции реакции, описанные выше. Следует учитывать, что эта реакция не будет вызвана, если для события введена специализированная функция: например, если в модель введена реакция на выполнение такта расчета (см. §3.8.3.1 на стр. 290), в ответ на событие RDS\_BFM\_MODEL будет вызвана именно она, а функция `rdsbcppOther` с параметром `CallMode`, равным RDS\_BFM\_MODEL, вызвана не будет.

### §3.9. Настройки стандартного модуля автокомпиляции

Описываются параметры модуля автокомпиляции, при помощи которых можно настраивать его на работу с конкретным компилятором C++, управлять автоматическим формированием исходного текста программы и создавать шаблоны моделей.

#### §3.9.1. Общие настройки модуля

Рассматриваются общие настройки модуля автокомпиляции, в которых указываются пути к различным файлам и папкам установленного в системе компилятора C++, а также задаются некоторые параметры редактора модели.

Настройки модуля автокомпиляции не относятся к какой-либо конкретной модели. Они определяют, как именно по данным модели и введенным пользователем фрагментам исходного текста формируется полный текст программы с моделью блока, где находится внешний компилятор, который собирает из этого полного текста исполняемую библиотеку (DLL), как этот компилятор запускается. Кроме того, к настройкам модуля относится также набор шаблонов стандартных моделей (см. стр. 22 и §3.7.4.4 на стр. 168) и некоторые общие параметры пользовательского интерфейса редактора модели. В стандартных модулях, предназначенных для работы с поддерживаемыми компиляторами языка C++, подключение и настройка которых описаны в §3.2 (стр. 10), все параметры уже настроены, и их необдуманное изменение может привести к тому, что модели перестанут компилироваться правильно. Поэтому начинающим пользователям **крайне не рекомендуется** изменять какие-либо настройки работающих модулей, за исключением набора шаблонов моделей и параметров интерфейса редактора. Все параметры модуля, которые могут понадобиться начинающему пользователю, рассматриваются в этом параграфе и в §3.9.2 (стр. 331). Остальные параметры, необходимые, в основном, для настройки модулей на работу с нестандартными компиляторами, описываются, начиная с §3.9.3 (стр. 333).

Вызвать настройки одного из стандартных модулей автокомпиляции в РДС можно двумя способами. Если открыто окно редактора модели, обслуживаемой этим модулем, проще всего выбрать в окне редактора пункт меню “модель | настройка модуля автокомпиляции” (см. §3.6.1 на стр. 32). Если же окна моделей закрыты или в РДС не загружена схема, следует вызвать на передний план главное окно РДС (если оно перекрыто окнами подсистем, можно сделать это нажатием клавиши F10) и выбрать в его меню пункт “сервис | автокомпиляция”. При этом откроется окно со списком зарегистрированных модулей автокомпиляции (см. стр. 11), подробно описанное в §2.19.1 части I. В этом окне следует дважды щелкнуть мышью на названии модуля, параметры которого должны быть изменены.

Следует иметь в виду, что модуль, настройки которого нужно изменить, должен быть уже зарегистрирован в РДС. Регистрация модулей для работы со стандартными компиляторами описана в §3.2 (стр. 10). Подключение модуля для нестандартного компилятора рассматривается в §3.9.3 (стр. 333) и требует от пользователя хорошего знания параметров этого компилятора и умения работы с ним через командную строку.

В открывшемся после описанных выше действий окне настройки модуля исходно выбрана вкладка “компилятор” (рис. 484). На ней задаются пути к исполняемым файлам и



папкам компилятора, с которым работает модуль, а также находится кнопка “дополнительно”, открывающая окно дополнительных параметров этого модуля. У модулей, предназначенных для работы с разными компиляторами, внешний вид этой вкладки будет немного различаться. На рис. 484 приведена вкладка окна настроек модуля, работающего с компилятором Borland C++ 5.5 (см. §3.2.1 на стр. 10), а на рис. 489 (стр. 335) – вкладка окна универсального модуля, который пользователь настраивает полностью вручную. Начинаящему пользователю на этой вкладке может понадобиться только самое первое поле ввода, в котором указывается папка установки компилятора. Все остальные поля ввода автоматически заполняются при изменении этого, и пользователю, работающему только со стандартными компиляторами, поддержка которых встроена в модуль, не нужно о них заботиться. Нажимать на кнопку “дополнительно” ему тоже нет необходимости – все дополнительные настройки для стандартных компиляторов уже введены. Смысл отдельных полей вкладки “компилятор” будет описан в §3.9.3 (стр. 333), здесь мы не будем на ней останавливаться.

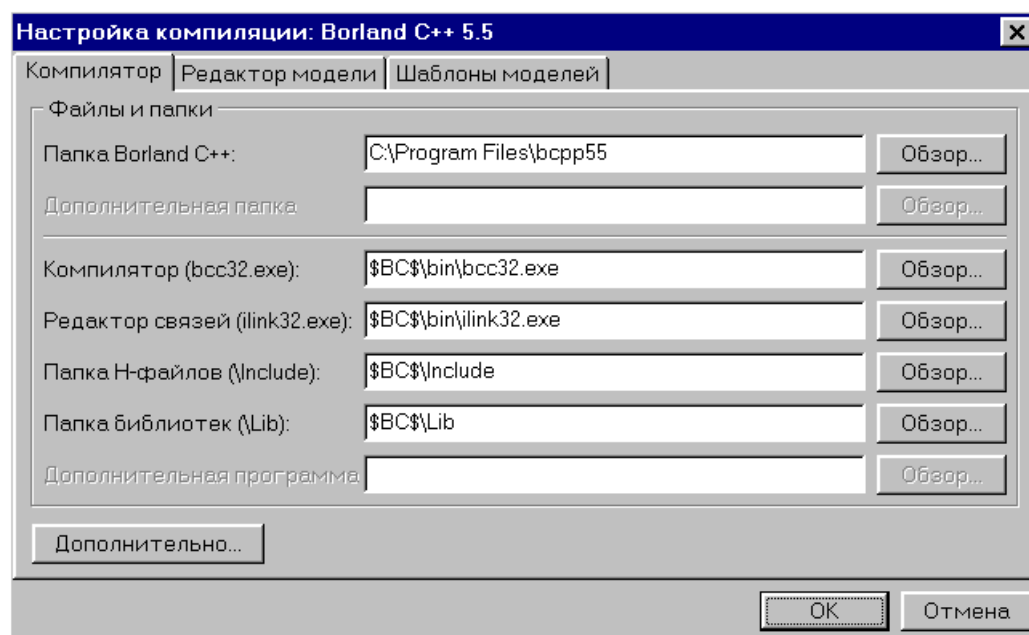


Рис. 484. Вкладка “компилятор” окна настроек модуля

На вкладке “редактор модели” (рис. 485) можно задать шрифт текста в окне редактора и некоторые параметры поведения самого редактора. Если на панели “шрифт редактора” установлен флажок “по умолчанию”, при вводе фрагментов программ в редакторе модели будет использоваться моноширинный (непропорциональный) шрифт Courier New (см. пример текста в окне – рис. 329 на стр. 32). Для текстов программ лучше всего использовать именно моноширинные шрифты, поскольку в этом случае символы во всех строках текста всегда находятся точно друг под другом, что упрощает оформление текста. В центре панели показывается образец выбранного шрифта – в качестве текста образца используется название этого шрифта и его размер. Пользователь может, при желании, изменить шрифт или его размер, установив на панели флажок “другой” и нажав кнопку “изменить” в правой ее части. Следует учитывать, что для того, чтобы изменение шрифта было принято, может потребоваться закрыть все открытые в данный момент окна редакторов моделей и открыть их заново.

Ниже панели выбора шрифта располагаются дополнительные флажки, управляющие сохранением модели.

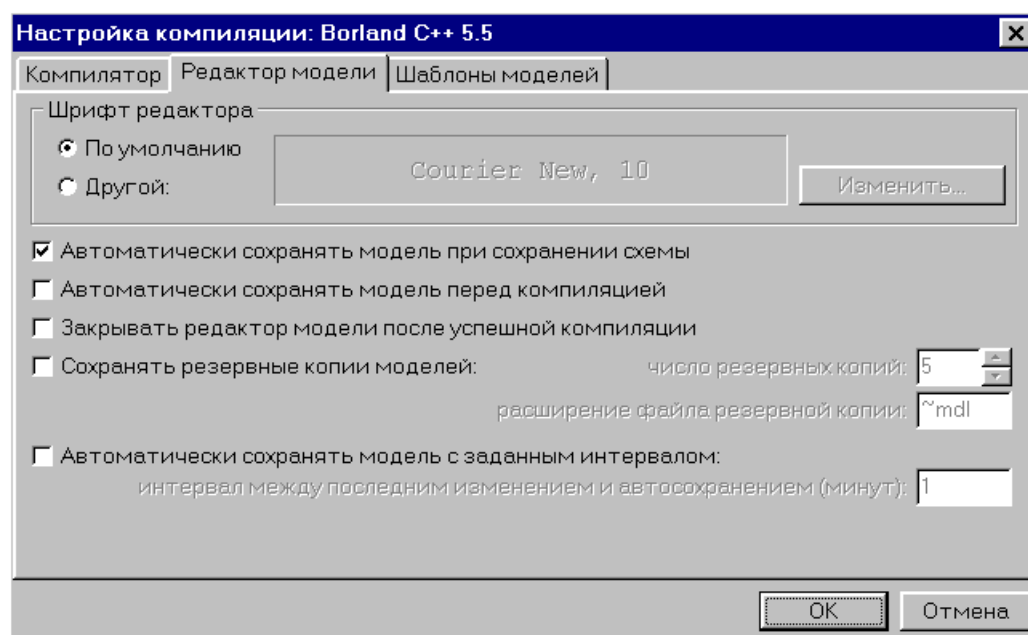


Рис. 485. Вкладка “редактор модели” окна настроек модуля

Флажок “автоматически сохранять модель при сохранении схемы”, включенный по умолчанию, заставляет модуль сохранять все несохраненные модели в момент сохранения пользователем схемы. Чаще всего пользователь, сохраняя схему, желает сохранить и все ее модели, поэтому этот флажок лучше оставить включенным. Выключать его имеет смысл только при экспериментах с моделью, когда еще не до конца ясно, следует ли сохранять внесенные в нее экспериментальные изменения.

Установка флажка “автоматически сохранять модель перед компиляцией” включает дополнительное автосохранение модели: если в нее были внесены какие-либо изменения, перед ее компиляцией она будет автоматически записана на диск. Поскольку скомпилированные модели блоков размещаются в динамически подключаемых библиотеках (DLL), они работают в одном адресном пространстве с главным приложением, и критические ошибки в них могут привести к аварийному завершению РДС. Это, в свою очередь, приведет к потере всех не сохраненных изменений в модели. Таким образом, автоматическое сохранение модели перед компиляцией позволяет забывчивым пользователям, которые не сохраняют изменения при первой возможности, не терять данные. С другой стороны, установка этого флажка лишает пользователя возможности внести изменения в модель и, не сохраняя их, попробовать, как она работает, чтобы, если результат ему не понравится, выйти из редактора без сохранения изменений, возвращаясь к прежней версии модели. Какой из вариантов удобнее, решать пользователю.

Установка флажка “закрывать редактор модели после успешной компиляции” может несколько ускорить работу, если пользователь постоянно вносит в модель мелкие изменения и проверяет, как они отразились на ее поведении. Как и следует из названия флажка, в этом случае после каждой успешной компиляции окно редактора будет автоматически закрываться и перестанет скрывать за собой окно подсистемы, в которой находится блок с тестируемой моделью – пользователь может сразу запустить расчет, подать на блок новые значения и т.п. Однако, если пользователь не тестирует каждое внесенное изменение и компилирует модель время от времени только чтобы увидеть, не допустил ли он на данный момент каких-либо синтаксических ошибок, о которых ему сообщит компилятор, этот флажок лучше оставить выключенным.

Флажок “сохранять резервные копии моделей” позволяет при сохранении файла модели не стирать старый файл, а переименовывать его, присваивая ему другое расширение

(аналогичная настройка есть и для схем в общих настройках РДС, см. §2.18 части I). Число таких хранимых резервных копий и их расширение задается в отдельных полях ввода. Самая последняя резервная копия получает указанное в настройках расширение (по умолчанию – “~mdl”), более старые – это же расширение с добавлением числа тем большего, чем старше копия (“~mdl01” – предыдущая, “~mdl02” – перед предыдущей и т. д.). Если количество резервных копий превысит заданное в настройках число, самые старые из них будут стираться.

Флажок “автоматически сохранять модель с заданным интервалом” включает автоматическое сохранение модели через заданное время после внесения в нее последнего изменения. Как в случае других флажков, связанных с автосохранением, включение этого лишает пользователя возможности экспериментировать с моделью без сохранения изменений.

Третья вкладка окна настроек модуля, на которой можно управлять шаблонами стандартных моделей, рассмотрена в §3.9.2.

### §3.9.2. Добавление и изменение шаблонов моделей

Описывается работа с шаблонами моделей, позволяющими создавать новые модели по образцам, в которых уже введены некоторые переменные, параметры и реакции.

Шаблоны моделей – это обычные файлы моделей, находящиеся в папке “Common” внутри стандартной папки шаблонов РДС (см. §2.18 части I). Они используются для создания новых моделей в качестве образцов – если при создании нового автокомпилируемого блока выбрать вариант “создать модель по шаблону” (см. рис. 317 на стр. 22), модель сразу получит реакции на события, переменные и параметры, заданные в выбранном шаблоне, так что пользователю останется только изменить их по своему желанию.

Шаблон создается путем сохранения какой-либо модели, которую пользователь желает использовать в качестве образца для создания других моделей. Создать новый шаблон можно несколькими способами. Можно, например, выполнить следующие действия:

- загрузить в РДС схему, в которой находится блок с моделью, которая будет использована в качестве шаблона;
- открыть окно редактора этой модели;
- выбрать в меню редактора пункт “модель | параметры модели” и на вкладке “описание” открывшегося окна параметров модели (см. рис. 356 на стр. 74) ввести текстовое описание шаблона, поясняющее его назначение – этот текст увидит пользователь, который будет выбирать шаблон для создания новой модели (см. рис. 316 на стр. 22);
- выбрать в меню редактора пункт “файл | сохранить как шаблон” и указать в диалоге сохранения имя файла для сохраняемого шаблона (по умолчанию в этом диалоге будет выбрана стандартная папка для шаблонов моделей, не нужно ее менять).

Добавленный таким образом шаблон можно переименовывать, удалять и перемещать в списке вверх и вниз на вкладке “шаблоны моделей” окна настройки модуля автокомпиляции (рис. 486). На ней же находится кнопка, позволяющая создать шаблон другими способами. Порядок вызова окна настройки с этой вкладкой описан на стр. 328.

Большую часть вкладки занимает список шаблонов моделей, то есть список файлов в папке “Common” внутри стандартной папки шаблонов РДС. В списке, как правило, отображаются не имена файлов, а более информативные названия шаблонов – их можно ввести здесь же способом, описанным ниже. Если для шаблона не задано название, будет отображаться имя файла.

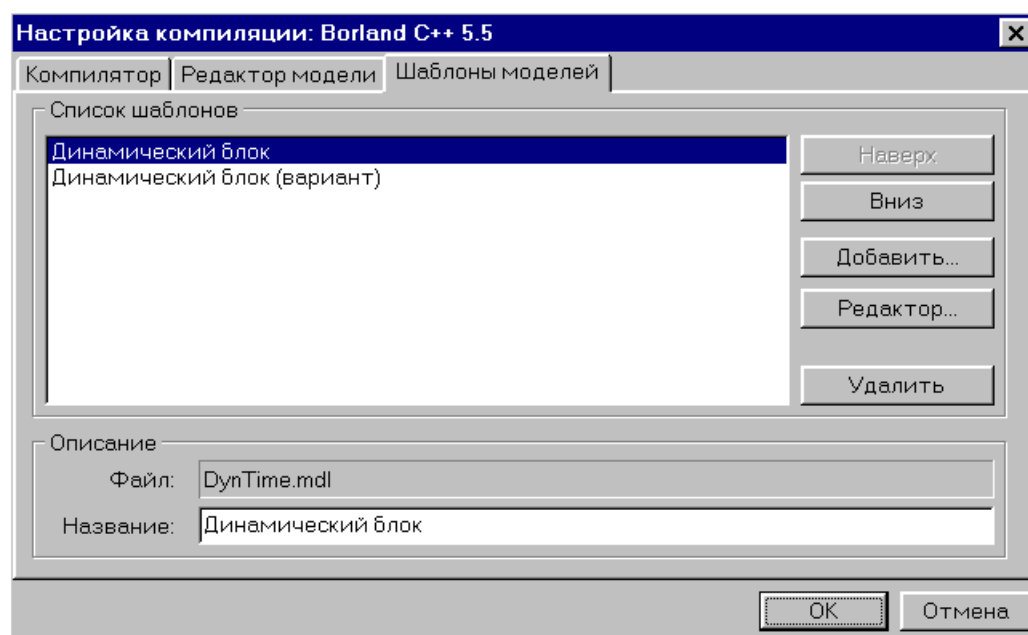


Рис. 486. Вкладка “шаблоны моделей” окна настроек модуля

В нижней части вкладки для выбранного в списке шаблона отображается имя файла, в котором он находится. Под именем файла располагается поле ввода “название”, в котором можно ввести название шаблона, то есть строку, которая будет показана для этого шаблона в списках и на этой вкладке, и в окне выбора шаблона для создания модели (см. рис. 316 на стр. 22). Как правило, в названии коротко указывают назначение шаблона, более подробный многострочный текст описания шаблона можно ввести в редакторе модели (см. рис. 356 на стр. 74).

Справа от списка находятся кнопки, позволяющие менять порядок шаблонов в списке, добавлять, удалять и редактировать шаблоны:

| <i><b>Кнопка</b></i> | <i><b>Действие</b></i>                                                |
|----------------------|-----------------------------------------------------------------------|
| Наверх               | Перемещает выбранный в списке шаблон на одну строку вверх             |
| Вниз                 | Перемещает выбранный в списке шаблон на одну строку вниз              |
| Добавить             | Открывает меню добавления шаблона                                     |
| Редактор             | Открывает редактор модели для выбранного в списке шаблона             |
| Удалить              | Удаляет файл выбранного шаблона с диска (запрашивается подтверждение) |

При нажатии кнопки “добавить” открывается дополнительное меню (рис. 487), в котором выбирается способ добавления шаблона. При выборе в нем пункта “создать новый шаблон” откроется диалог сохранения файла, в котором нужно ввести имя файла шаблона для сохранения. Если ввести это имя и нажать в диалоге кнопку “сохранить”, в папке шаблонов и в списке шаблонов на вкладке появится новый пустой шаблон с указанным именем – теперь можно ввести для него название в нижней части вкладки и изменить шаблон в редакторе модели, нажав кнопку “редактор”.

Пункт меню “создать из существующего” позволяет скопировать какой-либо уже имеющийся шаблон под новым именем. При его выборе открывается диалог с заголовком “копировать шаблон из...”, в котором нужно выбрать существующий шаблон и нажать кнопку “открыть”. Выбранный шаблон будет скопирован в папку шаблонов под новым, автоматически присвоенным ему именем (как правило, к старому имени файла при этом просто прибавляется цифра), и под этим же именем он появится в конце списка шаблонов на

вкладке. Теперь он доступен для редактирования, как и все остальные шаблоны.

Последний пункт меню, “создать из модели”, позволяет скопировать какую-либо уже существующую модель блока в папку шаблонов. При его выборе открывается уже описанный выше диалог с заголовком “копировать шаблон из...”, в котором нужно выбрать файл копируемой модели. После нажатия в диалоге кнопки “открыть” выбранный файл модели будет скопирован в папку шаблонов и появится в конце списка шаблонов на вкладке.

Редактирование шаблона при нажатии кнопки “редактировать” ничем не отличается от редактирования обычной модели – при этом открывается то же самое окно редактора (см. §3.6 на стр. 32).

Следует учитывать, что все стандартные модули автокомпиляции, входящие в состав РДС и описанные в §3.2 (стр. 10), используют один и тот же набор шаблонов моделей. Таким образом, изменения, сделанные на вкладке “шаблоны моделей” окна настройки любого из этих модулей, распространяются сразу на все модули.

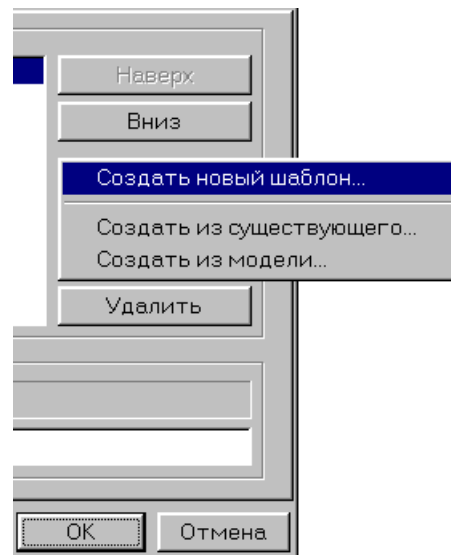


Рис. 487. Меню создания шаблона

### §3.9.3. Подключение универсальных модулей и настройка путей

Рассматривается подключение универсальных модулей автокомпиляции, которые можно настроить на работу с компилятором C++, не поддерживаемым стандартными модулями. Описывается задание путей к различным файлам и папкам этого компилятора.

Стандартные модули автокомпиляции, входящие в состав РДС, поддерживают только небольшое количество стандартных компиляторов языка C++ – полный список этих компиляторов приведен в §3.1 (стр. 6). Среди этих компиляторов есть бесплатные, установка которых достаточно проста, поэтому для создания автокомпилируемых моделей **настоятельно рекомендуется** выбрать какой-либо компилятор из приведенного списка. Тем не менее, если разработчик моделей привык использовать для своих целей какой-либо другой компилятор, существует возможность настроить один из универсальных модулей автокомпиляции на работу с ним. Для этого компилятор должен удовлетворять следующим условиям:

- он должен поддерживать язык C++ – в автокомпилируемых моделях используются классы, поэтому компилятор “чистого” C не подходит;
- он должен управляться из командной строки;
- он должен иметь возможность создавать динамически подключаемые библиотеки (DLL) Windows.

Настройка модуля автокомпиляции на работу с компилятором, поддержка которого не включена в модуль – достаточно сложная задача. Пользователь, выполняющий эту настройку, должен очень хорошо представлять себе, как при помощи выбранного им компилятора создать DLL с функцией, отвечающую всем требованиям, предъявляемым к функции модели блока РДС. Эти требования подробно описаны в руководстве программиста [1]. Лучше всего перед тем, как приступить к настройке модуля, создать при помощи выбранного компилятора несколько моделей блоков полностью вручную, и добиться того, чтобы эти модели успешно работали в схемах. После этого станет ясно, как именно нужно вызывать компилятор для создания DLL и какие описания нужно автоматически добавлять в исходный текст программы. Выяснив это, нужно будет записать необходимые параметры в настройки модуля. Для успешной настройки модуля на работу с компилятором нужно знать:

- как называется исполняемый файл компилятора и какими должны быть его параметры командной строки для создания DLL из файла исходного текста на языке C++;
- если редактор связей (linker) – отдельная программа, как называется его исполняемый файл и какими должны быть параметры его командной строки;
- как в результате компиляции на основе имени функции формируется экспортированное имя для библиотеки (экспортированное имя может не совпадать с именем функции в программе, такое изменение имени называется “name mangling”);
- как должна называться главная функция DLL;
- как устроены текстовые сообщения об ошибках компилятора и редактора связей (нужно будет указать модулю, как разбирать их для предъявления пользователю);
- какие стандартные файлы заголовков нужно автоматически включать в исходный текст для правильной компиляции модели;
- нужны ли для работы компилятора специфические переменные окружения.

Пользователям, которые не смогли ответить на один из приведенных выше вопросов, или которым не понятен смысл этих вопросов и использованные в них термины, лучше не пытаться настраивать модули автокомпиляции самостоятельно.

Для работы с исходно не поддерживаемым компилятором можно изменить настройки какого-либо модуля, предназначенного для другого компилятора, но лучше всего использовать для этого один из универсальных модулей с именами “UserComp1”, “UserComp2” и “UserComp3”. Рассмотрим регистрацию такого модуля в РДС для подключения к гипотетическому компилятору.

Подключение универсального модуля автокомпиляции, как и подключение специализированных (см. §3.2 на стр. 10), выполняется в окне со списком модулей. Откроем это окно, для чего выберем в главном меню РДС пункт “сервис | автокомпиляция”. Нажмем в правой части этого окна кнопку со знаком “+” – в конце списка модулей появится новый, пока не настроенный, модуль. В нижней части окна заполним для него поля ввода следующим образом (рис. 488):

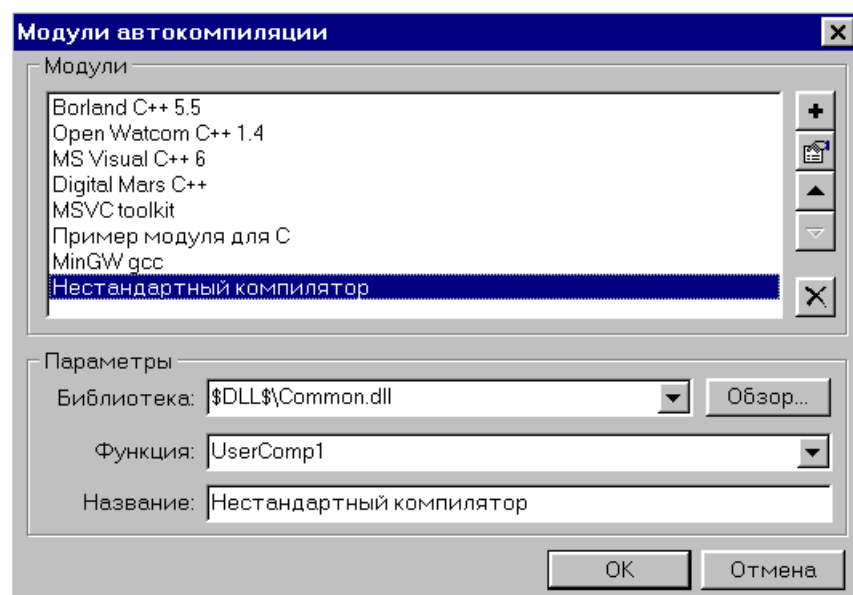


Рис. 488. Модуль для нестандартного компилятора в списке модулей

- “библиотека” – “\$DLL\$Common.dll”;
- “функция” – “UserComp1” (важно соблюдать регистр символов);
- “название” – “Нестандартный компилятор” (или любое другое название по желанию).

Таким образом, мы зарегистрировали в РДС модуль, обслуживаемый функцией “UserComp1”, находящейся в библиотеке “Common.dll” в стандартной папке DLL РДС, и

дали ему название “нестандартный компилятор” (под этим названием этот модуль будет видеть пользователь, оно не влияет на работу модуля). Теперь нужно указать для него пути к файлам компилятора и прочие параметры, которые на момент настройки мы должны уже знать.

Дважды щелчком мышью на названии нового модуля в списке – откроется окно его настройки с выбранной вкладкой “компилятор” (рис. 489). Здесь мы должны ввести все необходимые для работы пути.

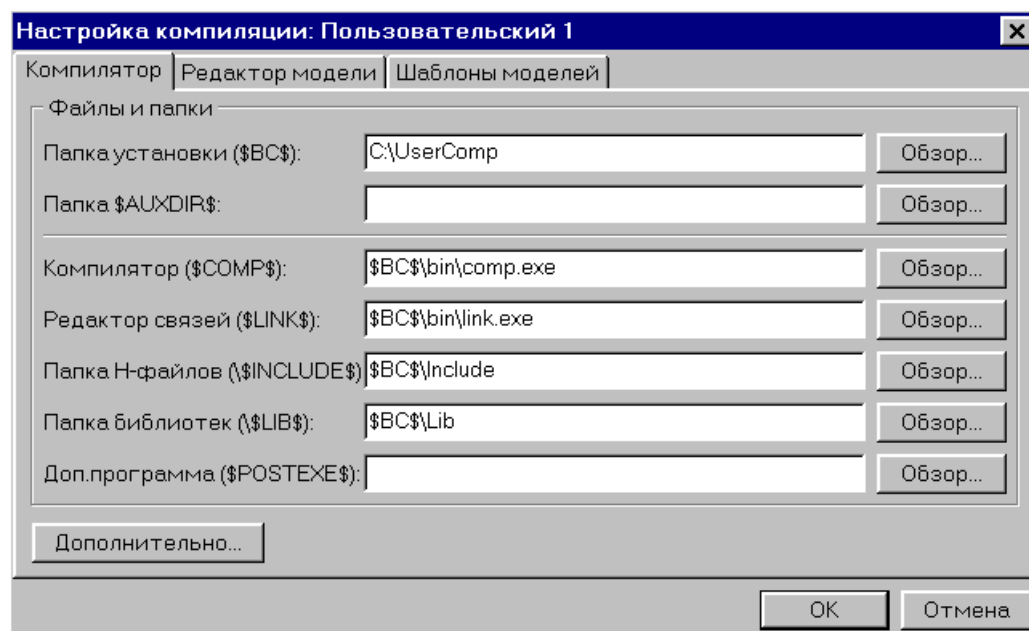


Рис. 489. Настройка путей для компилятора

Допустим, что наш гипотетический нестандартный компилятор установлен в папку “C:\UserComp”, внутри которой в папке “bin” находятся исполняемые файлы компилятора и редактора связей “comp.exe” и “link.exe” соответственно. Кроме того, в папке установки компилятора находится папка “include” с файлами заголовков, необходимыми для компиляции программ, и папка “lib” с библиотеками, необходимыми для сборки исполняемых файлов. Есть множество способов записать эти пути в поля ввода вкладки “компилятор” – сначала рассмотрим самый удобный из них.

Первое поле вкладки называется “папка установки”, в него вводится путь к папке, внутри которой размещаются все прочие файлы и папки, имеющие отношение к компилятору. В нашем случае это “C:\UserComp”. Путь к папке можно либо ввести с клавиатуры, либо, нажав кнопку “обзор”, выбрать ее в стандартном диалоге выбора папки Windows. Это относится и ко всем остальным полям ввода на вкладке: любой путь можно как ввести вручную, так и выбрать в диалоге, открываемом по кнопке “обзор”.

Теперь, когда мы ввели путь к общей папке установки компилятора, во всех остальных путях вместо этого пути можно указывать символическое обозначение “\$BC\$” (оно указано в названии поля). Так мы и поступим. Исполняемый файл гипотетического компилятора “comp.exe” размещается в папке “bin” внутри папки установки, значит, используя символическое обозначение, в качестве пути к нему можно ввести “\$BC\$\bin\comp.exe”. Если вместо ручного ввода пути мы воспользуемся кнопкой “обзор” для выбора исполняемого файла компилятора, символическое обозначение “\$BC\$” будет подставлено в путь автоматически. Точно так же, с помощью символического обозначения, можно указать пути к исполняемому файлу редактора связей (“\$BC\$\bin\link.exe”), папке заголовков (“\$BC\$\include”) и папке библиотек (“\$BC\$\lib”).



Использование символического обозначения “\$BC\$” при указании всех путей удобно тем, что, если нам потребуется перенести компилятор в другую папку, в настройках модуля автокомпиляции нужно будет изменить только одно поле – поле папки установки компилятора. Поскольку все остальные пути указаны относительно этой папки, а внутренняя структура папок компилятора при переносе не изменилась, эти пути останутся верными. Тем не менее, никто не заставляет пользователя использовать это символическое обозначение. При желании, можно вообще не указывать папку установки, а все пути ко всем файлам и папкам указать полностью (рис. 490): в качестве пути к исполняемому файлу компилятора ввести “C:\UserComp\bin\comp.exe”, к папке библиотек – “C:\UserComp\Lib” и т.п. Следует только учитывать, что при переносе компилятора в этом случае потребуется исправлять все введенные пути вручную.

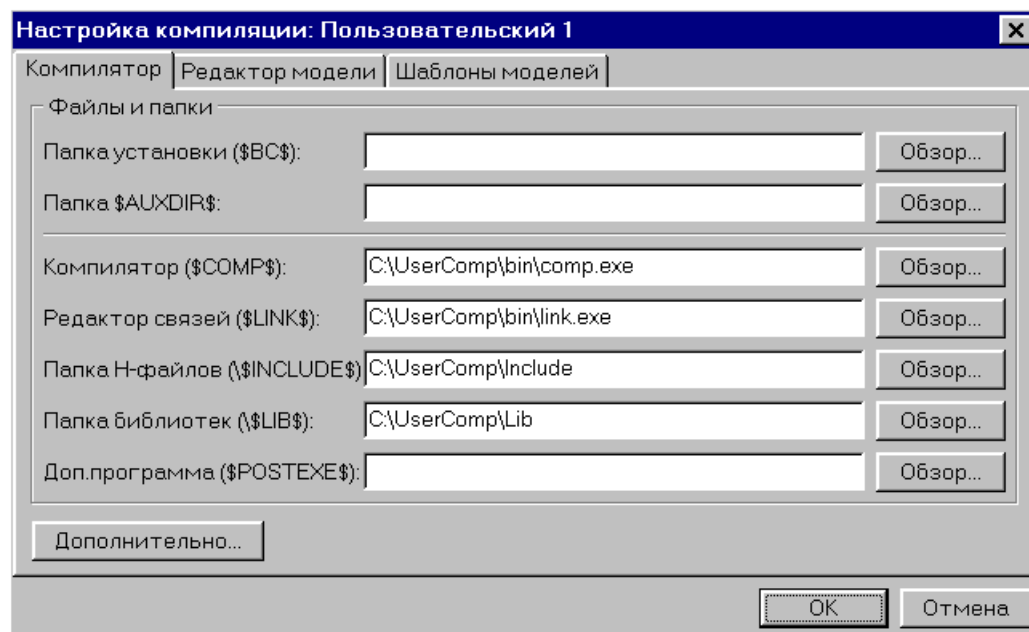


Рис. 490. Альтернативный способ задания путей для компилятора

Пути на вкладке “компилятор” вводятся только для того, чтобы при настройке всех остальных параметров можно было не указывать каждый путь вручную, а использовать символические обозначения этих путей. Для справки эти обозначения выводятся в названии каждого поля: вместо введенного пути к компилятору можно везде указывать “\$COMP\$”, вместо пути к папке заголовков – “\$INCLUDE\$” и т.п. Полный список доступных символических обозначений приведен в §3.9.4 на стр. 337 (кроме обозначений для введенных в настройках путей туда входят и обозначения для некоторых стандартных папок РДС). Использовать такие символические обозначения гораздо удобнее, чем указывать везде все пути полностью. Кроме того, как было показано выше, обозначения позволяют проще переносить компилятор из папки в папку. При желании можно вообще не вводить на вкладке “компилятор” никаких путей, а все пути прописывать вручную при задании параметров вызова компилятора (о них речь пойдет в §3.9.6 на стр. 342). Однако, это крайне неудобно.

На вкладке “компилятор” можно также указать два дополнительных пути, которые в большинстве компиляторов не используются. Во-первых, можно указать путь к произвольной дополнительной папке: он вводится в поле “папка \$AUXDIR\$”. В этой папке могут находиться какие-либо дополнительные объекты, необходимые компилятору для работы. В параметрах можно сослаться на нее при помощи символического обозначения “\$AUXDIR\$”. Например, для работы с Microsoft Visual C++ 2003 (см. §3.2.5 на стр. 16) необходимо дополнительно установить Microsoft Platform SDK в отдельную папку – можно указать путь к этой папке в этом дополнительном поле.



Во-вторых, модуль автокомпиляции позволяет запустить произвольную программу после того, как компилятор завершит работу. Путь к исполняемому файлу этой программы можно указать в самом последнем поле ввода с заголовком “доп. программа”. В параметрах на этот путь можно ссылаться при помощи символического обозначения “\$POSTEXE\$”. Такая дополнительная программа может быть создана разработчиком самостоятельно и использоваться для того, чтобы, например, скопировать готовый файл DLL в нужную папку, если компилятор, по каким-то причинам, не способен сразу записать ее туда.

Для успешной компиляции модели нестандартным компилятором одного задания путей на вкладке “компилятор” недостаточно. Необходимо указать модулю автокомпиляции, с какими параметрами запускать исполняемые файлы, пути к которым указаны на вкладке, как формировать текст программы из введенных пользователем фрагментов и т.п. Все эти параметры задаются в отдельном окне, открывающемся при нажатии на кнопку “дополнительно” в нижней части вкладки. Это окно описано в §3.9.5 (стр. 341) и далее.

### §3.9.4. Символические имена параметров в настройках

Описываются символические обозначения путей и параметров, которые можно использовать при задании настроек модуля автокомпиляции. Использование обозначений вместо самих значений параметров позволяет сделать настройки более гибкими: при изменении какого-либо параметра не нужно будет исправлять его значение во всех полях окна настроек, где он встречается.

Для того, чтобы вызвать компилятор и редактор связей, нужно передать операционной системе командную строку с указанием вызываемой программы (компилятора и редактора связей) с полным путем к этой программе и ее параметрами, указывающими, какой именно файл исходного текста необходимо обработать, в каких папках находятся библиотеки и файлы заголовков и т.п. Эта командная строка вводится в параметрах модуля автокомпиляции. В ней можно использовать специальные символические обозначения, вместо которых модуль подставит пути к различным папкам и файлам, заданным в настройках РДС и самого модуля. Использование таких обозначений гораздо удобнее ручного прописывания всей путей в командной строке.

Проиллюстрируем удобство символических обозначений на простом примере. При использовании компилятора Open Watcom C++ (см. §3.2.2 на стр. 12) пути к файлам заголовков необходимо указывать в его командной строке в параметре “/i”. В состав компилятора входит папка “h”, внутри которой находятся необходимые заголовки. Кроме того, заголовки, которые нужно включить, находятся внутри этой папки “h” во вложенных папках “sys” и “nt”. Пусть сам компилятор находится в папке “C:\Compilers\OpenWatcom”. Если указывать все пути вручную, в командную строку придется добавить такой параметр:

```
/i="C:\Compilers\OpenWatcom;C:\Compilers\OpenWatcom\sys;
C:\Compilers\OpenWatcom\nt"
```

С использованием символических обозначений этот параметр будет значительно короче:

```
/i="$INCLUDE$;$INCLUDE$\sys;$INCLUDE$\nt"
```

Разумеется, чтобы можно было использовать обозначение “\$INCLUDE\$”, путь к папке заголовков должен быть задан на вкладке “компилятор” (см. §3.9.3 на стр. 333).

Стандартный модуль автокомпиляции позволяет использовать следующие символические обозначения:

\$AUXDIR\$

Полный путь к дополнительной папке, заданный в настройках модуля на вкладке “компилятор”.

\$BC\$

Полный путь к папке установки компилятора, заданный в настройках модуля, без символа обратной косой черты (“\”) в конце.

\$COMP\$

Полный путь к исполняемому файлу компилятора, указанный в настройках модуля.

\$CPPFILE\$

Имя исходного файла текста программы модели без пути с расширением "cpp". Этот файл автоматически формируется модулем, компилятор должен преобразовать его в объектный.

\$DLLFILE\$

Имя скомпилированного файла DLL без пути. Этот файл создается редактором связей из объектного файла и необходимых библиотек.

\$FILE\$

Имя исходного файла текста программы модели без пути и расширения. Этот файл автоматически формируется модулем.

\$FUNC\$

Имя функции модели блока (всегда "rdsbcppBlockEntryPoint").

\$FUNCLC\$

Имя функции модели блока, приведенное к нижнему регистру (всегда "rdsbcppblockentrypoint").

\$FUNCUC\$

Имя функции модели блока, приведенное к верхнему регистру (всегда "RDSBCPPBLOCKENTRYPOINT").

\$INCLUDE\$

Полный путь к папке стандартных заголовков компилятора, указанной в настройках модуля, без символа обратной косой черты ("\") в конце. Обычно файлы заголовков имеют расширения "h" и "hpp".

\$LIB\$

Полный путь к папке стандартных библиотек компилятора, указанной в настройках модуля, без символа обратной косой черты ("\") в конце. Обычно файлы библиотек имеют расширения "lib" и "obj".

\$LINK\$

Полный путь к исполняемому файлу редактора связей, указанный в настройках модуля. Некоторые компиляторы способны сами вызывать редактор связей по окончании компиляции – в этом случае путь к редактору связей в настройках модуля может не указываться, и это символическое обозначение использовать нельзя.

\$MODELDIR\$

Полный путь к папке, внутри которой размещается файл модели блока, без символа обратной косой черты ("\") в конце. Файл модели имеет расширение "mdl", на его основе модуль автокомпиляции формирует текст программы, который затем передается компилятору.

\$OBJFILE\$

Имя скомпилированного объектного файла без пути с расширением "obj". Этот файл создается компилятором и передается редактору связей для окончательной сборки DLL.

\$POSTEXE\$

Полный путь к исполняемому файлу дополнительной программы, вызываемой после редактора связей, если этот путь задан в настройках модуля. Большинство компиляторов не требует никаких дополнительных программ, и этот путь задается редко.

#### \$RDSINCLUDE\$

Полный путь к папке заголовков РДС, указанной в настройках РДС (см. §2.18 части I), без символа обратной косой черты (“\”) в конце. Эти заголовки необходимы для успешной компиляции моделей, поскольку в них описаны типы, структуры и классы, используемые модулем при автоматическом формировании текста программы модели.

#### \$RDSTEMP\$

Полный путь к папке временных файлов, указанной в настройках РДС, без символа обратной косой черты (“\”) в конце. Именно в этой папке модуль автокомпиляции формирует временные файлы, включая файл с параметрами (response file), описанный ниже.

#### \$RESPONSE\$

Имя файла с параметрами компилятора или редактора связей, если он используется. Многие компиляторы позволяют вместо указания всех параметров непосредственно в командной строке запуска записать их в текстовый файл (т.н. response file), а в командной строке указать только имя этого файла. При большом количестве параметров этот путь предпочтительнее, поскольку размер командной строки ограничен и его нельзя превышать. Настройки запуска компилятора и редактора связей (см. §3.9.6 на стр. 342) позволяют организовать передачу параметров обоими способами.

Для пояснения работы описанных символических обозначений рассмотрим небольшой пример. Допустим, что мы используем модуль автокомпиляции, предназначенный для работы с компилятором Borland C++ 5.5. На вкладке “компилятор” окна настройки этого модуля заданы следующие пути (следует учитывать, что у модулей, рассчитанных на конкретные компиляторы, названия полей ввода на вкладке немного отличаются от универсальных, изображенных на рис. 489, хотя смысл их сохраняется):

- папка установки компилятора (папка Borland C++) – “C:\Program Files\bcpp55”;
- путь к компилятору – “\$BC\$\bin\bcc32.exe”;
- путь к редактору связей – “\$BC\$\bin\ilink32.exe”;
- папка заголовков (H-файлов) – “\$BC\$\Include”;
- папка библиотек – “\$BC\$\Lib”.

Пусть РДС находится в папке “C:\Rds”, и в настройках РДС на вкладке “папки” указаны следующие пути:

- файлы заголовков – “\$RDS\$\Include\”;
- настройки – “\$RDS\$”;
- временные файлы – “\$INI\$\Temp\”.

Пусть теперь мы компилируем модель “model.mdl”, находящуюся в папке “C:\User\Schemes\”. При вызове компилятора символические обозначения, перечисленные выше, будут заменены на следующие значения:

| <b>Обозначение</b> | <b>Подставляемое значение</b>                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------|
| \$AUXDIR\$         | Нет (для Borland C++ 5.5 дополнительная папка не задается)                                             |
| \$BC\$             | “C:\Program Files\bcpp55”                                                                              |
| \$COMP\$           | “C:\Program Files\bcpp55\bin\bcc32.exe”                                                                |
| \$CPPFILE\$        | “model.cpp” (имя файла исходного текста формируется из имени файла модели заменой расширения на “cpp”) |
| \$DLLFILE\$        | “model.dll”                                                                                            |
| \$FILE\$           | “model”                                                                                                |

| <i>Обозначение</i> | <i>Подставляемое значение</i>                                              |
|--------------------|----------------------------------------------------------------------------|
| \$FUNC\$           | "rdsbcppBlockEntryPoint" (это имя жестко встроено в модуль автокомпиляции) |
| \$FUNCLC\$         | "rdsbcppblockentrypoint"                                                   |
| \$FUNCUC\$         | "RDSBCPPBLOCKENTRYPOINT"                                                   |
| \$INCLUDE\$        | "C:\Program Files\bcpp55\Include"                                          |
| \$LIB\$            | "C:\Program Files\bcpp55\Lib"                                              |
| \$LINK\$           | "C:\Program Files\bcpp55\bin\ilink32.exe"                                  |
| \$MODELDIR\$       | "C:\User\Schemes"                                                          |
| \$OBJFILE\$        | "model.obj"                                                                |
| \$POSTEXE\$        | Нет (для Borland C++ 5.5 дополнительная программа не задается)             |
| \$RDSINCLUDE\$     | "C:\Rds\Include"                                                           |
| \$RDSTEMP\$        | "C:\Rds\Temp"                                                              |
| \$RESPONSE\$       | "response.txt" (это имя жестко встроено в модуль автокомпиляции)           |

Из таблицы видно, что вместо символических обозначений всегда подставляются полностью развернутые значения. Вместо обозначения "\$COMP\$", например, подставится не строка "\$BC\$\bin\bcc32.exe", указанная в настройках в качестве пути к компилятору, а, поскольку "\$BC\$" тоже является обозначением, строка "C:\Program Files\bcpp55\bin\bcc32.exe". Точно так же путь к папке временных файлов РДС, обозначаемый "\$RDSTEMP\$", будет развернут до полного пути "C:\Rds\Temp", несмотря на то, что в настройках РДС он указан со специфическим для РДС обозначением "\$INI\$". Полный список символических обозначений стандартных папок РДС, отличающихся от обозначений модуля автокомпиляции, рассматриваемых в этом параграфе, приводится в руководстве программиста [1].

Для вызова компилятора в настройках модуля, работающего с Borland C++ 5.5, используется следующая строка:

```
"$COMP$" @"$RDSTEMP$\$RESPONSE$"
```

При этом для файла параметров (response file) задан такой текст:

```
-I"$INCLUDE$;$INCLUDE$\sys"
-I"$RDSINCLUDE$"
-I"$MODELDIR$"
-O2 -Vx -Ve -X- -a8 -k- -vi -tWD -tWM -c -w-inl -w-aus -q
"$CPPFILE$"
```

С учетом описанных выше подстановок, в Windows для вызова компилятора будет передана строка

```
"C:\Program Files\bcpp55\bin\bcc32.exe"
@"C:\Rds\Temp\response.txt"
```

В файл параметров "response.txt" в папке "C:\Rds\Temp" модуль запишет следующий текст:

```
-I"C:\Program Files\bcpp55\Include;C:\Program
Files\bcpp55\Include\sys"
-I"C:\Rds\Include"
-I"C:\User\Schemes"
-O2 -Vx -Ve -X- -a8 -k- -vi -tWD -tWM -c -w-inl -w-aus -q
"model.cpp"
```

В этой командой строке и этом файле параметров указаны полные пути ко всем папкам и конкретные имена файлов, которые компилятор сможет обработать.

### §3.9.5. Настройка переменных окружения компилятора

Описывается ввод значений для переменных окружения Windows, которые необходимы некоторым компиляторам для работы.

Переменные окружения (переменные среды, environment variables) – это текстовые переменные операционной системы, в которых, как правило, хранится различная информация о настройках системы и отдельных программ. Некоторым компиляторам для нормальной работы необходима настройка таких переменных, в которых обычно указываются пути к различным папкам и исполняемым файлам. Имена переменных и значения, которые им необходимо присвоить, должны быть указаны в описании конкретного компилятора.

Модуль автокомпиляции позволяет индивидуально настроить переменные окружения для запуска компилятора. Для того, чтобы ввести значения этих переменных, следует вызвать окно настройки модуля (см. стр. 328) и на его вкладке “компилятор” нажать кнопку “дополнительно”. В открывшемся дополнительном окне настроек следует выбрать вкладку “окружение” (рис. 491).

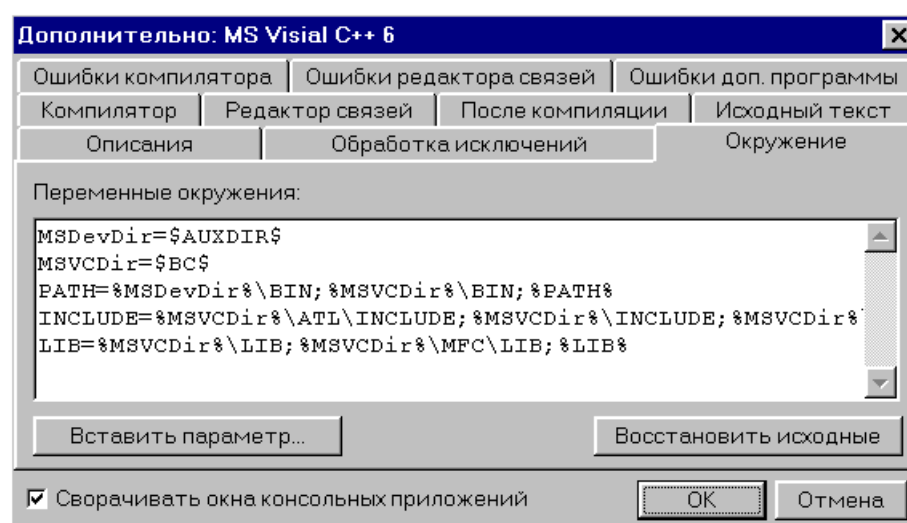


Рис. 491. Настройка переменных окружения компилятора  
(на примере Microsoft Visual C++)

На вкладке расположено большое многострочное поле ввода, в котором вводятся значения переменных в формате “*имя=значение*”, по одной переменной на строку. В правой части выражений можно использовать значения других переменных окружения, заключив их имена между знаками процентов (“%”). Например, выражение

```
PATH=C:\Compilers\VC\bin;%PATH%
```

добавит в начало переменной “PATH” строку “C:\Compilers\VC\bin;”. Такой же синтаксис применяется в команде “SET” операционной системы, с помощью которой устанавливаются значения переменных окружения в командных файлах – например, в “autoexec.bat”.

В правой части выражений можно также использовать любые символические обозначения, описанные в §3.9.4 (стр. 337) – при установке переменных окружения перед запуском компилятора эти обозначения будут заменены их значениями. Чтобы не набирать эти обозначения вручную, можно установить курсор в том месте поля ввода, куда нужно вставить обозначение, нажать кнопку “вставить параметр” в левой нижней части вкладки, и выбрать нужное обозначение в появившемся меню (рис. 492).

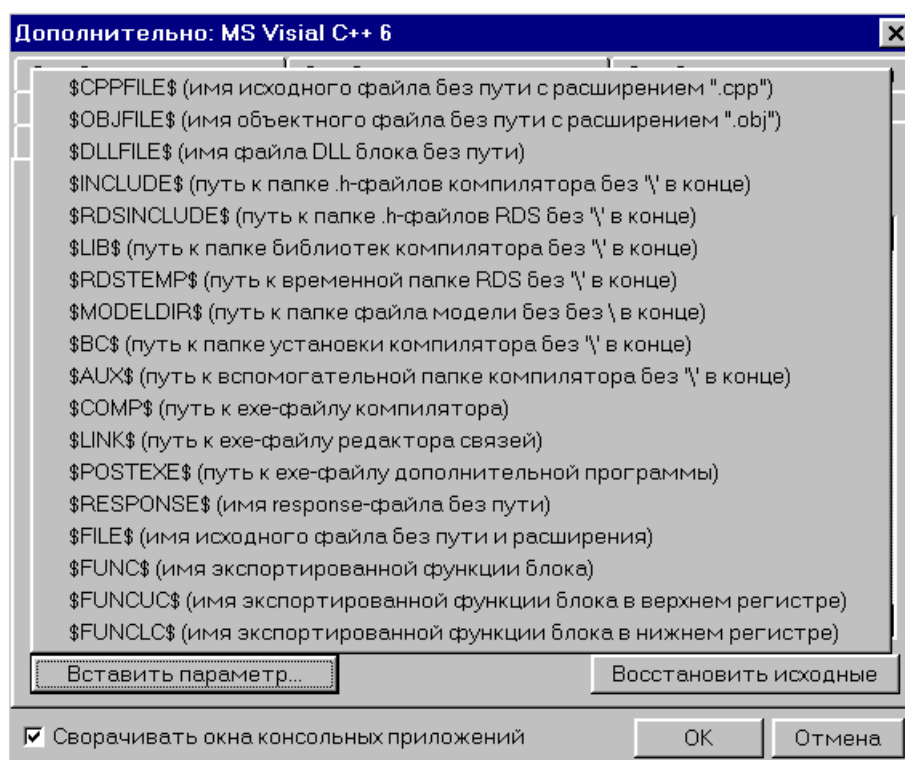


Рис. 492. Меню вставки параметра (символического обозначения)

Кнопка “восстановить исходные”, расположенная в правой нижней части вкладки, записывает в поле ввода переменных окружения набор значений, которые разработчики модуля считают подходящими для компилятора, для которого предназначен настраиваемый модуль. При настройке универсальных модулей эта кнопка просто очищает поле ввода.

### §3.9.6. Запуск компилятора и редактора связей

Описывается настройка командной строки для запуска компилятора и редактора связей.

Для того, чтобы модуль мог скомпилировать модель блока, мало указать пути к исполняемым файлам компилятора и редактора связей на вкладке “компилятор” окна его параметров (см. §3.9.3 на стр. 333). Нужно также задать параметры командной строки для запуска этих программ со всеми параметрами, необходимыми для правильной компиляции DLL Windows.

Для того, чтобы задать командную строку запуска компилятора, следует вызвать окно настройки модуля (см. стр. 328) и на его вкладке “компилятор” нажать кнопку “дополнительно”. В открывшемся дополнительном окне настроек следует выбрать вкладку “компилятор” (рис. 493).

В верхней части вкладки вводится командная строка для запуска компилятора. В нее должно входить имя исполняемого файла с полным путем к нему и, после него, все необходимые параметры (иногда называемые ключами). В командной строке можно использовать любые символические обозначения файлов и папок, описанные в §3.9.4 (стр. 337) – таким образом, для указания полного пути к исполняемому файлу компилятора достаточно написать “\$COMP\$”.

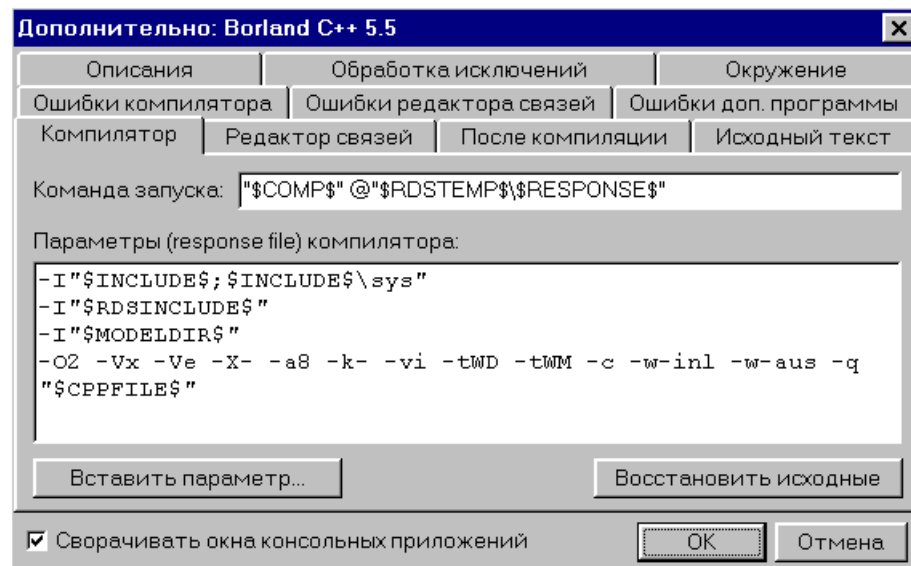


Рис. 493. Настройка запуска компилятора (на примере Borland C++ 5.5)

Любые пути в командной строке следует вводить в двойных кавычках. Это связано с тем, что разделителем параметров в командной строке Windows является пробел, и, если пробел окажется внутри пути к файлу, строка будет разобрана неправильно. Допустим, например, что мы хотим запустить компилятор “bcc32.exe”, находящийся в папке “C:\Program Files\bcpp55\bin\”. Можно попытаться записать командную строку в следующем виде:

```
C:\Program Files\bcpp55\bin\bcc32.exe
```

При этом пробел в имени папки “Program Files” будет распознан как разделитель параметров, и Windows будет пытаться запустить исполняемый файл “C:\Program” и передать ему параметр “Files\bcpp55\bin\bcc32.exe”, что, разумеется, не увенчается успехом. Ситуация изменится, если заключить строку в двойные кавычки:

```
"C:\Program Files\bcpp55\bin\bcc32.exe"
```

Двойные кавычки дадут Windows понять, что весь текст между ними является одним параметром, и будет запущен правильный файл.

На всякий случай, командную строку всегда следует завершать одним лишним пробелом. Хотя это и не является обязательным требованием, некоторые программы неправильно разбирают переданные им параметры без этого пробела.

Командная строка должна начинаться с полного пути к исполняемому файлу компилятора (то есть с текста “\$COMP\$” в двойных кавычках), за которым следуют все параметры этого компилятора, необходимые для компиляции DLL Windows – их можно узнать из описания к компилятору. Перед настройкой модуля автокомпиляции настоятельно рекомендуется научиться компилировать модели блоков вручную – так можно проверить действие параметров компилятора и определить, какие именно параметры требуются. Все пути, присутствующие в параметрах, необходимо заключать в двойные кавычки по указанной выше причине.

Параметры можно указывать не только в командной строке, длина которой ограничена. Многие компиляторы позволяют записать параметры в текстовый файл (так называемый response file), а в командной строке указать только имя этого файла. При большом количестве параметров этот путь предпочтительнее. Текстовый файл с параметрами всегда создается модулем автокомпиляции в папке временных файлов РДС, поэтому в качестве его имени с полным путем можно использовать сочетание символических обозначений “\$RDSTEMP\$\\$RESPONSE\$”. Как правило, для указания использования такого

файла в командной строке компилятора предусмотрен специальный параметр – например, в Borland C++ 5.5 полный путь к файлу параметров указывается после символа “@”.

Содержимое файла параметров вводится в большом многострочном поле непосредственно под полем для командной строки (см. рис. 493 выше). В этом поле тоже можно использовать любые символические обозначения файлов и папок. Чтобы не вводить эти обозначения вручную, можно установить курсор в том месте поля ввода командной строки или файла параметров, в котором нужно вставить обозначение, нажать кнопку “вставить параметр” в левой нижней части вкладки, и выбрать нужное обозначение в появившемся меню (см. рис. 492 на стр. 342). Кнопка “восстановить исходные” присваивает командной строке и тексту файла параметров значения по умолчанию, подходящие для компилятора, для которого предназначен модуль (в универсальных модулях оба поля просто очищаются).

После завершения работы компилятора исходный текст программы, сформированный модулем, преобразуется в объектный файл. Для получения из него DLL необходимо обработать его редактором связей. Некоторые компиляторы способны сами вызывать редактор связей, но, чаще всего, приходится настраивать его вызов вручную. Для этого служит вкладка “редактор связей” (рис. 494) окна дополнительных настроек модуля, открывающегося кнопкой “дополнительно” вкладки “компилятор” основного окна настроек (см. стр. 328).

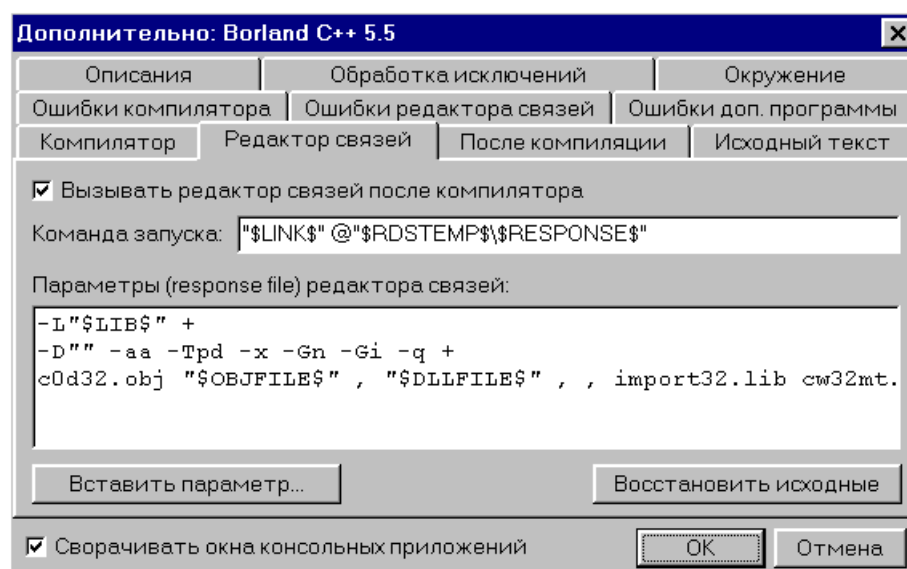


Рис. 494. Настройка запуска редактора связей (на примере Borland C++ 5.5)

В верхней части вкладки находится флажок “вызывать редактор связей после компилятора” – для отдельного запуска редактора связей он должен быть установлен. Если компилятор сам вызывает редактор связей, флажок устанавливать не нужно, но в этом случае все параметры, необходимые для сборки DLL, должны быть указаны в параметрах вызова компилятора согласно его описанию.

Ниже флажка располагаются поля ввода для командной строки редактора связей и файла параметров, если он используется. Файл параметров для редактора связей, как и для компилятора, создается модулем автокомпиляции в папке временных файлов РДС. В обоих полях ввода можно использовать символические обозначения файлов и папок (см. §3.9.4 на стр. 337) – их можно вводить вручную или выбирать из меню, открывающегося при нажатии кнопки “вставить параметр” в нижней части вкладки. Кнопка “восстановить исходные” присваивает обоим полям значения по умолчанию.



После завершения работы редактора связей должен быть создан исполняемый файл DLL с моделью блока, пригодный для подключения к РДС. Как правило, никаких дополнительных действий с этим файлом делать не нужно. Тем не менее, модуль автокомпиляции поддерживает запуск еще одной, дополнительной, программы после редактора связей. Необходимость ее запуска, ее командная строка и содержимое файла параметров, если он нужен, задаются на вкладке “после компиляции” в том же окне (рис. 495). Ни один из стандартных компиляторов не требует запуска такой дополнительной программы.

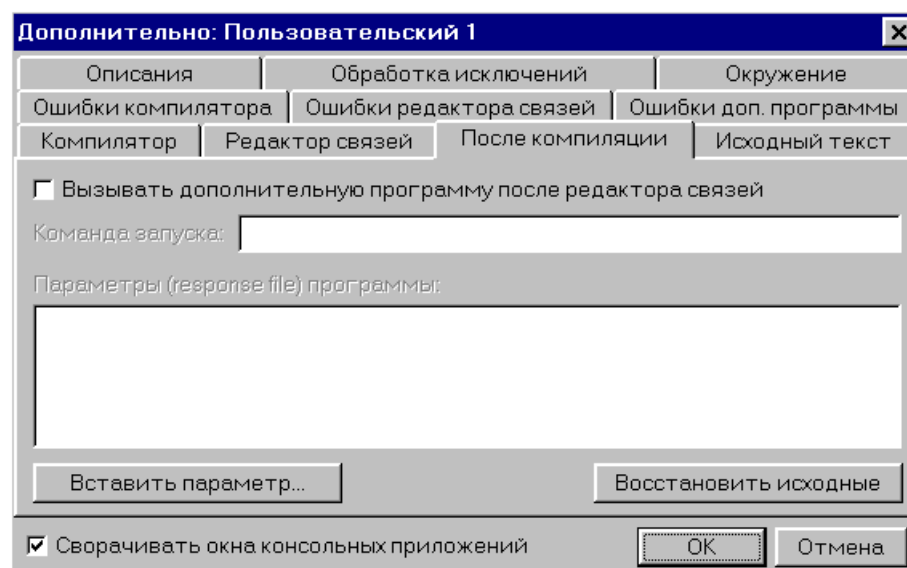


Рис. 495. Настройка запуска дополнительной программы

В нижней части всего дополнительного окна настроек, слева от кнопок “ОК” и “отмена”, находится включенный по умолчанию флажок “сворачивать окна консольных приложений”. Его установка позволяет “спрятать” дополнительные окна, открывающиеся в момент запуска компилятора и редактора связей. Эти окна, которые открываются на короткое время и, как правило, имеют черный цвет, могут сбить пользователя с толку, поэтому лучше держать их скрытыми. В них обычно нет информации, которая могла бы пригодиться пользователю – при правильной настройке модуля автокомпиляции сообщения об ошибках, выдаваемые компилятором и редактором связей, будут разобраны и показаны пользователю в окне редактора модели.

### §3.9.7. Разбор ошибок компиляции

Описывается настройка автоматического разбора создаваемых компилятором отчетов для выделения из них сообщений об ошибках. Эти сообщения показываются пользователю непосредственно в окне редактора компилируемой модели.

Если в процессе компиляции модели в ее тексте обнаружены какие-либо ошибки, модуль автокомпиляции должен показать список этих ошибок пользователю в окне редактора на специальной панели под вкладками с текстом реакций модели на события (см. рис. 329 на стр. 32). При этом желательно, чтобы двойной щелчок на ошибке в списке автоматически открывал вкладку с фрагментом исходного текста, в котором она возникла, и устанавливал курсор в строку с ошибкой. Для этого модуль должен уметь разбирать отчеты, выдаваемые компилятором и редактором связей.

Практически все компиляторы выдают текстовые отчеты о своей работе. Эти тексты имеют сложную структуру: обычно они содержат имя и версию компилятора, имена обработанных файлов, обнаруженные ошибки, сводные данные о результатах компиляции и

т.п. В настройках модуля автокомпиляции задаются правила обработки таких текстов. Необходимо указать, откуда модуль должен считывать эти тексты отчетов, как в них отмечены строки, содержащие ошибки и предупреждения, где в этих строках находится номер строки исходного текста, а где – описание ошибки. Поскольку редактор связей чаще всего является отдельной программой со своим форматом отчетов, правила разбора текстов для него задаются отдельно. Настройка разбора отчетов – довольно сложная задача, поэтому, перед тем, как приступить к ней, настоятельно рекомендуется изучить описание используемого компилятора и провести несколько компиляций вручную, специально внося в текст программы различные ошибки и наблюдая, как они отражаются в отчетах.

Для того, чтобы задать правила разбора отчета компилятора, следует вызвать окно настройки модуля (см. стр. 328) и на его вкладке “компилятор” нажать кнопку “дополнительно”. В открывшемся дополнительном окне настроек следует выбрать вкладку “ошибки компилятора” (рис. 496).

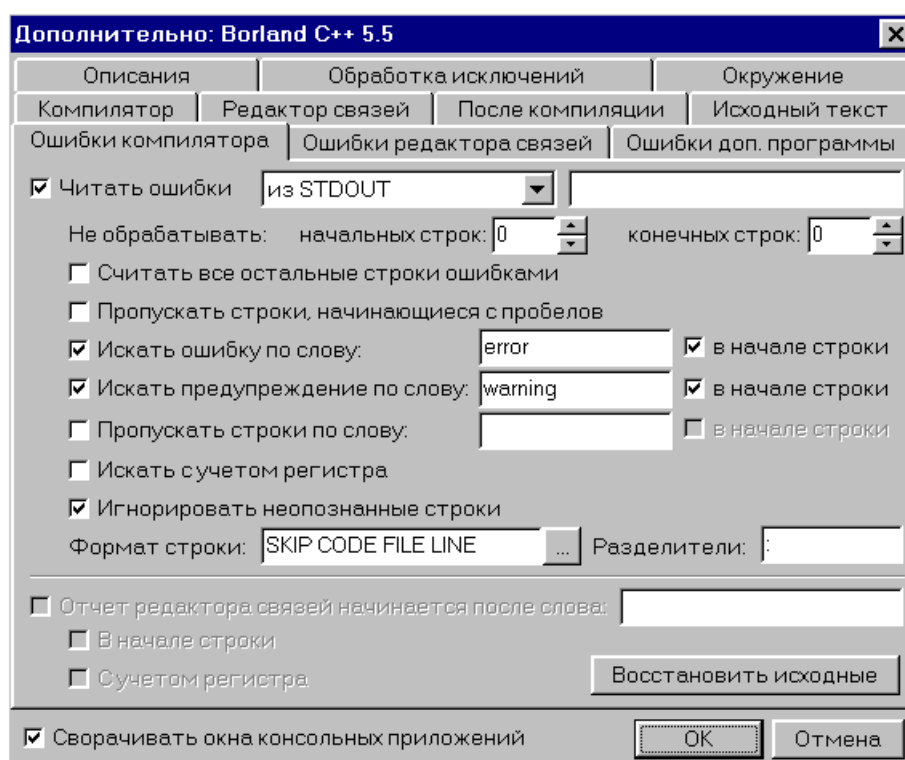


Рис. 496. Настройка разбора ошибок компилятора (на примере Borland C++ 5.5)

Самый верхний флажок на вкладке, называющийся “читать ошибки”, разрешает обработку отчета компилятора. Если его не установить, модуль автокомпиляции не будет обрабатывать отчет. Пользователь в этом случае сможет узнать о том, что в его программе есть ошибки, только по сообщению о невозможности создать DLL с моделью, которое модуль выдает самостоятельно, без анализа работы компилятора. Этого следует всячески избегать, поскольку пользователю в этом случае будет очень сложно обнаружить ошибку. Нормально настроенный модуль должен обрабатывать ошибки, и этот флажок должен быть установлен.

Справа от флажка находится выпадающий список, в котором нужно указать, куда именно компилятор выводит отчет о своей работе. Следует выбрать один из трех вариантов: “из STDOUT”, “из файла” или “из STDERR”. Большинство компиляторов – консольные приложения Windows, а консольные приложения обычно работают с тремя стандартными потоками ввода-вывода: потоком ввода данных “stdin”, потоком вывода данных “stdout” и потоком вывода сообщений об ошибках “stderr”. Если компилятор выводит сообщения об ошибках в потоки “stdout” или “stderr”, модуль автокомпиляции, запустивший компилятор,

может перед его запуском перенаправить эти потоки и получить из них данные, которые будут анализироваться. В какой именно поток компилятор выводит сообщения об ошибках, следует узнать из его описания или определить опытным путем, скомпилировав программу с ошибкой и перенаправив стандартные потоки вручную (перенаправление вывода в командной строке производится при помощи символа “>”, подробнее об этом можно узнать в описании пользователя Windows). В выпадающем списке при этом следует выбрать вариант “из STDOUT” или “из STDERR”, в зависимости от используемого компилятором потока. Если же компилятор выводит отчет не в стандартный поток, а в отдельный файл, в выпадающем списке следует выбрать вариант “из файла”, а в поле ввода справа от списка указать имя этого файла (в имени можно использовать символические обозначения путей, рассмотренные в §3.9.4 на стр. 337).

Ниже находятся два поля ввода: “не обрабатывать — начальных строк” и “конечных строк”, которые позволяют исключить из рассмотрения заданное число начальных и конечных строк отчета компилятора. В начальных строках часто выводится название и версия компилятора, а в конечных — сводные данные о результатах компиляции. К ошибкам эти строки не имеют прямого отношения, поэтому их обрабатывать не нужно.

Далее расположен флажок “считать все остальные строки ошибками”. Его следует устанавливать только в том случае, если сообщения об ошибках компилятора не имеют четкой формальной структуры и не могут анализироваться способами, описанными ниже. В этом случае любая строка отчета, не входящая в число пропускаемых начальных и конечных, будет показана пользователю как сообщение об ошибке без привязки к какой-либо строке введенного им текста. Это очень неудобно, поскольку поиск ошибки в тексте в этом случае ложится на пользователя, и установки этого флажка следует, по возможности, избегать.

Флажок “пропускать строки, начинающиеся с пробелов”, как следует из его названия, исключает из рассмотрения строки отчета компилятора, которые начинаются с пробелов или знаков табуляции. В таких строках обычно содержится различная дополнительная информация, которую можно не анализировать.

Ниже расположены три похожих флажка, управляющие выделением из отчета интересующих модуль строк: “искать ошибку по слову”, “искать предупреждение по слову” и “пропускать строки по слову”. После каждого из этих флажков находится поле ввода, в котором указывается искомое слово, и дополнительный флажок, требующий нахождения этого слова в начале строки отчета. Кроме того, под этими флажками находится флажок “искать с учетом регистра”, управляющий поиском слова в строке.

Если установлен флажок “искать ошибку по слову”, сообщениями об ошибках будут считаться только строки отчета, содержащие указанное после флажка слово (с учетом или без учета регистра символов, определяется флажком “искать с учетом регистра”). При этом, если установлен дополнительный флажок “в начале строки”, строки, в которых найденное слово не находится в начале строки отчета, будут исключены из рассмотрения. Точно так же работает флажок “искать предупреждение по слову”. Флажок “пропускать строки по слову” исключает их рассмотрения строки, содержащие заданное слово, даже если они содержат слова, указанные в качестве слов-признаков ошибок и предупреждений (так можно пропускать диагностические сообщения компилятора). Настройки, приведенные в качестве примера на рис. 496, указывают модулю считать сообщениями об ошибках любые строки, начинающиеся со слова “error” в любом регистре (то есть подойдут и “ERROR”, и “Error”), а предупреждениями — строки, начинающиеся со слова “warning”. Пропуск строк по слову в этих настройках отключен.

Установка флажка “игнорировать неопознанные строки” исключает из рассмотрения строки отчета, в которых не оказалось ни слова, отмечающего сообщение об ошибке, ни слова, отмечающего предупреждение. Без установки одного из флажков поиска, описанных выше, установка этого флажка не рекомендуется, поскольку это заставит модуль игнорировать все строки отчета, так как все они окажутся неопознанными.

Далее располагаются два поля ввода, управляющие анализом опознанных строк отчета, то есть строк, в которых обнаружено либо слово, отмечающее ошибку, либо слово, отмечающее предупреждение (для этого хотя бы один из флажков “искать ошибку по слову” и “искать предупреждение по слову” должен быть установлен). В поле ввода “формат строки” через пробел указываются условные обозначения, отмечающие последовательность слов в сообщениях об ошибке и предупреждениях. Можно использовать следующие обозначения:

- FILE – имя файла исходного текста;
- LINE – номер строки исходного текста;
- CODE – код ошибки;
- MARK – пропустить все слова до обнаруженного слова-маркера ошибки или предупреждения (то есть одного из слов, указанных после флажков “искать ошибку по слову” и “искать предупреждение по слову”) включительно;
- SKIP – пропуск одного слова.

Эти обозначения можно как вводить вручную с клавиатуры, так и выбирать из меню, открываемого кнопкой с многоточием рядом с полем.

Справа от поля “формат строки”, в поле “разделители”, подряд вводятся символы, которые в строке отчета должны считаться разделителями слов. Пробел и символ табуляции всегда считаются разделителями, и их не нужно указывать отдельно. Строка отчета, опознанная как сообщение об ошибке или предупреждение, разбивается на слова согласно указанному списку разделителей, а затем ее начало разбирается согласно обозначениям из поля “формат строки”: оттуда извлекается код ошибки, номер строки и т.п. Не разобранный конец строки считается описанием ошибки и показывается пользователю.

На рис. 496 формат строки указан в виде “SKIP CODE FILE LINE”, а в качестве разделителя указан символ двоеточия. Проиллюстрируем разбор сообщения об ошибке на примере следующего отчета компилятора Borland C++ 5.5:

```
src_p2_4.cpp:
Error E2303 src_p2_4.cpp 20: Type name expected
*** 1 errors in Compile ***
```

Согласно настройкам на рисунке, в этом тексте только вторая строка будет опознана модулем как сообщение об ошибке: в ее начале содержится слово “error”. Таким образом, только она будет разобрана по формату “SKIP CODE FILE LINE”. Разбор будет выглядеть следующим образом:

|         |        |              |        |             |                       |
|---------|--------|--------------|--------|-------------|-----------------------|
| SKIP    | CODE   | FILE         | LINE   |             | остаток строки        |
| Error   | E2303  | src_p2_4.cpp | 20     | :           | Type name expected    |
| пропуск | код    | имя файла    | номер  | разделитель | текст описания ошибки |
|         | ошибки |              | строки |             |                       |

Ошибка опознана как ошибка “type name expected” (“требуется имя типа”) с кодом E2303 в строке номер 20. На данный момент извлеченные из строки код ошибки и имя файла никак не используются модулем автокомпиляции, используется только номер строки и текст описания. В окне редактора пользователь увидит сообщение об этой ошибке в следующем виде:

```
[Ошибка C++] Описания (2): Type name expected
```

Здесь “описания” – название фрагмента введенного пользователем фрагмента текста модели, а 2 – номер строки в этом тексте. Модуль автоматически находит фрагмент текста по номеру строки из сообщения об ошибке и выводит в сообщении номер строки не во всем автоматически сформированном тексте программы, который обработал компилятор, а внутри этого найденного фрагмента. При двойном щелчке на этом сообщении автоматически откроется вкладка глобальных описаний модели (см. §3.8.1 на стр. 282) и курсор будет установлен на вторую строку этой вкладки, то есть на ту строку, в которой компилятор нашел ошибку.

В нижней части вкладки находится кнопка “восстановить исходные”, заполняющая вкладку параметрами, которые разработчики модуля считают подходящими для компилятора, для которого предназначен настраиваемый модуль. При настройке универсальных модулей использовать эту кнопку не имеет смысла.

Чаще всего редактор связей для преобразования скомпилированного объектного файла в DLL вызывается как отдельная программа (см. §3.9.6 на стр. 342). Однако, некоторые компиляторы способны сами вызывать редактор связей, и, в этом случае, у компилятора и редактора связей будет один общий отчет. Если редактор связей запускается самим компилятором, то есть если на вкладке “редактор связей” не установлен флажок “вызывать редактор связей после компилятора” (см. рис. 494 на стр. 344), на вкладке “ошибки компилятора” под горизонтальной чертой будет активна группа флажков, задающая разбиение общего отчета на отчет компилятора и отчет редактора связей (рис. 497).

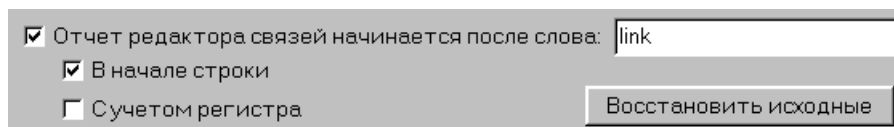


Рис. 497. Настройка выделения отчета редактора связей

Общий отчет всегда начинается с отчета компилятора. Отчет редактора связей обычно начинается с какого-то текста – например, с имени редактора связей. Этот текст можно указать в поле ввода после флажка “отчет редактора связей начинается после слова”. При этом флажок “в начале строки” будет требовать того, чтобы этот текст находился в начале строки отчета (обычно так и бывает), а флажок “с учетом регистра” будет требовать совпадения регистра всех символов найденного текста с введенным образцом. Как только строка с указанным текстом будет найдена, все строки после нее будут считаться отчетом редактора связей и обрабатываться по другим правилам.

Правила обработки отчета редактора связей (не важно, вызывается он отдельно или запускается автоматически самим компилятором) задаются на вкладке “ошибки редактора связей” того же окна дополнительных настроек модуля (рис. 498).

Поля этой вкладки совпадают с полями вкладки разбора ошибок компилятора (см. рис. 496 на стр. 346) и разбор отчета производится по таким же правилам. Следует учитывать, что редактор связи работает не с исходным текстом программы, а с уже скомпилированным объектным файлом, поэтому в его сообщениях об ошибках номера строк, как правило, отсутствуют. Редактор связей может, например, определить, что какая-то функция описана в программе, но тело ее отсутствует – у такой ошибки в принципе не может быть номера строки. По этой причине ошибки редактора связей, чаще всего, не разбираются и показываются пользователю “как есть”. Кроме того, если файл не скомпилирован из-за ошибок, редактор связей тоже выдаст ошибку из-за отсутствия объектного файла, который он должен обработать. Такие вторичные ошибки сами исчезают после исправления ошибок, о которых сообщил компилятор.

Ранее был приведен пример отчета компилятора с ошибкой “type name expected” (“требуется имя типа”) – такая ошибка делает невозможным компиляцию и, таким образом, создание объектного файла. В результате пользователь, допустивший ее, увидит сразу три сообщения об ошибках:

```
[Ошибка C++] Описания (2): Type name expected
[Ошибка ilink32]: Fatal: Unable to open file 'src_p2_4.obj'
[Анализ]: Файл DLL не создан
```

Первая строка в этом списке – обнаруженная и разобранный ошибка компилятора, уже рассмотренная выше. Вторая строка – обнаруженная ошибка редактора связей, который не смог открыть объектный файл, не созданный компилятором из-за первой ошибки. Третья строка – сообщение самого модуля о том, что, по каким-то причинам, в результате

компиляции не был создан файл DLL с моделью блока. Это третье сообщение не связано с разбором отчетов компилятора и редактора связей, модуль выводит его автоматически. Если отключить в параметрах модуля анализ отчетов, это сообщение будет единственным, которое увидит пользователь.

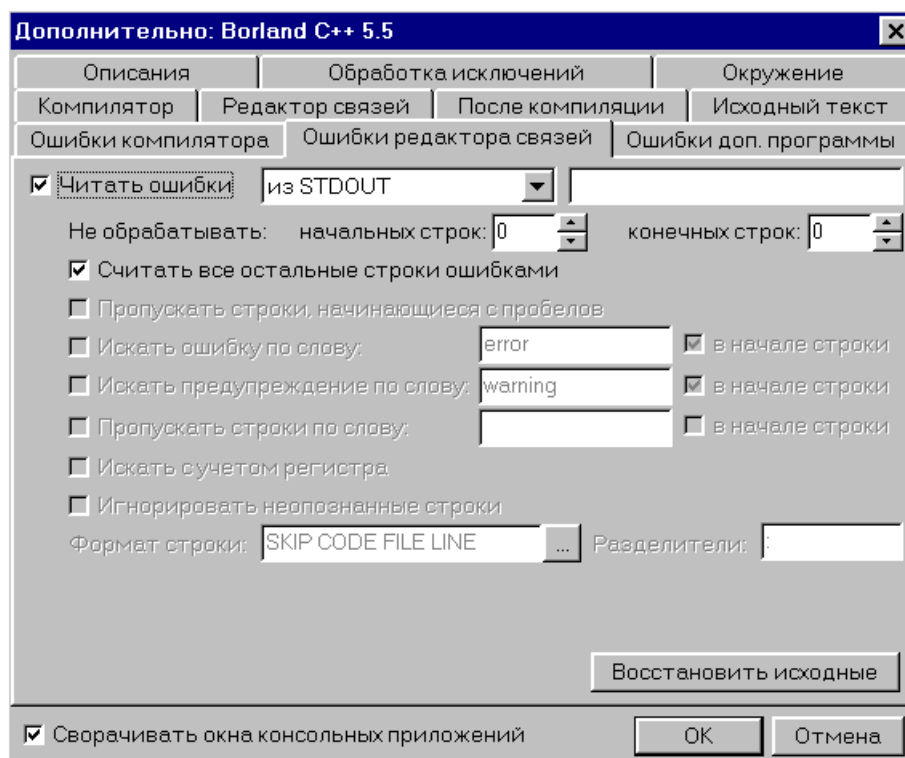


Рис. 498. Настройка разбора ошибок редактора связей

Если в параметрах модуля автокомпиляции задан запуск дополнительной программы после редактора связей (см. рис. 495 на стр. 345), модуль может также разобрать отчет о работе, формируемый этой программой. Правила его разбора задаются на вкладке “ошибки доп.программы” того же окна дополнительных настроек модуля. Все поля этой вкладки совпадают с полями вкладки разбора отчета редактора связей (см. рис. 498 на стр. 350) и имеют тот же смысл, поэтому она здесь не описывается.

В процессе настройки модуля на разбор отчетов нестандартного компилятора большую помощь может оказать функция показа отчета компилятора, которую можно вызвать пунктом меню “модель | показать отчет компилятора” из окна редактора модели. Если запуск компилятора и редактора связей уже настроен, эта функция покажет на отдельной вкладке командные строки и параметры запуска этих программ, а также полные тексты сформированных ими отчетов.

### §3.9.8. Общие описания в программе

Описывается ввод фрагмента исходного текста, который будет автоматически добавляться в начало каждой формируемой модулем программы. В этот текст обычно записывают команды включения стандартных заголовочных файлов, необходимых для использования функций API Windows и библиотек языка C.

Для того, чтобы модель блока могла быть успешно скомпилирована, помимо автоматически формируемых модулем описаний классов и функций необходимо включить в нее описания типов и функций Windows, математических библиотек и т.п. Эти глобальные описания – общие для всех возможных моделей, и от параметров самой модели они не зависят (модель может иметь свои собственные глобальные описания, см. §3.8.1 на стр. 282). Общие описания можно ввести в параметрах модуля – для этого следует вызвать окно

настройки модуля (см. стр. 328) и на его вкладке “компилятор” нажать кнопку “дополнительно”. В открывшемся дополнительном окне настроек следует выбрать вкладку “описания” (рис. 499).

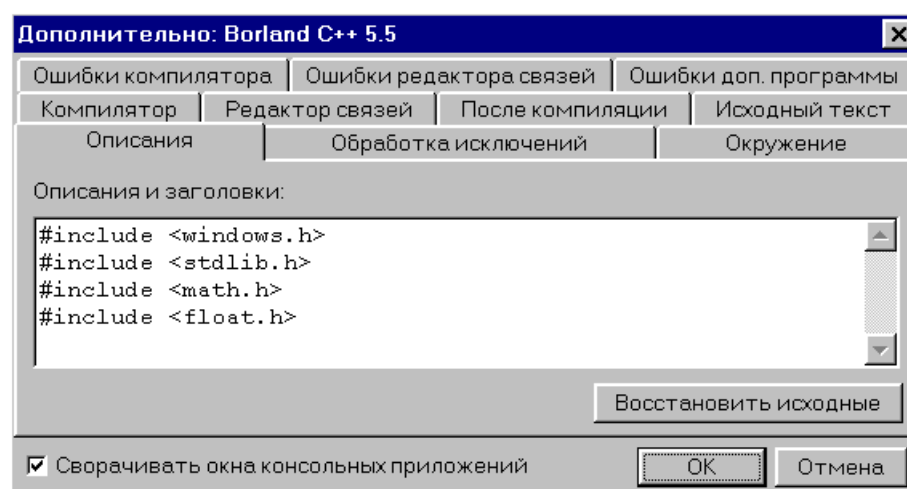


Рис. 499. Настройка общих описаний всех моделей

Эта вкладка содержит одно большое многострочное поле ввода “описания и заголовки”, в которое вводятся все необходимые описания в синтаксисе языка С. Эти описания будут вставлены в самое начало формируемого модулем исходного текста, передаваемого компилятору (см. также стр. 80 и стр. 283). Как правило, в это поле вводят команды “#include” для включения стандартных, необходимых для работы, описаний типов, констант и функций. На рисунке модуль настроен так, чтобы для каждой модели в текст включались стандартные описания Windows (“windows.h”), часто используемые библиотечные функции (“stdlib.h”), описания и функции математической библиотеки (“math.h”) и описания и функции для работы с вещественными числами (“float.h”). Эти описания необходимы для исходных текстов, формируемых стандартными модулями автокомпиляции. При желании, к ним можно добавить и другие, которые будут использоваться в большом количестве моделей. Описания, редко используемые в моделях, лучше добавлять в сами модели в раздел глобальных описаний. Добавление описаний, специфичных для РДС, настраивается на другой вкладке этого же окна – “исходный текст” (см. §3.9.9 ниже).

В нижней части вкладки находится кнопка “восстановить исходные”, записывающая в поле ввода четыре стандартных команды “#include”, изображенных на рисунке.

### §3.9.9. Параметры формирования исходного текста

Описываются различные настройки, управляющие формированием исходного текста программы по введенным пользователем фрагментам: как называется главная функция DLL, как выглядит заголовок экспортированной функции, какие описания РДС нужно автоматически включать в программу и т.п.

В настройках модуля автокомпиляции можно указать, как именно формируются те части исходного текста программы модели, за которые не отвечает пользователь: главная функция DLL, экспортированная функция модели, код инициализации глобальных переменных и т.п. Чтобы настроить эти параметры, следует вызвать окно настройки модуля (см. стр. 328) и на его вкладке “компилятор” нажать кнопку “дополнительно”. В открывшемся дополнительном окне настроек следует выбрать вкладку “исходный текст” (рис. 500).



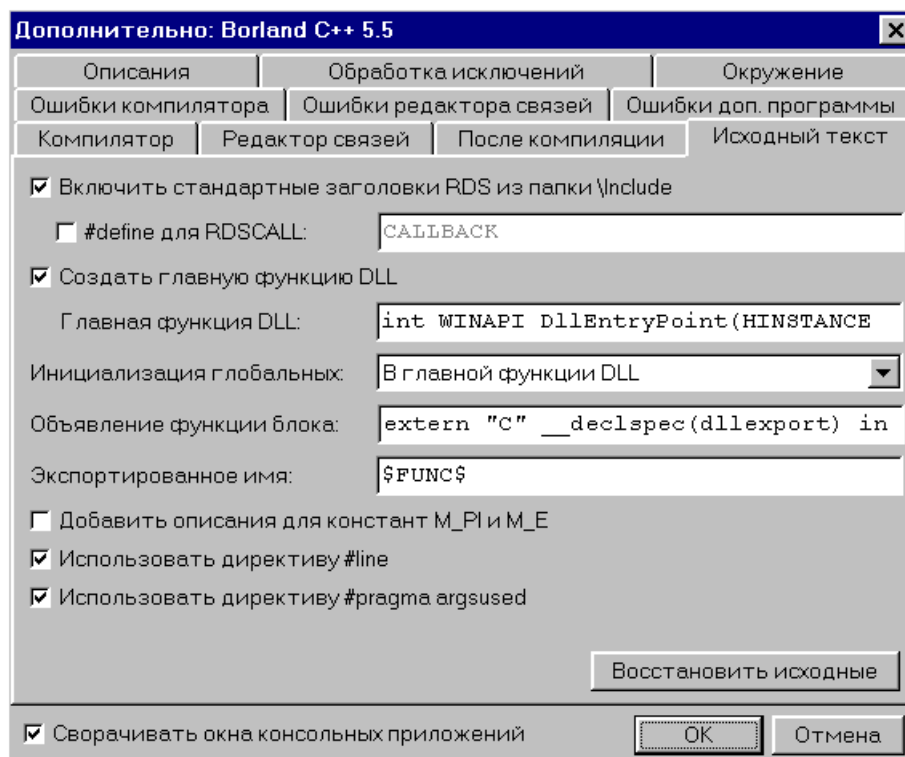


Рис. 500. Настройка параметров исходного текста (на примере Borland C++ 5.5)

Первый флажок на вкладке отвечает за автоматическое включение в текст программы файлов, содержащих описания, необходимые для взаимодействия модели блока с РДС. Включение флажка добавляет в текст непосредственно после общих описаний следующий фрагмент:

```
#include <RdsDef.h>
#define RDS_SERV_FUNC_BODY rdsbcppGetService
#include <RdsFunc.h>
#include <CommonBl.h>
#include <CommonAC.hpp>
```

Он содержит команды “#include” для стандартных файлов заголовков, входящих в состав РДС. Чтобы эти команды могли выполняться, путь к папке с этими файлами должен быть передан в параметрах запуска компилятора (имена параметров для указания папок заголовков должны быть указаны в описании компилятора). В настройках его запуска (см. §3.9.6 на стр. 342) для этого пути можно использовать символическое обозначение “\$RDSINCLUDE\$” (см. §3.9.4 на стр. 337). Перед включением файла “RdsFunc.h” расположена команда “#define”, вводящая макроопределение RDS\_SERV\_FUNC\_BODY – его наличие приведет к автоматической вставке в текст программы функции для доступа к сервисным функциям DLL, которая, в данном случае, будет называться rdsbcppGetService (подробнее об этом – в §2.2 руководства программиста [1]).

Если не устанавливать флажок “включать стандартные заголовки RDS”, приведенные выше описания добавлены не будут, а без них модель скомпилировать невозможно. Все необходимые команды в этом случае следует добавить вручную в общие описания модуля (см. §3.9.8 на стр. 350). Это менее удобно, поэтому отключать автоматическое добавление описаний РДС следует только в том случае, если в них необходимо что-либо изменить – например, ввести какие-либо настроечные макроопределения (см. §A.5.1 приложения к руководству программиста [2]).

Ниже флажка “включать стандартные заголовки RDS” расположен зависящий от него флажок “#define для RDSCALL” – если не включен первый, второй будет заблокирован.



Справа от него можно ввести значение для используемой в РДС константы RDSCALL, обозначающей тип вызова всех сервисных функций (подробнее об этом – в §A.1 приложения к руководству программиста). Если не включать в текст программы определение для RDSCALL, она будет автоматически правильно определена при включении файла “RdsDef.h” как CALLBACK – это именно тот тип, который нужен, поэтому устанавливать флажок “#define для RDSCALL” в большинстве случаев не нужно. Он оставлен в настройках для совместимости со старыми версиями файла “RdsDef.h”, в которых автоматическое определение для RDSCALL отсутствовало. Если включить его, в поле ввода справа необходимо ввести либо “CALLBACK”, либо другой тип вызова, поддерживаемый компилятором, при котором аргументы функции передаются в стеке справа налево и стек освобождается вызванной функцией. Например, для Borland C++ можно использовать тип “\_\_stdcall” (с двумя знаками подчеркивания в начале слова). При этом непосредственно перед командой “#include” для “RdsDef.h” будет вставлено описание:

```
#define RDSCALL введенный в поле текст
```

Ниже располагается флажок “создать главную функцию DLL”, при установке которого главная функция DLL будет добавлена в текст автоматически. Главная функция (точка входа) – это функция, которая вызывается при загрузке библиотеки в память и ее выгрузке (подробнее об этом – в §A.2.2 приложения к руководству программиста). Каждая DLL должна иметь главную функцию, поэтому, если не включать этот флажок, эту функцию нужно вручную добавить в общие описания модуля. Заголовок формируемой функции вводится полностью в одну строчку в поле “главная функция DLL”, расположенное непосредственно под флажком. В этот заголовок должно входить полное описание функции – тип, имя и список параметров, причем второй параметр в списке **обязательно** должен называться “reason” – внутри автоматически формируемой функции используется его значение. После этой строки заголовка модуль автокомпиляции вставляет открывающую фигурную скобку и начинает тело функции, поэтому точку с запятой в конце заголовка добавлять **не следует**. Имя главной функции DLL обычно указывается в описании используемого компилятора. Например, для Borland C++ следует использовать такой заголовок:

```
int WINAPI DllEntryPoint(HINSTANCE hinst,unsigned long reason,void
*lpReserved)
```

(здесь он не уместился на одну строчку, но в поле ввода он вводится подряд, без переводов строки). При этом автоматически сформированная главная функция DLL будет иметь примерно следующий вид:

```
#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst,unsigned long reason,void
*lpReserved)
{ if(reason==DLL_PROCESS_ATTACH)
 { if(!RDS_SERV_FUNC_BODY())
 { MessageBox(NULL,
 "Невозможно получить доступ к сервисным "
 "функциям RDS",RDSBCPP_MODELNAME,
 MB_OK | MB_ICONERROR);
 return 0;
 }
 }
 else // Есть доступ к сервисным функциям
 { rdsGetHugeDouble(&rdsbcppHugeDouble);
 // Регистрация функций блока
 rdsfuncControlValueChanged.Register(
 "Common.ControlValueChanged");
 }
}
return 1;
```

```
}
//-----
```

Конкретное наполнение тела главной функции DLL зависит от настроек модуля автокомпиляции и от наличия функций блока в создаваемой модели (см. §3.6.5 на стр. 57).

Далее располагается выпадающий список “инициализация глобальных”, который указывает модулю, в какое именно место формируемого текста программы следует добавить команды присвоения начальных значений глобальным переменным, необходимым для работы модели. К этим переменным относятся переменные-указатели на сервисные функции РДС, глобальные объекты функций блоков и т.п. Возможен выбор одного из трех вариантов:

- “в главной функции DLL” – команды присвоения начальных значений будут добавлены внутрь автоматически формируемой главной функции DLL, как в примере выше (инициализация при этом производится в момент загрузки DLL в память);
- “при первом вызове функции блока” – команды будут добавлены в автоматически формируемую функцию модели блока, изнутри которой вызываются все пользовательские реакции (инициализация при этом производится в момент первого обращения к модели блока);
- “не инициализировать” – глобальные переменные автоматически не инициализируются, все необходимые команды нужно добавлять вручную – например, в реакцию на инициализацию блока (см. §3.8.2.1 на стр. 286).

Выполнять инициализацию глобальных переменных по возможности рекомендуется в главной функции DLL. Если при настройке модуля на работу с компилятором возникли проблемы, связанные с главной функцией (например, пришлось добавлять ее вручную в общие описания модуля), инициализацию следует проводить при первом вызове функции модели блока. Выбирать в выпадающем списке вариант “не инициализировать” рекомендуется только в самом крайнем случае, когда компилятор по каким-либо причинам выдает ошибки в автоматически добавленных командах инициализации.

Следует учитывать, что, хотя регистрация функций блока тоже относится к инициализации глобальных переменных (см. §3.7.13.5 на стр. 279), она будет выполнена даже в том случае, если в выпадающем списке выбран вариант “не инициализировать”. В этом случае регистрация функций будет выполнена в конструкторе класса блока.

Под выпадающим списком выбора места инициализации глобальных переменных находятся поля ввода “объявление функции блока” и “экспортированное имя”. С их помощью задается тип и имя автоматически формируемой функции модели блока, то есть общей функции, отвечающей за всю работу блока, и имя, под которым эта функция видна “снаружи” библиотеки при ее подключении.

В поле “объявление функции блока” вводится тип и имя функции модели блока **без списка параметров** – этот список будет добавлен автоматически. Вместо имени функции следует использовать символическое обозначение “\$FUNC\$” (см. §3.9.4 на стр. 337), которое при формировании текста будет автоматически заменено на жестко встроенное в модуль автокомпиляции имя “rdsbcppBlockEntryPoint”. Перед этим именем должны находиться все описания, необходимые для того, чтобы эта функция была экспортирована из библиотеки, то есть чтобы указатель на нее можно было бы получить функцией Windows API GetProcAddress. Кроме того, эта функция должна иметь тип вызова RDSCALL.

На рис. 500 (стр. 352) приведен пример описания, подходящего для Borland C++. На рисунке видно только начало этого описания (весь текст на рисунке не уместился), но, на самом деле, в поле ввода введен следующий текст:

```
extern "C" __declspec(dllexport) int RDSCALL $FUNC$
```

Когда модуль автокомпиляции будет формировать функцию модели блока по этому описанию, она будет выглядеть следующим образом:

```
#pragma argsused
extern "C" __declspec(dllexport)
int RDSCALL rdsbcppBlockEntryPoint(
 int CallMode, // Режим вызова
 RDS_PBLOCKDATA BlockData, // Данные блока
 LPVOID ExtParam) // Дополнительные данные
{

}
```

Во введенном в поле тексте “\$FUNC\$” заменено на “rdsbcppBlockEntryPoint” и получившийся текст добавлен перед открывающей скобкой списка параметров функции, который, как и ее тело, формируется модулем автоматически согласно требованиям РДС.

В поле “экспортированное имя” вводится экспортированное имя функции модели, которое может не совпадать с именем функции в программе (изменение имен экспортированных функций называется “name mangling” и указывается в описании каждого компилятора). Именно экспортированное имя будет аргументом вызова GetProcAddress, при помощи которого РДС подключает функцию модели к блоку. В поле ввода этого имени можно использовать символические обозначения “\$FUNC\$”, “\$FUNCLC\$” и “\$FUNCUC\$” (см. §3.9.4 на стр. 337). На рис. 500 в примере для Borland C++ экспортированное имя совпадает с именем самой функции, поэтому в поле введено обозначение “\$FUNC\$”. Но, например, компилятор Digital Mars C++ по умолчанию добавляет к экспортированному имени дополнительные символы. Функция блока с именем “rdsbcppBlockEntryPoint” и требуемым списком параметров получает в этом компиляторе имя “\_rdsbcppBlockEntryPoint@12” – это можно определить опытным путем, скомпилировав модель блока с таким именем вручную и посмотрев список экспортированных функций получившегося файла DLL (как правило, в состав компилятора входят специализированные программы для такого просмотра). Таким образом, при настройке модуля на работу с Digital Mars C++ в поле “экспортированное имя” нужно вводить текст “\_ \$FUNC\$@12”.

В нижней части вкладки размещаются флажки, управляющие дополнительными описаниями и разрешающие использование некоторых директив препроцессора.

Флажок “добавить описания для констант M\_PI и M\_E” вставляет в формируемый текст программы следующие команды:

```
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
#ifndef M_E
#define M_E 2.71828182845904523536
#endif
```

При этом в программе для числа  $\pi$  можно использовать константу M\_PI, а для основания натурального логарифма – константу M\_E. Эти имена констант общеприняты, и многие компиляторы уже содержат их описания в своих стандартных файлах заголовков, поэтому, в большинстве случаев, этот флажок можно не устанавливать.

Флажок “использовать директиву #line” разрешает модулю автокомпиляции вставлять в формируемый текст директиву “#line”, изменяющую номера строк в сообщениях об ошибках. При использовании этой директивы снижается вероятность того, что модуль неправильно соотнесет номер строки в общем сформированном файле с номером строки в одном из фрагментов текста, введенных пользователем, что приведет к тому, что при двойном щелчке на сообщении об ошибке в редакторе модели курсор будет установлен не в ту строку, где эта ошибка возникла. Если компилятор поддерживает директиву “#line”, этот флажок лучше установить.

Флажок “использовать директиву `#pragma argsused`” разрешает модулю вставлять указанную директиву перед каждой автоматически сформированной функцией с параметрами. Эта директива отключает для следующей за ней функции предупреждение компилятора о том, что параметры функции не используются в ее теле. Это предупреждение, хотя и может быть полезно для пользовательских функций (с его помощью можно обнаружить и исключить ненужные параметры), бессмысленно для автоматически формируемых функций реакции на события: модуль всегда передает в них все возможные параметры, а уже пользователь решает, нужны ли они ему. Например, функция, формируемая для реакции на нажатие кнопки мыши (см. §3.7.11 на стр. 226 и §3.8.4.1 на стр. 295), всегда получает в качестве параметра указатель на структуру, содержащую координаты курсора, нажатую кнопку и т.п. Пользователь пишет только тело этой функции, и, для самых простых реакций, ему не нужна вся эта информация – достаточно самого факта нажатия кнопки. Но компилятор, обнаружив, что указатель на структуру описания события в теле функции нигде не использован, выдаст предупреждение, которое увидит пользователь, и оно может сбить его с толку. По этой причине, если компилятор поддерживает директиву “`#pragma argsused`”, рекомендуется включать этот флажок. Если директива не поддерживается, рекомендуется запретить вывод предупреждения о неиспользованных параметрах функций в настройках запуска компилятора (как правило, в командной строке компиляторов предусмотрены параметры, позволяющие включать и выключать вывод отдельных предупреждений). В настройках некоторых стандартных модулей автокомпиляции это предупреждение по умолчанию выключено, хотя компиляторы, для которых предназначены модули, и поддерживают “`#pragma argsused`”.

В правой нижней части вкладки находится кнопка “восстановить исходные”, заполняющая все поля ввода значениями, предлагаемыми разработчиками модуля (для поддерживаемых компиляторов это будут значения, совместимые с ними).

### §3.9.10. Настройка обработки исключений и ошибок

Описывается настройка процедур и способов обработки ошибок, возникающих при работе модели. Эти процедуры могут включаться в формируемый текст программы по желанию пользователя.

В настройках модуля автокомпиляции можно указать, какими операторами можно перехватывать возникшие в модели исключения и как модель может перехватывать ошибки в математических функциях. Такой перехват ошибок включается индивидуально в настройках каждой модели (см. §3.6.7 на стр. 69), но, чтобы его вообще можно было включить, в настройках модуля должны быть заданы способы обработки исключений и ошибок. Чтобы настроить эти способы, следует вызвать окно настройки модуля (см. стр. 328) и на его вкладке “компилятор” нажать кнопку “дополнительно”. В открывшемся дополнительном окне настроек следует выбрать вкладку “обработка исключений” (рис. 501).

В верхней части вкладки настраивается перехват исключений. Чтобы описать его, необходимо установить одноименный флажок и ввести в поля ввода “оператор `try`” и “универсальный `catch`” операторы начала блока обработки исключений (`try`) и начала блока реакции на исключение (`catch`) соответственно. Желательно ввести операторы для перехвата **любого** исключения, то есть самые универсальные из них.

Исключения (exceptions, исключительные ситуации) – это события, возникающие при каких-либо серьезных ошибках и прерывающие нормальный ход выполнения программы. Эти события часто создаются различными библиотечными функциями, разработчики которых, по каким-то причинам, решили сообщать об ошибках именно таким образом, а не возвратом какого-либо признака ошибки. Могут исключения возникать и при математических операциях, например, при делении на ноль.

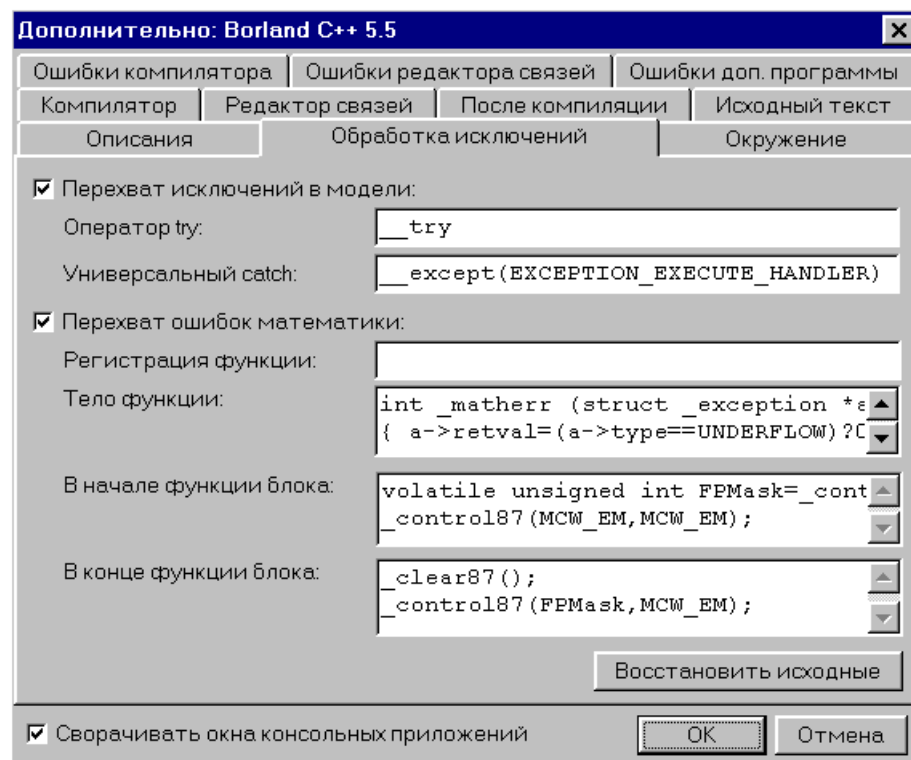


Рис. 501. Настройка обработки исключений и ошибок (на примере Borland C++ 5.5)

Обработка исключений обычно устроена следующим образом: часть программы, в которой могут возникнуть исключения, помещают в блок обработки исключений, чаще всего описываемый оператором `try`. За ним размещают один или несколько блоков реакции на исключение, чаще всего описываемых оператором `catch`. При возникновении исключения внутри блока `try` управление немедленно передается ближайшему к нему блоку `catch`, предназначенному для обработки исключения соответствующего типа. Если такого блока не найдется в данной функции, управление будет передано выше. Если же его не будет во всей программе, ее выполнение будет прервано. Для моделей блоков в РДС это будет означать, что расчет остановится и РДС выдаст стандартное сообщение об ошибке (см. рис. 354 на стр. 70). Более подробно обработка исключений описывается в учебниках по языку C++ [4].

При включении флажка “перехват исключений в модели” модуль автокомпиляции добавляет в текст формируемой программы описания трех констант:

```
// Обработка исключений включена
#define RDSBCPP_EXCEPTIONS
// Оператор try
#define RDSBCPP_TRY текст_первого_поля
// Оператор catch
#define RDSBCPP_CATCHALL текст_второго_поля
```

Например, для настроек на рис. 501 будут добавлены такие описания:

```
// Обработка исключений включена
#define RDSBCPP_EXCEPTIONS
// Оператор try
#define RDSBCPP_TRY __try
// Оператор catch
#define RDSBCPP_CATCHALL __except(EXCEPTION_EXECUTE_HANDLER)
```

В операторах перехвата исключений (`catch`) может указываться тип перехватываемого исключения. В данной настройке следует вводить наиболее универсальный обработчик – например, “`catch(...)`”, или, как в примере, оператор “`__except(EXCEPTION_EXECUTE_HANDLER)`”. Компиляторы поддерживают различные

операторы обработки исключений, но введенный оператор перехвата должен соответствовать оператору начала блока обработки: для “try” нужно указывать “catch”, для “\_\_try” – “\_\_except”, и т.п. Способы обработки исключений и используемые для этого операторы приводятся в описании конкретного компилятора.

Добавленные константы используются следующим образом: наличие определения константы RDSBCPP\_EXCEPTIONS говорит о том, что в параметрах модуля автокомпиляции указаны операторы обработки исключений (то есть флажок “перехват исключений в модели” установлен), константа RDSBCPP\_TRY может быть использована как оператор начала блока исключений, а константа RDSBCPP\_CATCHALL – как оператор начала блока перехвата. Использование этих констант вместо конкретных операторов позволяет создавать модели, которые можно использовать с любым компилятором, если параметры модуля, работающего с этим компилятором, настроены правильно.

Рассмотрим простой пример. Допустим, в модели блока есть вещественные переменные x1, x2 и y, и в реакции на нажатие кнопки мыши мы хотим присвоить y результат деления x1 на x2. Простейший текст реакции будет выглядеть так:

```
y=x1/x2;
```

Однако, если x2 будет иметь нулевое значение, при делении возникнет исключение, расчет будет остановлен, и пользователь увидит сообщение, подобное изображенному на стр. 70. Можно перехватить это исключение следующим образом:

```
RDSBCPP_TRY
{ y=x1/x2; }
RDSBCPP_CATCHALL
{ rdsMessageBox("Деление на ноль", "Ошибка",
 MB_OK|MB_ICONERROR);
 y=rdsbcppHugeDouble;
}
```

Здесь мы заключили деление в блок обработки исключений, причем вместо конкретного оператора try использовали константу RDSBCPP\_TRY – вместо нее компилятором будет подставлен оператор, указанный в настройках модуля. В оператор перехвата, записанного после константы RDSBCPP\_CATCHALL, мы включили вывод сообщения об ошибке и присвоение переменной y значения-маркера ошибки (см. стр. 84). Расчет при этом остановлен не будет.

Приведенный пример, хотя и позволяет использовать для модели разные компиляторы, имеет один недостаток: если в настройках модуля не указан способ обработки исключений (флажок “перехват исключений в модели” не установлен), константы RDSBCPP\_TRY и RDSBCPP\_CATCHALL будут определены таким образом, чтобы обработка исключений игнорировалась и блок перехвата никогда не выполнялся. Для этого RDSBCPP\_TRY будет иметь пустое значение, а RDSBCPP\_CATCHALL будет иметь значение “if(0)”, то есть будет заменяться на никогда не выполняющийся оператор if. Модель при этом будет работать так, как будто никакой обработки исключений нет, однако, при компиляции “if(0)” будет все время выдаваться предупреждение “условие всегда ложно”. Отключать это предупреждение в параметрах компилятора опасно, поскольку оно позволяет обнаружить потенциальные алгоритмические ошибки в программе. Лучше всего переписать приведенный пример с использованием константы RDSBCPP\_EXCEPTIONS:

```
RDSBCPP_TRY
{ y=x1/x2; }
#ifdef RDSBCPP_EXCEPTIONS
RDSBCPP_CATCHALL
{ rdsMessageBox("Деление на ноль", "Ошибка",
 MB_OK|MB_ICONERROR);
}
```

```

 y=rdsbcppHugeDouble;
 }
#endif

```

Здесь весь блок перехвата исключений заключен внутри конструкции “#ifdef...#endif”. Таким образом, если константа RDSBCPP\_EXCEPTIONS будет определена, то есть если в настройках модуля описана обработка исключений, блок перехвата будет скомпилирован и перехватит возникшее исключение при работе модели. Если же константа будет отсутствовать, весь блок перехвата скомпилирован не будет и не выдаст никаких предупреждений.

Вторая часть вкладки “обработка исключений” служит для настройки перехвата ошибок математики. Эти ошибки возникают только в функциях математической библиотеки и могут обрабатываться отдельно. Их обработка позволяет не прерывать выполнение программы в случае ошибки, а подставлять вместо результата операции, которая не может быть выполнена, какое-либо значение – например, маркер ошибки rdsbcppHugeDouble. Следует помнить, что деление на ноль не может быть перехвачено таким образом, оно всегда создает исключение, обработка которого рассматривается выше. Обработка ошибок математики включается индивидуально в параметрах каждой модели флажком “перехватывать ошибки математических функций” (см. рис. 353 на стр. 69).

Флажок “перехват ошибок математики” разрешает модулю автокомпиляции добавлять в текст формируемой программы функции для такого перехвата. В поле ввода “регистрация функции” вводится оператор, который может потребоваться для инициализации перехвата математических ошибок – он будет добавлен к процедуре инициализации глобальных переменных, место вставки которой указывается в настройках модуля на вкладке “исходный текст” (см. стр. 354). Следует отметить, что ни один из поддерживаемых модулями компиляторов не требует никаких специальных действий для инициализации обработки ошибок математики, поэтому это поле ввода в настройках модулей всегда пустое. Тем не менее, оно может оказаться полезным для какого-либо нестандартного компилятора.

В многострочное поле “тело функции” вводится функция перехвата математических ошибок в том виде, который требуется компилятору. Функция вводится полностью, вместе с заголовком и телом. Имя этой функции и ее параметры приводятся в описании конкретного компилятора. Для Borland C++, например, подходит такая функция:

```

int _matherr(struct _exception *a)
{ a->retval=(a->type==UNDERFLOW)?0.0:rdsbcppHugeDouble;
 return 1;
}

```

Здесь анализируется возникшая ошибка и, в случае потери точности, результат математической операции считается нулем, а для всех остальных ошибок результатом будет маркер ошибки rdsbcppHugeDouble. Возврат функцией единицы указывает на то, что функция успешно обработала ошибку.

В поля ввода “в начале функции блока” и “в конце функции блока” вводятся команды, позволяющие, если в настройках модели разрешен перехват ошибок математики, временно блокировать специфические математические исключения (команды в поле “в начале функции блока”) на время работы модели, а затем разрешать их снова (команды в поле “в конце функции блока”). Функции для управления математическими исключениями и их параметры описываются в руководстве по Windows API и в описаниях компиляторов. Команды, записанные в этих полях по умолчанию, в начале функции модели блокируют все исключения, а в конце – восстанавливают их прежнее состояние.

## Список литературы

1. Рошин А.А. Расчет Динамических Систем (РДС) Руководство для программистов. М.: ИПУ РАН, 2011. – 656 с.
2. Рошин А.А. Расчет Динамических Систем (РДС). Руководство для программистов. Приложение: описание функций и структур. М.: ИПУ РАН, 2012. – 719 с.
3. Бахвалов Н.С. Численные методы (анализ, алгебра, обыкновенные дифференциальные уравнения). – М.: Наука. Гл. ред. физ.-мат. лит., 1973
4. Пол Ирэ. Объектно-ориентированное программирование с использованием C++: Пер. с англ./Ирэ Пол. – К.: НИПФ “ДиаСофт Лтд.”, 1995. – 480 с.



## Алфавитный указатель

|                                             |                  |
|---------------------------------------------|------------------|
| главная функция DLL.....                    | 47, 85, 283, 353 |
| дифференциальные уравнения.....             | 146              |
| захват мыши.....                            | 88, 232          |
| компилятор.....                             |                  |
| Borland C++.....                            | 10, 359          |
| Digital Mars C++.....                       | 13, 355          |
| Microsoft Visual C++ 2003.....              | 16, 336          |
| Microsoft Visual C++ 6.....                 | 18               |
| MinGW GCC.....                              | 15               |
| Open Watcom C++.....                        | 12               |
| компиляция модели.....                      | 26               |
| метод Эйлера.....                           | 146, 162         |
| настроечные параметры.....                  | 60               |
| окно.....                                   |                  |
| выбора шаблона модели.....                  | 22               |
| групповой установки параметров блоков.....  | 77               |
| добавления нового параметра.....            | 62               |
| дополнительных настроек модуля.....         | 341              |
| компиляции.....                             | 24               |
| настройки блока.....                        | 61, 62           |
| настройки модуля.....                       | 328, 331         |
| параметров блока.....                       | 20, 27           |
| параметров вкладки.....                     | 66               |
| параметров динамической переменной.....     | 43               |
| параметров модели.....                      | 69               |
| параметров поля ввода.....                  | 66               |
| параметров функции блока.....               | 59               |
| поиска и замены текста.....                 | 36               |
| поиска текста.....                          | 36               |
| редактирования существующего параметра..... | 62               |
| редактора модели.....                       | 24, 32           |
| редактора переменных.....                   | 25, 41           |
| ошибка компиляции.....                      | 33               |
| переменная.....                             |                  |
| вход.....                                   | 39               |
| выход.....                                  | 39               |
| динамическая.....                           | 42, 129          |
| логическая.....                             | 39               |
| массив.....                                 | 40               |
| матрица.....                                | 40               |
| подписка на динамическую.....               | 130              |
| сигнал.....                                 | 40, 117          |
| статическая.....                            | 38               |
| строка.....                                 | 40, 61           |
| структура.....                              | 40               |
| тип.....                                    | 39               |
| BOOL.....                                   | 61               |
| char.....                                   | 39               |
| COLORREF.....                               | 61               |

|                               |                              |
|-------------------------------|------------------------------|
| double.....                   | 39, 61                       |
| DrawData.....                 | 175                          |
| DynTime.....                  | 42                           |
| float.....                    | 39                           |
| int.....                      | 39, 61                       |
| rdsbcppHugeDouble.....        | 39, 70, 81, 84, 95, 102, 124 |
| rdsbcppString.....            | 61, 112, 258                 |
| Ready.....                    | 41                           |
| ResizeData.....               | 193                          |
| short.....                    | 39                           |
| Start.....                    | 41, 93                       |
| переменные окружения.....     | 341                          |
| планировщик.....              | 42, 97, 130                  |
| растровая операция.....       | 179                          |
| расчет.....                   | 25, 290                      |
| инициализационный.....        | 72, 290                      |
| предварительный.....          | 72, 290                      |
| файл.....                     |                              |
| .dll.....                     | 9, 24                        |
| .mdl.....                     | 7, 24                        |
| .rds.....                     | 9                            |
| autoexec.bat.....             | 15                           |
| Common.dll.....               | 6                            |
| CommonAC.hpp.....             | 86, 281                      |
| dynvars.lst.....              | 43                           |
| freecommandlinetools.exe..... | 10                           |
| math.h.....                   | 98                           |
| mingw-get-inst.....           | 15                           |
| PSDK-x86.exe.....             | 17                           |
| RdsCtrl.dll.....              | 55, 322                      |
| RdsDef.h.....                 | 88                           |
| response file.....            | 339, 343                     |
| stdio.h.....                  | 114, 212                     |
| VCToolkitSetup.exe.....       | 17                           |
| функция блока.....            | 57, 245, 301                 |
| шаблон модели.....            | 22, 35, 73, 168, 331         |
| _clear87.....                 | 83                           |
| _control87.....               | 83                           |
| _matherr.....                 | 70, 81, 85                   |
| #line.....                    | 84, 355                      |
| #pragma argsused.....         | 172, 356                     |
| \$AUXDIR\$.....               | 336, 337                     |
| \$BC\$.....                   | 335, 337                     |
| \$COMP\$.....                 | 338                          |
| \$CPPFILES\$.....             | 338                          |
| \$DLL\$.....                  | 6                            |
| \$DLLFILES\$.....             | 338                          |
| \$FILES\$.....                | 338                          |
| \$FUNC\$.....                 | 338                          |
| \$FUNCLC\$.....               | 338                          |
| \$FUNCUC\$.....               | 338                          |

|                                 |                                  |
|---------------------------------|----------------------------------|
| \$INCLUDE\$.....                | 338                              |
| \$INIS.....                     | 9                                |
| \$LIB\$.....                    | 338                              |
| \$LINK\$.....                   | 338                              |
| \$MODELDIR\$.....               | 338                              |
| \$MODEL\$S.....                 | 9, 24                            |
| \$OBJFILES.....                 | 338                              |
| \$POSTEXES.....                 | 337, 338                         |
| \$RDSINCLUDE\$.....             | 339, 352                         |
| \$RDSTEMP\$.....                | 339                              |
| \$RESPONSE\$.....               | 339                              |
| BCpp55.....                     | 6, 10                            |
| Broadcast.....                  | 249, 253                         |
| c_str.....                      | 112, 209, 260                    |
| Call.....                       | 248, 259, 273                    |
| CheckLink.....                  | 136                              |
| COLORREF.....                   | 179, 198                         |
| Cols.....                       | 100                              |
| Common.ControlValueChanged..... | 261                              |
| CP1251.....                     | 116                              |
| Create.....                     | 207                              |
| DEFAULT_CHARSET.....            | 191                              |
| Delete.....                     | 207                              |
| DigitalMars.....                | 7, 13                            |
| DLL_PROCESS_ATTACH.....         | 85                               |
| DllEntryPoint.....              | 81, 85, 283, 353                 |
| Exists.....                     | 70, 133, 207, 209, 225           |
| Gcc_MinGW.....                  | 7, 15                            |
| GetLink.....                    | 136                              |
| GetProcAddress.....             | 354                              |
| GetPtr.....                     | 87, 98                           |
| GetTickCount.....               | 241                              |
| HasData.....                    | 100, 104                         |
| HUGE_VAL.....                   | 84, 92                           |
| Id.....                         | 248                              |
| IsEmpty.....                    | 100, 104, 112                    |
| IsProvider.....                 | 250                              |
| Item.....                       | 100, 104                         |
| Length.....                     | 112                              |
| MSVCpp6.....                    | 7, 18                            |
| MSVCTK2003.....                 | 7, 16                            |
| name mangling.....              | 334, 355                         |
| NotifySubscribers.....          | 50, 131, 138, 143, 205, 209, 294 |
| OpenWatcomCpp.....              | 6, 12                            |
| Provider.....                   | 250, 282                         |
| PS_NULL.....                    | 180                              |
| PS_SOLID.....                   | 179                              |
| R2_COPYPEN.....                 | 179                              |
| RDS_BCALL_ALLOWSTOP.....        | 253                              |
| RDS_BCALL_SUBSYSTEMS.....       | 253                              |
| RDS_BEN_OUTPUTS.....            | 263                              |

|                                 |                        |
|---------------------------------|------------------------|
| RDS_BEN_TRACELINKS.....         | 263                    |
| RDS_BFM_AFTERLOAD.....          | 52, 276, 306           |
| RDS_BFM_AFTERSAVE.....          | 52, 306                |
| RDS_BFM_BEFORESAVE.....         | 52, 305                |
| RDS_BFM_CALCMODE.....           | 50, 293                |
| RDS_BFM_CLEANUP.....            | 48, 90, 209, 287       |
| RDS_BFM_CONTEXTPOPUP.....       | 56, 326                |
| RDS_BFM_DRAW.....               | 53, 174, 219, 233, 315 |
| RDS_BFM_DRAWADDITIONAL.....     | 54, 225, 316           |
| RDS_BFM_DYNVARCHANGE.....       | 50, 71, 295            |
| RDS_BFM_EDITMODE.....           | 50, 293                |
| RDS_BFM_FUNCTIONCALL.....       | 303                    |
| RDS_BFM_INIT.....               | 48, 90, 287            |
| RDS_BFM_KEYDOWN.....            | 51, 300                |
| RDS_BFM_KEYUP.....              | 51, 301                |
| RDS_BFM_LOADSTATE.....          | 52, 307                |
| RDS_BFM_LOADTXT.....            | 51, 209, 304           |
| RDS_BFM_MANUALDELETE.....       | 49, 289                |
| RDS_BFM_MANUALINSERT.....       | 49, 288                |
| RDS_BFM_MENUFUNCTION.....       | 56, 327                |
| RDS_BFM_MODEL.....              | 49, 71, 291            |
| RDS_BFM_MOUSEDBLCLICK.....      | 51, 298                |
| RDS_BFM_MOUSEDOWN.....          | 50, 228, 296           |
| RDS_BFM_MOUSEMOVE.....          | 51, 298                |
| RDS_BFM_MOUSEUP.....            | 50, 297                |
| RDS_BFM_MOVED.....              | 53, 315                |
| RDS_BFM_NETCONNECT.....         | 54, 318                |
| RDS_BFM_NETDATAACCEPTED.....    | 54, 319                |
| RDS_BFM_NETDATARECEIVED.....    | 54, 317                |
| RDS_BFM_NETDISCONNECT.....      | 54, 319                |
| RDS_BFM_NETERROR.....           | 54, 320                |
| RDS_BFM_POPUPHINT.....          | 55, 79, 211, 325       |
| RDS_BFM_REMOTEMSG.....          | 55, 323                |
| RDS_BFM_RENAME.....             | 55, 321                |
| RDS_BFM_RESETCALC.....          | 50, 292                |
| RDS_BFM_RESIZE.....             | 53, 313                |
| RDS_BFM_RESIZING.....           | 53, 193, 314           |
| RDS_BFM_SAVESTATE.....          | 52, 307                |
| RDS_BFM_SAVETXT.....            | 52, 305                |
| RDS_BFM_SETUP.....              | 55, 68, 79, 208, 322   |
| RDS_BFM_STARTCALC.....          | 49, 93, 291            |
| RDS_BFM_STOPCALC.....           | 49, 292                |
| RDS_BFM_TIMER.....              | 55, 323                |
| RDS_BFM_UNLOADSYSTEM.....       | 49, 289                |
| RDS_BFM_VARCHECK.....           | 56, 90, 327            |
| RDS_BFM_WINDOWKEYDOWN.....      | 311                    |
| RDS_BFM_WINDOWKEYUP.....        | 311                    |
| RDS_BFM_WINDOWMOUSEDLCLICK..... | 310                    |
| RDS_BFM_WINDOWMOUSEDOWN.....    | 309                    |
| RDS_BFM_WINDOWMOUSEMOVE.....    | 310                    |
| RDS_BFM_WINDOWMOUSEUP.....      | 309                    |

|                              |                             |
|------------------------------|-----------------------------|
| RDS_BFM_WINDOWOPERATION..... | 52, 308                     |
| RDS_BFM_WINREFRESH.....      | 55, 324                     |
| RDS_BFR_DONE.....            | 91, 227, 296, 300, 313, 325 |
| RDS_BFR_NOTPROCESSED.....    | 215, 227, 296, 325          |
| RDS_BFR_SHOWMENU.....        | 228, 231, 253, 296          |
| RDS_BFR_STOP.....            | 300, 313                    |
| RDS_BHANDLE.....             | 88                          |
| RDS_BLOCKDATA.....           | 88                          |
| RDS_CONTEXTPOPUPDATA.....    | 56, 326                     |
| RDS_DRAWDATA.....            | 53, 175, 224, 315, 316      |
| RDS_DVPARENT.....            | 207                         |
| RDS_DVROOT.....              | 207                         |
| RDS_DVSELF.....              | 207                         |
| RDS_DWORDVERDATE.....        | 74                          |
| RDS_DYNVARLINK.....          | 50, 294                     |
| RDS_FUNCTIONCALldata.....    | 60, 301                     |
| RDS_GFCOLOR.....             | 192                         |
| RDS_GFFONTALLHEIGHT.....     | 191                         |
| RDS_GFS_EMPTY.....           | 181                         |
| RDS_GFS_SOLID.....           | 179                         |
| RDS_GFSTYLE.....             | 180, 181                    |
| RDS_INITCALC.....            | 72                          |
| RDS_INITCALCFIRST.....       | 72                          |
| RDS_INITIALCALCDATA.....     | 290                         |
| RDS_INTVERSION.....          | 74                          |
| RDS_KALT.....                | 227, 243, 296, 300          |
| RDS_KCTRL.....               | 227, 243, 296, 300          |
| RDS_KEYDATA.....             | 51, 299                     |
| RDS_KSHIFT.....              | 227, 296, 300               |
| RDS_LS_LOADCLIPBRD.....      | 288                         |
| RDS_LS_LOADFROMFILE.....     | 288                         |
| RDS_MANUALDELETEDATA.....    | 49, 289                     |
| RDS_MANUALINSERTDATA.....    | 49, 288                     |
| RDS_MENU_DISABLED.....       | 240                         |
| RDS_MENU_DIVIDER.....        | 240                         |
| RDS_MENU_SHORTCUT.....       | 243                         |
| RDS_MENU_UNIQUECAPTION.....  | 243                         |
| RDS_MENUFUNCData.....        | 56, 240, 326                |
| RDS_MLEFTBUTTON.....         | 227, 231, 253, 296, 300     |
| RDS_MMIDDLEBUTTON.....       | 227, 296, 300               |
| RDS_MOUSECAPTURE.....        | 88, 234                     |
| RDS_MOUSEDATA.....           | 50, 226, 295                |
| RDS_MOVEDATA.....            | 53, 314                     |
| RDS_MR_DRAG.....             | 314                         |
| RDS_MR_KEYBOARD.....         | 314                         |
| RDS_MR_SET.....              | 314                         |
| RDS_MR_UNDOREDO.....         | 314                         |
| RDS_MRRIGHTBUTTON.....       | 227, 296, 300               |
| RDS_NETACCEPTDATA.....       | 54, 319                     |
| RDS_NETCONNDATA.....         | 54, 318                     |
| RDS_NETERR_ACCEPT.....       | 320                         |

|                                   |                   |
|-----------------------------------|-------------------|
| RDS_NETERR_CLIENTCONN.....        | 320               |
| RDS_NETERR_DISCONNECT.....        | 320               |
| RDS_NETERR_GENERAL.....           | 320               |
| RDS_NETERR_NOBLOCK.....           | 320               |
| RDS_NETERR_RECEIVE.....           | 320               |
| RDS_NETERR_SEND.....              | 320               |
| RDS_NETERRORDATA.....             | 55, 320           |
| RDS_NETRECEIVEDDATA.....          | 54, 317           |
| RDS_PDYNVARLINK.....              | 136               |
| RDS_POINTDESCRIPTION.....         | 265               |
| RDS_POPUPHINTDATA.....            | 56, 214, 325      |
| RDS_REMOTEMSGDATA.....            | 55, 323           |
| RDS_RESIZEDATA.....               | 53, 193, 312, 313 |
| RDS_SERV_FUNC_BODY.....           | 352               |
| RDS_SETFLAG.....                  | 237               |
| RDS_STARTSTOPDATA.....            | 49, 291           |
| RDS_STDICON_YELCIRCEXCLAM.....    | 225               |
| RDS_SWO_CLOSE.....                | 308               |
| RDS_SWO_OPEN.....                 | 308               |
| RDS_TIMERS_WINREF.....            | 324               |
| RDS_WINOPERATIONDATA.....         | 52, 308           |
| RDS_WINREFRESHDATA.....           | 55, 324           |
| rdsActivateOutputConnections..... | 261               |
| rdsAdditionalContextMenuEx.....   | 239               |
| rdsAddToDynStr.....               | 222               |
| RDSBCPP_CATCHALL.....             | 357               |
| RDSBCPP_EXCEPTIONS.....           | 357               |
| RDSBCPP_TRY.....                  | 357               |
| rdsbcppAfterLoad.....             | 306               |
| rdsbcppAfterSave.....             | 305               |
| rdsbcppBeforeSave.....            | 305               |
| rdsbcppBlockClass.....            | 82, 87            |
| rdsbcppBlockData.....             | 87                |
| rdsbcppBlockEntryPoint.....       | 83, 89, 354       |
| rdsbcppBlockMoved.....            | 314               |
| rdsbcppBlockRename.....           | 321               |
| rdsbcppBlockResized.....          | 312               |
| rdsbcppCalcMode.....              | 293               |
| rdsbcppCleanup.....               | 287               |
| rdsbcppContextPopup.....          | 325               |
| rdsbcppDraw.....                  | 315               |
| rdsbcppDrawAdditional.....        | 316               |
| rdsbcppDynVarChange.....          | 136, 293          |
| rdsbcppDynVarsOk.....             | 82, 89            |
| rdsbcppEditMode.....              | 292               |
| rdsbcppGetService.....            | 352               |
| rdsbcppInit.....                  | 287               |
| rdsbcppKeyDown.....               | 299               |
| rdsbcppKeyUp.....                 | 300               |
| rdsbcppLoadState.....             | 307               |
| rdsbcppLoadText.....              | 303               |

|                                   |               |
|-----------------------------------|---------------|
| rdsbcppManualDelete.....          | 288           |
| rdsbcppManualInsert.....          | 288           |
| rdsbcppMenuFunc.....              | 326           |
| rdsbcppModel.....                 | 290           |
| rdsbcppMouseDbClick.....          | 297           |
| rdsbcppMouseDown.....             | 295           |
| rdsbcppMouseMove.....             | 298           |
| rdsbcppMouseUp.....               | 297           |
| rdsbcppNetAccepted.....           | 319           |
| rdsbcppNetConnect.....            | 317           |
| rdsbcppNetDisconnect.....         | 318           |
| rdsbcppNetError.....              | 320           |
| rdsbcppNetReceive.....            | 316           |
| rdsbcppOther.....                 | 327           |
| rdsbcppPopupHint.....             | 325           |
| rdsbcppRemoteMessage.....         | 322           |
| rdsbcppResetCalc.....             | 292           |
| rdsbcppResizing.....              | 313           |
| rdsbcppSaveState.....             | 307           |
| rdsbcppSaveText.....              | 305           |
| rdsbcppSetupFunc.....             | 322           |
| rdsbcppStartCalc.....             | 291           |
| rdsbcppStaticVarsInit.....        | 327           |
| rdsbcppStopCalc.....              | 291           |
| rdsbcppString.....                | 203           |
| rdsbcppSystemUnload.....          | 289           |
| rdsbcppSysWinKeyDown.....         | 311           |
| rdsbcppSysWinKeyUp.....           | 311           |
| rdsbcppSysWinMouseDbClick.....    | 310           |
| rdsbcppSysWinMouseDown.....       | 309           |
| rdsbcppSysWinMouseMove.....       | 310           |
| rdsbcppSysWinMouseUp.....         | 309           |
| rdsbcppSysWinOperation.....       | 308           |
| rdsbcppTimer.....                 | 323           |
| rdsbcppWinRefresh.....            | 324           |
| rdsBlockByFullName.....           | 248, 259      |
| rdsBroadcastFuncCallsDelayed..... | 248           |
| rdsCalcProcessIsRunning.....      | 244           |
| rdsCalcProcessNeverStarted.....   | 244           |
| rdsDtoA.....                      | 190, 213      |
| rdsDynStrCat.....                 | 214, 273      |
| rdsEnumConnectedBlocks.....       | 88, 262       |
| rdsForceBlockRedraw.....          | 226, 316      |
| rdsFree.....                      | 190, 213, 273 |
| rdsGetBlockDimensionsEx.....      | 313           |
| rdsGetBlockLink.....              | 262           |
| rdsGetFullFilePath.....           | 276           |
| rdsGetHugeDouble.....             | 82, 86, 95    |
| rdsGetMouseObjectId.....          | 231, 253      |
| rdsGetPointDescription.....       | 262           |
| rdsItoA.....                      | 222           |

|                              |              |
|------------------------------|--------------|
| rdsLoadSystemState.....      | 52, 306      |
| rdsMessageBox.....           | 244, 271     |
| rdsOpenSystemWindow.....     | 88           |
| rdsReadBlockData.....        | 52, 307      |
| rdsRegisterFunction.....     | 57           |
| rdsRegisterMenuItem.....     | 241          |
| rdsRemoteReply.....          | 323          |
| rdsRenameBlock.....          | 321          |
| rdsReportVersion.....        | 74           |
| rdsResetSystemState.....     | 292          |
| rdsSaveSystemState.....      | 52, 307      |
| rdsSetBlockTimer.....        | 55           |
| rdsSetHintText.....          | 56, 211, 325 |
| rdsWriteBlockData.....       | 52, 307      |
| rdsXGDrawStdIcon.....        | 225          |
| rdsXGEllipse.....            | 188, 219     |
| rdsXGGetStdIconSize.....     | 225          |
| rdsXGGetTextSize.....        | 191          |
| rdsXGLineTo.....             | 190, 219     |
| rdsXGMoveTo.....             | 190, 219     |
| rdsXGRectangle.....          | 179          |
| rdsXGSetBrushStyle.....      | 179          |
| rdsXGSetClipRect.....        | 235          |
| rdsXGSetFont.....            | 190          |
| rdsXGSetPenStyle.....        | 179          |
| rdsXGTextOut.....            | 191          |
| RegisterProvider.....        | 249          |
| Resize.....                  | 100, 105     |
| Rows.....                    | 100          |
| RUSSIAN_CHARSET.....         | 191          |
| SearchComp.....              | 7, 18        |
| Size.....                    | 104          |
| sprintf.....                 | 114, 212     |
| Subscribe.....               | 207          |
| Subscribed.....              | 250          |
| SubscribeToProvider.....     | 249          |
| UnregisterProvider.....      | 249          |
| Unsubscribe.....             | 207          |
| UnsubscribeFromProvider..... | 250          |
| UserComp1.....               | 7, 334       |
| UserComp2.....               | 7, 334       |
| UserComp3.....               | 7, 334       |