АННОТАЦИЯ К ДИПЛОМНОМУ ПРОЕКТУ

«Разработка и создание программы-отладчика для языка Object RSL на Microsoft Visual C с применением библиотеки Microsoft Foundation Classes»

Данный дипломный проект посвящён проекированию и реализации интерактивной системы отладки в рамках проекта по созданию визуальной среды разработки распределённых приложений с использованием языка Object RSL.

В дипломе рассматриваются ошибки, возникающие в программном обеспечении, их категории причины возникновения, возможности их устранения, общие проблемы отладки и методы отладки. Для обеспечения программистов, использующих в качестве языка программирования язык Object RSL, отладочного инструмента, был реализован интерактивный отладчик распределённых RSL-приложений.

В процессе работы были сформированы требования к рассматриваемому инструменту и методика решения и инструмент реализации. Указанные требования основаны на опыте, полученном автором при рассмотрении отладчиков различных сред программирования, с учётом специфики разрабатываемых на языке Object RSL распределённых приложений. Исходя из упомянутых требований был выбран инструмент и метод реализации, которые позволили создать современный, устойчывый, масштабируемый отладочный инструмент, обладающий качествествнным интерфейсом.

В результате внедрения описываемого отладчика значительно улучшилась работа программистов, применяющих в язык Object RSL. Разработчики получили возможность использовать все современные средства, предоставляемые отладчиком для производства надёжного и устойчивого программного обеспечения

ОГЛАВЛЕНИЕ

1. Ошибки и отладка	7
1.1. Определение понятия отладки	7
1.2. Категории ошибок	7
1.3. Причины ошибок	8
1.4. Процесс отладки	10
1.4.1. Планирование отладки	10
1.4.2. Предпосылки для отладки	11
1.4.3. Процесс отладки	12
2. Постановка задачи	17
3. Краткое описание языка Object RSL	19
3.1. Элементы языка	19
3.1.1. Служебные слова	19
3.1.2. Имена	19
3.1.3. Комментарии	21
3.1.4. Объекты языка	21
3.1.5. Типы данных	23
3.1.6. Базовые стандартные типы	23
3.1.7. Выражения	24
3.1.7.1.Синтаксис	24
3.1.7.2.Семантика	25
3.2 Структура программы	26
3.3. Конструкции языка RSL	28
3.3.1 Определение переменных VAR	29
3.3.2 Определение классов и объектов	30
3.3.3. Определение символических констант CONST	31
3.3.4. Определение процедуры MACRO	31
3.3.5. Процедуры языка RSL	32
3.3.5.1 Передача параметров	32
3.3.6. Определение массивов	33
3.3.7. Определения FILE и RECORD	33
3.4. Поддержка технологии ActiveX в RSL	33
3.4.1. Поддержка стандартных коллекций	34
3.4.2. Обращение к свойствам объектов с параметрами	34
3.4.3. Обработка событий	35
3.4.4. Передача параметров	35
3.4.5. Обработка ошибок времени выполнения	35
3.5. Автоматическое создание объектов Object RSL	36
3.6. Организация ввода/вывода	36
3.7. Работа с файлами	36
3.8. Встроенные процедуры	36
3.9. Средство разработки расширений для языка RSL (DLM SDK)	37
3.9.1. Создание и использование DLM-модулей	37
4. Структура отладчика	39
4.1. Состав отладчика, инсталляция	39
4.3. Основное окно отладчика	40
4.4. Отладка макропрограмм	41

4.5. Точки останова	42
4.6. Список импортируемых модулей	44
4.7. Окно проверки	45
4.8. Стек вызовов процедур	56
4.9. Окно переменных	57
4.10. Окно выражений	58
4.11. Редактирование переменной	50
4.12. Настройка отладчика	50
4.13. Список горячих клавиш отладчика	51
5. Устройство отладчика	52
5.1. Способ реализации, используемые библиотеки	52
5.2. Загрузка и инициализация отладчика	52
5.3. Особенности окон приложения	53
5.4. Отладка в локальном режиме	54
5.5. Отладка в удалённом режиме	54
6. Перспективы развития	55
7. Заключение	56

1. ОШИБКИ И ОТЛАДКА

1.1. Определение понятия отладки

Отладка – это этап разработки программного обеспечения, при котором проверяется правильность написания программы, выявляются и исправляются ошибки. Обычно отладка выполняется на контрольных примерах, близких к реальным, но с известными результатами. Во всех современных средах программирования существуют специальные программы - отладчики, упрощающие этот процесс.

1.2. Категории ошибок

Наличие ошибок в программном продукте вызывает потери двух видов. Первый сязан с тем, что, пользователи чаще обращаются за помощью, следовательно требуется больше времени и денег на поддержку существующего продукта, в то время как конкуренты уже работают над новой версией. Второй определяется законами экономики, согласно которым, пользователи начинают покупать не продукт с ошибками, а продукты других производителей. Так как программнсе обеспечение все чаще поставляется в виде услуги, необходимость в безошибочном программном обеспечении постоянно расёт. Скоро пользователи смогут менят программное обеспечение, переходя с одного Web-сайта на другой. Это обернется уменьшением гарантии занятости для разработчиков программного обеспечения в том случае, если их продукты будут полны ошибок и потребуется создание новых, более качественных.

Ошибкой можно считать все, что создает для пользователя неприятные ситуации, и подразделяю их на три следующих категории:

- непоследовательный пользовательский интерфейс;
- неоправданные ожидания;
- низкая производительность;
- сбой программы или повреждение данных

Рассмотрим эти категории более подробно:

Непоследовательный пользовательский интерфейс. Эта проблема не является самой серьёзной, но тем не менее, обращает на себя внимание. Одна из причин популярности Microsoft Windows в том, что все приложения Windows в целом посторены одинаково. Если приложение отклоняется от стандарта Windows, то причиняет пользователю неудобства. Примером такого раздражающе нестандартного поведения является комбинация клавиш для команды Find в Microsoft Outlook. Во всех остальных англоязычных приложениях Microsoft Windows комбинация клавиш Ctrl+F вызывает диалоговое окно Find для поиска текста в окне приложения. Но в Outlook эта комбинация предназначена для пересылки (forward) сообщения. Проблем с непоследовательным пользовательским интерфейсом можно избежать, следуя рекомендациям книг, посвящённых этому вопросу, напимер работы [1], [2] и [3].

Неоправданные ожидания. Одна из самых сложных ошибок - это неоправданные ожидания пользователя. Эта ошибка обычно возникает в самом начале проекта, если компания недостаточно исследует реальные нужды пользователя. Для обоих типов поставки - "коробочной" (написание программного обеспечения для продажи) или Information Technology (информационная

технология, IT, то есть создание приложений для внутреннего использования) - причина этой ошибки кроется в том, что разработчики не общаются непосредственно с пользователями продуктов, поэтому не знают, что им нужно. В идеале программисты должны посещать рабочие места пользователей, чтобы узнать, как используется их продукт. Это позволит правильно истолковывать требования, предъявляемые пользователями к продукту.

Ошибки этого типа также возникают, если ожидания пользователя превышают возможности продукта. Это - классический результат преувеличенной рекламы, поэтому следует всячески пресекать неверное комментирование возможностей программного продукта. Не получая от программы желаемого, пользователям кажется, что в нем намного больше ошибок, чем на самом деле. Чтобы избежать подобной ситуации, нужно придерживаться правила - никогда не обещать того, что невозможно выполнить, и всегда выполнять то, что обещается.

Низкая производительность. Пользователи очень расстраиваются, если ошибки снижают производительность приложения при работе с реальными данными. Несомненно, корень всех ошибок, связанных с низкой производительностью, кроется в недостаточно тщательном тестировании - приложение выглядело великолепно во время разработки, но не было протестировано на реальных данных. Существует два способа справиться с подобными ошибками. Во-первых, требования к производительности приложения должны быть определены заранее. Чтобы выявить низкую производительность, нужен ориентир, с которым можно было бы ее сравнивать. Важно поддерживать показатели производительности приложения. Если его производительность становится ниже этих показателей на 10% и более, следует определить причину и предпринять меры по ее устранению. Во-вторых, необходимо убедиться, что тестирование приложения выполняется для сценариев его использования, максимально приближенных к реальным. И делается это в цикле разработки на возможно ранних этапах.

Сбой программы или повреждение данных. Под ошибками пользователи и разработчики чаще всего подразумевают те, что связаны со сбоями или повреждением данных. Данные ошибки являются критическими, так как если ошибками вышеописанных типов пользователи могут как-то справиться, то сбои приводят к невозможности дальнейшей работы. Кроме того, этот тип ошибок встречается чаще других.

1.3. Причины ошибок

Поставлять программное обеспечение без ошибок можно только в том случае, если уделять достаточно внимания деталям. Ошибки в этой отрасли неизбежны, но необходимо минимизировать их число. Именно этого добиваются производители высококачественных продуктов. Причины ошибок можно классифицировать следующим образом:

- слишком жесткий или невыполнимый график выполнения;
- подход "вначале программировать, а потом думать";
- непонятные требования;
- недостаточная подготовка разработчиков;
- недостаточный контроль качества

Слишком жесткий или невыполнимый график. Все разработчики сталкивались с менеджерами, которые устанавливали контрольные сроки выпуска продукта

наугад; но зачастую в планировании большинства нереальных графиков виноваты не менеджеры. График обычно составляют с учетом производительности программистов. А программисты часто недооценивают, сколько времени им потребуется на создание законченного продукта. Независимо от того, кто назначил нереальную дату поставки - менеджеры, программисты или и те и другие, - в конечном счете нереальный график проекта приводит к "срезанию углов" и выпуску низкокачественного продукта.

Подход "вначале программировать, а потом думать". Выражение "вначале программировать, а потом думать" описывает обычную ситуацию при разработке программ. Все программисты в той или иной мере пользутся этим подходом, так как интереснее писать код и выполнять отладку, чем заниматься планированием. Но именно из-за отсутствия планирования и возникают ошибки. Вместо того чтобы обдумать, как избежать ошибок, некоторые программисты предпочитают исправлять ошибки в процессе написания кода. Очевидно, что подобная тактика только усложняет проблему, поскольку в процессе исправления одних ошибок неизбежно появятся другие.

Решить эту проблему так же несложно: необходимо уделять достаточное время планированию проектов. В [5], [6], [7] и [8] определены требования по планированию проекта.

Непонятые требования. Правильное планирование также устраняет одну из частых причин возникновения ошибок при разработке - "ползучий улучшизм" (feature creep). Это - добавление возможностей, которые не предпологались изначально (что является признаком плохого планирования и неправильно заданных требований). Добавление новых свойств в последний момент из-за конкуренции, по прихоти разработчика или менеджера порождает большое количество ошибок.

Программирование требует внимания к деталям. Чем больше вопросов выявится и решится перед написанием кода, тем меньше шансов вщзникновения ошибок. Единственный способ добиться необходимого внимания к деталям - планировать этапы проектирования и реализации.

Очень немногие компании занимаются обучением программистов в проблемной области. И часто даже дипломированные программисты обычно хорошо не знают, как пользователи будут использовать их продукты. Если бы компании заранее помогали свои специалистам лучше разбираться в проблемной области, то смогли бы устранить множество ошибок, вызванных непониманием требований. Но не только компании виновны в этом. Программисты также должны стремиться изучить проблемную область. Некоторые полагают, что создают инструменты, допускающие решения задачи, и поэтому могут отстраниться от проблемной области. Но программисты должны решать задачи, а не доказывать существование их решения. Только тогда написанное программное обеспечение расширяет возможности пользователя и улучшает качество его работы.

Недостаточная подготовка. Другая существенная причина ошибок состоит в том, что разработчики недостаточно хорошо понимают операционную систему, язык программирования или технологию, используемые в их продуктах.

Во многих случаях подобное невежество, скорее, не личный недостаток, а примета современной разработки программного обеспечения. Сейчас разработка включает так много уровней и взаимосвязей, что порой одному человеку трудно разобраться во всех тонкостях операционных систем, языков и технологий,

применяемых в проекте. Поэтому очень важно определить навыки и профессиональный уровень каждого члена коллектива, так как это позволит команде в целом лучше распределить задачи внутри коллектива, и следовательно, каждая часть работы будет выполнена более качественно.

Недостаточный контроль качества. Последняя причина ошибок в проектах, на мой взгляд, самая важная. Многие программисты, с которыми я общался, считают, что они - сторонники качества. Они гордятся своими продуктами и готовы тратить усилия на все части разработки, а не только на необходимые. Но на деле часто получается совсем не так. Например, не утруждая себя сложностями алгоритма, они предпочитают найти более простой алгоритм, чтобы как можно лучше протестировать его Но это не всегда означает, что конечный продук получится более высококачественным. Объективно можно сказать, что обеспечивать необходимый контроль качества должна компания. От того, следует ли она основным этапам разработки програмного обнспечения (планирование, кодирование, и т.д.), уделяет ли достаточно внимания деталям (набору сотрудников, обеспечению необходиыми инструментами, зависит, будет ли продукт соответствующего качества вовремя поставлен.

1.4. Процесс отладки

Теперь, когда я подробно разобрал типы и происхождение ошибок и описал, как избегать или устранять их, настало время перейти к процессу отладки.

1.4.1. Планирование отладки.

Об отладке следует подумать с самого начала, на этапе определения требований к продукту. Чем тщательнее проект будет спланирован, тем меньше времени и денег позже будет затрачено на его отладку. Как уже упоминалось ранее, добавление незапланированных свойств может создать большое количество проблем. Это чаще всего приводит к ошибкам и нарушению правильной работы продукта. Тем не менее это не означает, что планы не могут изменяться. Иногда бывает необходимо добавить или изменить какую-либо возможность продукта из-за конкуренции или чтобы лучше удовлетворить потребности пользователя. Но прежде чем заняться кодом, нужно точно определить и спланировать, что именно будет меняться. Следует помнить, что добавление новых возможностей затрагивает не только код, но отражается и на тестировании, документировании, а и иногда даже затрагивает маркетинговые заявления. Добавление новых возможностей является причиной изменения графика выпуска; здесь необходимо придерживаться правила: время на добавление или удаление возможности растет экспоненциально по мере продвижения в цикле разработки.

В работе [9] автор приводит затраты на устранение ошибки. Эти затраты минимальны на этапах определения требований к продукту и планирования. Но в процессе развития продукта стоимость устранения ошибки , как и стоимость отладки, растет экспоненциально и в основном по такому же сценарию, что и при удалении или добавлении новых возможностей.

Планирование отладки должно выполняться одновременно с планированием тестирования. При этом следует рассматривать различные способы ускорения и улучшения обоих процессов. Одна из лучших мер предосторожности, которые можно предусмотреть, заключается в написании программ, выдающих дампы данных для файлов, делающих проверку внутренних структур данных, а при

необходимости и двоичных файлов. Если в проекте выполняется запись данных в двоичный файл и чтение данных из него, необходимо автоматически выделить время для написания тестовой программы, выводящей данные из файла в текстовый файл в удобной для чтения форме. Эта программа должна также проверять данные и взаимосвязи в двоичном файле данных. Такой шаг облегчит и отладку, и тестирование.

Правильное планирование отладки позволяет уменьшить затраты времени на нее, к чему следует стремиться.

1.4.2. Предпосылки для отладки.

Перед тем как перейти к содержанию отладки, необходимо перечислить необходимые знания, требующиеся хорошему специалисту по отладке. Как правило, эксперты в этой области, сами являются квалифицированными разработчиками, так как нельзя быть хорошим экспертом в этой области, не будучи хорошим разработчиком, и наоборот.

Необходимые знания и умения. Хорошие специалисты по отладке, а следовательно, и разработчики должны не только обладать отличными навыками решения специфичных задач программирования, но и они понимать взаимосвязь всех частей проекта. Ниже перечислены области, в которых такой человек должен быть одним из лучших специалистов.

- проект;
- язык программирования;
- используемую технологию;
- операционную систему;
- работу центрального процессора.

Проект. Отличное знание проекта - первая линия обороны от ошибок пользовательского интерфейса, логики работы и производительности. Зная, как и где реализованы функциональные возможности в различных файлах исходного кода, специалист по отладке может быстро определить, что и где происходит.

К сожалению, поскольку все проекты различны, единственный способ узнать проект - ознакомиться с его документацией, если она существует, и исполнением кода в отладчике. Например при работе с исходным кодом C++ может помочь просмотр файлов, содержащих перекрестные ссылки (Browser files). Кроме того, есть несколько инструментов для преобразования существующего кода в диаграммы UML (Unified Modeling Language - унифицированный язык моделирования). Даже плохо документированный исходный код лучше, чем ничего.

Язык программирования. Хорошего знания используемого в проекте языка (или языков) программирования, то есть понимания как язык реализован, механизмов его действия, используемых технологий, а не только умения программировать на нём. Например, разработчики иногда забывают, что локальные переменные, являющиеся классами C++ или перегруженными операторами C++, могут создавать временные объекты в стеке. Или оператор присваивания может выглядеть невинно, но приводить к выполнению большого объема кода. В Microsoft Visual Basic значительный объем кода генерируется неявно пользователя. Однако многие ошибки, особенно проблемы низкой производительности -

результат неправильного использования языка, поэтому стоит потратить время на изучение всех его тонкостей.

Используемая технология. Знание используемых технологий - первый важный шаг в устранении наиболее трудных ошибок. Например, если известно, каким образом создаются объекты COM (Component Object Model - модель компонентных объектов) и возвращается их интерфейс, то легко определить, почему запрос конкретного интерфейса завершился неудачей. То же самое относится и к библиотеке MFC (Microsoft Foundation Class - библиотека базовых классов Microsoft). Если в документе возникают проблемы с приемом сообщений Windows, разработчик обязан знать, как распространяется поток сообщений в архитектуре документ-вид. Всё это не означает, что нужно помнить наизусть строку из исходного кода или книги. Необходимо иметь общее представление об используемых технологиях, а главное, точно знать, где найти более подробную информацию, если она понадобится.

Операционная система. Знание операционной системы отличает целенаправленный поиск ошибки от случайного блуждания. Вот некоторые вопросы об операционной системе, на которые специалист, занимающийся отладкой, должен ответить: что такое библиотека DLL? Как работает загрузчик исполняемых файлов? Как работает реестр?

При вызове функций операционной системы многие коварные ошибки возникают в результате передачи неверных данных или непонимания последствий выполнения такого вызова.

Например, в программе происходит утечка памяти, и не получается найти, в каком именно модуле. Знание операционной системы поможет ответить на этот вопрос. Разработчик должен знать, что вся память, в конечном счете, выделяется при вызове функции API VirtualAlloc. Поэтому, установив точку останова в этой функции, можно выяснить при просмотре стека вызовов, какой из модулей выполнил вызов.

1.4.3. Процесс отладки.

Начать рассматривать практическую отладку необходимо с обсуждения ее процесса. Определить данный процесс, работающий для любых ошибок, даже странных (неожиданно возникающих и не имеющих объяснения) достаточно сложно.

Как будет показано ниже, в процессе отладки нет ничего необычного. Сложно лишь всегда придерживаться этого процесса. Ниже перечислены этапы процесса отладки:

1. воспроизведение ошибки;
2. описание ошибки;
3. определение типа ошибки;
4. определение сущности ошибки, её расположения;
5. выбор пути устранения ошибки;
6. выбор необходимых инструментов;
7. глубокая отладка, устранение ошибки;
8. тестирование;
9. учёт приобретённого опыта.

В зависимости от типа ошибки можно пропустить некоторые этапы, если, например, местоположение и причина очевидны. Но начинать следует всегда с первого шага и обязательно выполнить второй. В диапазоне шагов 3-7 будет найдено решение проблемы и устранена ошибка. Затем нужно перейти к шагу 8 и выполнить тестирование. На рис. 1.1 приведены шаги процесса отладки.

Шаг 1: воспроизведение ошибки

Этот шаг наиболее критичен в процессе отладки. Иногда сделать это достаточно сложно или даже невозможно, но без воспроизведения ошибки ее не удастся и устранить. Возможно, для этого потребуются радикальные меры.

После того как удалось воспроизвести ошибку при определенной последовательности шагов, следует попытаться повторить ее при другой их последовательности. Один путь может выявлять одни ошибки, а другой - иные. Цель в том, чтобы рассмотреть поведение программы с различных точек зрения. Воспроизводя ошибку различными способами, можно лучше узнаеть, какие данные и условия приводят к ее возникновению. Кроме того, как известно, некоторые ошибки могут маскировать другие. Чем больше будет найдено способов воспроизведения ошибки, тем лучше.

Даже если невозможно воспроизвести ошибку, ее все равно следует занести в систему отслеживания ошибок. Даже если встречается невоспроизводимая ошибка, ее описание следует внести в систему. Если за эту часть кода отвечает другой программист, то он, по крайней мере, будет знать - что-то не в порядке. При записи в систему отслеживания ошибки, которую нельзя воспроизвести, следует описать ее как можно подробнее, чтобы впоследствии устранить.

Шаг 2: описание ошибки

Навыки описания для программиста не менее важны, чем опыт программирования, поскольку ошибки нужно уметь описывать устно и письменно. Столкнувшись с серьезной ошибкой, разработчик обязан ее воспроизвести и тут же описать. В идеальном варианте ее всегда следует отображать при помощи системы отслеживания ошибок.

Шаг3: определение типа ошибки

Определить тип ошибки очень важно, так как только после этого будет ясно, как её можно усттранить. Следует отметить, что только небольшой процент ошибок является результатом неправильной работы компилятора или операционной системы. Если встречается ошибка, вероятно, виноват программист. При ошибке в коде ее можно устранить; хуже, если причина в компиляторе или операционной системе. Возможную ошибку следует всегда искать сначала в собственном коде, прежде чем тратить время на поиски в других местах.

Шаг 4: определение сущности ошибки, её расположения

После того, как ошибка воспроизведена и описана, надо попытайтесь определить, в чем она заключается и где может находиться. Прежде всего, следует убедиться в правильности предположений. Иногда можно начать с поверхностной отладки, которая заключается в проверке состояний и значений переменных, а не в работе над случайно выбранными участками кода, пытаясь нащупать или угадать решение проблемы. Если за несколько минут не удается сформулировать гипотезу, следует попробовать заново оценить ситуацию. На этом шаге об ошибке известно немного больше, поэтому теперь можно пересмотреть свою гипотезу и сделать еще одну попытку.

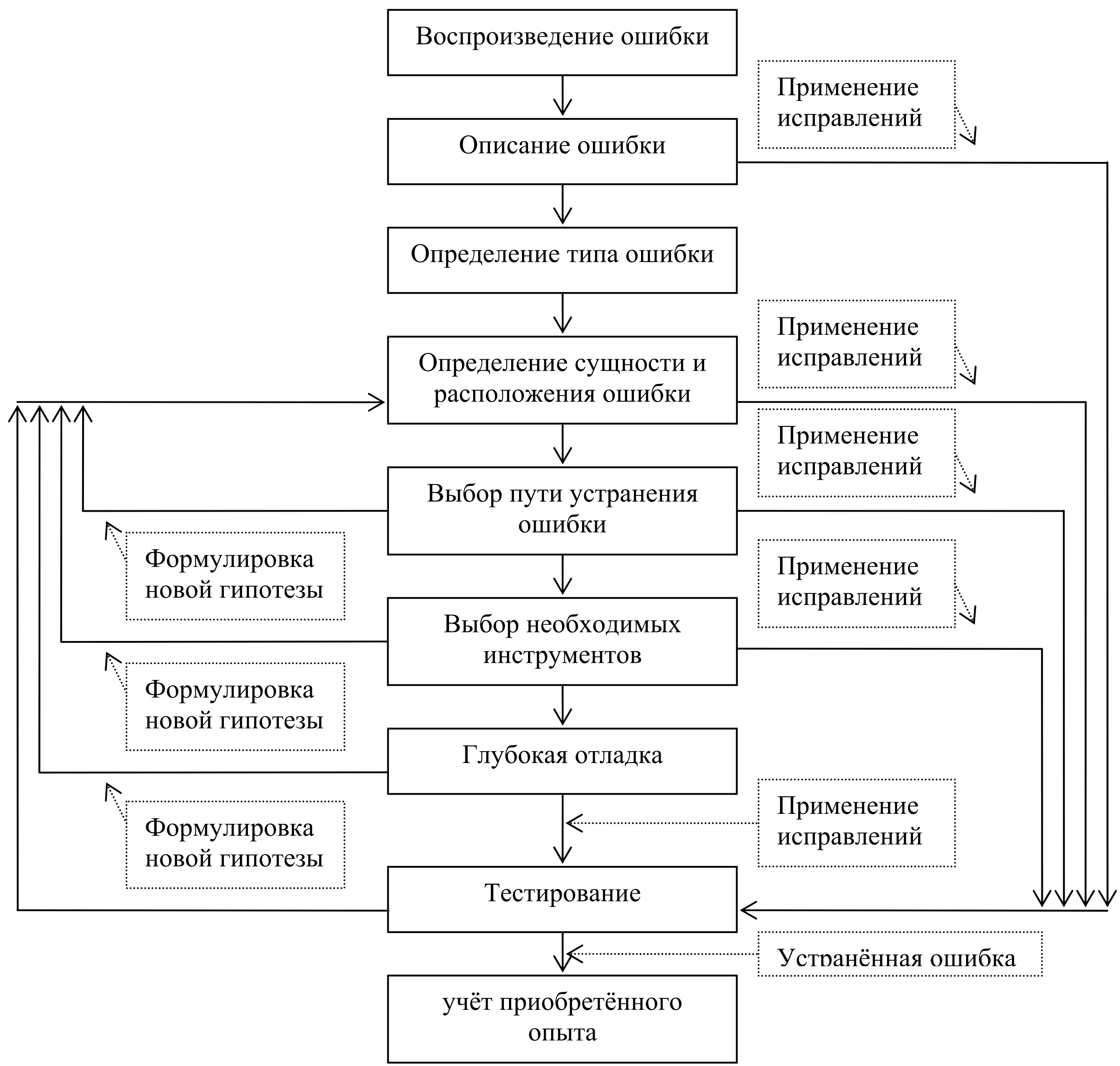

Рис. 1.1. Процесс отладки

Шаг 5: выбор пути устранения ошибки

Если ошибка, которую требуется устранить, относится к трудно воспроизводимым, которые возникают только на определённых компьютерах, следует рассмотреть её с разных точек зрения. На этом этапе необходимо анализировать, например, возможные конфликты версий библиотек DLL, различия в операционных системах и другие внешние факторы.

Шаг 6: выбор необходимых инструментов

Наличие вспомогательного инструмента может в большой степени помочь разработчикам найти ошибку. Например, при реализации проекта на Microsoft Visual C такими вспомогательными инструментами выступают программы BoundsChecker (инструмент обнаружения ошибок), TrueTime

(инструмент для измерения производительности) и TrueCoverage (инструмент для измерения покрытия кода) компании Compuware NuMega или аналогичные продукты других производителей (Rational Software, ParaSoft, Mutek Solutions, Intel). Инструмент для обнаружения ошибок определяет попытки некорркткного доступа к памяти, неправильные параметры, передаваемые системным API и интерфейсам COM, утечки памяти и ресурсов и т.д. Инструмент измерения производительности позволяет увидеть время выполнения отдельных частей программы. Инструмент для измерения покрытия кода показывает строки кода, которые не выполняются при работе программы. Эта информация также полезна при отладке, поскольку при поиске ошибки необходимо проверять только выполняеме строки кода. Дополнительный инструмент также может понадобиться и на шаге 9.

Шаг 7: глубокая отладка, устранение ошибки

Глубокая отладка отличается от поверхностной, которая упоминалась выше в шаге 1, по действиям, выполняемым в отладчике. При поверхностной отладке проверяется только несколько состояний программы и значения нескольких переменнх. А при выполнении глубокой отладки исследуется работа всей программы. Именно на этом этапе следует использовать дополнительные функции отладочного инструмента.

И при поверхностной, и при серьёзной отладкенеобходимо уже приблизительно представлять, где находится ошибка, до использования отладчика, а затем с его помощью подтвердить или опровергнуть это предположение

Во время глубокой отладки следует также регулрно описывать изменения, которые были внесены в отладчике для устранения ошибка. Это особенно важно на поздних этапах проекта, когда нужно быть предельно осторожным, чтобы не дестабилизировать базовый код.

Шаг 8: тестирование

Считая, что ошибка устранена, необходимо несколько раз протестировать сделанные исправления. Если ошибка находится в изолированном модуле и строка кода с ошибкой вызывается только один раз, протестировать её не предстовляет трудностей. Но когда исправлений сделано в одном из основных модулей, особенно обрабатывающем структуры данных, нужно быть очень осторожным, чтобы оно не вызвало проблем или побочных эффектов в других частях проекта.

При тестировании исправления, особенно в критическом участке кода, следует проверить, что оно правильно работает для всех данных, корректных и некоректных.

Шаг 9: учёт приобретённого опыта

Каждый раз, когда удается избавиться от ошибки, которую было сложно найти и устранить, следует обобщить то, чему научились. Лучше всего, записать подобные ошибки в журнал, чтобы позднее увидеть, что было сделано правильно и неправильно для их поиска и устранения. Это поможет научиться избегать тупиков и быстрее избавляться от ошибок.

Как было сказано выше, на этапе тестирования такжеможет понадобиться дополнительный инструмент. При тестировании приложения часто возникает необходимость проверить функциональность, которая активизируется только после

совершения определённых действий со стороны пользователя. Например, выбор каких-либо опций в различных диалогах. Для того, чтобы воспроизвести всю последовательноть, требуется большое количество времени. В данном случае, разработчикам следует воспользоваться code regression программами. Такие программы (например, Rational Visual Test) запоминают последовательность действий пользователя, а при тестировании сами её воспроизваодят. Подобный инструмент также может помочь в том случае, когда ошибка возникает при повторении каких-либо действий много раз (например, открыть-закрть окно).

В этой части работы было дано определение ошибок и описание причин их появления. Затем обсуждались необходимые знания для отладки. И наконец описан процесс отладки, которого следует придерживаться.

Наилучший способ отладки - избегать ошибок. Если проект тщательно спланирован, контролируется качество и хорошо известно, как работают технологии, операционная система и центральный процессор, затрачиваемое на отладку время будет сведено к минимуму.

2. ПОСТАНОВКА ЗАДАЧИ

Программисты, применяющие в своей работе язык Object RSL не имели возможности использовать какой-либо отладочный инструмент. Единственной возможностью исследовать поведение программы было использование отладочной печати. Такой подход не позволял произвести качественную отладку, т.к. возможности отладочной печати весьма ограничены, особенно это проявлялось при отладке таким образом распределённых приложений. Соответственно, возникла необходимость реализации отладочного инструмента.

Создаваемый отладочный инструмент должен обладать следующими основными функциональными возможностями:

1. производить пошаговое выполнение программ, в том числе и распределённых;
2. работать с точками останова;
3. отображать отладочную информацию;
4. отображать текущие значения локальных и глобальных переменных, а также изменять их в режиме выполнения;
5. вычислять значения пользовательских выражений;
6. работать со стеком вызовов

При пошаговом выполнении программы должны быть доступны такие сервисные команды как: пошаговое выполнение инструкций с заходом в процедуры, выполнение процедуры как единой инструкции, выполнение оставшейся части процедуры как единой инструкции и остановка на следующей за процедурой выполняемой инструкции, выполнение программы до инструкции, на которой стоит курсор (эта команда используется, например, для пропуска больших циклов). Также в задачу отладчика входит возможность переходить между модулями RSL программы, работающими на различных уровнях распределённой архитектуры.

Точки останова – это точки, на которых прерывается выполнение программы, и команды пошагового выполнения программы. Точки останова могут быть активными и неактивными. В первом случае отладчик должен активизироваться, во втором случае – нет.

Проектируемая система должна позволять просматривать отладочную информацию, формируемую в процессе работы программы с помощью специальной процедуры ***trace***.

Отображение переменных и их значений необходимо для того, чтобы программист мог анализировать текущее состояние программы. В рамках контекста процедуры выводятся локальные переменные этой процедуры и их значения, а процедуры модуля – локальные и глобальные переменные модуля и их значения. Значения переменных можно изменять. Изменённые значения должны сразу попадать в среду выполнения.

Рассматриваемая система должна позволять также просматривать значения пользовательских выражений, в качестве которых могут выступать любые инструкции языка RSL, например переменная или вызов функции.При входе приложения в режим прерывания, определенные пользователем выражения должны вычисляться в контексте соответствующей процедуры.

Стек вызовов содержит имена процедур, вызванных в процессе работы программы; вершине соответствует текущая исполняемая процедура. При выборе в этом списке любого элемента автоматически должны отображаться соответствующие ему модуль, инструкция, переменные и их значения, а также значения пользовательских выражений.

3. КРАТКОЕ ОПИСАНИЕ ЯЗЫКА OBJECT RSL

Наличие хорошо структурированного языка, тесно связанного с базой данных, значительно расширяет возможности системы. Язык может быть использован для создания силами работников банка специфических отчетов, отсутствующих в системе, или других процедур, реализующих дополнительные процедуры.

Язык Object RSL - это язык высокого уровня. Он достаточно прост в изучении и применении и одинаково хорош как для квалифицированного, так и для неквалифицированного пользователей системы, так как имеет широкий спектр функциональных возможностей. В отличие от других подобных ему языков, язык интерпретатора интегрирован с системами, разработанными программистами компании R-Style Software Lab. Он вызывается непосредственно из меню системы, обеспечивает доступ к базе данных, использует одинаковые с ней форматы данных, работает с экранными формами.

3.1. Элементы языка

3.1.1. Служебные слова

Служебное, или зарезервированное слово - это имя, с которым в языке жестко сопоставлены определенные смысловые значения, и которое не может быть использовано ни для каких других целей. Ниже приведен список служебных слов языка RSL (они записываются в любом регистре).

and	If	record
array	Import	return
const	Macro	this
class	Not	true
elif	Null	var
end	Onerror	with
false	Or	while
file	Local	Private

Кроме этого, существует отдельный список ключевых слов, действующих в пределах определений FILE и RECORD.

btr	key	txt
dbf	mem	sort
dialog	normal	write

Значение каждого служебного слова поясняется при описании конструкций языка.

3.1.2. Имена

Под именем (идентификатором) в программе понимается не являющаяся служебным словом последовательность букв (латинских и русских) и цифр, начинающаяся с буквы; символ "_" (подчеркивание) рассматривается как буква. Длина идентификатора в RSL не должна превышать 80 символов. Прописные и строчные буквы не рассматриваются как различные символы, то есть регистр игнорируется. Имена используются для идентификации объектов RSL.

Кроме обычных идентификаторов, в RSL применяются имена специальных переменных, которые могут содержать любые символы, включая специальные, без

ограничений. Такие имена должны быть заключены в фигурные скобки "{}", которые также входят в имя переменной.

Специальные переменные являются глобальными и используются обычно для передачи значений из вызывающих модулей, написанных на языках C или C++, в программу на языке RSL. Для обозначения полей в структурах и файлах, описанных в словаре базы данных, используются составные имена. Составные имена состоят из имени, идентифицирующего файл или структуру, после которого следует точка и название поля либо индекс поля в круглых скобках.

Пример:

Счет	- простое имя
Сумма010	- простое имя
{Debet account 52.5}	- имя специальной переменной
Клиенты.Name	- составное имя ссылающееся на поле Name в структуре Клиенты
Клиенты(10)	- составное имя ссылающееся на десятое поле в структуре Клиенты

Если имя переменной совпадает с именем переменной среды, оно инициализируется ее значением.

Пример:

PATH	- переменная будет проинициализирована значением системной переменной PATH, содержащей список каталогов для поиска программ

Присвоение значений этим переменным не изменяет содержимого системных переменных.

Область видимости имен.

Глобальными именами считаются те, которые объявлены вне любой процедуры RSL. Имена специальных переменных, независимо от места их появления, считаются глобальными. Для размещения глобальных объектов используется специальная область памяти, через которую осуществляется доступ к их значениям из любого места программы RSL или из вызывающего внешнего модуля.

Локальными именами считаются те, которые объявлены в любой процедуре RSL. Локальные объекты создаются заново при каждом вызове процедуры, в которой были объявлены их имена, и доступны только в этой процедуре, а также во всех вложенных в нее процедурах RSL. Следует отметить, что определенные явно имена локальных объектов могут совпадать с именами глобальных объектов. В этом случае имя локального объекта перекрывает имя глобального, и доступ к глобальному объекту будет блокирован.

Каждая RSL-программа образует глобальную область видимости. Каждая процедура, содержащаяся в программе, образует свою, вложенную область видимости. Из вложенной области видимости доступны имена этой области и всех вышележащих. В каждой области видимости можно "перекрывать" имена из вышележащих областей, то есть во вложенной области могут быть определены переменные с такими же именами, как и в вышележащих.

При входе в область видимости переменные конструируются, при выходе разрушаются. Таким образом, время жизни переменной ограничено областью ее видимости.

3.1.3. Комментарии

Любой фрагмент текста, заключенный в скобки вида /* */, является комментарием. Язык RSL разрешает наличие вложенных комментариев:

Пример:

```
/*********************************************************
      Комментарий на языке RSL
   /*
          Вложенный комментарий
   */

**********************************************************/
```

Данный пример содержит два комментария, один из которых вложен в другой.

3.1.4. Объекты языка

Объект RSL представляет собой совокупность информации, с которой оперирует язык. Объектами языка RSL являются:

- Переменная - объект, содержащий значение одного из типов данных, который может изменять величину и тип хранимого значения. Это происходит при присвоении переменной нового значения операцией присваивания (при этом старое значение теряется).
- Объект типа FILE хранит информацию о Btrieve-файле (структуру файла и текущие значения полей).
- Объект типа RECORD хранит информацию о структуре файла или диалоговой панели. Объект типа RECORD, в отличие от объекта типа FILE, не имеет явной связи с файлом. Если это необходимо, то связь с файлом должна быть задана при помощи процедур RSL или неявно в вызывающем модуле написанном на языке С или С++.
- Объект типа DBFFILE хранит информацию о файле формата DBF (структуру файла и текущие значения полей).
- Объект типа TXTFILE хранит информацию о текстовом файле (структуру файла и текущие значения полей).
- Объект типа ARRAY представляет собой вектор переменных, который можно индексировать.
- Объект типа CLASS RSL представляет собой совокупность свойств и методов и служит для создания экземпляра класса - объекта.

Все объекты RSL, кроме переменных, требуют явного определения их имени. Имена переменных нет необходимости объявлять явно - по умолчанию они считаются объявленными в момент их первого появления в тексте программы. Однако допускается объявлять имена переменных явно при помощи определения VAR.

3.1.5. Типы данных

Под типом понимается совокупность информации, определяющей данное: сколько именно нужно выделить для него памяти, какой вид имеет его представление, какие операции над ним определены. В Object RSL поддерживаются следующие типы данных:

- Variant - переменная данного типа может принимать значения любых типов данных Object RSL; используется по умолчанию для всех переменных.
- Integer - четырехбайтовое целое число.
- Money - восьмибайтовое число для денежных величин.
- Double - восьмибайтовое число с плавающей точкой.
- DoubleL - десятибайтовое число с плавающей точкой.
- String - строка символов.
- Bool - логическая величина, которая может принимать значения TRUE или FALSE.
- Date - дата.
- Time - время.
- Dttm - дата/время.
- Object - ссылка на объект.
- ProcRef - ссылка на макропроцедуру.

Значения любого из описанных типов данных хранятся в переменных RSL. Переменные RSL могут быть представлены базовыми стандартными типами или ссылками на объект:

3.1.6. Базовые стандартные типы

- V_INTEGER - для целых чисел со знаком длиной 4 байта.
- V_STRING - для символьных строк.
- V_DOUBLE - для чисел с плавающей точкой длиной 8 байт.
- V_DOUBLEL - для чисел с плавающей точкой длиной 10 байт. При задании констант этого типа необходимо в конце числа указать символ L (например: 1234.567L).
- V_BOOL - для логических переменных.
- V_MONEY - для денежных сумм (суммы представляются в копейках в виде целого числа длиной 8 байт и выводятся в отчетах с точкой, отделяющей копейки; внутреннее представление имеет тип V_DOUBLE).
- V_DATE - для дат (даты представляются в виде ДД.ММ.ГГГГ).
- V_TIME - для времени (время представляется в виде ЧЧ:ММ:СС).
- V_DTTM - для даты и времени совместно.
- V_UNDEF - для значений неопределенного типа.

Ссылки

- V_FILE - для ссылок на файл базы данных (объект типа FILE).
- V_STRUC - для ссылок на структуру, описанную в словаре базы данных (объект типа RECORD).
- V_ARRAY - для ссылок на массив (объект типа ARRAY).
- V_TXTFILE - для ссылок на текстовый файл (объект типа TXTFILE).
- V_DBFFILE - для ссылок на файл формата DBF (объект типа DBFFILE).
- V_GENOBJ - для ссылок на объект класса.

Переменная получает или изменяет тип в момент присваивания ей значения, то есть когда она присутствует в левой части операции присваивания. Если переменной не присвоено никакого значения, в том числе при явном определении ее имени (см. Yb;t), она получает тип V_UNDEF.
При использовании структуры, описанной в словаре, все ее поля получают те типы, которые были заданы в словаре.
Правила преобразования типов описаны ниже.
Кроме автоматического определения типа в Object RSL имеется возможность декларировать типы:

- переменных,
- параметров макропроцедур и методов,
- возвращаемых значений макропроцедур и методов.

Декларируемый тип переменной или формального параметра указывается через двоеточие после объявления. Тип возвращаемого значения макропроцедуры или метода класса указывается после двоеточия, которое следует за списком формальных параметров, а в случае их отсутствия - непосредственно после имени макропроцедуры или метода класса.

```
Пример:
Macro MyStrLen (str: string): integer
     Return strlen (str);
End;
```

Для декларирования типов данных можно использовать и имена классов:

```
Пример.
Определим переменной для хранения ссылки на объект тип
"MyClass", который является именем класса:
Var obj: MyClass;
```

Следует отметить, что декларирование типов не приводит к производительности RSL-программы.

3.5. Константы

Под константой понимается значение одного из типов данных.
Символьные константы имеют тип V_STRING (максимальная длина - 256 символов) и задаются в виде последовательности символов, заключенных в апострофы:

```
Примеры:
"a"
"абв_12"
"\t"
```

Последовательности, начинающиеся с обратной косой черты, ничем не отличаются от таковых в языке C:

- \n - новая строка;
- \r - возврат каретки;
- \t - символ табуляции;
- \f - перевод формата;
- \xHH, \XHH - символ, заданный кодом (здесь HH - две шестнадцатеричные цифры);
- \\ - задает одиночный символ '\'.

Предопределенные константы для работы с диалоговыми окнами описаны в разделе, посвященном вводу/выводу.

3.1.7. Выражения

3.1.7.1. Синтаксис

Выражения представляют собой последовательность операндов, разделенных знаками операций.

Пример:

Name	- выражение, состоящее из одного операнда – простого имени;
MgFun(10)	- выражение, состоящее из одного операнда – сложного имени;
a+b	- сумма двух переменных;
b*10+c	- произведение и сумма переменных.

В приведенной ниже таблице операции располагаются в порядке возрастания их приоритета при вычислении выражений. Приоритет операций в каждой группе одинаков.

Таблица видов операций:

Обозначение	Операция
" = "	Присваивание
" == " " != " " < " " > " " <= " " >= "	Равно Не равно Меньше Больше Меньше или равно Больше или равно
" + " " – " " OR "	Сложение Вычитание Дизъюнкция
" * " " / " " AND "	Умножение Деление Конъюнкция
" – " " + "	Унарный минус Унарный плюс
" NOT "	Отрицание
" @ "	Получение ссылки на процедуру

Для того чтобы явно задать порядок вычисления, необходимо применять круглые скобки.

Все операции, кроме операции присваивания, имеют левую ассоциативность. Это означает, что операции с одинаковым приоритетом выполняются слева направо.

Например, выражение a+b+c интерпретируется как (a+b)+c. Для использования другого порядка вычисления необходимо использовать скобки.
Операция присваивания имеет правую ассоциативность. То есть выражение a = b = 10 интерпретируется как a = (b= 10).
Последовательность вычисления операндов в выражении не определена. Исключение составляют операции "AND" и "OR". Для них гарантируется строгий порядок вычисления операндов: сначала вычисляется значение левого операнда, и если после этого результат операции уже известен, вычисление правого операнда не производится.
Все перечисленные операции используются при работе с любыми переменными, за исключением переменных, хранящих ссылки на объекты RSL-классов, которые можно сравнивать друг с другом или с константой NULL с помощью операций == и !=.

3.1.7.2.Семантика

В зависимости от типов операндов операции определены различным образом (см. далее). Если в выражении используются операнды разных типов, и для них не описаны правила выполнения операции и определения результата, то производится повышение типа операнда с меньшим приоритетом до типа операнда с большим приоритетом. Используется следующая строка приоритетов (от меньшего к большему):
V_INTEGER -> V_MONEY -> V_DOUBLE -> V_BOOL
или
V_STRING ->V_BOOL
При необходимости могут быть явно выполнены следующие преобразования типа операнда:

- V_INTEGER => V_DOUBLE, V_MONEY, V_BOOL, V_STRING
- V_MONEY => V_DOUBLE, V_BOOL, V_STRING, V_MONEYL
- V_DOUBLE => V_BOOL, V_INTEGER, V_MONEY, V_STRING, V_DOUBLEL
- V_STRING => V_BOOL, V_INTEGER, V_DOUBLE
- V_TIME => V_STRING
- V_DATE => V_STRING
- V_DTTM => V_STRING
- V_GENOBJ => V_STRING
- V_FILE => V_STRING

Такие преобразования осуществляются или с помощью специальных встроенных процедур, или в процессе присвоения значений переменных полям в файлах и структурах, то есть полям объектов типа FILE или RECORD.
Преобразование также происходит при присвоении значения переменной с явно декларированным типом данных.

Логические операции AND, OR и NOT определены только для операндов типа V_BOOL. Все арифметические операции определены для всех числовых типов данных (V_INTEGER, V_DOUBLE, V_MONEY). Тип результата бинарных операций будет таким же, как у операнда с большим приоритетом, кроме особых случаев с операндами типа V_MONEY.

Операция "+" определена для операндов типа V_STRING как конкатенация (соединение строк):

Операции "+" и "-" определены для операндов типа V_DATE и V_INTEGER. Результат операции имеет тип V_DATE:

Операция "-" определена для операндов типа V_DATE. Результат имеет тип V_INTEGER и представляет собой количество дней между двумя датами:

Операции "+" и "-" определены для операндов типа V_TIME. Результат операции имеет тип V_TIME:

Результатом операций "+" и "-" с операндами типа V_MONEY будет значение типа V_MONEY. Результатом операций "*" и "/" с операндами типа V_MONEY будет значение типа V_DOUBLE.

Результатом операций "*" между операндами типа V_MONEY и V_INTEGER будет значение типа V_MONEY. Результатом операций "*" между операндами типа V_MONEY и V_DOUBLE будет значение типа V_DOUBLE.

Результатом операций "/" между операндами типа V_MONEY и V_INTEGER будет значение типа V_MONEY. Результатом операций "/" между операндами типа V_MONEY и V_DOUBLE будет значение типа V_DOUBLE.

Результатом операций "/" между операндами типа V_INTEGER будет значение типа V_INTEGER, равное целой части от деления:

Операции сравнения определены для всех типов данных. Для данных типа V_BOOL определены операции сравнения равно "==" и не равно "!=".

Результатом операций сравнения является значение типа V_BOOL - ложь или истина.

3.2 Структура программы

Программа на языке RSL представляет собой последовательность конструкций языка, разделенных знаком ";".

Конструкции языка - это либо определения объектов RSL, либо определения процедур, либо выполняемые инструкции. Программа, как и процедура, может заканчиваться словом END.

Выполняться RSL-программа начинает с первой встретившейся выполняемой инструкции, которая может находиться как в текущем файле, так и в файле, подключенном посредством директивы IMPORT.

Вот пример самой простой программы на RSL:

[Привет, Мир!]

Этот пример выводит в стандартный выходной поток строку "Привет, Мир!".

Вот более сложный пример, в котором вычисляется самое длинное имя клиента, хранящееся в нашей базе данных:

```
file f (client);
var  n = 0,
      maxLen = 0,
      curLen,
      maxName = "";
macro Report
[ Самое длинное название клиента ####
  #                     ]
(maxLen:l,maxName)
end;
while (next (f))
   n = n + 1;
```

```
   message ("Обработано записей ",n);
   curLen = strlen (f.Name_Client);
 if (curLen > maxLen)
      maxLen = curLen;
      maxName = f.Name_Client
   end
end;
Report;
```

Конструкции языка, использованные в этом и последующих примерах, будут описаны в следующем разделе.

Текст программы должен находиться в текстовом файле с расширением "mac". Содержимое файла с таким расширением образует макромодуль. В принципе, всю RSL программу, содержащуюся в модуле, можно рассматривать как процедуру инициализации, имя которой - это имя файла, в котором находится программа. Синтаксически единственным отличием программы от процедуры является отсутствие у программы заголовка MACRO. Заканчиваться программа может, как и процедура, словом END.

Пример:

```
[ Это пример программы ]
end
```

Все, что расположено после слова END, игнорируется интерпретатором.

Макромодуль, или сокращенно модуль, может включать в себя определения переменных, объектов, процедур, а также выполняемые инструкции. Все инструкции модуля неявно образут процедуру инициализации модуля.

По умолчанию все переменные, процедуры и классы, объявленные в макромодуле, автоматически приобретают статус глобальных. Они становятся доступными не только внутри макромодуля, но и во всех остальных модулях.

Чтобы запретить извне доступ к переменной, процедуре модуля или классу, определенному в модуле, используется модификатор private. Таким образом, переменная или процедура становятся недоступными извне и видны только в рамках данного модуля. Если private стоит перед определением свойства или метода класса, то это свойство или метод становятся недоступными извне класса.

Если данный модификатор используется при определении переменной, то для ее декларации обязательно должно присутствовать ключевое слово VAR.

В языке RSL используется также модификатор local, который при установке перед объектом или переменной указывает, что данный объект RSL, переменная или процедура является локальным объектом процедуры инициализации или конструктора объекта. Данный модификатор применим только для процедур инициализации модуля и конструктора класса!

Локальные переменные модуля доступны только локальным процедурам модуля. Невозможно обратиться к локальной переменной внутри любой другой процедуры модуля.

Если модификатор local стоит перед свойством класса, то по аналогии с модулем, оно перестает быть свойством класса как таковым и становится локальной переменной конструктора, то есть может быть доступно только для инструкций конструктора.

Процедуры RSL могут быть вызваны как явно при помощи инструкции вызова процедуры, так и неявно вызывающим модулем, написанным на языке C,

С++. Примером неявно вызываемой процедуры является BtrError, которая вызывается неявно при возникновении ошибок Btrieve (конечно, если процедура с именем BtrError присутствует в тексте программы).

Программа, написанная на языке RSL, может использовать макромодули, хранимые в Btrieve-файле btrmac.ddf. Этот файл должен располагаться в текущем каталоге или в каталоге фалов Btrieve. В тексте программы может находиться директива IMPORT. Использование этой директивы позволяет включить в текст макромодуля информацию из других файлов, что дает возможность вынести часто используемые процедуры в отдельный файл и потом при необходимости подключать этот файл директивой IMPORT.

IMPORT blnc, common;

Примечание.

Директива IMPORT должна находиться вне определения макропроцедур.

Помимо включения файлов, директива IMPORT позволяет активизировать процедуры из стандартных модулей, находящихся в вызывающем модуле, написанном на языках С или С++. В этом случае в директиве IMPORT указываются имена стандартных модулей (эти имена могут содержать русские буквы).

IMPORT BankInter, Баланс;

Поиск указанных в директиве IMPORT файлов выполняется системой в следующей последовательности:

- среди имен стандартных модулей, написанных на языке С/С++;
- среди имен DLM-модулей, созданных с помощью инструмента DLM SDK;
- среди имен макромодулей, хранимых в Btrieve-файле btrmac.ddf;
- среди имен текстовых файлов, имеющих расширение "mac".

3.3. Конструкции языка RSL

RSL предоставляет программисту достаточно широкие возможности по управлению данными. Количество конструкций языка невелико, но вполне позволяет писать хорошо структурированные и эффективные программы. В качестве конструкций языка используются:

- выполняемые инструкции;
- определение объектов и процедур.

При описании конструкций языка используются следующие соглашения:

- синтаксические элементы, которые являются необязательными, заключаются в квадратные скобки [...];
- синтаксические элементы, которые могут повторяться произвольное число раз или отсутствовать, заключаются в фигурные скобки {...}.

1. Пустая инструкция состоит только из разделителя - точки с запятой. Она используется там, где по правилам языка могла бы находится какая-либо инструкция, а по логике программы там ничего выполнять не надо:
2. Инструкция "выражение". Данная инструкция представляет собой любое выражение RSL. Вычисленное значение выражения отбрасывается.
3. Условная инструкция IF
   Условная инструкция в RSL имеет следующий вид:
   IF '('условное выражение')' список инструкций
   { ELIF '('условное выражение')' список инструкций }
   [ELSE список инструкций] END

Количество конструкций ELIF не ограничено.

Первым анализируется условное выражение после IF. Если оно истинно, выполняется список инструкций. В противном случае последовательно, сверху вниз анализируются условные выражения после ELIF, если они есть. Если результатом сравнения какого-либо из них является истина, выполняется список инструкций, который следует за данным условным выражением.

Остальные условия при этом не анализируются.

Если в конструкции присутствует ELSE, то соответствующий список инструкций выполняется в том случае, если все условные выражения ложны.

4. Инструкция цикла WHILE

Для организации циклов в RSL используется инструкция WHILE.

Цикл WHILE имеет следующий синтаксис:

WHILE '('условное выражение')'

список инструкций

END

Список инструкций выполняется до тех пор, пока остается истинным условное выражение. Если условие ложно до входа в цикл, то список инструкций не выполняется ни разу.

5. Инструкция возврата RETURN

Инструкция RETURN применяется для выхода из процедуры или завершения всей RSL-программы. Во втором случае она должна быть указана вне любой процедуры. Применяется следующая синтаксическая форма:

RETURN [выражение];

Выражение, заданное после слова RETURN, определяет возвращаемое процедурой значение. Если в инструкции завершения RSL-программы в качестве выражения задать текст, содержащийся в символьной константе или переменной типа V_STRING, он будет выведен на стандартное устройство вывода.

6. Инструкция вывода

Инструкция вывода позволяет определить в тексте программы форму для вывода в стандартный выходной поток. Выводимая форма заключается в квадратные скобки:

[Этот текст без изменений будет выведен в стандартный выходной поток]

Внутри текста формы могут находиться поля, в которые необходимо выводить посчитанные значения из переменных языка RSL. Поля внутри формы задаются последовательностью символов '#'. Значения, которые необходимо вывести в поля, указываются сразу после формы в круглых скобках:[Номер счета ########## Остаток ##########] (Account, Summa)

В результате выполнения данного примера в выходной поток будет выведен номер счета из переменной Account и остаток на счете из переменной Summa.

3.3.1 Определение переменных VAR

Имена переменных RSL по умолчанию считаются объявленными в момент их первого появления в тексте программы. Однако при этом достаточно часто возникают ошибки из-за неточного написания имени в различных частях программы. Если вы хотите избежать этого, необходимо объявлять имена переменных явно, используя определение VAR. Кроме объявления имени, можно проинициализировать его каким-либо значением.

Синтаксическая форма определения VAR имеет следующий вид:

```
[local | private]VAR идентификатор [: имя типа ] [ '=' выражение ] { ',' идентификатор [: имя типа ] [ '=' выражение ] }
```

Список идентификаторов может содержать произвольное количество имен переменных. Инициализация заключается в присвоении идентификатору переменной значения какого-либо выражения. Допускается инициализировать не все объявленные переменные. Конструкции, относящиеся к отдельным переменным, должны быть разделены запятой.

После того, как определение VAR будет обнаружено компилятором в тексте программы хотя бы один раз, любая необъявленная явно переменная в текущем RSL-модуле приведет к сообщению об ошибке.

3.3.2 Определение классов и объектов

В языке RSL имеется возможность создавать классы и объекты. Классы и объекты, например, используются для поддержки программирования визуальной среды, в котором очень удобен объектно-ориентированный подход: такие ее специфические черты, как отчеты, кнопки, поля редактирования, переключатели всегда можно представить в качестве объектов со своими свойствами и методами.
Синтаксическая форма определения классов выглядит следующим образом:

```
[local | private] Class [ '('идентификатор')'] идентификатор [' ('список формальных параметров')' ]
     <свойства класса>;
     <методы класса>;
End
```

Идентификатор класса - это имя конструктора класса, которое используется при создании экземпляра данного класса.

Начиная с версии интерпретатора 852, в Object RSL добавлена возможность автоматического создания объектов классов при первом обращении к декларированной переменной класса:

Пример:

```
Class TestClass
     Var prop = 10;
End;
Var Ob:TestClass; /* Декларируем переменную типа TestClass */
Println (Ob.prop);  /* Объект автоматически создается */
```

Обращение к свойствам и методам класса осуществляется следующим образом:

```
Obj.Имя;
Obj.Отчет;
```

Кроме того, Object RSL поддерживает обращение к свойствам RSL-классов не только по имени, но и по индексу.

Пример:

```
Class Test
   Var prop1, prop2;
End;
Ob = Test;
A = ob.prop1; /* Доступ к свойству по имени */
B = ob (0); /* Доступ к свойству по индексу */
```

Для совместимости со стандартом автоматизации в Object RSL добавлена поддержка свойства по умолчанию. Свойством по умолчанию считается свойство, имя которого в выражении можно не указывать.

Классы Object RSL могут иметь и свойства с параметрами. В настоящее время такие классы можно создать, например, при помощи специального инструмента DLM SDK. Параметры свойства аналогично параметрам метода указываются в круглых скобках.

Классы Object RSL могут содержать деструктор, определяемый пользователем. Для его использования в определении класса необходимо указать метод с предопределенным именем Destructor. Деструктор вызывается автоматически при "разрушении" объекта.

Если деструктор не определен, то все выделенные при создании объекта ресурсы освобождаются автоматически.

Object RSL позволяет наследовать классы объектов от других классов. Базовый класс указывается в круглых скобках после ключевого слова CLASS. Для инициализации базового класса необходимо вызвать предопределенный метод, название которого образуется путем добавления к имени класса приставки "Init". Вызов инициализатора базового класса может располагаться в любом месте определения дочернего класса.

Множественное наследование в Object RSL не поддерживается.

Все методы классов являются виртуальными. Добавление в дочерний класс метода с именем, которое уже используется для одного из методов базового класса, вызовет замену метода родительского класса методом дочернего класса. Применение наследования с использованием виртуальных методов позволяет модифицировать функциональность базовых классов. Кроме того, в Object RSL можно заменять методы конкретного экземпляра класса. Для этого используется стандартная процедуру GenAttach.

RSL-классы можно наследовать от классов, определенных в DLM-модулях, которые создаются с помощью специального инструмента DLM SDK.

3.3.3. Определение символических констант CONST

Все символические константы должны быть объявлены явно при помощи определения CONST. Кроме объявления имени, необходимо проинициализировать его каким-либо значением.

Список идентификаторов константы будет соответствовать типу заданной величины.

Синтаксическая форма определения CONST имеет следующий вид:

```
 [local | private]CONST идентификатор [: имя типа ]  '='
выражение { ',' идентификатор [: имя типа ]'=' выражение }
```

Список идентификаторов может содержать произвольное количество проинициализированных имен констант, разделенных запятыми. Попытка изменить значение любой из объявленных констант приведет к сообщению об ошибке при компиляции.

3.3.4. Определение процедуры MACRO

Определение MACRO используется для описания процедуры RSL:

```
[local | private] MACRO идентификатор [ '('список формальных параметров')'] [: имя типа ]
     список определений и инструкций
END
```

Идентификатор процедуры - это ее имя, которое используется в инструкции вызова процедуры для выполнения.
Список формальных параметров заключается в круглые скобки.

3.3.5. Процедуры языка RSL

Процедуры являются неотъемлемой частью любой программы, написанной на RSL. Они являются фрагментом программы со своим именем, к которому можно обратиться для выполнения необходимых действий. Кроме этого, допускается использовать стандартные процедуры, входящие в состав языка RSL. Обращение к встроенным процедурам так же осуществляется по имени. Для косвенного вызова процедуры (определенной пользователем или стандартной) служит стандартная процедура execmacro. Процедуры RSL могут присутствовать в выражениях, при этом вместо процедуры в выражение будет подставлено значение, возвращаемое процедурой. Если процедура явно не возвращает значение, то возвращаемое значение будет иметь тип V_UNDEF.

3.3.5.1 Передача параметров

При описании процедуры может быть указан список параметров:
macro My (a,b,c,d)
Здесь a,b,c,d являются формальными параметрами, создаваемыми и инициализируемыми значениями фактических аргументов при входе в процедуру.
После завершения процедуры эти переменные прекращают свое существование.
Нумерация параметров начинается с нуля. При вызове процедуры после ее имени задаются значения фактических параметров, которые заменят формальные при выполнении.
Передача параметров производится по значению, поэтому изменение формальных параметров в вызванной процедуре не приводит к изменению значений соответствующих им фактических аргументов в вызывающей.
Однако пользователю предоставляется возможность изменить значения фактических параметров при помощи процедуры setparm. В этом случае в качестве изменяемого фактического параметра должна быть переменная. Если же это условие не выполняется, значение фактического параметра не изменится, при этом система не сгенерирует сообщения об ошибке.

Если количество фактических параметров меньше количества формальных, то вместо недостающих параметров передаются неопределенные значения типа V_UNDEF. Если количество фактических параметров больше количества формальных, то к избыточным параметрам можно получить доступ через обращение к процедуре getparm.
При описании процедуры можно вообще не указывать формальные параметры. В этом случае доступ к фактическим параметрам производится только через обращение к процедуре getparm.
Наличие процедуры getparm позволяет писать процедуры, принимающие произвольное количество параметров.

3.3.6. Определение массивов

В языке RSL можно объявлять только одномерные массивы. Для этого предусмотрены следующие конструкции:

- конструкция ARRAY;
- стандартный класс TArray.

3.3.7. Определения FILE и RECORD

Конструкции FILE и RECORD используются для объявления объектов типа FILE, DBFFILE, TXTFILE и RECORD. Эти определения практически одинаковы. Различие между ними заключается в следующем: конструкция FILE по умолчанию определяет объект типа FILE, а конструкция RECORD - объект типа RECORD.
Определение имеет следующий синтаксис:

```
[local | private] FILE идентификатор (имя объекта [, имя словаря])   [список параметров объекта]
[local | private] RECORD идентификатор (имя объекта [, имя библиотеки диалогов])   [список параметров объекта]
```

Идентификатор - это имя, через которое можно ссылаться на определяемый объект. В выражении идентификатор преобразуется к переменной типа V_FILE, V_STRUC, V_TXTFILE, V_DBFFILE в зависимости от типа определенного объекта. Благодаря этому идентификатор объекта можно присваивать переменным и передавать в качестве параметра процедурам.

3.4. Поддержка технологии ActiveX в RSL

В Object RSL предусмотрена поддержка ActiveX-объектов (объектов автоматизации). Пользователи Object RSL могут создавать ActiveX-объекты и обращаться к их свойствам и методам из RSL-программы.
Чтобы воспользоваться этой возможностью, необходимо импортировать модуль rslx и вызвать конструктор ActiveX-объектов. Синтаксическая форма конструкции ActiveX выглядит следующим образом:
ACTIVEX (имя объекта [, имя удаленного компьютера [, ссылка на объект]])
В качестве первого параметра передается идентификатор необходимого класса COM-объекта (progID). Вторым (необязательным) параметром является имя компьютера, на котором создается объект. В качестве третьего параметра можно передать значение TRUE, в этом случае конструктор будет пытаться вернуть ссылку на уже существующий объект, а не создавать его новый экземпляр.

```
Пример.
1) Запишем строку в MS Word:
ob = ActiveX ("Word.basic");
ob.FileNew;
ob.Insert ("Hello, World!");
ob.AppShow;
2) Подключимся к объекту автоматизации Excel.Applicacation и добавим рабочую книгу Excel. Если приложение Excel уже было запущено, то его новая копия запускаться не будет.
import rslx;
ob = ActiveX ("Excel.Application", null, TRUE);
ob.Visible = TRUE;
ob.Workbooks.add;
```

```
MsgBox ("Нажмите любую клавишу для выхода из Excel");
ob.Quit
```

Как видно из примера, способ обращения к свойствам и методам объекта автоматизации ничем не отличается от обращения к свойствам и методам объекта RSL.

ActiveX является стандартным классом языка RSL, и ActiveX-объект может быть создан с помощью процедуры GenObject. В этом случае синтаксис определения имени класса выглядит следующим образом:

"ActiveX\\<имя объекта>[\\<имя удаленного компьютера>]"

```
Пример.
Создадим объект Word.Basic:
ob = GenObject ("ActiveX\\Word.basic")
```

3.4.1. Поддержка стандартных коллекций.

Object RSL позволяет использовать стандартные коллекции объектов автоматизации. Для создания таких коллекций и доступа к их содержимому существует специальный стандарт, в значительной степени облегчающий их использование.

Так, например, в приложении Excel есть коллекция рабочих книг. Пусть переменная ob содержит ссылку на объект Excel.Application. Для создания коллекции язык RSL позволяет создать объект-перечислитель при помощи предопределенного метода CreateEnum, который автоматически добавляется к каждому ActiveX-объекту.

```
Пример:
En = ob.Workbooks.CreateEnum;
While (En.Next )
      MsgBox (En.Item.Name)
End;
```

3.4.2. Обращение к свойствам объектов с параметрами

Кроме обычных свойств, объекты автоматизации могут содержать свойства с параметрами, к которым можно обращаться из Object RSL. Рассмотрим пример:

```
ob = ActiveX ("XArray.XArray");
ob.ReDim (0,10,0,20);   /* У массива будет две размерности 
0-10 и 0- 20 */
i = 0;
while (i < 10)
   ob.Value (i,0) =  "String "+i;
    i = i + 1
end;
```

Объект XArray.Xarray реализует двумерный массив. Элементы этого массива хранятся в свойстве Value, а индексы элементов определяются параметрами данного свойства.

3.4.3. Доступ к объектам Object RSL из других языков

Все объекты классов языка Object RSL являются объектами автоматизации и, следовательно, могут быть доступны из других языков программирования. Рассмотрим пример обращения к объекту Object RSL из Visual BASIC. Объект Object RSL может быть передан в процедуру Visual BASIC, например, в качестве

параметра. Предположим, что на языке Visual BASIC реализован простой сервер автоматизации, к которому можно обратиться, указав его идентификатор VbServer.VbClass. Пусть этот сервер экспортирует следующую процедуру:

```
Public Sub TetRSL (ob As Object)
   ob.Test "From Basic", " Parm1 ", "Parm2"
End Sub
```

Теперь на языке Object RSL создадим два объекта: один - класса RSL, другой - класса Visual BASIC. Вызов метода Test класса Object RSL в языке Visual BASIC осуществляется следующим образом:

```
import rslx;
class   TestRSLClass
    macro Test (p1,p2,p3)
          MsgBox ("Вызван метод Test с параметрами: ", p1,
" ", p2, " ", p3)
     end;
end;
vb = ActiveX ("VbServer.VbClass");
rsl  = TestRSLClass;
vb.TestRSL (rsl);
```

3.4.3. Обработка событий

Объекты автоматизации могут иметь не только свойства и методы, но и события, то есть способы уведомления клиента объекта о происходящих в объекте изменениях. Object RSL позволяет обрабатывать эти события, то есть принимать сообщения от объектов автоматизации и выполнять необходимые действия. Для этого предусмотрен стандартный класс TRslEvHandler.

3.4.4. Передача параметров

Как известно, в Object RSL параметры передаются процедурам по значению. Тем не менее, вызванная процедура может изменить значение фактического параметра в процедуре, ее вызвавшей, - это делается при помощи стандартной процедуры SetParm. Что же касается методов объектов автоматизации, то они могут принимать параметры по ссылке. Чтобы передать параметры иным способом, в Object RSL предусмотрены специальные модификаторы (атрибуты):

- :i - данный атрибут используется для передачи параметров из Object RSL по ссылке;
- :iv - данный атрибут используется для передачи параметров из Object RSL как ссылок на вариант, так как некоторые методы ActiveX-объектов могут принимать параметры как ссылки на вариант, благодаря чему эти методы могут менять не только значение параметра, но и его тип;
- :v - данный атрибут используется для передачи параметров по значению в метод ActiveX-объекта. Необходимость в данном атрибуте возникает в Visual RSL, который имеет специальную настройку, согласно которой все параметры передаются методам ActiveX-объектов по умолчанию по ссылке.

3.4.5. Обработка ошибок времени выполнения

В Object RSL можно осуществить "перехват" любой ошибки времени выполнения, не допустив аварийного завершения RSL-программы. Для этого

любая макропроцедура RSL или метод класса может иметь обработчик ошибок. Записывается это следующим образом:

```
Macro Test
   <инструкции RSL>
OnError
   <инструкции для обработки ошибок>
End;
```

Если какая-либо инструкция в теле макропроцедуры Test или инструкция в процедурах, вызванных из Test, генерирует ошибку времени выполнения, то управление передается на первую инструкцию для обработки ошибок после ключевого слова OnError, е аварийного завершения RSL-программы не происходит. Если обработчику ошибок нужна информация о произошедшей ошибке, после ключевого слова OnError в скобках необходимо указать имя переменной (эта переменная может не декларироваться в теле макропроцедуры при помощи ключевого слова VAR), которая после возникновения сбоя получит ссылку на специальный объект класса TrslError, содержащий информацию о данной ошибке.

3.5. Автоматическое создание объектов Object RSL

Объекты классов создаются автоматически при первом обращении к декларированной переменной типа класса Object RSL.

3.6. Организация ввода/вывода

Ввод данных возможен двумя способами:

- интерактивным - ввод данных с клавиатуры;
- из файлов - Btrieve, текстовых и файлов формата DBF.

Вывод осуществляется:

- На стандартное устройство - файл, в который выводят процедуры print, println и инструкция вывода. Имя этого файла определяется программой, вызывающей RSL программу.
- В файлы - текстовые, Btrieve файлы и DBF файлы.

3.7. Работа с файлами

Для работы с файлами в Object RSL предусмотрены следующие конструкции:

- конструкция FILE;
- стандартный класс Tbfile;
- стандартный класс TRecHandler.

3.8. Встроенные процедуры

В данном разделе представлен список всех стандартных процедур RSL. Они охватывают довольно широкий диапазон обработки данных:

- типы и значения переменных;
- работа со строками;
- параметры процедур;
- обработка ошибок и прочие.

Как правило, в результате успешной работы эти процедуры возвращают TRUE, в противном случае - FALSE.

Стандартные процедуры ввода данных с клавиатуры
Стандартные процедуры вывода
Типы и значения переменных
Работа со строками
Параметры процедур
Математические процедуры
Внешние программы
Файлы и структуры
Классы и объекты
Обработка меню
Обработка диалоговых окон
Обработка скроллинга
Переменные
Макропроцедуры
Другие процедуры

3.9. Средство разработки расширений для языка RSL (DLM SDK)

Язык RSL является прекрасным средством настройки программного комплекса RS-Bank под требования конкретного банка. Он позволяет создавать макропрограммы, которые формируют нестандартные отчеты, проводят анализ или диагностику базы данных, выполняют другие функции. Встречаются ситуации, когда для решения поставленных задач стандартных средств языка не хватает. Для устранения данной проблемы был создан инструмент DLM SDK.

DLM SDK предназначен для разработки на языке C/C++ динамически загружаемых модулей. Использование данного инструмента позволяет дополнить RSL новыми эффективными процедурами и даже классами объектов, используя для этого все мощь компилируемых языков C/C++.

В состав дистрибутивного комплекта DLM SDK входят:

- библиотеки;
- заголовочные файлы;
- документация;
- примеры использования.

Полная документация с примерами приведена в дистрибутиве.

3.9.1. Создание и использование DLM-модулей

Двоичные DLM-модули (DLM-ы), созданные при помощи DLM SDK, загружаются в память динамически во время выполнения RSL-программы. Данный инструмент позволяет создавать модули для использования в программах:

- реального режима DOS (real mode);
- защищенных режимов DOS:
- DPMI16;
- DPMI32.

Для того, чтобы DLM модуль стал доступен RSL-программе, имя модуля без его расширения необходимо указать в директиве RSL import.

```
Пример:
import demo; //Подключение файла, содержащего код DLM-
модуля
```

Помимо процедур, вызываемых из RSL-программ, при помощи инструмента DLM SDK можно создавать целые классы объектов со свойствами и методами. Объекты классов в RSL создаются и уничтожаются с помощью механизма поставщиков объектов. В DLM-модуль можно включить код, реализующий поставщик объектов требуемого типа, и, используя функцию AddModuleObjects, зарегистрировать этого поставщика.

В этой части работы было дано краткое описание проблемно-ориентированного языка Object RSL. RSL - современный язык, разработанный с учетом нынешних требований к программному обеспечению. В компании R-Style Software Lab ведется постоянная работа по его развитию и совершенствованию.

4. СТРУКТУРА ОТЛАДЧИКА

При создании собственных программных модулей с помощью языка RSL пользователь может использовать специальную компоненту – отладчик для поиска ошибок и отладки RSL-программ.

Отладчик реализован в виде отдельного модуля и имеет графический интерфейс. С его помощью можно отлаживать любые приложения на языке RSL, как графические, так и консольные.

Отладчик можно использовать как в двухзвенной, так и в трехзвенной архитектуре. Если RSL-программа работает на удаленной машине-сервере приложений, а пользователь работает с программой-терминалом, то отладчик будет активизироваться там же, где работает программа-терминал. Соответственно, компоненты отладчика должны быть установлены там же, где и программа-терминал.

4.1. Состав отладчика, инсталляция

В состав отладчика входят следующие компоненты:

- основные компоненты отладчика:
  - RSLDBG.dll – собственно отладчик; данная компонента должна быть доступной при работе с отлаживаемой программой: она должна находиться либо в текущем каталоге, либо путь к ней должен быть определен в переменной среды PATH;
  - rdbg.dll – русские ресурсы отладчика; если эта компонента не найдена, то интерфейс отладчика будет только на английском языке;
- модули, специфичные для трехзвенной архитектуры:
  - rsexts.d32 – серверная компонента, которая представляет собой dlm-файл, позволяющий вызывать RSL-программы на терминале; его можно импортировать в макрос, который работает на сервере приложений и вызывает макропроцедуры на терминале;
  - rsextt.d32, rsextt.ini, rsscript.dll – модули, предназначенные для работы на терминале; эти компоненты используются для запуска макропрограмм на терминале, а также для взаимодействия с отладчиком RSL.

4.2. Запуск отладчика

Чтобы активизировать отладчик RSL, прикладная программа должна находиться в режиме отладки макрофайла. Если пользователь работает с АБС RS-Bank, то он должен воспользоваться режимом "Отладка макрофайла". Переход в режим отладки макропрограммы можно осуществить одним из следующих способов:

1 способ – находясь в текстовом редакторе, нажать клавишу [F11]. Это означает, что программа будет запущена на выполнение и будет вызван отладчик для первой команды в тексте программы. При этом будет

активизировано окно отладчика, и текущей инструкцией будет первая инструкция в программе.

2 способ – запустить программу на выполнение с помощью клавиши [Alt+F10] и во время выполнения нажать комбинацию клавиш [Ctrl+Break]. В этом случае активизируется окно отладчика, и текущей инструкцией будет та, которая следует за последней выполненной инструкцией.

3 способ – непосредственно в код программы вставить инструкцию DebugBreak. Далее программу необходимо запустить на выполнение с помощью клавиши [Alt+F10]. В результате этого программа прервет свое выполнение, и в окне отладчика текущей будет инструкция, следующая за инструкцией DebugBreak.

4 способ – при возникновении ошибок времени выполнения. При этом выдается диалоговое окно с информацией об ошибке и предложением запустить отладчик. Если на это предложение ответить утвердительно, то загружается отладчик. Текущей инструкцией при этом устанавливается инструкция, вызвавшая данную ошибку. В режиме отладки можно попытаться устранить причину ошибки и продолжить выполнение

4.3. Основное окно отладчика

После входа в режим отладки на экране появляется основное окно отладчика (см. Рис. 4.1). В этом окне мы видим текст программы модуля, в контексте которого был активизирован отладчик. Этот модуль является текущим. Его имя приведено в заголовке основного окна. Красным цветом подсвечена текущая инструкция, то есть инструкция, которая будет выполняться первой при запуске программы.В процессе работы с отладчиком данное окно активизируется при нажатии комбинации клавиш [Alt+0], то есть оно автоматически появляется на экране и в него переносится фокус ввода.

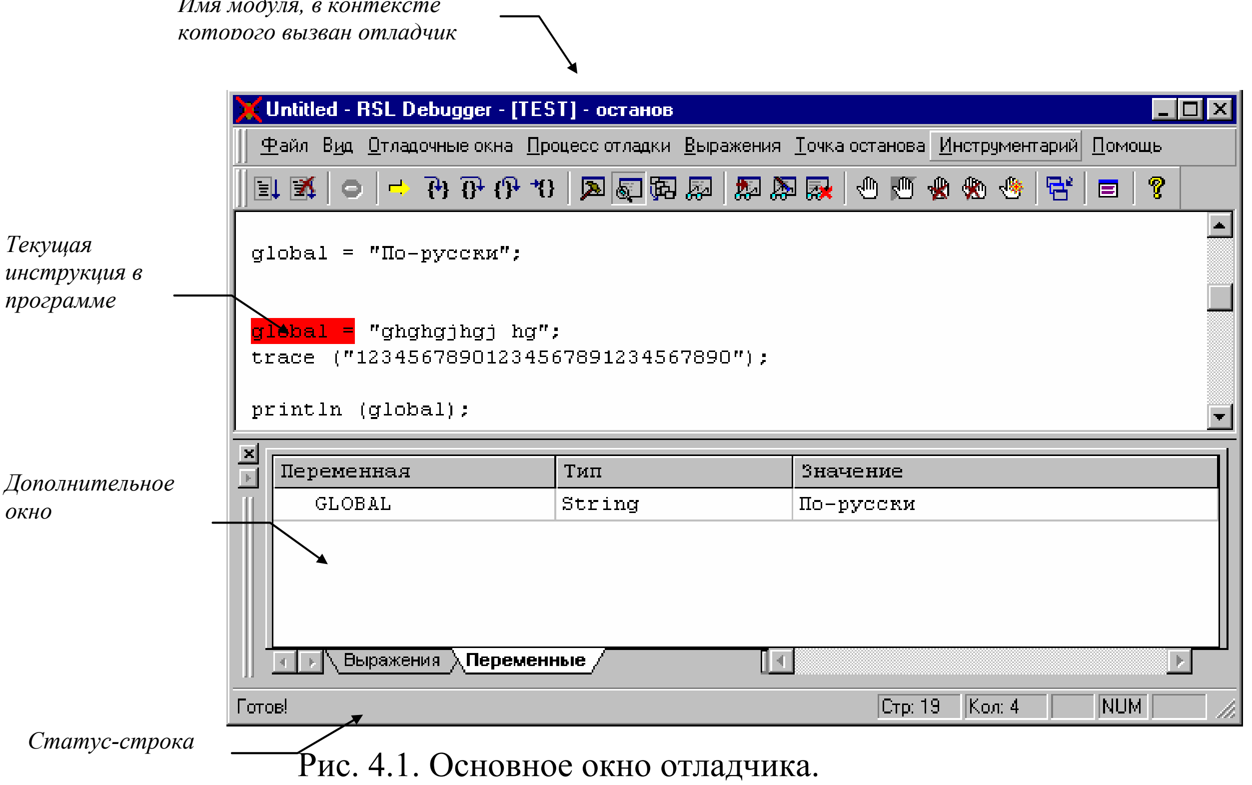

Рис. 4.1. Основное окно отладчика.

В нижней части окна окне может находиться одно или несколько вспомогательных окон, используемых при отладке программы.

Внизу располагается статус-строка. Чтобы скрыть или вывести ее на экран, нужно в меню "Вид" ("View") выбрать пункт "Строка состояния" ("Status Bar").

Управление работой отладчика осуществляется с помощью команд, сгруппированных в меню, кнопок, расположенных на панели инструментов, а также "горячих" клавиш. Каждая команда меню имеет аналог вызова с помощью "горячих" клавиш. Действия кнопок дублируют команды меню.

Пользователь может вывести на экран или убрать панель инструментов, выбрав подпункт "Отладка" ("Debug Toolbar") пункта "Панель инструментов" ("Toolbars") меню "Вид" ("View").

4.4. Отладка макропрограмм

Основным действием при отладке программ является исследование ее выполнения.

При отладке программы удобно использовать точки останова, то есть точки, на которых прерывается выполнение программы, и команды пошагового выполнения программы, описание которых приведено в данной главе. Используя команды пошагового выполнения и точки останова, можно эффективно исследовать выполнение работы программы.

– выполнение инструкции с заходом в процедуры. Эта команда позволяет проследить действие всех инструкций на переменные. Аналогом этой команды является выбор пункта "Шаг с заходом" ("Step into") меню "Процесс отладки" ("Debug actions") или нажатие клавиши [F11].

– выполнение процедуры как единой инструкции. Эта команда используется вместо предыдущей, чтобы продвигаться по вызовам процедур, не входя внутрь. Аналогом этой команды является выбор пункта "Шаг с обходом" ("Step over") меню "Процесс отладки" ("Debug actions") или нажатие комбинации клавиш [F10].

– выполнение оставшейся части процедуры как единой инструкции и остановка на следующей за процедурой выполняемой инструкции. Эта команда используется в том случае, если нет больше необходимости пошагово выполнять процедуру. Аналогом этой команды является выбор пункта "Шаг с выходом" ("Step out") меню "Процесс отладки" ("Debug actions") или нажатие комбинации клавиш [Shift+F11].

– выполнение программы до инструкции, на которой стоит курсор. Эта команда используется, например, для пропуска больших циклов. Аналогом этой команды является выбор пункта "Выполнить до текущей позиции" ("Run to cursor") меню "Процесс отладки" ("Debug actions") или нажатие клавиши [Ctrl+F10].

– продолжение выполнения программы. В результате выбора этой команды программа будет выполнена до конца либо до следующей точки останова. Аналогом этой команды является выбор пункта "Продолжить" ("Run") меню "Процесс отладки" ("Debug actions") или нажатие клавиши [F5].

– прерывание выполнения программы. Эта кнопка активизируется в тот момент, когда начинает выполняться RSL-программа. Нажав на эту кнопку, мы заставляем RSL-программу прервать свое выполнение и активизировать отладчик на текущей исполняемой инструкции. Нажатие данной кнопки эквивалентно выбору пункта "Прервать отладчик" ("Interrupt") меню "Процесс отладки" ("Debug actions") или нажатие комбинации клавиш [Ctrl+Scroll lock]. Это же действие можно выполнить с помощью нажатия комбинации клавиш [Ctrl+Break] в прикладной программе, если программа поддерживает прерывание выполнения RSL-программ по [Ctrl+Break].

– возврат на текущую инструкцию. Эта команда используется при работе с большими программами. С ее помощью можно быстро перейти к текущей инструкции и, тем самым, сократить время поиска нужной строки программы. Аналогом этой команды является нажатие клавиш [Alt+Grey*].

Для дополнительного контроля в режиме отладки предусмотрено использование дополнительных средств контроля, таких как:

- просмотр значений любых переменных и выражений;
- просмотр результатов выполнения программы в специальном окне вывода;
- просмотр стека вызовов процедур;
- просмотр списка модулей, импортируемых программой.

Чтобы прервать работу с отладчиком, необходимо нажать кнопку или выбрать в меню "Процесс отладки" ("Debug actions") команду "Закончить работу" ("Abort"), эквивалентную нажатию клавише [Shift+F5]. В результате этого выполнение программы прерывается, управление передается RSL-программе, в среде RSL появляется сообщение "Прерывание пользователя", и окно отладчика закрывается.

Закрытие окна отладчика в процессе отладки приведет к тому, что режим отладки будет прерван, и программа выполнится до конца или до следующей точки останова.

4.5. Точки останова

В окне с исходным кодом программы можно устанавливать точки останова (см. Рис. 4.2), чтобы приостановить выполнение программы на определенной инструкции в процедуре, например там, где может возникать ошибка. Если точки останова больше не нужны, их можно удалить.

Чтобы установить точку останова, необходимо поставить курсор в на, интересующиее выражение, и в меню "Точка останова" ("Breakpoint") выбрать команду "Установить" ("Set Breakpoint"), либо воспользоваться кнопкой , расположенной на панели инструментов. В результате выбранная инструкция будет подсвечена зеленым цветом. Помеченная таким образом точка останова является активной, то есть при достижении данной инструкции процесс выполнения программы будет остановлен

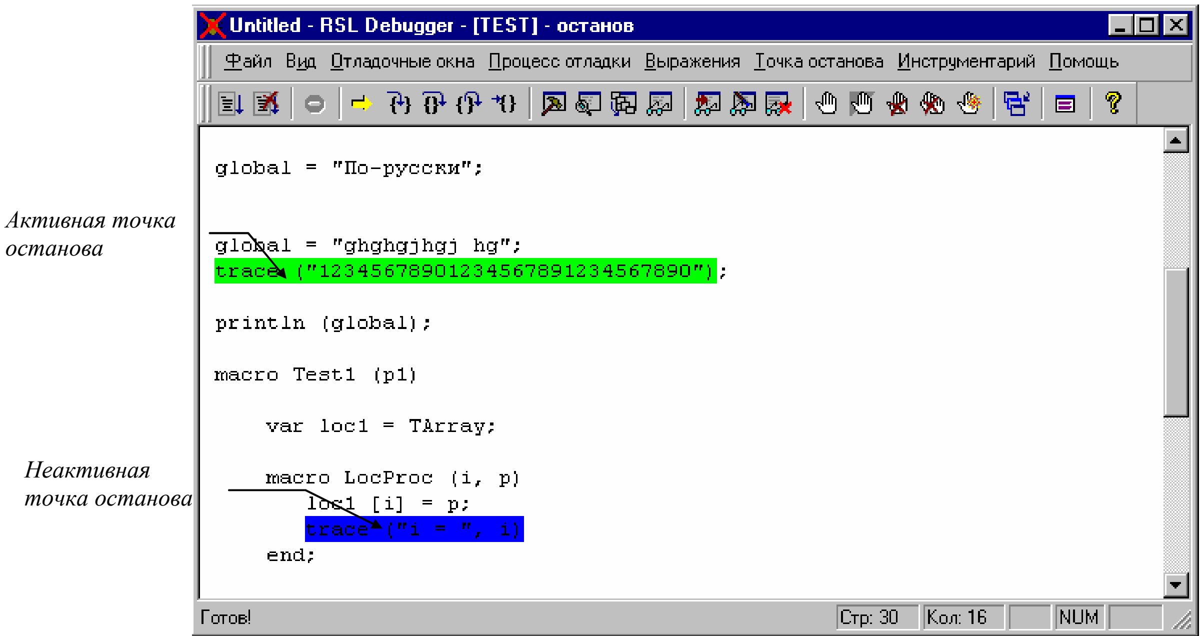

Рис. 4.2. Точки останова.

.

В отладчике имеется возможность сделать активную точку останова неактивной, то есть разрешить программе не останавливаться на данной инструкции. Для этого необходимо установить курсор на точке останова и воспользоваться командой "Разрешить/Запретить" ("Enable/Disable bp") меню "Точка останова" ("Breakpoint") или кнопкой , расположенной на панели инструментов. Неактивная точка останова подсвечивается синим цветом. При повторном выполнении этой же команды неактивная точка останова снова станет активной.

Чтобы удалить точку останова, необходимо установить на ней курсор и выбрать команду "Убрать" ("Remove Breakpoint") меню "Точка останова" ("Breakpoint") или нажать кнопку на панели инструментов. В результате этого подсветка со строки будет убрана. Если точек останова много, и нужно отменить их все сразу, для этого можно воспользоваться командой "Убрать все" ("Remove all") из меню "Точка останова" ("Breakpoint") или кнопкой .

В отладчике имеется возможность просмотра информации о всех установленных точках останова в специальном окне.

Список точек останова

Чтобы активизировать список точек останова, установленных в программе, необходимо выбрать в меню "Процесс отладки" ("Debug actions") команду "Показать список" ("Manage BP's"), эквивалентную нажатию комбинации клавиш [Ctrl+B], или нажать кнопку на панели инструментов.

В результате этого на экране появляется окно (см. Рис. 4.3), в котором содержится список всех точек останова, установленных в программе. Для каждой точки останова отображается следующая информация:

♦ признак активности точки останова;

♦ наименование модуля, в котором установлена точка;

♦ номер строки в модуле, на которой установлена точка останова.

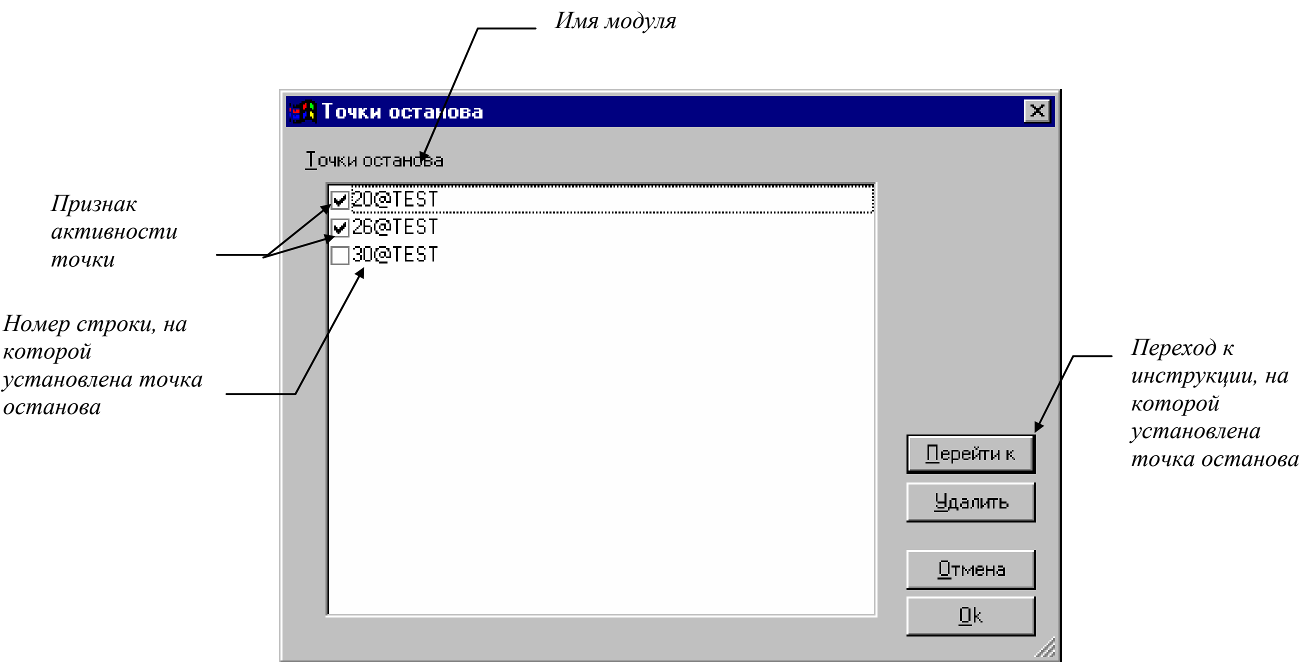

Рис. 4.3. Список точек останова.

В этом списке можно выполнить следующие операции:

♦ сделать активную точку неактивной и наоборот; для этого необходимо, соответственно, убрать или установить отметку для выбранной точки, щелкнув мышью в соответствующем поле;

♦ удалить из списка точку останова, воспользовавшись кнопкой Удалить (Remove);

♦ спозиционироваться на точке останова и перейти на нее в тексте программы в исходном коде, воспользовавшись кнопкой Перейти к (Go to).

4.6. Список импортируемых модулей

Чтобы вызвать список импортируемых программой модулей, необходимо в меню "Процесс отладки" ("Debug actions") выбрать пункт "Список модулей" ("Manage modules") или нажать кнопку на панели инструментов. Действие этой команды аналогично нажатию комбинации клавиш [Ctrl+0].В появившемся на экране окне (см. Рис. 4.4) будет представлен список модулей, которые написаны на языке RSL. Встроенные и dlm-модули в этом списке не показываются. С помощью данного списка в процессе отладки можно переключаться из одного модуля в другой. Для этого необходимо выбрать имя нужного модуля в списке и нажать кнопку Показать. В результате в основном окне появится текст указанного модуля.

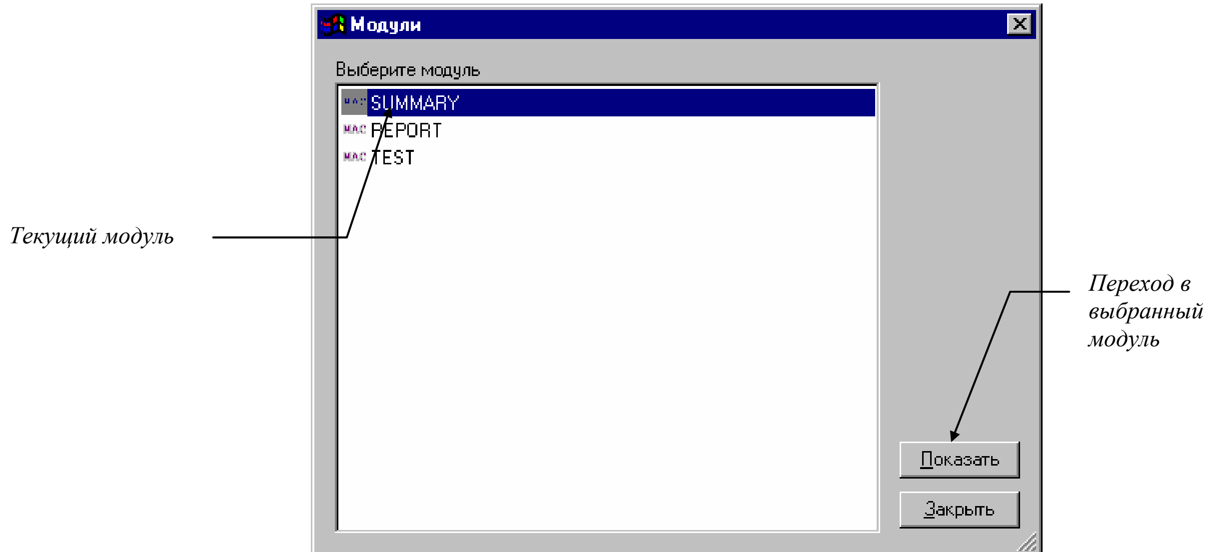

Рис. 4.4. Список модулей.

4.7. Окно проверки

Окно проверки (см. Рис. 4.5) можно вывести на экран или убрать с экрана путем выбора в меню "Отладочные окна" ("Debug window") пункта "Окно проверки" ("Show output") или нажатия кнопки на панели инструментов. Действие этой команды аналогично нажатию комбинации клавиш [Ctrl+4]. Чтобы активизировать это окно, то есть вызвать его и перенести в него фокус ввода, нужно нажать комбинацию клавиш [Alt+4].

Это окно служит для отображения информации, выводимой в процессе работы программы с помощью специальной процедуры trace. Данная процедура является специальной процедурой отладчика. Синтаксически она аналогична процедуре Println. Формат данной процедуры имеет следующий вид:

trace (<выражение>)

Таким образом, используя окно проверки, удобно наблюдать за работой программы.

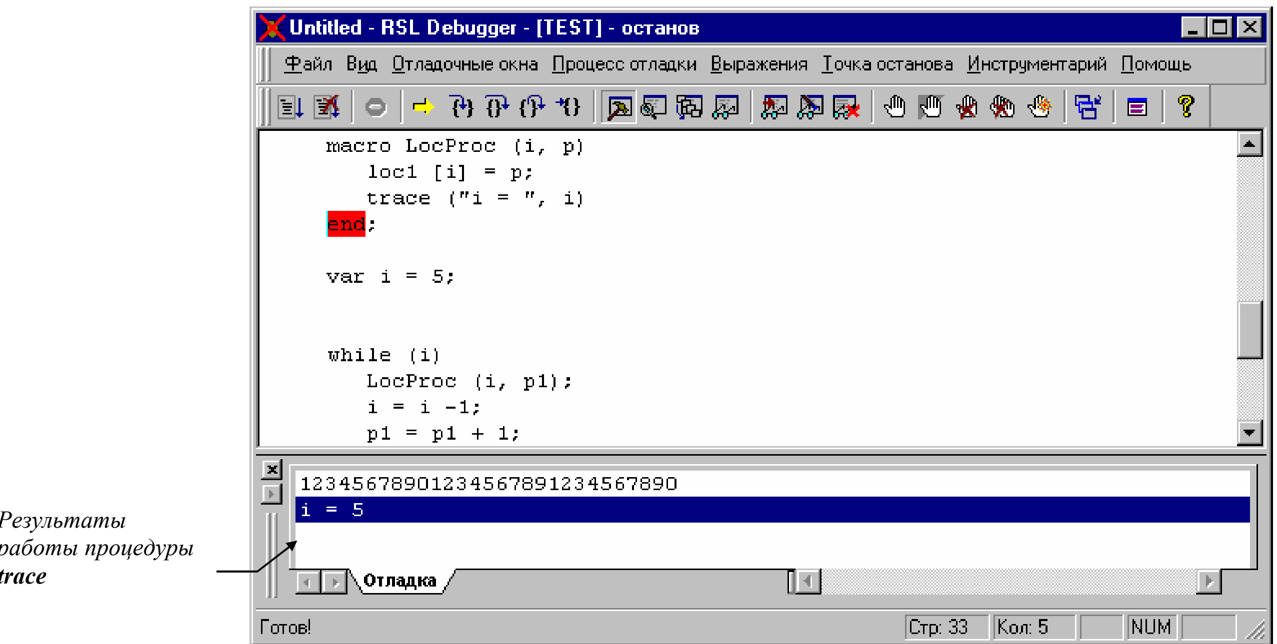

Рис. 4.5. Окно проверки.

4.8. Стек вызовов процедур

Окно, содержащее стек вызовов процедур (см. Рис. 4.6), можно вывести на экран или убрать с экрана путем выбора в меню "Отладочные окна" ("Debug window") пункта "Список вызовов" ("Show call stack") или нажатия кнопки на панели инструментов. Действие этой команды аналогично нажатию комбинации клавиш [Ctrl+3]. Чтобы вызвать данное окно и перенести в него фокус ввода, нужно нажать комбинацию клавиш [Alt+3].

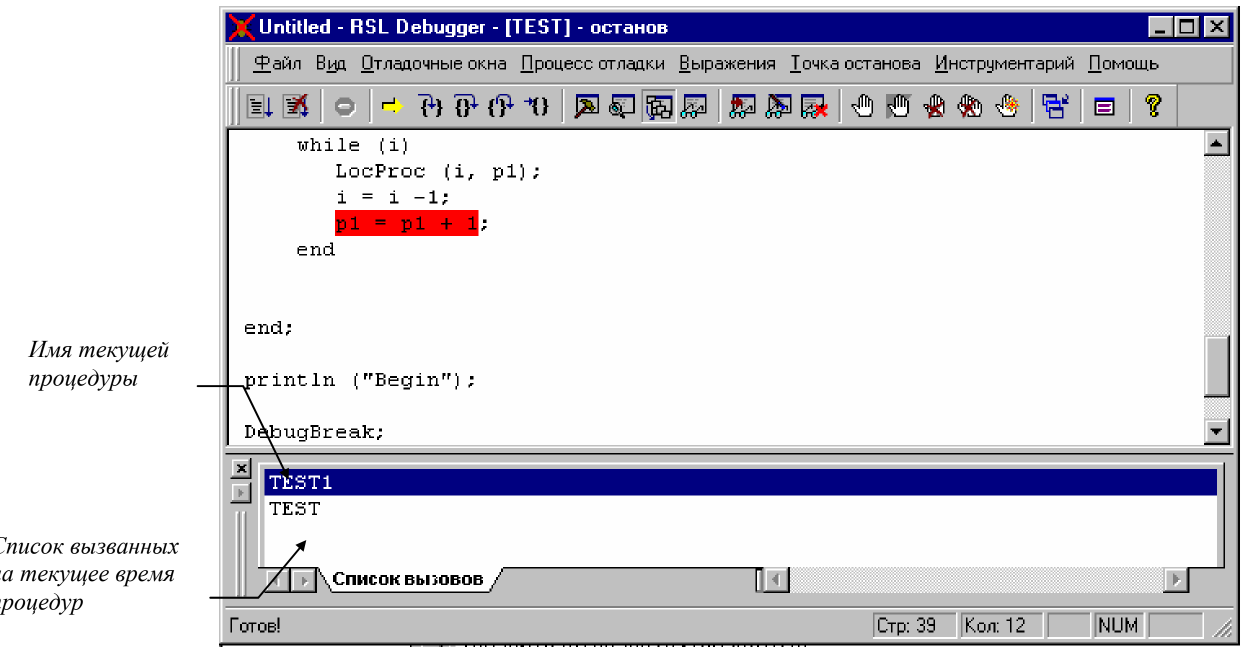

Рис. 4.6. Стек вызовов процедур.

Стек содержит имена процедур, вызванных в процессе работы программы. На его "вершине" располагается текущая исполняемая процедура. На "дне" стека находится имя модуля, содержащего эту процедуру.

В данном списке можно выбирать любую процедуру. В результате этого автоматически в окне с исходным кодом будет подсвечиваться инструкция, соответствущая выбранному элементу в стеке. При этом в окне переменных будут отображаться значения переменных выбранной процедуры.

4.9. Окно переменных

В процессе отладки удобно пользоваться окном, в котором автоматически отображаются текущие значения локальных переменных для текущей процедуры (см. Рис. 4.7).

Чтобы вывести на экран или убрать это окно, нужно в меню "Отладочные окна" ("Debug window") выбрать пункт "Переменные" ("Debug window") или нажать кнопку на панели инструментов. Действие этой команды аналогично нажатию комбинации клавиш [Ctrl+1]. Чтобы вызвать окно переменных и перенести в него фокус ввода, нужно нажать комбинацию клавиш [Alt+1].

Данное окно становиться доступным также при выборе закладки Переменные (Variables), если в окне отладчика присутствует окно выражений.

В данном окне отображаются:

- для процедур – локальные переменные этой процедуры и их значения;
- для процедуры модуля – локальные и глобальные переменные модуля и их значения.

Если мы осуществляем навигацию по элементам стека вызовов, то в окне переменных отображаются переменные и их значения для выбранной в стеке процедуры.

Значения локальных переменных можно изменять. Для этого нужно щелкнуть мышью в графе Значение (Value) на соответствующей строке списка переменных и ввести нужное значение.

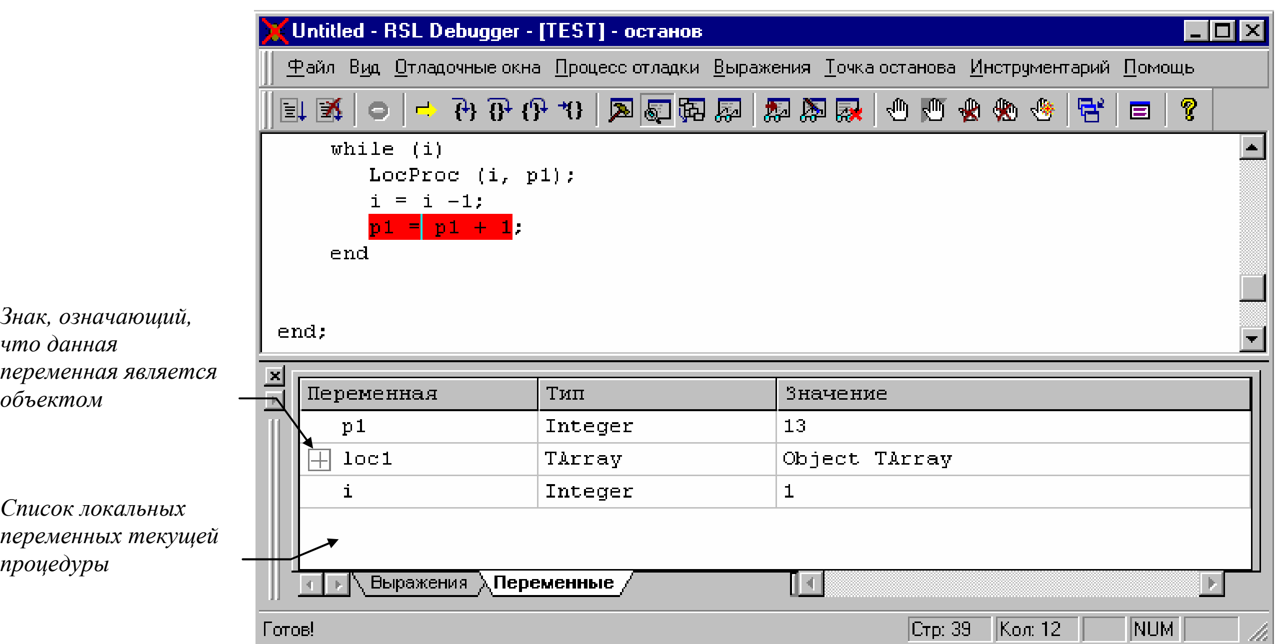

Рис. 4.7. Окно для просмотра значений локальных переменных.

Если переменная является объектом, то рядом с ее именем устанавливается знак "+". Щелкнув по этому знаку мышью, объект можно раскрыть в виде дерева и просмотреть и при необходимости изменить его свойства (см. Рис. 4.8). Эта возможность относится как к объектам стандартных классов, так и к любым другим объектам языка RSL.

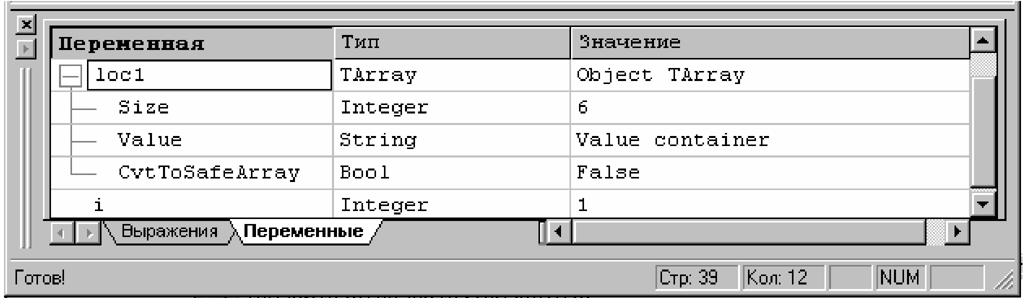

Рис. 4.8. Пример просмотра свойств объекта.

4.10. Окно выражений

Помимо окна переменных, при отладке используется окно, в котором можно просматривать любые контрольные выражения, которые нужны пользователю для дополнительного контроля в процессе отладки (см. Рис. 4.9).

Чтобы вывести на экран или убрать с экрана это окно, необходимо в меню "Отладочные окна" ("Debug window") выбрать пункт "Выражения" ("Show watch") или нажать кнопку на панели инструментов. Действие этой команды аналогично нажатию комбинации клавиш [Ctrl+2]. Данное окно становиться доступным также при выборе закладки Выражения (Variables), если в окне отладчика присутствует окно переменных. Чтобы вызвать окно переменных и перенести в него фокус ввода, нужно нажать комбинацию клавиш [Alt+2].

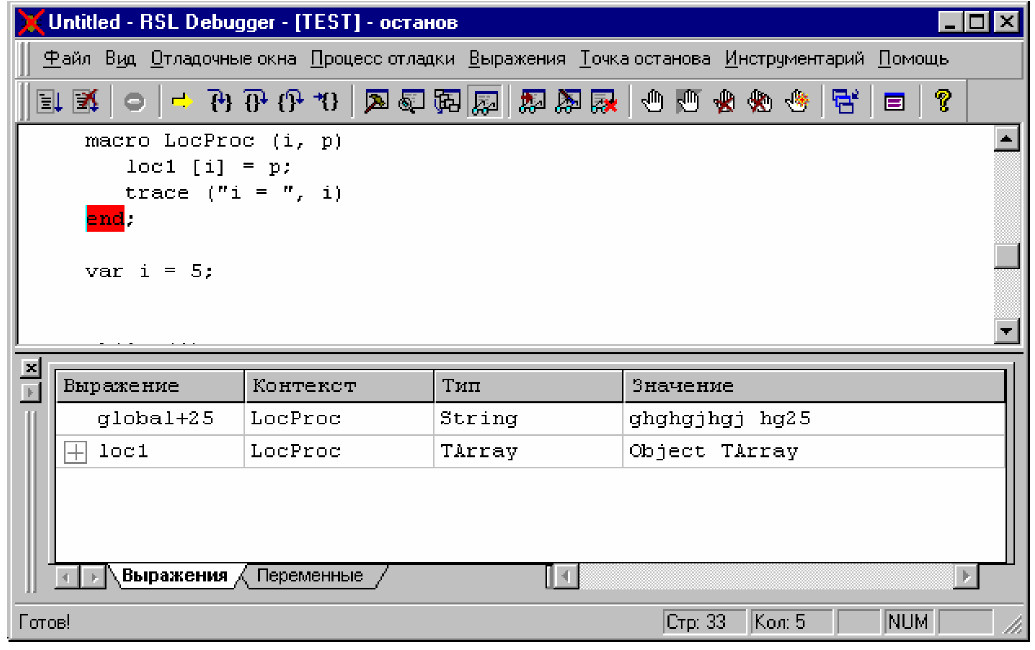

Рис. 4.9. Окно для просмотра значений выражений.

В отличие от окна переменных, в которых автоматически отображаются переменные процедур, данное окно при входе в режим отладки не содержит никаких выражений. Поэтому пользователь должен сам ввести в него нужные выражения с помощью специального режима.

Как только приложение пользователя входит в режим прерывания, определенные пользователем выражения появляются в данном окне. Если выражение является объектом RSL, то можно просматривать и редактировать его свойства так же, как и в окне переменных.

В данном окне предусмотрен специальный режим редактирования выражения.

Удалить выражение можно, воспользовавшись пунктом "Удалить" ("Delete watch") в меню "Выражения" ("Watch"), пунктом "Удалить" ("Delete watch") в дополнительном меню, вызываемом при нажатии правой кнопки мыши, или нажав кнопку на панели инструментов. Данная команда эквивалентна нажатию комбинации клавиш [Ctrl+D].

Ввод выражения

Чтобы ввести выражение, необходимо активизировать специальное окно для ввода (см. Рис. 10) одним из следующих способов:

- в меню "Выражения" ("Watch") выбрать пункт "Добавить" ("Immediate window");
- в дополнительном меню, вызываемом при нажатии правой кнопки мыши, выбрать пункт "Добавить" ("Add watch");
- нажать кнопку на панели инструментов;
- нажать комбинацию клавиш [Ctrl+A].

Чтобы ввести выражение, достаточно в верхней строке окна ввести его и нажать кнопку Добавить (Add). В результате это выражение будет помещено в список.

Перед вызовом окна можно выделить в окне с исходным кодом интересующее выражение и оно автоматически появится в диалоговом окне.

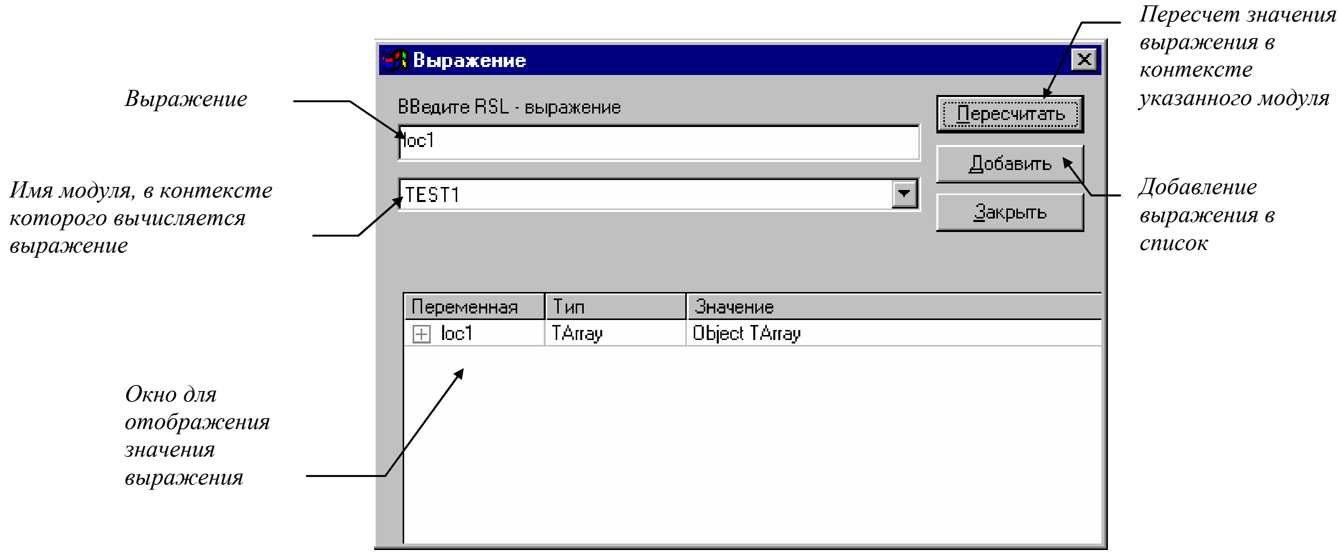

Рис. 10. Ввод и вычисление значения выражения.

В качестве выражения можно использовать переменную, вызов функции или любое выражение RSL. Если при вводе присвоить переменной тип какого-либо объекта, то в процессе вычисления его ззначения будет вызван конструктор этого объекта.

Во второй строке окна указывается контекст, в котором нужно вычислять значение данного выражения. По умолчанию там содержится текущее значение в стеке. В этой строке можно с помощью раскрывающегося списка выбрать любую позицию стека и, нажав кнопку Пересчитать (Recalculate), вычислить выражение в контексте этой позиции. Далее, нажав кнопку Добавить (Add), мы поместим в список выражение, которое будет всегда вычисляться в указанном контексте.

4.11. Редактирование переменной

Чтобы редактировать выражение, необходимо выполнить одно из следующих действий:

- в меню "Выражения" ("Watch") выбрать пункт "Изменить" ("Edit watch");
- в дополнительном меню, вызываемом при нажатии правой кнопки мыши, выбрать пункт "Изменить" ("Edit watch");
- нажать кнопку на панели инструментов;
- нажать комбинацию клавиш [Ctrl+E].

В результате этого на экране появится окно для редактирования, аналогичное окну для ввода выражения (см. Рис. 10).

С помощью этого окна можно:

- изменить значение выражения путем редактирования значений в полях Тип (Type) и Значение (Value);
- вычислить значение выражения в контексте какой-либо позиции стека;
- просмотреть значение какой-либо переменной (локальной или глобальной), которая находится в области стека; если этой переменной нет в области видимости, то ее значение будет не определено;

  Примечание:

  Следует отметить, что изменять значения можно только для выражений, которые принадлежат к так называемому типу LValue. То есть, переменной А можно присвоить значение, а значение выражения А+20 изменить нельзя.

4.12. Настройка отладчика

Чтобы выполнить настройку отладчика, необходимо выбрать в меню "Инструментарий" ("Tools") пункт "Настройки" ("Preferences") или нажать кнопку на панели инструментов.

С помощью данного режима (см. Рис. 11) можно выбрать язык, который будет применяться в интерфейсе (русский или английский), а также выбрать шрифт в окне с исходным кодом и во всех остальных окнах. При первом запуске отладчика по умолчанию устанавливается английский интерфейс.

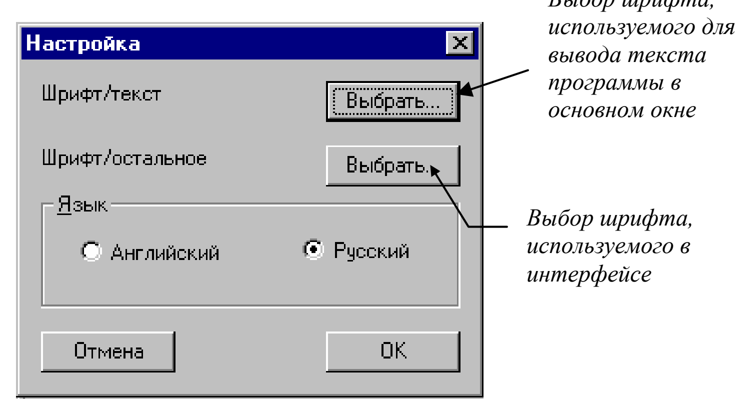

Рис. 11. Настройка параметров отладчика.

Установленные с помощью данного режима настройки сохраняются в системном реестре по следующему пути: HKEY_CURRENT_USER\SoftWare Lab\RSL Debugger.

4.13. Список горячих клавиш отладчика

Закончить работу отладчика	[Shift+F5]
Шаг с обходом	[F10]
Выполнить до текущей позиции	[Ctrl+F10]
Шаг с заходом	[F11]
Шаг с выходом	[Shift+F11]
Прервать выполнение программы	[Ctrl+Break]
Установить точку останова	[F9]
Удалить точку останова	[Ctrl+F9]
Сделать точку останова активной\неактивной	[Alt+F9]
Стек вызовов процедур	[Ctrl+3]
Окно локальных переменных	[Ctrl+1]
Окно значений	[Ctrl+2]
Окно вывода	[Ctrl+4]
Список импортируемых модулей	[Ctrl+0]
Редактировать выражение	[Ctrl+E]
Удалить выражение	[Ctrl+D]
Добавить выражение	[Ctrl+A]
Список точек останова	[Ctrl+B]
Активизировать окно, содержащее текст программы	[Alt+0]
Активизировать окно выражений	[Alt+2]
Активизировать окно локальных переменных	[Alt+1]
Активизировать стек вызовов	[Alt+3]
Активизировать окно вывода	[Alt+4]
Закрыть окно	[Alt+F4]

5. УСТРОЙСТВО ОТЛАДЧИКА.

5.1. Способ реализации, используемые библиотеки

Отладчик представляет собой SDI-приложение, созданное на основе архитектуры «документ-вид-контроль» библиотеки MFC (Microsoft Foundation Classes) со следующими изменениями и дополнениями:

1. Отсутствует поддержка класса документа
2. Изменена концепция класса приложения, т.к. отладчик не является самодостаточным приложением.
3. Для реализации современного интерфейса (например пристыковыающихся окон) была использваона библиотека Business Component Gallery

5.2. Загрузка и инициализация отладчика

При загрузке библиотеки отладчика происходит процедура первичной инициализации отладчика и соединение его с ядром языка Object RSL:
Интерпретатор вызывает у отладчика фкнкцию TRslDbg * RSDBG RslGetInterface (void), в ответ на которую отладчик возвращает таблицу указателей на управляющие функции:

- DbgVersion возвращает версию отладчика
- DbgCount возвращает количество копий отладчика
- DbgInit выолнение инициализации отладчика
- DbgDone завершение работы экземляра отладчика
- DbgBreak отработка точки останова
- DbgTrace отработка вывода отладочной информации
- DbgAddModule отработка добавления модуля при интерпретации программы
- DbgRemModule отработка удаления модуля
- DbgSetMode установка режима работы программы

Сначала интерпретатор вызывает DbgVersion, чтобы определить версию отладчика и инициализировать свои структуры в соответствие с возвращаемым значением.
DbgSetMode вызывается для того, чтобы отладчик знал, отлаживается консольное или оконное приложение – в зависиости от этого соответствующим образом будет происходить передача строк между отладчиком и интерпретатором. Далее, интерпретаор вызывает DbgInit, передавая в качестве параметра указатель на экземляр интепретатора и указатель на таблицу функций, благодаря которым отладчик имеет возможность совершать свою работу. Такая структура инициализации нужна для того, чтобы одновременно можно было отлаживать несколь программ в рамках одного макроса (язык Object RSL позволяет создавать новые экземпляры интепретаора, например, указывая среде, что макрос create_report.mac следует выполнить на клиенте). После этого создаётся собственно экземпляр отладчика. Само же окно отладчика возникнет лишь в тот момент, когда пользователю понадобиться его увидеть (напрмер, интерпретатор дойдёт до выполнения инструкции DebugBreak). В этот момент для нужд отладчика создастся дополнительный поток выполнения, в котором будет работать его message – процедура. Такая процедура инициализации необходима для того, чтобы освободить ресурсы компьютера – например, если интерпретатор работает в

отладочном режиме, но по ходу выполнения программы нет ни ошибок, ни процедуры DebugBreak.
(исходного текст процедуры инициализации приведён в приложении.)

5.3. Особенности окон приложения

Окна отладчика

1. Тескт программы
2. Стек вызовов
3. Списки переменных и наблюдаемых выражений
4. Диалоги

Окно с текстом программы.

Реализовано на основе элемента упраления RichEdit. MFC по умолчанию работает с версией бибиотеки 1.0 этого контрола. Эта версия не поддерживает такой важной функциональности, как поддержка различных цветов для текста, что позволило бы сделать подсветку нужных выражений. В ругих же программах, работающих на основе этого элемента управления, можно наблюдать текст, например, выделенный различными цветами. Для того, чтобы поддержать вторую версию richedit необходимо добавить определение (define) _RICHEDIT_VER 0x0200. После этого появилась возможность подсвечивать точки останова различными цветами (активная инструкция – красным, активная точка остнова зелёным, неактиваная красным). Также подобный поход позволил обойти такой недостаток остальных отладчиков, как неорпеделённость процедуры в строке, в которой установлена точка, т.к. другие ставят на всю строчку целико, а не на конкретную инструкцию.

Окно стека вызовов:

В этом окне содержится стек вызовов. При выборе эемента из списка, загрузится соответствующий модуль и обновятся данные в окне переменных и нблюдаемых выражений. Работа с этим окном подробно описана в части 2.

Окна списка переменных и списка наблюдаемых выражений:

Подробно эти окна описаны ниже, в разделе «описание работы отладчика». Здесь же я хочу описать интересную задачу, связанную с реализацией отображение списка переменных и отображения объектов и их свойств в этих окнах. Успешное решение этой задачи было очень важным вопросом, т.к. бóльшая часть работы с отладчиком происходит именно в этих окнах. Для этого был реализован элемент упраления grid обладающий следующими возможностями:

1. отображать данные в виде таблицы
2. отображать дерево произвольной глубины
3. позволять динамически добавлять/удалять элементы в/из этого дерева
4. позволять изменять значения в ячейках

Такая задача была решена. О важности этого элемента управления говорит тот факт, что при общем объёме проекта немногим более 25 тысяч строк, реализация описваемого грида составляет более 13 тысяч строк.

Особенностью списка наблюдаемых выражений в описываемом отладочном инструменте является работа с контекстом, в рамках которого выисляются выражения.

5.4. Отладка в локальном режиме

После загрузки и инициализации библиотеки отладчика взаимодействие между отладочным инструментом и ядром языка RSL происходит по следующей схеме (рисунок 5.1):

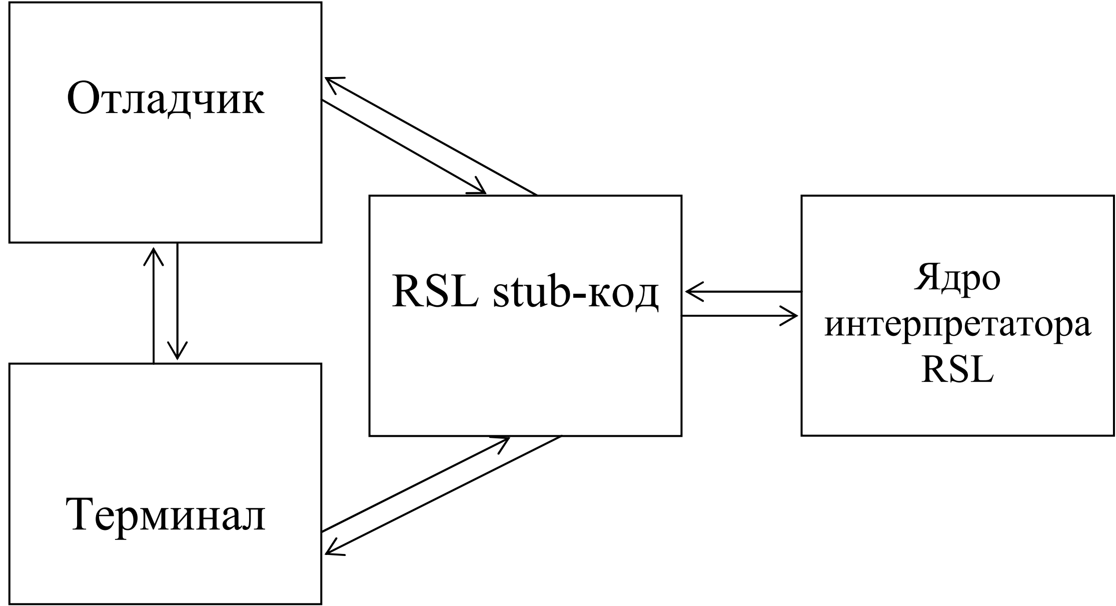

Рис. 5.1. Схема отладки в локальном режиме

- Отладчик – рассматриваемый отладочный инструмент.
- Терминал – программа-терминал, в которой происходит выполнение RSL программы.
- Ядро интерпретатора RSL – программа, осуществляющая саму работу RSL – программы
- RSL stub код – часть программы, которая 1) осуществляет маршалинг переменных, 2) позволила ввести в язык поддержку процедуры `trace`, 3) позволило отладчику работать в отдельном потоке выполнения.

5.5. Отладка в удалённом режиме

Рассматриваемый отладчик позволяет отлаживать распределённые RSL приложения; также он позволяет переходить между модулями RSL программы, работающими на различных уровнях распределённой архитектуры. Под распределённой архитектурой подразумевается трёхзвенная архитектура программ семейства RS-Bank, описанная в приложении.

Эменты схемы, представленной на рисунке 4.2, совпадают с элементами предыдущей схемы. Разница состоит в том, что взаимодействие между rsl stub-кодом и ядром происходит по сети. Это взаимодействие может осуществляется по протоколам TCP/IP, NetBIOSm IPX/SPX или по именованым каналам (named pipes), в открытом или зашифрованном виде; протокол, шифрование и метод авторизации определяются настройками терминала.

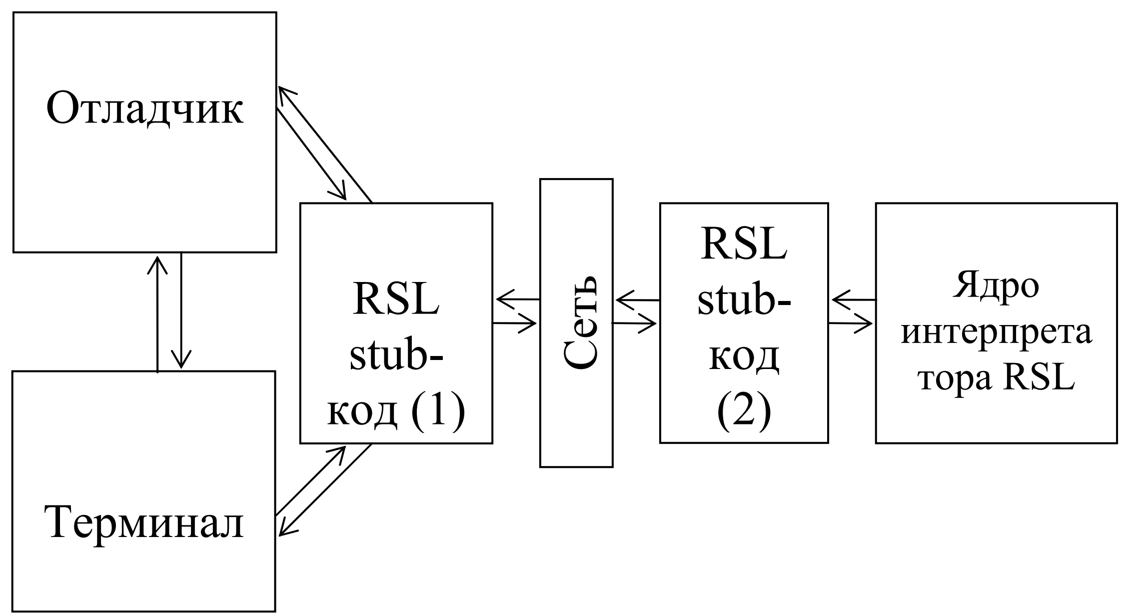

Рис. 5.2. Схема отладки в распределённом режиме

6. ПЕРСПЕКТИВЫ РАЗВИТИЯ

В качестве дальнейших этапов развития отладчика можно указать следующие пункты:

- встаривание отладчика в создаваемую среду разработки распределённых приложений RS-Forms [11];
- реализовать подсветку синтаксиса;
- сохранение раскрытых веток в списках переменных и наблюдаемых значений;
- сохранение в файл списка наблюдаемых выражений и точек останова с дальнейшей возможностью загрузки сохранённых данных.

7. ЗАКЛЮЧЕНИЕ

В результате выполнения дипломной работы был создан интерактивный отладчик распределённых RSL-приложений. Для этого

1. Изучена предметная облать. Рассмотрены 1) общие проблемы отладки, 2) отладочный интерфейс языка Object RSL, 3) отладочные инструменты различных сред разработки, а также соответствующие вспомогательные инструменты
2. Сформулированы следующие требования к отладочному инструменту:
    - производить пошаговое выполнение программ, в том числе и распределённых;
    - работать с точками останова;
    - отображать отладочную информацию;
    - отображать текущие значения локальных и глобальных переменных, а также изменять их в режиме выполнения;
    - вычислять значения пользовательских выражений;
    - работать со стеком вызовов
3. Выбран метод реализации - в виде отдельной динамически подключаемой библиотеки, что позволяет использовать отладчик как в создаваемой визуальной среде разработки, так и в используемом в настоящее время консольном инструменте.
4. Выбран инструмент реализации - Microsoft Visual C с применением библиотеки MFC.
5. Произведена отладка и тестирование готового приложения.
6. Выполнено внедрение отладчика.
7. Сформированы перспективы развития созданного отладочного инструмента.

СПИСОК ЛИТЕРАТУРЫ

1. Пользовательский интерфейс Microsoft Windows — М.: Русская редакция, 1999;
2. The Windows Interface Guidelines for Software Design: MSDN;
3. В. Головач, Дизайн пользовательского интерфейса – М.: Usethics, 2001;
4. Создание эффективных Win32-приложений – М.: Русская редакция, 2001;
5. Пользовательская документация языка Object RSL;
6. S. McConnel, Rapid Development – Microsoft Press, 1996;
7. S. Maguire, Debugging the Development Process, Microsoft Press, 1994;
8. J. McCarthy, Dynamics of Software Development, Microsoft Press, 1995;
9. S. McConnel, Code Complete – Microsoft Press, 1993;
10. Дж. Роббинс, Отладка Windows-приложений – М.: ДМК, 2001;
11. Б. Страуструп, Язык программирования C++ - М.: Бином, 1999;
12. Л. Аммерааль, STL для программистов на C++ - М.: ДМК, 1999;
13. Дж. Роджерсон, Основы COM - М.: Русская редакция, 1997;
14. Интернет ресурс Microsoft Developers Network;
15. Пользовательская документация NuMega Softice;
16. Пользовательская документация NuMega DevPartner Studio.

ПРИЛОЖЕНИЯ

1. Описание многоуровневых архитектур приложений семейства RS-Bank

Необходимым условием существования архитектуры «клиент—сервер» является наличие хотя бы двух независимых процессов обработки информации. Часто эти процессы работают на разных компьютерах. В этом случае их взаимодействие обеспечивает сеть.

Объект, функция которого — выполнять определенные действия и предоставлять результаты для дальнейшего использования другими объектами будем называть сервером. Очевидно, что в качестве сервера может выступать как компьютер, так и программа.

Результаты работы, предоставляемые сервером, мы считаем сервисом, а процесс, использующий их, — клиентом.

Когда в рамках одной системы существует несколько видов сервисов, предоставляемых различными серверами, мы имеем дело с многозвенной архитектурой «клиент—сервер»

Актуальность применения
Сейчас уже ни у кого не вызывает сомнения, что любое серьезное программное приложение, в том числе и финансовое, не может быть разработано без использования архитектуры «клиент—сервер». Естественно, что масштабируемые приложения, работающие как в локальных, так и в глобальных вычислительных сетях, наиболее продуктивно создаются при модульной разработке их компонент, т.е. с использованием архитектуры «клиент—сервер».

Выбирая эту технологию, разработчики руководствуются следующими факторами:

- возможность использовать широкий спектр аппаратных платформ, различных по архитектуре, производительности и другим параметрам:
  - персональные компьютеры;
  - компьютеры на основе RISC-процессоров;
  - мейнфреймы;
- использование огромного многообразия программных платформ: DOS; Windows; Windows NT; OS/2; UNIX; NetWare и др.;
- бурное развитие средств передачи информации:
  - оборудования для организации локальных и глобальных сетей, коммутируемых и выделенных каналов связи;
  - волоконно-оптических линий связи;
  - каналов радиосвязи и спутниковой связи;
- широкое распространение специализированного программного обеспечения удаленного выполнения заданий, внедрение в повседневную практику электронной почты, экспансия Internet;

- большое разнообразие СУБД и средств организации интерактивного взаимодействия пользователя с компьютером.

Перед тем, как приступить к реализации проектов, использующих архитектуру «клиент—сервер», фирме-разработчику программного обеспечения следует очертить себе круг программно-аппаратных средств, на которых будут базироваться эти проекты . Попытка «объять необъятное» — получить программные комплексы для любых компьютеров и любых ОС — вряд ли окажется удачной. Это подтверждают и многочисленные примеры отечественной и мировой практики.

Компания «R-Style Softlab» при создании своих программных продуктов в архитектуре «клиент—сервер» рассматривает следующие программно-аппаратные платформы (см. Таблицу 1).

Таблица 1. Состав аппаратно-программных платформ в разработках «R-Style Softlab»

Компоненты	Состав
Компьютеры	Рабочие станции: • IBM-совместимые компьютеры на базе Intel-процессоров Серверы(Здесь термин «сервер» используется в обычном значении — «ведущий компьютер локальной сети».): • IBM-совместимые компьютеры на базе Intel-процессоров; • • компьютеры на базе RISC-процессоров (в частности производства фирм «Hewlett Packard» и SUN)
Операционные системы	DOS; Windows95; Windows NT; NetWare; UNIX
СУБД	Btrive; SyBase; MS SQL
Линии связи	• локальные сети, работающие по протоколам TCP/IP; IPX/SPX; • коммутируемые и выделенные каналы

Используя данные программы и оборудование компания «R-Style Softlab» создала программное обеспечение для оснащения банков и в настоящее время работает над новыми проектами.

В зависимости от цели использования, АБС, как и другое финансовое ПО, условно можно разделить на три компоненты:

1. Клиентская часть — программное обеспечение взаимодействия пользователя с системой.
2. Бизнес-процедуры — процедуры специализированной обработки данных.
3. База данных — средство хранения и поиска информации.

А теперь рассмотрим реализацию этих компонент на различных программно-аппаратных платформах.

Схемы построения приложений в архитектуре «клиент—сервер»

Существуют следующие схемы построения банковских приложений в архитектуре «клиент—сервер»:

- Удаленный доступ к данным;
- Эмуляция сервера приложений;
- Удаленный вызов процедур;
- Интегрированный сервер приложений;
- Выделенный сервер приложений.

Эти схемы могут одновременно использоваться в одном программном продукте, а также взаимодействовать между собой, находясь в составе нескольких связанных друг с другом программных комплексов. Компания «R-Style Softlab» применяет их при разработке своих программных продуктов.

Удаленный доступ к данным (RDA-модель)

Эта схема построения программного комплекса реализована во всех существующих версиях АБС RS-Bank, начиная с версии 3.1 (1993 г.) до выпущенной в 1996 г. версии 4.21.

Состав программно-аппаратной платформы, на которой была реализована схема удаленного доступа к данным приведен в Таблице 2.

Таблица 2. Программно-аппаратная платформа и распределение прикладных компонент

Компоненты	Клиент	Сервер БД
Аппаратная платформа	IBM-совместимый компьютер	Компьютер на базе Intel-процессора
Программная платформа	DOS; Windows95; Windows NT WorkStation	NetWare; Windows NT Server
СУБД	Btrieve Requester	Btrieve MKDE
Прикладные компоненты	Клиентская часть; Бизнес-процедуры	База данных

В схеме удаленного доступа к данным клиентом являются бизнес-процедуры, размещенные на рабочей станции. В качестве сервера рассматривается Btrieve MKDE (Micro Cernel Database Engine) — микроядро СУБД Btrieve. Основная функция Btrieve MKDE — предоставление бизнес-процедурам интерфейса прикладного программирования Btrieve — Application Programming Interface (API). Следовательно, API Btrieve в данной схеме является сервисом (см. Рис.1).

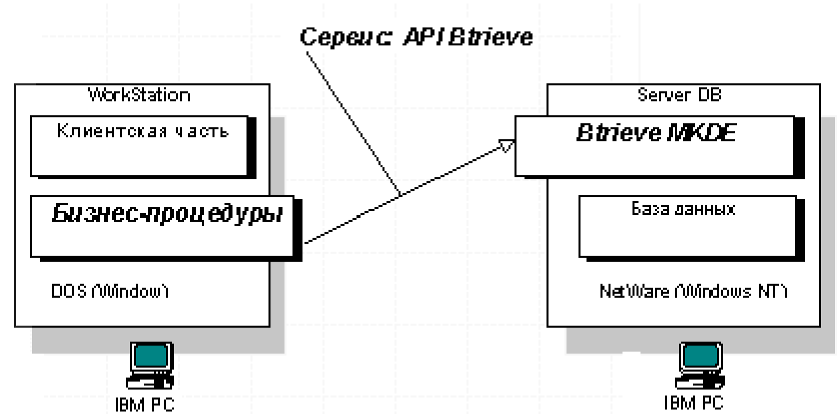

Рис.1. Удаленный доступ к данным.

Приведенная на рис.1 схема взаимодействия компонент в архитектуре «клиент—сервер», пожалуй, наиболее простая. Однако многолетний опыт эксплуатации программных продуктов (RS-Bank и RS-Balance), построенных по этой схеме, позволяет говорить о ее высокой надежности и эффективности.

К недостаткам схемы удаленного доступа к данным можно отнести высокие технические требования к рабочим станциям для последней версии (4.21) программного комплекса RS-Bank: процессор не ниже 486DX, RAM не менее 4 Кб, а также то, что пользователь имеет фактически прямой доступ к базе данных. В некоторых случаях это может провоцировать возможность попытки несанкционированного доступа к данным.

Развитие архитектуры «клиент—сервер» позволило разработать схему, лишенную указанных недостатков.

Эмуляция сервера приложений

Схема «Эмуляция сервера приложений» использована в очередной версии АБС RS-Bank v. 4.3. В пользовательский интерфейс новой версии программного комплекса было внесено очень мало изменений (надеемся, что перейти от АБС RS-Bank v. 4.21 к RS-Bank v. 4.3 пользователям будет просто), а вот архитектура построения системы изменена коренным образом.

В Таблице 3 представлен состав программно-аппаратной платформы, использованной для эмуляции сервера приложений.

Таблица 3. Программно-аппаратная платформа и распределение прикладных компонент

Компоненты	Клиент	Сервер приложений	Сервер БД

Аппаратная платформа	IBM-совместимый компьютер	Компьютер на базе Intel-процессора	Компьютер на базе Intel-процессора
Программная платформа	Windows95; Windows NT WorkStation	Windows NT Server	Windows NT Server; NetWare;
Специализированное ПО	Коммуникационная компонента на базе TCP/IP	Монитор прикладных процессов; Коммуникационная компонента на базе TCP/IP	Btrieve MKDE
Прикладные компоненты	ПО эмуляции специализированного оконного терминала	Верхний уровень ПО клиентской части; Бизнес-процедуры	База данных

Схема эмуляции сервера приложений в архитектуре «клиент—сервер» имеет двухзвенную структуру (см. Рис.2):

I звено обеспечивает взаимодействие пользователя с системой. Сервером в этом звене является монитор прикладных процессов. При помощи предоставляемого им сервиса — команд специализированного оконного терминала — осуществляется доступ клиента к информационным ресурсам системы. II звено во многом аналогично описанной выше схеме удаленного доступа.

Основным отличием можно считать то, что бизнес-процедуры (клиент этого звена схемы) — 32-разрядные приложения. Сервером, как и в предыдущей схеме, здесь является Btrieve MKDE, а сервисом, который он предоставляет, — API Btrieve.

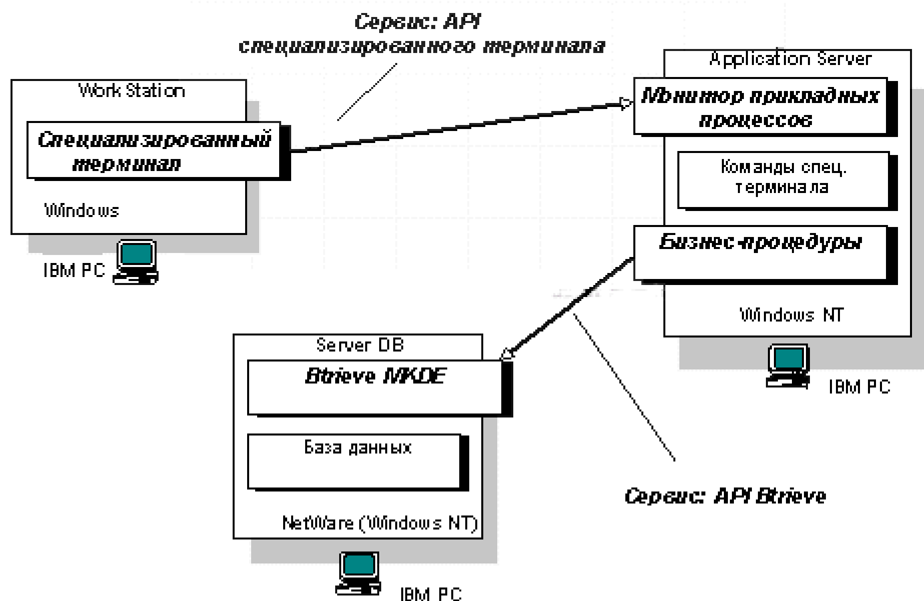

Рис.2. Схема эмуляции сервера приложений.

Эта схема имеет ряд преимуществ по сравнению со схемой, приведенной на рис.1. Остановимся на них подробно.

Прежде всего следует сказать о том, что значительно увеличивается быстродействие системы. Происходит это, во-первых, вследствие использования 32-разрядных приложений, а, во-вторых, за счет того, что намного упрощается состав информации, передаваемой с терминала пользователя в систему: времени на передачу одной команды специализированного терминала уходит значительно меньше, чем на обработку инструкции управления данными.

Повышается уровень защиты данных — пользователь получает доступ к данным только через сервер процедур Windows NT Server, который обеспечивает защиту степени C2.

Кроме того, в значительной мере снижаются требования к производительности рабочих станций.

Появляется возможность организовать эксплуатацию удаленных рабочих мест в режиме реального времени (с использованием выделенных или коммутируемых каналов связи). Такой путь значительно эффективнее, чем применение WinView и WinFrame в ранних версиях АБС RS-Bank.

Удаленный вызов процедур
Одним из важных преимуществ программных комплексов RS-Bank и RS-Balance является их открытость: язык интерпретатора RSL дает возможность пользователям разрабатывать собственные процедуры обработки данных, необходимость которых диктует технология работы, применяемая в конкретной организации.

Дальнейшее развитие этого преимущества возможно при использовании схемы удаленного вызова процедур, программно-аппаратная платформа которой приведена в Таблице 4.

Таблица 4. Программно-аппаратная платформа и распределение прикладных компонент схемы удаленного вызова процедур

Компоненты	Клиент	Сервер процедур
Аппаратная платформа	IBM-совместимый компьютер	Компьютер на базе Intel-процессора
Программная платформа	DOS; Windows95; Windows NT WorkStation	Windows NT Server
Специализированное ПО	RSL в составе ПО клиента; Клиент RS-Mail или Host RS-Mail	Менеджер обработки запросов процедур;

		RSL в составе ПО сервера; Клиент RS-Mail или Host RS-Mail
Прикладные компоненты	Клиентская часть;	База данных, Бизнес-процедуры

Клиентом в этой схеме (см. Рис. 3) выступают программы, написанные пользователем на языке интерпретатора RSL. Для их работы необходим специальный сервис, который создается пользователем также на языке интерпретатора RSL, но эти макропрограммы располагаются на другом компьютере — сервере системы. В качестве сервера выступает менеджер запросов на выполнение удаленных макропроцедур, который входит в состав ПО удаленного центра обработки информации. Сервер позволяет использовать все функциональные возможности другой системы. Передача данных обеспечивается при помощи системы электронной почты RS-Mail.

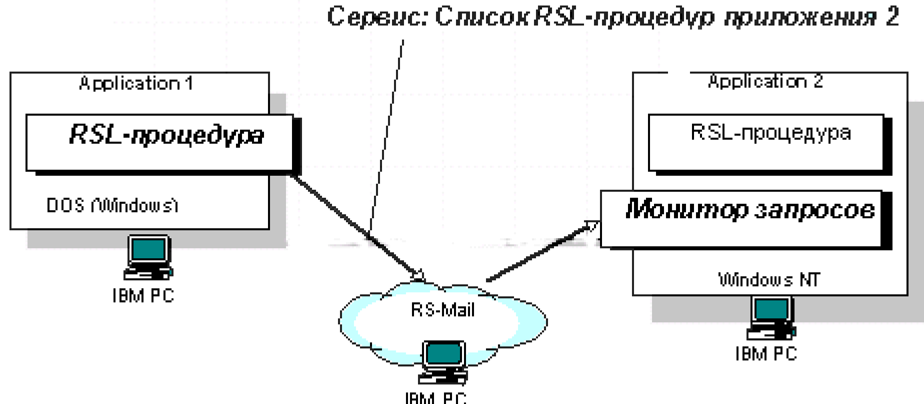

Рис. 3. Схема организации удаленного доступа.

Схема организации удаленного доступа позволяет запускать с рабочей станции одной ЛВС (на которой установлена система удаленного доступа) или с локального компьютера процедуры, написанные на языке RSL и находящиеся в другой ЛВС или на другой локальной машине. При этом монитор запросов контролирует наличие вызываемой процедуры и права доступа к ней конкретного исполнителя.

Программный комплекс, построенный в такой архитектуре, позволяет, например, обеспечить работу дирекции многофилиального банка в режиме сеансовой обработки информации с базами данных отделений: ввод ограничений на выполнение некоторых операций; получение отчетов; внесение изменений в справочники и т.п. Выполнение перечисленных функций и получение результатов можно выполнять без участия самого филиала.

Аналогичную схему работы системы в архитектуре «клиент—сервер» предполагается использовать и для обеспечения проекта «Финансовое кольцо» (см.

статью В. Овсия «Финансовое кольцо компании "R-Style Softlab"» — «RS-Club», 1996, Нулевой номер, с. 9—11).

Схема организации удаленного доступа заинтересует тех пользователей программных комплексов RS-Bank и RS-Balance, которые с успехом применяют в своей работе язык интерпретатора RSL. У них появится возможность организации собственных серверов, сервис которых можно создавать на привычном RSL.

Интегрированный сервер приложений
Все ранее описанные схемы построения «клиент—серверных» приложений не могут полностью удовлетворить требования многофилиальных банков, с разветвленной структурой отделений, которые ведут учет операций в единой базе данных. Связано это, в первую очередь, с тем, что для обеспечения работы в режиме реального времени сети, насчитывающей несколько тысяч пользователей, недостаточно вычислительных ресурсов серверов на базе Intel-процессоров.

Здесь нужна другая программно-аппаратная платформа (см. Таблицу 5).

Использование в качестве сервера компьютера на базе RISC-процессора, операционной системы UNIX и СУБД Sybase, а также реализация бизнес-функций в виде объектов базы данных (т.е. хранимых процедур) позволит обеспечить необходимое быстродействие и оптимальную эффективность работы системы.

Таблица 5. Программно-аппаратная платформа и
распределение прикладных компонент схемы «Интегрированный сервер базы данных»

Компоненты	Клиент	Сервер БД	Сервер приложений
Аппаратная платформа	IBM-совместимый компьютер	Компьютер на базе RISC-процессора	Компьютер на базе RISC-процессора
Программная платформа	Windows95; Windows NT WorkStation	UNIX, (Windows NT Server)	UNIX (Windows NT Server)
Специализированное ПО, СУБД	—	Sybase (MS SQL)	Open-Server
Прикладные компоненты	Клиентская часть	База данных; Бизнес-процедуры	Бизнес-процедуры

Структура такой системы наиболее близка к «классическому» определению архитектуры «клиент—сервер».

Схема системы (см. Рис.4), так же как и схема эмуляции сервера приложений, двухзвенная:

I звено — обеспечение организации взаимодействия пользователя с процедурами обработки данных.

Клиентом в этом звене системы являются модули организации интерфейса пользователя. При их разработке используются средства визуализации Microsoft и язык C++, что позволяет обеспечить максимальную дружественность интерфейса и повысить его быстродействие. Для работы с данными сервер (СУБД SyBase или MS SQL) предоставляет этим модулям в качестве сервиса API-бизнес-процедуры сервера базы данных.

II звено — выполнение приложений, которые не могут быть эффективно реализованы средствами СУБД (например: управление доступом, расчет процентов по кредитам и депозитам и т.п.).

Сервером этого звена является Open Server СУБД SyBase, клиентом — процедуры, расположенные на сервере приложений (32-разрядные программные модули), а сервисом — API бизнес-процедур сервера приложений.

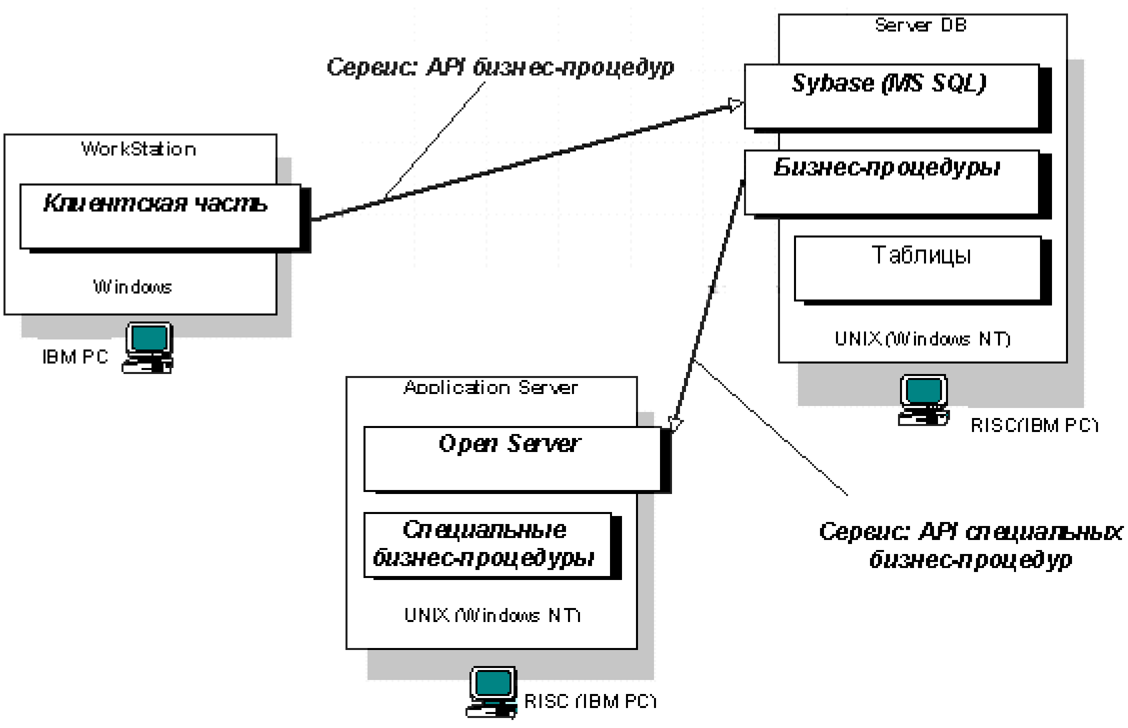

Рис. 4. Интегрированный сервер базы данных.

(Основные принципы АБС, в разработке которых планируется использовать схему «Интегрированный сервер базу данных» подробно описаны в статье В.Чаусова «Концептуальное построение банковской системы» — см. «RS-Club» № 3, с. 13—19 — Ред.)

Выделенный сервер приложений
Возможности интегрированного сервера приложений могут оказаться избыточными для построения относительно небольших (на несколько десятков или сотен рабочих станций) автоматизированных систем, а это негативно влияет на соотношение «цена/качество» таких программно-аппаратных комплексов.

Однако похожую схему взаимодействия компонент программного комплекса можно реализовать на другой программно-аппаратной платформе (см. Таблицу 6).

Таблица 6. Программно-аппаратная платформа
и распределение прикладных компонент схемы «Выделенный сервер приложений»

Компоненты	Клиент	Сервер приложений	Сервер БД
Аппаратная платформа	IBM-совместимый компьютер	Компьютер на базе Intel-процессора	Компьютер на базе Intel-процессора
Программная платформа	Windows95; Windows NT WorkStation	Windows NT Server	Windows NT Server; NetWare
Специализированное ПО	—	Монитор многонитиевых процессов	Btrieve - MKDE
Прикладные компоненты	Клиентская часть	Бизнес-процедуры	База данных

Как наглядно представлено в Таблице 6, в схеме «Выделенный сервер приложений» значительно снижены требования к программно-аппаратному обеспечению серверной части системы.

Схема взаимодействия компонент системы (см. Рис.5) несколько отличается от схемы интегрированного сервера приложений.

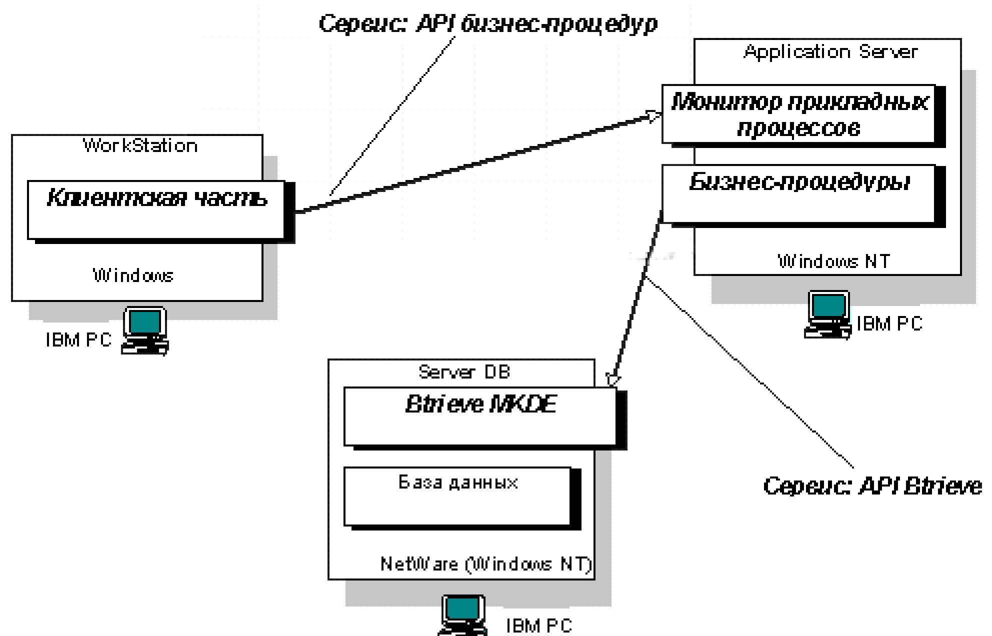

Рис. 5. Выделенный сервер приложений.

Принципиальное отличие этой схемы состоит в том, что запросы от клиента I звена системы (т.е. модулей организации интерфейса пользователя) передаются не на сервер БД, а на сервер приложений, монитору прикладных процессов. В качестве сервера этого звена следует рассматривать монитор многонитиевых процессов, работающий на сервере приложений. Сервисом, который он предоставляет клиенту, являются бизнес-процедуры сервера приложений.

Эти бизнес-процедуры, в свою очередь, можно считать клиентом II звена схемы. Клиент обращается к серверу — им является Btrieve MKDE, установленный на выделенном компьютере (сервере базы данных), — и получает сервис (API Btrieve), необходимый для обработки данных, хранящихся на сервере БД.

Схема «Выделенный сервер приложений», как и предыдущая — «Интегрированный сервер приложений», обеспечивает высокое быстродействие за счет использования 32-разрядных приложений и организации параллельного выполнения нескольких процессов. При этом снижаются требования и к объему оперативной памяти сервера, и к его производительности.

Разработка с использованием визуальных средств Windows и C++ интерфейса клиентской части, позволяет, как и в схеме на Рис. 4, обеспечить дружественность и эффективность его работы. Это выгодно отличает данную схему от схемы удаленного доступа к данным.

Поскольку в этой системе в качестве транспортного протокола используется протокол TCP/IP, можно организовать ее взаимодействие с программными комплексами, функционирующими в различных операционных системах.

Схема «Выделенный сервер приложений» реализует те же принципы построения, что и схема «Интегрированный сервер приложений», поэтому АБС, разработанные в обеих этих архитектурах, являются концептуально совместимыми.

2. Фрагменты исходного кода

Полностью исходный код программы составляет более 25 тысяч строк, поэтому здесь я приведу лишь самые важные его фрагменты.

2.1. Инициализация и загрузка отладчика

```
#include "StdAfx.h"
#include "export.h"
#include "dfif.h"
#include "prx_mes.h"
#include "prx_func.h"
#include "dbg.h"

/*флаги, соответствующие состоянию инициализации отладчика и его
интерактивного интерфейса */
BOOL        is_inited=FALSE;
BOOL        is_ui=FALSE;
/*дескриптор события, по которому происходит синхронизация задержки
rsl-stub во время создания интерфейса */
HANDLE           wnd_event=NULL;
```

```
/*дескрипторы потока выполнения отладчика и его окна*/
CWinThread*     wnd_thread = NULL;
CWnd*           p_wndmain=NULL;

/*коллекция указателей на экземпляры отладчиков*/
vector < auto_ptr < CDebug > > dbg;

/*инициализация пользовательского интерфейса*/
BOOL init_ui()
{
      done_ui();
      wnd_event = CreateEvent(NULL, FALSE, FALSE, NULL);
      wnd_thread = AfxBeginThread (RUNTIME_CLASS(CDFIFApp),
                        THREAD_PRIORITY_NORMAL,
                        0,
                        CREATE_SUSPENDED,
                        NULL);
      if(wnd_thread)
      {
        wnd_thread->m_bAutoDelete = FALSE;
          wnd_thread->ResumeThread ();

#ifdef _DEBUG
          VERIFY(WaitForSingleObject(wnd_event, INFINITE/*24000*/) ==
WAIT_OBJECT_0);
#else
          VERIFY(WaitForSingleObject(wnd_event, /*INFINITE*/24000) ==
WAIT_OBJECT_0);
#endif
          CloseHandle(wnd_event);
          wnd_event = NULL;
          if ( NULL == (p_wndmain = wnd_thread->m_pMainWnd)){
                return FALSE;
          } else {
                return TRUE;
          }
      }
      CloseHandle(wnd_event);
      wnd_event = NULL;
      return FALSE;
}

/*деинициализация ПИ*/
void done_ui(void){
      if (wnd_thread)
      {
         VERIFY(WaitForSingleObject(wnd_thread->m_hThread, INFINITE) ==
WAIT_OBJECT_0);
         delete wnd_thread;
         wnd_thread = NULL;
      }
}

/*обработка точки останова*/
BOOL process_bp(CDebug* _dbg, void* data){
      LRESULT         rv = 0;
      p_wndmain->ShowWindow(SW_SHOW);
      p_wndmain->SetForegroundWindow();
```

```
	rv = p_wndmain->SendMessage(MSG_BREAKPOINT, (WPARAM)data,
(LPARAM)_dbg);
	return (BOOL)rv;
}

/*Функции, которыt отладчик предоставляет RSL*/

/*возвращает количество экземпляров отладчиков*/
static int RSDBG DbgCount (void)
{
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	return dbg.size();
}

/*активвизация инициализации экземпляра отладчика*/
static HDBG RSDBG DbgInit (HRD inst,TDbgIntf * _dbg_ftable)
{
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	if(is_ui==FALSE){
		if(FALSE==init_ui()){
			return NULL;
		}
		is_ui=TRUE;
	}
	vector < auto_ptr < CDebug > >::iterator i;
	for(i = dbg.begin(); i != dbg.end(); ++i){
		if ((*i)->GetInst() == inst){
			return (HDBG)(i->get());
		}
	}
	CDebug* new_dbg = new CDebug(inst, _dbg_ftable);
	auto_ptr < CDebug > new_dbg_aptr(new_dbg);
	dbg.push_back(new_dbg_aptr);
	return (HDBG)new_dbg;
}

/*активизация деинициализации экземпляра отладчика*/
static void RSDBG DbgDone (HDBG hDBG)
{
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	if (1==DbgCount()){
		is_ui = FALSE;
		if (p_wndmain)
		   p_wndmain->SendMessage(MSG_FINISH/*WM_CLOSE*/, NULL,
NULL);
		p_wndmain = NULL;

		CDebug* aDBG = (CDebug* )hDBG;
		vector < auto_ptr < CDebug > >::iterator i;
		for(i = dbg.begin(); i != dbg.end(); ++i){
			if ((*i)->GetInst() == aDBG->GetInst()){
				dbg.erase(i);
				break;
			}
		}
		done_ui();
	} else if (DbgCount() > 1){
		CDebug* aDBG = (CDebug* )hDBG;
```

```
			vector < auto_ptr < CDebug > >::iterator i;
			int ctr = 0;
			int cur_ctr = 0;
			for(i = dbg.begin(); i != dbg.end(); ++i){
				if ((*i)->GetInst() == aDBG->GetInst()){
					if( ((CMainFrame* )p_wndmain)->m_curdbg == 
dbg[0].get()){
						if(cur_ctr != 1){
							ctr = 1;
						} else {
							ctr = cur_ctr - 1;
						}
					}
					((CMainFrame* )p_wndmain)->m_curdbg = 
dbg[ctr].get();
					dbg.erase(i);
					//
					break;
				}
				++cur_ctr;
			}
		}
}
/*активизация обработки точки останова*/
static int RSDBG DbgBreak (HDBG hDBG, uint32 data)
{
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	if( data && ((TBpData* )data)->bp_type == BP_DISABLED )
		return FALSE;
	else {
		CDebug* aDBG = (CDebug* )hDBG;
		return process_bp(aDBG, (void*)data);
	}
}

/*обработка процедуры отладочной печати*/
static void  RSDBG DbgTrace  (HDBG hinst,const char *str)
{
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	CDebug* pDebug = (CDebug* )hinst;
	CString aStr(str);
	if(pDebug && pDebug->NeedConvert())
		aStr.OemToAnsi();
	((CMainFrame* )p_wndmain)->AddTraceMsg(aStr, MSGLEVEL_NORMAL);
	p_wndmain->SendMessage(MSG_TRACE, (WPARAM)hinst, (LPARAM) NULL);
}

/*определение версии отладочного инструмента*/
static int  RSDBG DbgVersion (void)
{
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	return 2; //version 2, now supports decoding  dos<->win cp
}

/*Данная процедура вызывается при добавлении модуля в RSL-программе*/
static void RSDBG DbgAddModule(HDBG hinst,RSLMODULE hmod){
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
}
```

```
/* Данная процедура вызывается при удалении модуля в RSL-программе*/
static void RSDBG DbgRemModule(HDBG hinst,RSLMODULE hmod){
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	CDebug* pDebug = (CDebug* )hinst;
	if(!pDebug)
		return;
	vector < auto_ptr < CDebug > >::iterator i;
	for(i = dbg.begin(); i != dbg.end(); ++i){
		if ((*i)->GetInst() == pDebug->GetInst()){
			break;
		}
	}
	if(i != dbg.end()){
		vector < auto_ptr < TBpData > >::iterator j;
		CBPData* pBPData = (*i)->GetBP();
		for(j = pBPData->begin(); j!=pBPData->end(); ){
			if( hmod == (*j)->mod){
				pBPData->erase(j);
			} else {
				++j;
			}
		}

		CWatchV::iterator k;
		CWatchV* watch[3];
		watch[0] = (*i)->GetLocals();
		watch[1] = (*i)->GetSurvey();
		watch[2] = (*i)->GetQSurvey();
		int l;
		for ( l = 0; l < 3; l++ ){
			if ( watch[l] )
				for ( k = watch[l] -> begin(); k != watch[l] ->
end(); ++k )
					if ( (*k)->mod == hmod )
						(*k)->Free();
		}
	}
}

static void RSDBG DbgSetMode(HDBG hInst, int mode){
	AFX_MANAGE_STATE(AfxGetStaticModuleState());
	CDebug* pDebug = (CDebug* )hInst;
	if(0!=mode){
		pDebug->SetExternalMode(FALSE);
	}

}

/*интерфейс, предоставляемый отладчиком интерпретатору. Содержит все
описанные функции*/
static TRslDbg  dbgInterface =
{
	DbgVersion,
	DbgCount,
	DbgInit,
	DbgDone,
	DbgBreak,
	DbgTrace,
```

```
      DbgAddModule,
      DbgRemModule,
      DbgSetMode
};

/*Данная функция возвращает указатель на интерфейс отладчика*/
extern "C" __declspec(dllexport)
TRslDbg * RSDBG RslGetInterface (void)
{
      return &dbgInterface;
}
```

2.2. Обработка точки останова

```
/*вызывается при поступлении события MSG_BREAKPOINT*/
LRESULT CMainFrame::ProcessBreakPoint(WPARAM wParam, LPARAM lParam){
      ::ReplyMessage((LRESULT)TRUE);
      m_curdbg = (CDebug* )lParam;
      TBpData* cur_bpdata = (TBpData* )wParam;
      /*обновить информацию в окнах отладчика: стек, код, список
      выражений и переменных*/
      m_curdbg->UpdateDbgInfo();
      /*флаг состояния*/
      m_curdbg->SetDebugState(true);

      m_curdbg->SetCurModule((RSLMODULE)0xffffffff);
      CString str_trace;
      if (!str_trace.LoadString(IDR_STR_TRACE)){
            str_trace = "trace";
      }
      CString framename = frame_name();
      framename += " - ";
      framename += str_trace;
      SetWindowText(framename);

      //удалить точку останова, если текущая была установлена как
      //"выполнить до курсора"
      int offs, len;
      if(NULL != lParam && NULL != wParam && BP_INVISIBLE==cur_bpdata-
>bp_type){
            m_curdbg->do_ClrBreakPoint(m_curdbg->GetCurStatement(&offs,
&len));
            m_curdbg->DelBp(cur_bpdata);
      }

      m_curdbg->SetIndex(0);
      UpdateDbgInfo(0);
      return NULL;
}
```

2.3. Интерфес, предоставляемый интерпретатором RSL

```
typedef struct tagDbgIntf
{
   short int verHi;
   short int verLo;
   RSLSTACK  (RSDBG * EnumStack)  (HRD inst,RSLSTACK st,
```

```
 RSLPROC *prc, RSLMODULE *mod, RSLSTMT *stmt,int *offs,int *len, char
*procName, int szProcName, char *modName, int szModName);
   RSLMODULE (RSDBG * EnumModule) (HRD inst,RSLMODULE hmod, char
*modName, int szModName,int *modType);
   RSLSTMT   (RSDBG * GetStatementOfPos) (HRD inst,RSLMODULE hmod,int
offs,int len, int *realOffs,int *realLen,int *line);
   void      (RSDBG * BreakPointAt)  (HRD inst,RSLSTMT stmt,uint32
data);
   void      (RSDBG * ClrBreakPoint) (HRD inst,RSLSTMT stmt);
   int       (RSDBG * SetBreakPoint) (HRD inst,RSLMODULE hm,int
num,TRslBreakInfo *ptr);
   void      (RSDBG * RemBreakPoint) (HRD inst,RSLMODULE hm,int
num,TRslBreakInfo *ptr);
   RSLEXPCTX (RSDBG * ParseExp)      (HRD inst,RSLPROC proc,const char
*txt,int *isLval);
   void      (RSDBG * RemExp)        (HRD inst,RSLEXPCTX exp);
   RSLVALUE  (RSDBG * ExecExpAt)     (HRD inst,RSLEXPCTX exp,RSLSTACK
st, int *isObject,char *tpName, int szTpName, char *value, int szValue
);
   bool      (RSDBG * SetExpValue)   (HRD inst,RSLEXPCTX exp,RSLSTACK
st,const char *txt);
   RSLVINFO  (RSDBG * GetFirstLocalInfo) (HRD inst,RSLSTACK
hst,RSLVALUE *val,int *isObject, char *name, int szName, char *tpName,
int szTpName, char *value, int szValue);
   RSLVINFO  (RSDBG * GetFirstPropInfo) (HRD inst,RSLVALUE hv,RSLVALUE
*val,int *isObject, char *name, int szName, char *tpName, int
szTpName, char *value, int szValue);
   RSLVINFO  (RSDBG * GetNextInfo) (HRD inst,RSLVINFO hinfo,RSLVALUE
*val,int *isObject, char *name, int szName, char *tpName, int
szTpName, char *value, int szValue);
   void      (RSDBG * RefreshInfo) (HRD inst,RSLVINFO hinfo,RSLVALUE
*val,int *isObject, char *name, int szName, char *tpName, int
szTpName, char *value, int szValue);
   void      (RSDBG * DoneInfo) (HRD inst,RSLVINFO hinfo);
   bool      (RSDBG * SetNewInfo) (HRD inst,RSLVINFO hinfo,RSLSTACK
st, const char *txt);
   RSLSRC    (RSDBG * OpenSrc)  (HRD inst,RSLMODULE hmod,
TRslDbgCallBack proc, void *data);
   int       (RSDBG * ReadSrc)  (HRD inst,RSLSRC hsrc,char *buff, int
size);
   int       (RSDBG * GetSizeSrc)  (HRD inst,RSLSRC hsrc);
   void      (RSDBG * CloseSrc) (HRD inst,RSLSRC hsrc);
   void      (RSDBG * Interrupt) (HRD inst);
   void      (RSDBG * ExecContinue) (HRD inst, int traceFlag);
   void      (RSDBG * GetAppWnd) (HRD inst, HWND *wnd);
   int       (RSDBG * SetDbgFlag) (HRD inst,int newFlag);
   int       (RSDBG * GetVersion) (HRD inst,short int *verLo);
}  TDbgIntf;
```

2.4. Класс-оболочка интерфеса, предоставляемым интерпретатором RSL

```
class CDebugRoot
{
protected:
     TDbgIntf*  m_dbgftable;
     CString         m_error;
     HRD             m_inst;
public:
     CDebugRoot();
```

```
	virtual ~CDebugRoot();

public:
	bool do_EnumStack(RSLSTACK,RSLPROC*, RSLMODULE*, RSLSTMT*,
	                int*,int*, char*, int ,
	                char*, int,RSLSTACK*);
	bool do_EnumModule(RSLMODULE, char*, int,  int*,RSLMODULE*);
	bool do_GetStatementOfPos( RSLMODULE,
	                           int, int,
	                           int*, int*,
	                           RSLSTMT*,  int* line);
	bool do_BreakPointAt( RSLSTMT, uint32);
	bool do_ClrBreakPoint(RSLSTMT);
	bool do_ParseExp(RSLPROC, const char*, int*, RSLEXPCTX*);
	bool do_RemExp( RSLEXPCTX);
	bool do_ExecExpAt(RSLEXPCTX,RSLSTACK, int*,
	                           char*, int,
	                           char*, int,
	                           RSLVALUE*);
	bool do_GetFirstLocalInfo( RSLSTACK,  RSLVALUE*, int*,
	                           char*, int,
	                           char*, int,
	                           char*, int,
	                           RSLVINFO*);
	bool do_GetFirstPropInfo(RSLVALUE,RSLVALUE*, int*,
	                           char*, int,
	                           char*, int,
	                           char*, int,
	                           RSLVINFO*);
	bool do_GetNextInfo(  RSLVINFO,  RSLVALUE*, int*,
	                      char*, int,
	                      char*, int,
	                      char*, int,
	                      RSLVINFO*);
	bool do_RefreshInfo( RSLVINFO, RSLVALUE*,
	                      int*,char*,int, char*,
	                      int, char*,int);
	bool do_DoneInfo(RSLVINFO);
	bool do_SetNewInfo(RSLVINFO, RSLSTACK, const char*,bool*);
	bool do_OpenSrc(RSLMODULE, RSLSRC*);
	bool do_ReadSrc(RSLSRC, char*, int, int*);
	bool do_GetSizeSrc(RSLSRC, int*);
	bool do_CloseSrc(RSLSRC);
	bool do_Interrupt();
	bool do_ExecContinue(int);
	bool do_GetAppWnd(HWND*);
	bool do_SetDbgFlag(int, int* );
	bool do_GetVersion(short int*, int*);
	bool do_SetExpValue(RSLEXPCTX, RSLSTACK, const char*, bool*);
	void SetExternalMode(int mode);
	BOOL NeedConvert();
private:
	int m_mode;
};
```

Реализация функций этого класса имеет схожую структуру, поэтому я приведу лишь одну реализацию в качестве примера:

```
bool CDebugRoot::do_EnumModule(RSLMODULE hmod, char *modName, int
szModName, int* modtype, RSLMODULE* rv){
	CHECK_FTABLE(EnumModule);
	try{
		*rv = m_dbgftable->EnumModule(m_inst, hmod, modName,
szModName, modtype);
		if(NeedConvert()){
			Dos2Win(modName);
		}
		return true;
	} catch(...){
		m_error = ERR_UNKNOWN;
		return false;
	}
}
```