ПЛАТФОРМА RP Server X2

Введение в программирование

Редакция 2.1
от 08.2022
С.Степанов

Содержание

1 Общие сведения

Данное пособие не заменяет электронную документацию, поскольку не содержит описания встроенных функций языка X2 и многих интерфейсов, но в нем описаны принципы программирования и приведены примеры кода.

Основное назначение данного пособия – дать программисту возможность быстро найти нужную тему и сориентироваться в возможностях платформы и методиках программирования тех или иных аспектов.

Многие разделы данного пособия ссылаются на примеры, реализованные в демонстрационном модуле.

1.1 Назначение и архитектура платформы

Платформа RP Server^{X2} предназначена для создания и сопровождения бизнес-приложений, ориентированных на работу с базами данных. На данный момент платформа поддерживает две СУБД: Microsoft SQL Server и Postgres Pro.

В плане архитектуры платформа имеет следующие составляющие:

- Клиентская часть. Устанавливается на машину пользователя.
    - Среда разработки RPDesigner.exe. Используется разработчиками для создания прикладных приложений. В качестве аналогов можно назвать Microsoft Visual Studio или Delphi.
    - Среда исполнения RPExec.exe. Используется конечным пользователем для исполнения приложений, созданных в среде RPDesigner.
    - Набор вспомогательных утилит. Включает в себя средства для инсталляции приложений, установки и генерации ключей лицензионной защиты, работы с файлами помощи и т.д.
- Серверная часть. Устанавливается в рабочую БД.
    - Репозиторий. Предназначен для хранения кода прикладных приложений, служебных данных платформы и пользовательских настроек. Представляет собой набор объектов в целевой БД (таблицы, хранимые процедуры, триггеры, view и прочие).

1.2 Язык программирования X2

Платформа имеет встроенный язык программирования «X2». Это основной язык для создания приложений. Тем не менее, в коде на X2 можно использовать вставки на VBScript и JScript. Можно, также, задействовать библиотеки (dll), написанные на C++ или C#.

Язык X2 синтаксически близок к языку C, но, в отличие от C, имеет встроенную поддержку SQL. Это значит, что операторы SQL можно использовать непосредственно в коде программы, написанной на X2.

Кроме того, язык X2 позволяет манипулировать множеством преднастроенных объектов пользовательского интерфейса, используя их в качестве строительных блоков для создания приложения. Это списки, диалоги, меню, диаграммы, отчеты и прочие. Каждый из них имеет набор обработчиков событий, которые можно реализовать на языке X2, меняя тем самым поведение объекта.

1.3 Особенности платформы

- **Открытый код**

Приложения, созданные на платформе RP Server^{X2}, могут поставляться конечному пользователю с открытым кодом. Это позволяет специалисту сопровождения на стороне клиента самостоятельно перенастроить поведение каких-то объектов интерфейса при необходимости. Это является очень важным условием для успешного внедрения серийных версий больших проектов на предприятиях. Тем не менее, поддерживается возможность закрыть часть кода (или весь), сделав его недоступным для просмотра и редактирования.

- **Код приложения хранятся в целевой БД**

Приложения, написанные на данной платформе X2, хранятся в целевой базе данных, т.е. в той базе, где хранятся сами данные. Это позволяет обновлять версию приложения централизованно без установки на каждую локальную машину. Такой подход очень важен для крупных предприятий с разветвленной инфраструктурой, особенно, когда филиалы находятся в разных регионах страны.

- **Ключи лицензионной защиты**

Платформа позволяет защищать приложения с помощью ключей лицензионной защиты и имеет в составе средства для генерации таких ключей. Но, также, есть возможность создавать незащищенные приложения, которые для своей работы не требуют ключей.

- **Средства кастомизации**

Как было сказано выше, для успешного внедрения серийной версии приложения на предприятии, его, как правило, приходится дорабатывать. Для этого приложение должно поставляться с открытым кодом. Но при этом возникает другая проблема, связанная с тем, что при обновлении серийной версии все настройки и доработки, проведенные на предприятии, удаляются этим обновлением.

Платформа RP Server^{X2} имеет встроенные средства, позволяющие избежать полностью или же свести к минимуму потери, связанные с обновлением серийной версии. Для этого платформа предоставляет возможность хранить все изменения, сделанные в объектах серийной версии, отдельно от самих этих объектов. Таким образом, при обновлении все доработки сохраняются. Во многих случаях одного этого бывает достаточно. Тем не менее, обновление может затронуть логику работы каких-то объектов, настроенных пользователем. Тогда их придется, как минимум, протестировать. Чтобы выявить такие ситуации, платформа ведет историю обновлений.

1.4 X и X2 – сходства и различия

Данная глава предназначена для тех, кто уже знаком с платформой RP Server и языком X.

Платформа RP Server^{X2} радикально отличается от предыдущей реализации платформы RP Server.

Прежде всего, это связано с тем, что она использует новый язык программирования X2. В отличие от X-языка, X2 является компилируемым языком. Он имеет более строгий синтаксис. Устранены недостатки языка подстановок, такие, как неоднозначность и низкая производительность.

Кроме того, появились новые подходы к построению визуальных интерфейсов. Объекты стали более управляемыми, их программирование – более прозрачным и надежным. Для построения интерфейсов используется новая библиотека визуальных компонентов.

Также, обеспечивается поддержка UNICODE и возможность локализации системной части платформы.

Существующие возможности по кастомизации прикладных приложений полностью сохранены, но некоторые принципы изменены. В частности, вместо дополнительных сегментов кода используются объекты замещения с возможностью наследования.

Код на языке X2 не совместим с кодом на X-языке. Нет возможности автоматического переноса существующих кодов на новую платформу.

2 X2 – первые шаги

Этот раздел предназначен для тех, кто впервые знакомится с платформой X2. Он представляет собой небольшой практикум, который позволит программисту освоить некоторые основные приемы программирования и получить первые практические навыки. Здесь мы создадим простое приложение и рассмотрим некоторые возможности платформы.

Платформа поддерживает две СУБД: Microsoft SQL Server и Postgres Pro, но в этом разделе мы будем рассматривать пример, ориентированный на Microsoft SQL Server, поэтому нам понадобится именно он.

2.1 Необходимые знания и навыки

Для работы потребуются:

- Начальные знания языка SQL, хотя бы на уровне построения простых запросов
- Представление о принципах разработки реляционных баз данных
- Некоторые навыки работы с Microsoft SQL Server для того, чтобы установить его, выдать нужные права пользователю и настроить источник данных ODBC

Поскольку язык программирования X2 синтаксически близок к языку C, то знание C или C++ будет хорошим подспорьем. Тем не менее, это не обязательно.

2.2 Подготовительные работы

Прежде, чем начать практические занятия, нужно скачать и установить платформу, а также, настроить среду разработки с учетом своих предпочтений.

2.2.1 Скачать и установить

Если Вы еще не установили платформу RP Server^{X2}, то войдите на сайт rp-server.com и перейдите в раздел «Скачать RP Server^{X2}». На той страничке Вы найдете пошаговую инструкцию по установке платформы и демонстрационной базы данных.

Вам потребуется:

- Скачать и установить Microsoft SQL Server, если он еще не установлен
    - Если сервер уже установлен, и Вы не входите в группу системных администраторов, то Вам потребуются права VIEW SERVER STATE и VIEW ANY DEFINITION на уровне сервера
- Скачать и установить платформу RP Server^{X2}
    - Установить ключ для работы платформы

- Скачать и установить демонстрационную базу данных (с ней мы будем работать)
- Настроить источник данных ODBC на демонстрационную БД

Вся процедура достаточно проста, но на некоторых деталях стоит заострить внимание.

Платформа RP Server^{X2} поставляется бесплатно только для редакции Microsoft SQL Server Express. Это определяется ключом лицензионной защиты, который входит в комплект поставки платформы. Этот ключ не удастся установить на иную редакцию Microsoft SQL Server.

При настройке источника данных следует использовать 32-битную версию Администратора ODBC. На 64-битной версии Windows она лежит по пути

C:\windows\SysWOW64\odbcad32.exe

А по пути

C:\windows\System32\odbcad32.exe

лежит 64-битная версия Администратора. Она нам не подходит.

Источник данных (DSN) должен быть настроен на демонстрационную базу данных. Она имеет имя «X2_Demo», если только Вы не изменили его во время инсталляции. Имя DSN может быть любым. В нашем примере DSN имеет имя «_X2_Demo».

2.2.2 Запуск и настройка среды разработки

После того, как Вы установили платформу и настроили источник данных ODBC на демонстрационную БД, запустите среду разработки RPDesigner.exe.

Через главное меню вызовите панель настроек программы «Вид \ Панели \ Настройки программы». Здесь можно настроить внешний вид окон и выбрать шрифт для редакторов кодов. По умолчанию для окон редакторов включен режим закладок и используется шрифт MS Shell Dlg(8). Все это можно оставить, как есть, или поменять по своему усмотрению.

В дальнейшем на всех копиях экранов, которые будут здесь приведены, режим закладок отключен и установлен шрифт Fixedsys 10.

2.3 Приложение X2

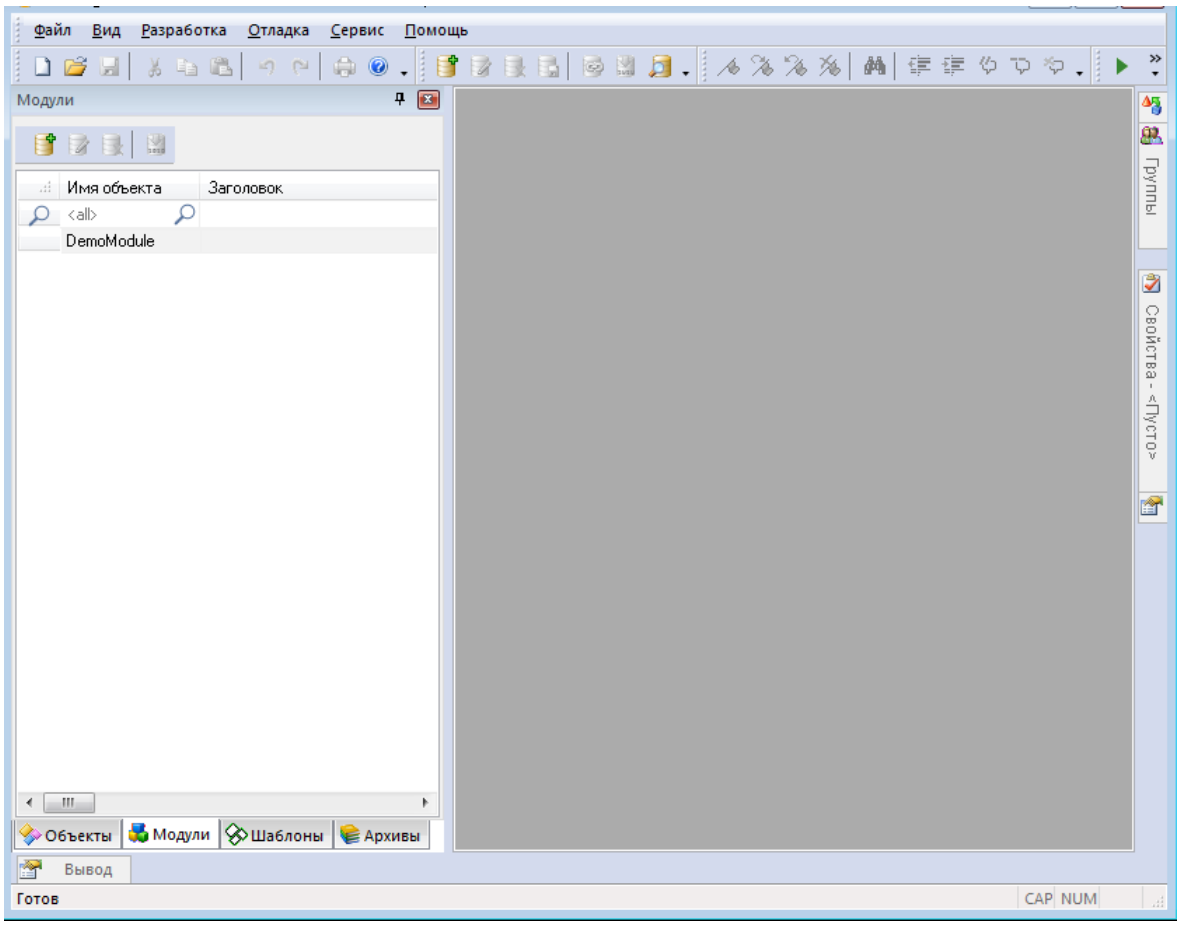
Приложение X2 представляет собой набор объектов. Все они хранятся в таблице x2Objects. Первый объект, который исполняется при старте приложения – это объект «модуль». Именно с этого объекта следует начинать разработку нового приложения.

К модулю может быть привязан объект «меню». Из обработчиков пунктов меню могут вызываться другие объекты. Эти объекты могут иметь собственные меню, либо вызывать следующие объекты из своих обработчиков событий. Таким образом и строится приложение.

Задача программиста сводится к тому, чтобы создать необходимый набор объектов и реализовать их обработчики в соответствии с требуемой логикой.

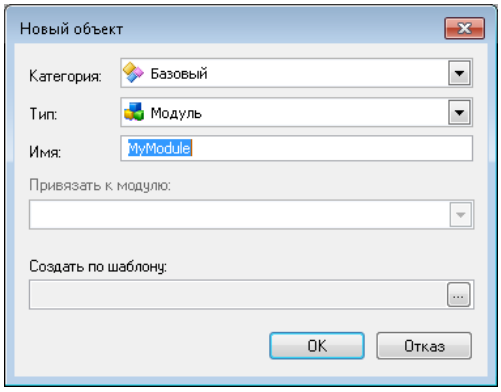
2.4 Создаем модуль

Попробуем создать наш первый модуль. Для этого в Диспетчере объектов переходим на закладку «Модули». Диспетчер объектов – это панель, которая содержит списки объектов. По умолчанию она располагается с левой стороны главного окна Дизайнера. Если Вы не видите закладку «Модули», то вызовите ее через меню «Вид \ Панели \ Модули».



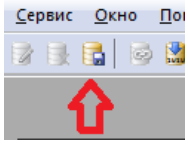
Поскольку мы находимся в демонстрационной БД, то в списке модулей уже присутствует один модуль DemoModule.

Нужно кликнуть мышкой в списке модулей, чтобы передать ему фокус ввода. Далее нажимаем клавишу «Ins», либо вызываем контекстное меню с помощью правой клавиши мыши и выбираем пункт «Новый».

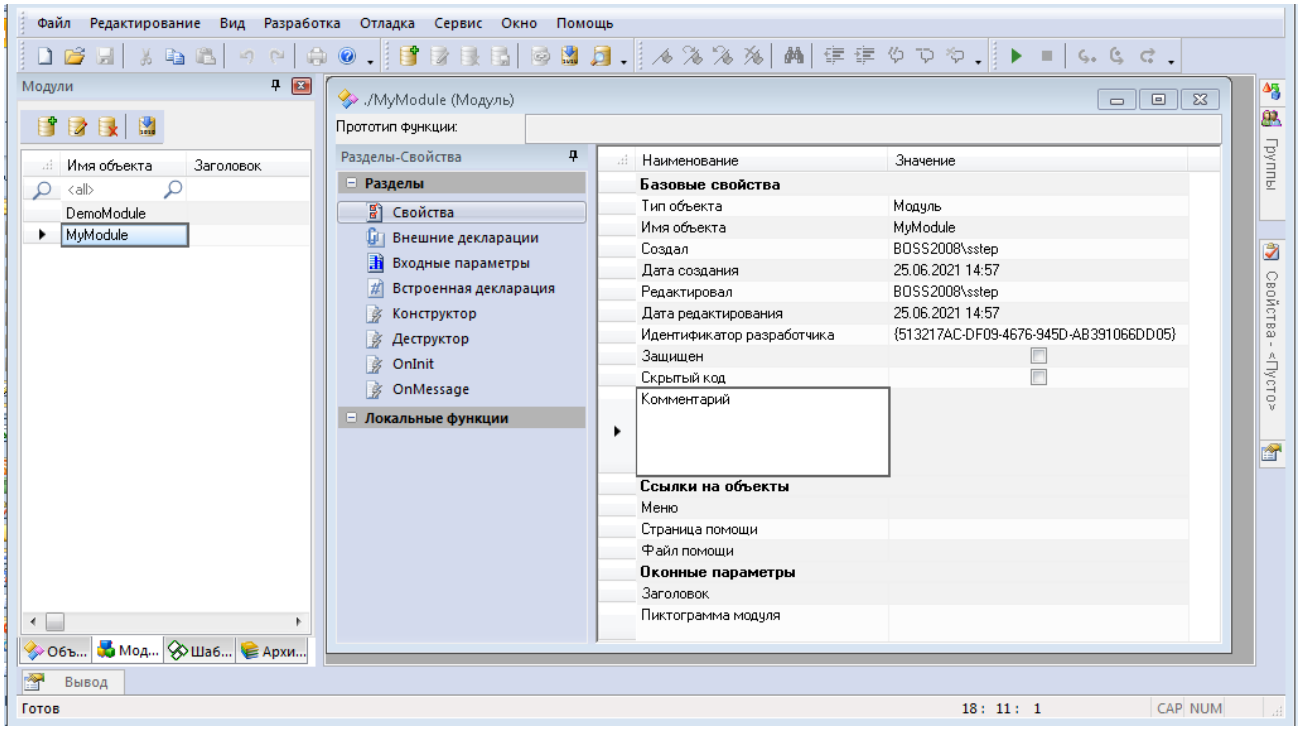


В диалоге создания объекта вводим имя модуля, к примеру, «MyModule». Нажимаем «OK», и на экране появится окно редактора модуля. Наш новый модуль пока еще существует только в оперативной памяти, он не сохранен в БД, поэтому мы не видим его в списке модулей.

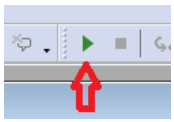
Теперь нажмем кнопку «Сохранить» в линейке инструментов,



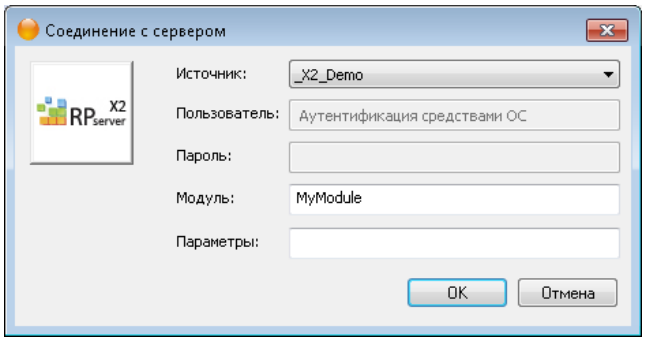
и наш модуль появится в списке модулей.



Наш модуль пока еще ничего не умеет, но его уже можно запустить на исполнение. Для этого нажимаем клавишу F7 или кнопку в линейке инструментов

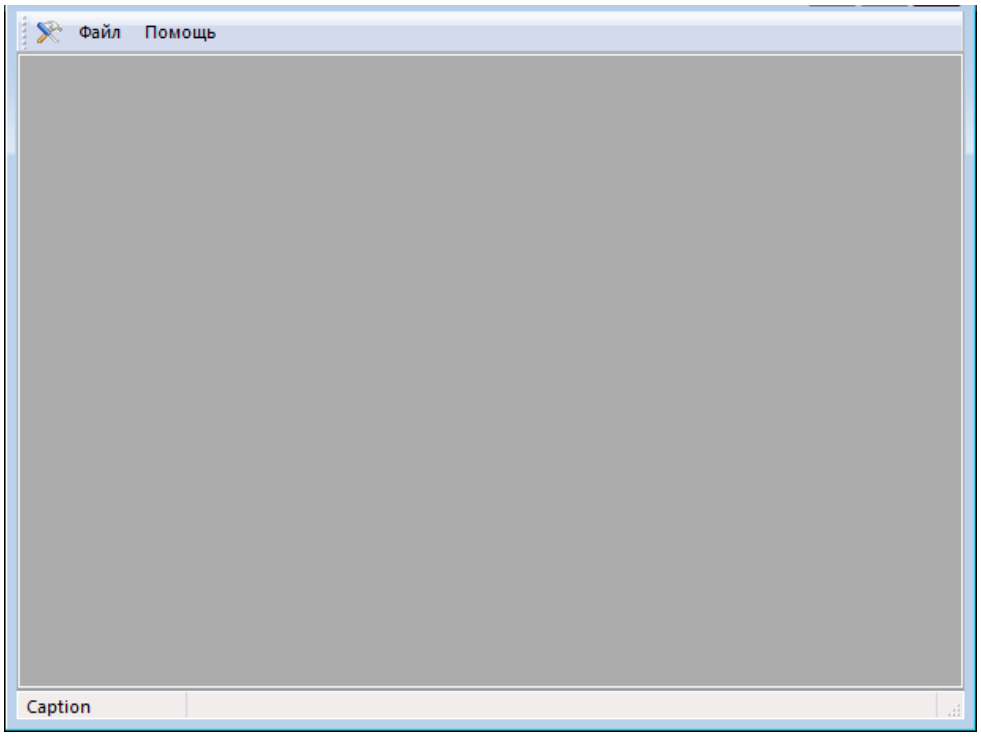


При этом запустится среда исполнения RPExec.exe и выдаст на экран диалог соединения с сервером. Это соединение и будет в дальнейшем использоваться для доступа к БД.



Нажимаем «OK», и модуль запущен.

Поскольку наш модуль пока еще ничего не делает, то мы увидим пустое окно. Это и есть главное окно приложения. В линейке меню мы видим две группы «Файл» и «Помощь». Это системное меню. Две эти группы будут присутствовать всегда. После того, как мы создадим наше собственное меню и вызовем его из модуля, оно будет размещено между этими группами.



Обратите внимание на эту кнопку  в левом верхнем углу. Это кнопка вернет нас в Дизайнер и поднимет на редактирование тот объект, который в данный момент активен. Она появляется только в том случае, когда среда исполнения запущена в режиме отладки, т.е. изнутри Дизайнера. Если же запустить RPExec.exe непосредственно из командной строки, то этой кнопки не будет.

Теперь закроем наше пустое окно и вернемся в редактор модуля.

2.4.1 Пиктограмма и заголовок модуля

В разделе «Свойства» мы можем редактировать свойства нашего модуля.

Установим заголовок главного окна и пиктограмму, которая будет отображаться в левом верхнем углу. После этого сохраним изменения в БД (Ctrl+S).

Пиктограмму можно выбрать только из справочника, который хранится в нашем репозитории в таблице x2Image, и она должна иметь формат ico. А что делать, если там нет нужной пиктограммы? Нужно ее добавить. Для этого идем в меню «Сервис\ Справочники\ Пиктограммы» и в списке пиктограмм нажимаем кнопку «Добавить».

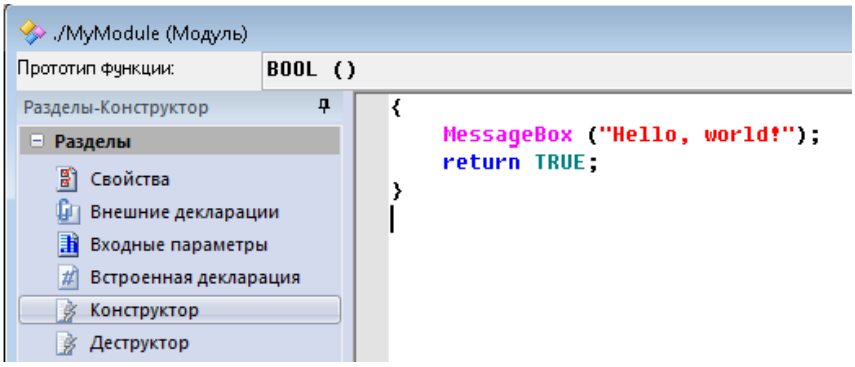
2.4.2 Hello, world!

Теперь настало время научить наш модуль чему-то полезному.

Для этого в разделах выберем «Конструктор» и напишем такой код:

```
{
        MessageBox ("Hello, world!");
        return TRUE;
}
```

Обратим внимание на то, что все кодовые сегменты окружаются фигурными скобками.



Теперь сохраним изменения (Ctrl+S) и запустим модуль (F7).

Получаем сообщение:

После нажатия на кнопку «OK» появится главное окно. Закроем его и вернемся к нашему модулю.

Теперь в операторе return заменим TRUE на FALSE, сохраним изменения и снова запустим модуль. В данном случае после выдачи сообщения приложение сразу же завершится, и главного окна не возникнет. Это может пригодиться нам в будущем, если мы захотим написать приложение, которое должно отработать в фоне без оконного интерфейса.

2.4.3 Протокол компиляции

Вы уже наверняка обратили внимание на то, что каждый раз при сохранении объекта выполняется его компиляция. Протокол компиляции выдается в панель вывода в нижней части окна.

```
Вывод
Компиляция объекта: TYPE: MODULE, NAME: MyModule ...

RESULT: OK
ERRORS: 0, WARNINGS: 0
```

Давайте целенаправленно создадим ошибку в нашем коде и посмотрим, что будет.

```
{
    MessageBox ("Hello, world!");
    BadName
    return FALSE;
}
```

Теперь вызовем компиляцию. Для этого сохраним объект (Ctrl+S) и получим новый протокол.

```
Вывод
Компиляция объекта: TYPE: MODULE, NAME: MyModule ...
Компиляция объекта: TYPE: MODULE, NAME: MyModule. Шаг 2.
Компиляция сегмента Local Function Parameters ...
Компиляция сегмента Constructor ...
ERROR: 45; Pos:(3, 2) BadName : Неопознанный идентификатор!

RESULT: ERROR
ERRORS: 1, WARNINGS: 0
Предупреждение : Объект сохранен без компиляции!
```

Двойной клик мыши на строке «ERROR: 45; Pos:(3, 2) BadName : Неопознанный идентификатор!» спозиционирует каретку на строке кода, вызвавшей ошибку.

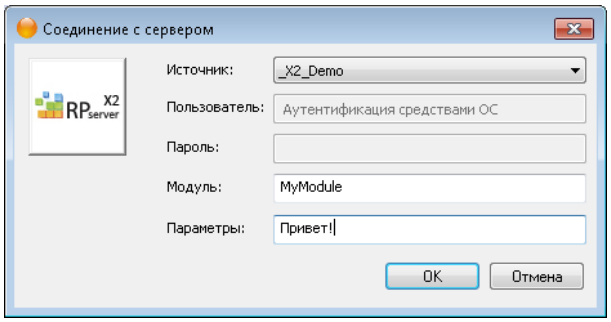
```
{
    MessageBox ("Hello, world!");
    BadName
    return FALSE;
}
```

2.4.4 Параметры модуля

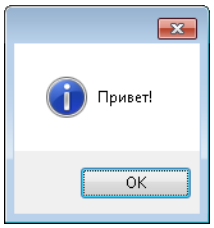
Модуль может принимать один строковый параметр, который можно передать через командную строку (ключ -i) или же указать в диалоге соединения с сервером. Он имеет имя strInput. Попробуем его задействовать. Для этого немного изменим код нашего конструктора модуля.

```
{
    MessageBox (strInput);
    return FALSE;
}
```

Сохраним изменения и запустим модуль. В диалоге соединения с сервером в поле «Параметры» введем «Привет!».



Нажмем «OK» и получим сообщение



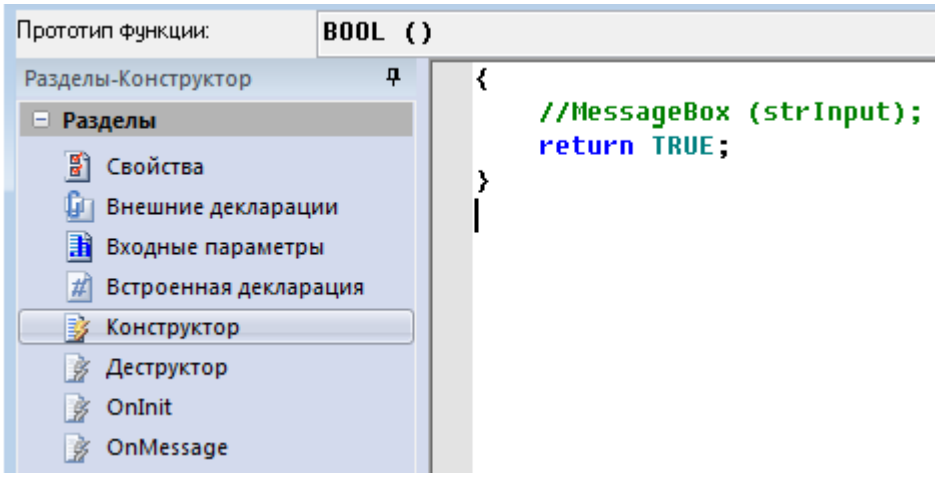
В сообщении отобразилась строка, которую мы передали в качестве параметра.

А вот как это будет выглядеть в случае командой строки:

RPExec.exe -d:_X2_DEMO -m:MyModule -i:"Привет!"

2.4.5 Сегмент должен вернуть значение

Теперь, когда мы научились работать с модулем, закомментируем вызов функции MessageBox, он нам больше не понадобится.



Обратите внимание, я закомментировал только функцию MessageBox, но оператор return оставил. Дело в том, что конструктор обязательно должен вернуть значение типа BOOL. Мы видим это в строке «Прототип функции». Если в кодовом сегменте есть хоть какие-то символы, то компилятор будет пытаться компилировать этот сегмент. Если же нет никаких символов, то компилятор будет считать, что данный сегмент возвращает значение по умолчанию. В случае конструктора это – TRUE.

Поэтому, если закомментировать все, то компилятор вернет ошибку.

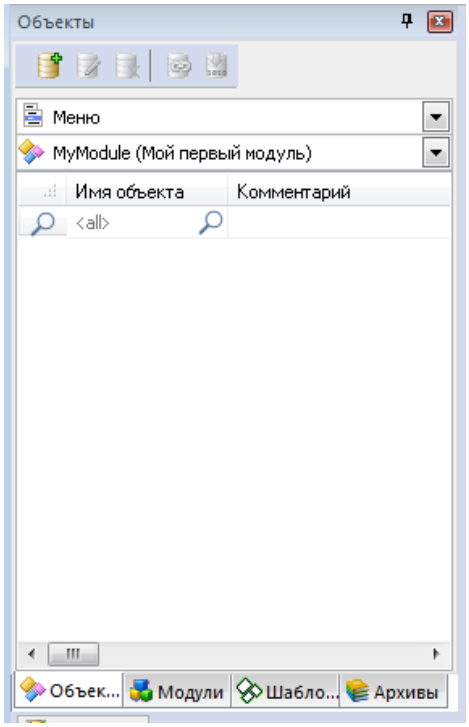
```
//{
    //MessageBox (strInput);
    //return TRUE;
//}
```

А вот если удалить весь текст полностью (Ctrl+A, Del), то компилятор отнесется к этому спокойно.

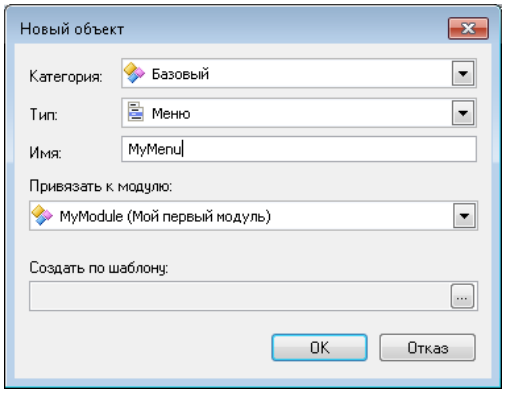
2.5 Создаем меню

Теперь, когда наш модуль готов, пора создать для него меню.

Чтобы создать меню, в Диспетчере объектов переключимся на закладку «Объекты», выберем тип объекта «Меню» и выберем наш модуль «MyModule».



Теперь кликнем мышкой в список, чтобы передать ему фокус ввода, и нажмем клавишу «Ins», либо правым кликом вызовем контекстное меню и выберем пункт «Новый».

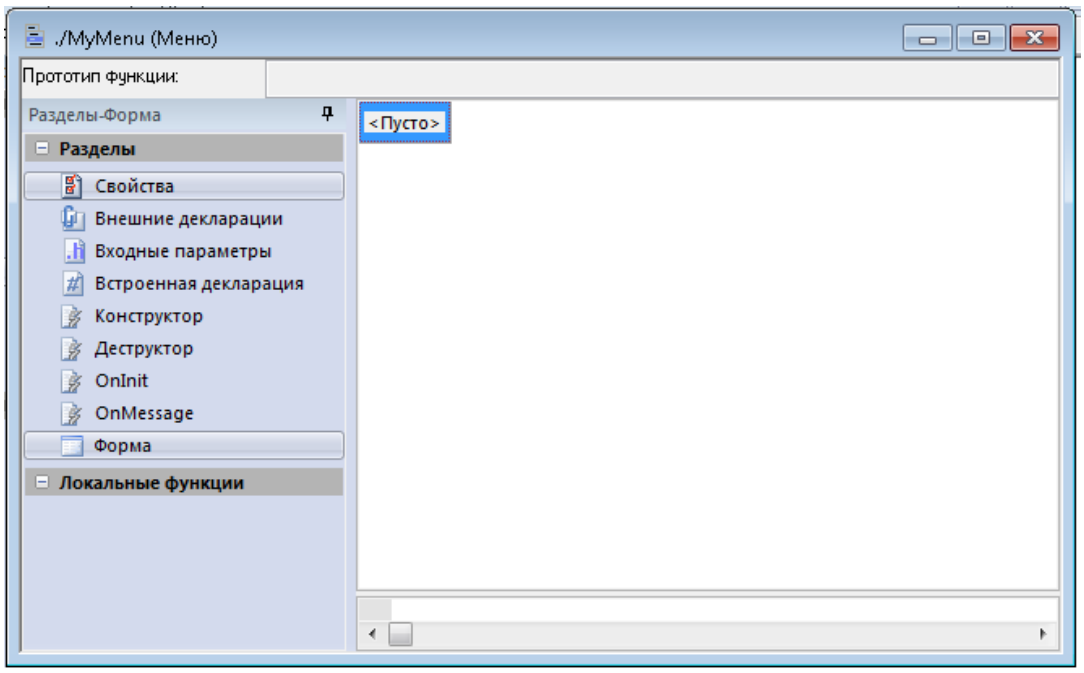


В диалоге введем имя нашего меню, например, «MyMenu».

Обратите внимание на поле «Привязать к модулю», там выбран наш модуль «MyModule». Это правльно, пусть так и останется. Об этом мы скажем чуть позже.

Теперь нажмем «OK» и на экране появится окно редактора меню.

Сразу же сохнаним наше меню в БД, и оно появитя в списке объектов.

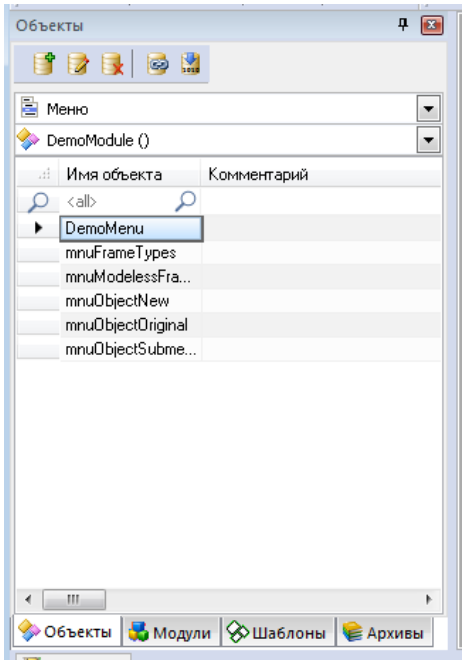


А теперь сделаем небольшое отступление и поговорим о привязке объектов к модулям.

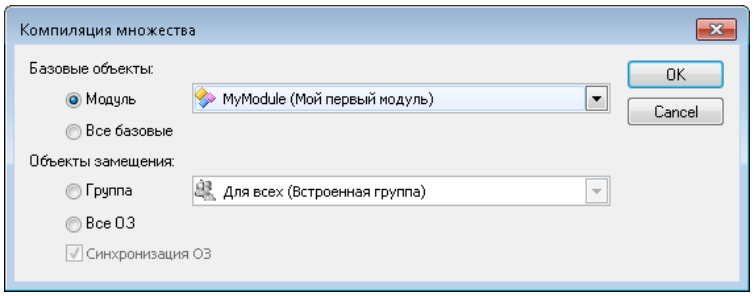
2.5.1 Привязка объекта к модулю

Дело в том, что в одной БД может быть мого модулей и много объектов. Их могут быть сотни, тысячи, десятки тыяч… К примеру, такой проект, как «БОСС-Кадровик» содержит более 6 тыс. объектов.

Это не удобно, когда все объекты хранятся общей кучей, поэтому объекты, логически отосящиеся к конкретному модулю, желательно привязывать к нему. Тогда в списке объектов можно будет накладывать фильтр. К примеру, если мы переключимся в нашем списке меню на демонстрационный модуль, то мы увидим только те меню, которые к нему относятся.



К тому же, при компиляции проекта можно будет указывать конкретный модуль (путь по меню: «Разработка \ Компилировать множество»).



В этом случае будет копилироваться не только сам объект «модуль», но и все объекты, привязанные к нему. Это бывает важно, когда разработка ведется группой программистов.

А что, если объект используется одновременно в нескольких модулях? Тогда его нужно привязать к нескольким модулям. В списке объектов нужно кликнуть на нем правой клавишей и в контекстном меню выбрать «Привязка к модулям…».

В сущности, привязка объектов к модулям интересна только на этапе разработки. В период исполнения она не играет ни какой роли. Даже если объект не привязан ни к одному модулю, это не помешает его исполнению.

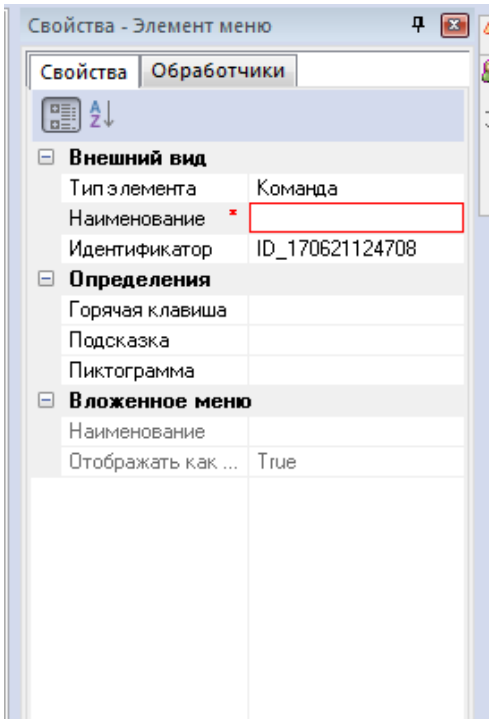
2.5.2 Структура меню

Вернемся к нашему меню и посмотрим, как это работает.

Объект меню устроен довольно просто. Пункт меню может выступать в роли команды, группы, разделителя или подменю. Команда имеет обработчик, который будет вызываться, когда пользователь выберет этот пункт. Группа всего лишь объединяет несколько команд или других групп. Они будут отображаться в виде выпадающего меню. Разделитель является простой горизонтальной линией, которая визуально разделяет другие пункты по смысловым категориям. А подменю является ссылкой на другой объект меню. Пример использования подменю можно найти в демонстрационном модуле (см. меню «DemoMenu», первый пункт).

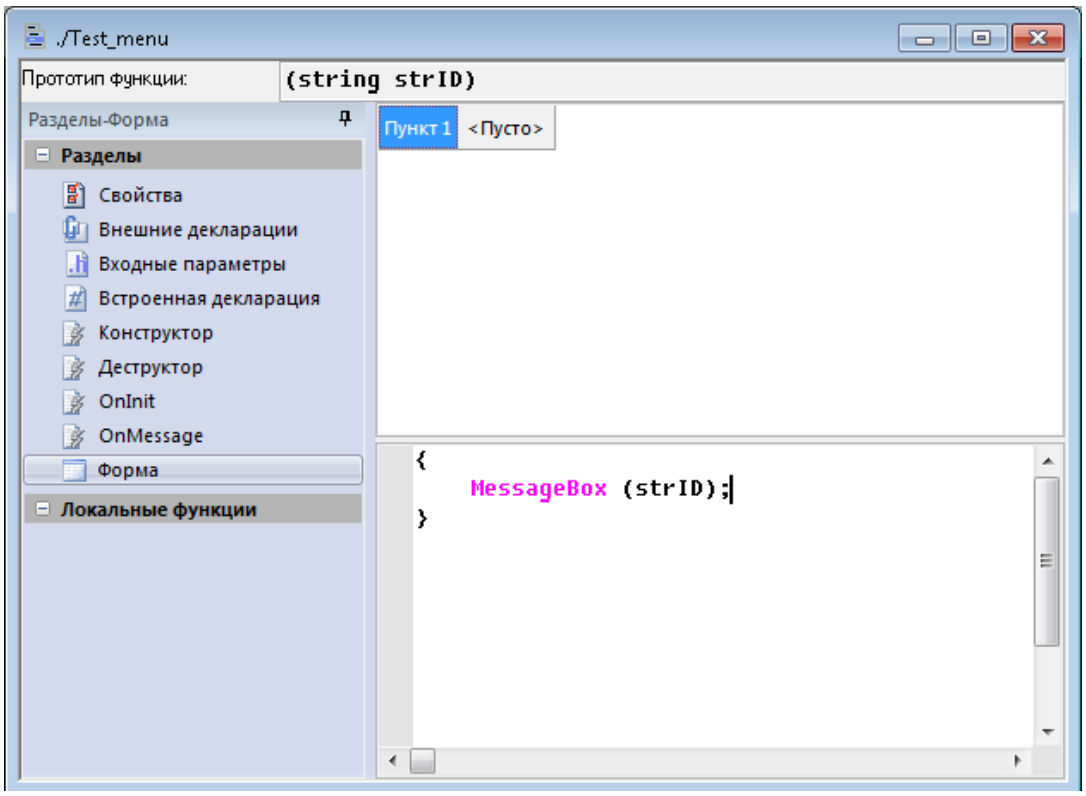
Мы же создадим в нашем меню пункт, который является командой, и напишем для него обработчик.

Поскольку наше меню пока еще не содержит ни одного пункта, то мы видим в нем один фиктивный элемент <Пусто>. Дважды кликнем на нем, и появится панель настроек «Свойства».

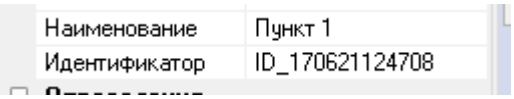


Введем наименование пункта меню, например, «Пункт 1». Кликнем мышкой в окне редактора меню, чтобы перевести фокус ввода, и увидим, что наш новый пункт появился. Напишем для него обработчик:

```
{
        MessageBox (strID);
}
```



Я написал обработчик, который просто выведет а экран сообщение. А что такое strID? Это идентификатор пункта меню. Каждый пункт меню имеет уникальный идентификатор. Он задается в паенли «Свойства».

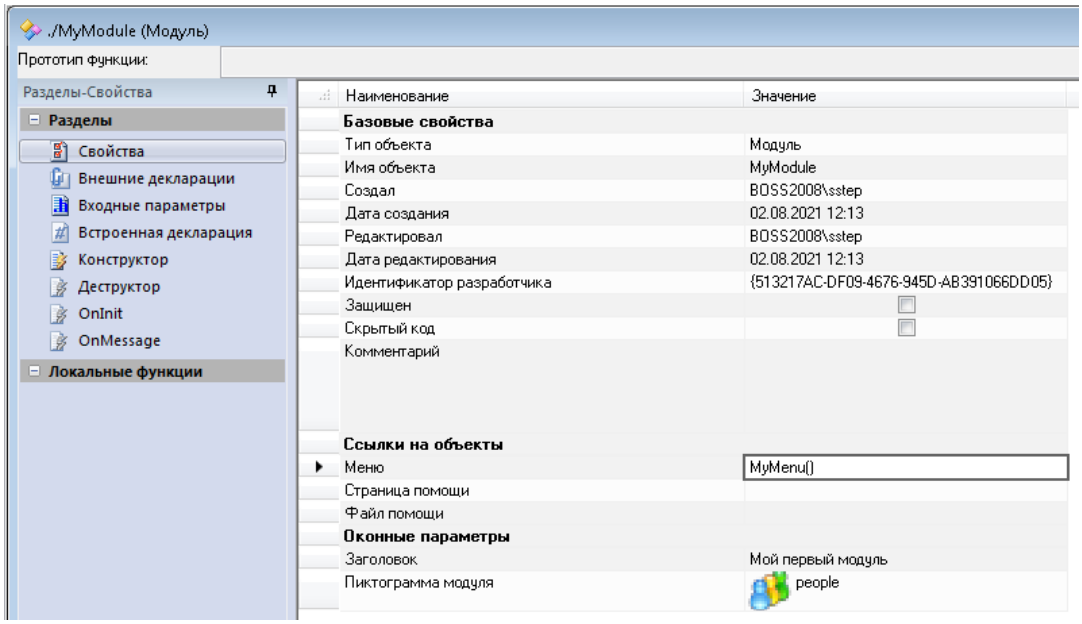


Среда разработки автоматически генерирует уникальные идентификаторы для каждого пункта меню. Можно оставить идентификатор, как есть, но можно поменять его на более осмысленный, например «ID_MENU_ITEM1». В дальнейшем идентификатор можно будет использовать для управления пунктом меню. Например, сделать пункт недоступным или поменять его иконку (функции MenuEnableItem и MenuSetItemInfo). Пример управления пунктами меню можно найти в демо-модуле (Меню \ Управление пунктами меню).

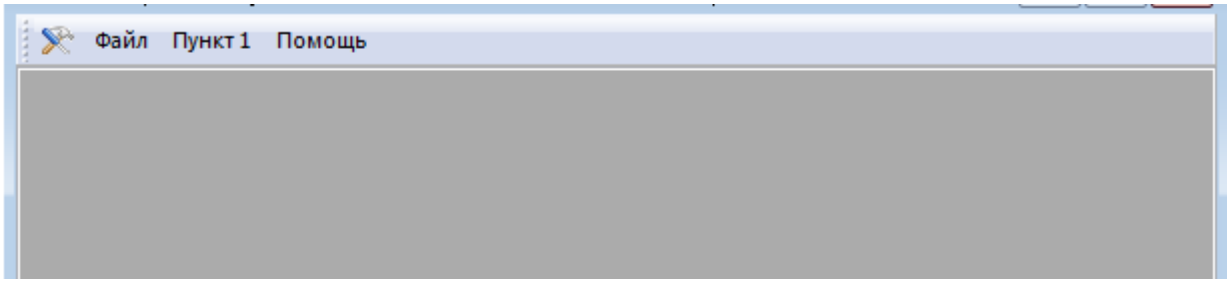
Как мы видим, обработчик команды меню получает один входной параметр strID, который имеет тип string. Его-то мы и выведем на экран с помощью функци MessageBox.

2.5.3 Вызов меню

Итак, сохраним наше новое меню в базу. Теперь вернемся в окно редактора модуля, перейдем в разел «Свойства» и установим ссылку на наш объект «MyMenu».



Сохраним изменения в базу и запустим модуль. Наше меню появилось в линейке меню главного окна.



Выберем «Пункт 1» и получим

Наше меню работает!

2.6 Поставим задачу

Теперь, когда у нас есть модуль и меню, попробуем решить какую-нибудь прикладную задачу. Создадим в нашей БД две таблицы с такой структурой:

```
CREATE TABLE Countries
(
	id_Country		int IDENTITY PRIMARY KEY,
	Name			nvarchar (255) NOT NULL,
	PopulationSize		int NOT NULL CHECK (PopulationSize >= 0)
)

CREATE TABLE Cities
(
	id_City			int IDENTITY PRIMARY KEY,
	Name			nvarchar (255) NOT NULL,
	PopulationSize		int NOT NULL CHECK (PopulationSize >= 0),
	Capital			int NOT NULL CHECK (Capital IN (1,0)),
	id_Country		int FOREIGN KEY (id_Country)
	REFERENCES Countries (id_Country) ON DELETE CASCADE
)
```

В таблице Countries будут храниться названия стран мира и численность их населения. В таблице Cities – названия городов, численность их населения и признак того, является ли город столицей. Как мы видам, в таблице Cities поле id_Country является внешним ключом, ссылающимся на таблицу Countries.

С этими двумя таблицами мы и будем работать. Мы создадим пользовательский интерфейс, позволяющий отображать данные и редактировать их, а также, построим диаграмму и простой отчет на основе наших данных.

2.6.1 Подготовка к работе

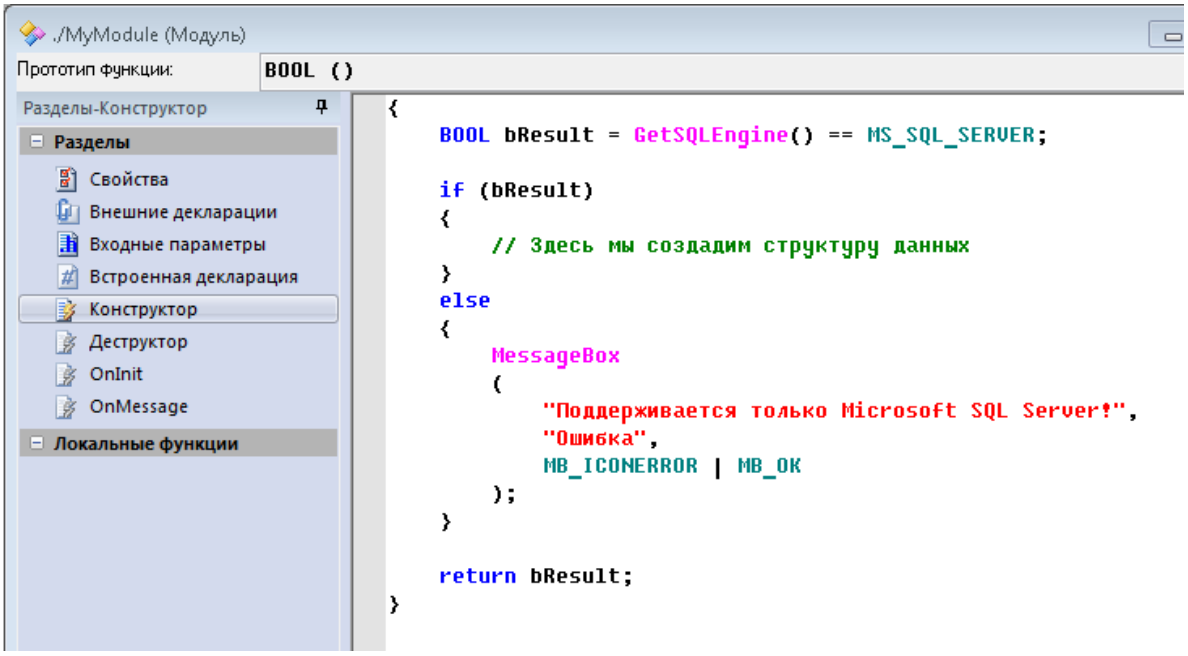
Для начала работы нам потребуется наполнить наши таблицы исходными данными. По ходу дела мы будем удалять и редактировать эти данные. Чтобы нам не приходилось каждый раз их восстанавливать вручную, мы напишем код, который будет делать это автоматически при каждом входе в модуль.

Кроме этого, нам нужно решить еще одну задачу. Поскольку наш пример рассчитан на Microsoft SQL Server и на Postgres Pro работать не будет, было бы правильным при запуске модуля проверить, на какой СУБД мы работаем. Это мы и сделаем.

2.6.1.1 Проверка СУБД

Войдем в конструктор модуля и напишем такой код:

```
{
	BOOL bResult = GetSQLEngine() == MS_SQL_SERVER;

	if (bResult)
	{
		// Здесь мы создадим структуру данных
	}
	else
	{
		MessageBox
		(
			"Поддерживается только Microsoft SQL Server!",
			"Ошибка",
			MB_ICONERROR | MB_OK
		);
	}

	return bResult;
}
```



Поясним смысл этого кода.

Функция GetSQLEngine вернет нам константу, определяющую тип СУБД. Если она равна MS_SQL_SERVER, то в переменную bResult ляжет 1 (TRUE), иначе там будет 0 (FALSE).

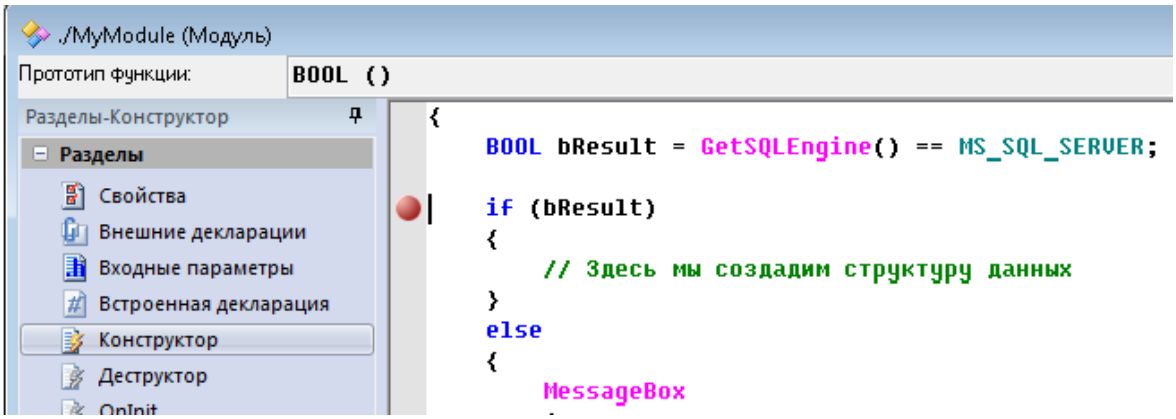
Дальше мы выполняем проверку переменной bResult. Если такая форма записи оператора if не вполне понятна, то поясню, что это эквивалентно

```
if (bResult != 0)
```

Ну и мы уже знаем, что, если конструктор модуля вернет 0 (FALSE), то приложение завершится.

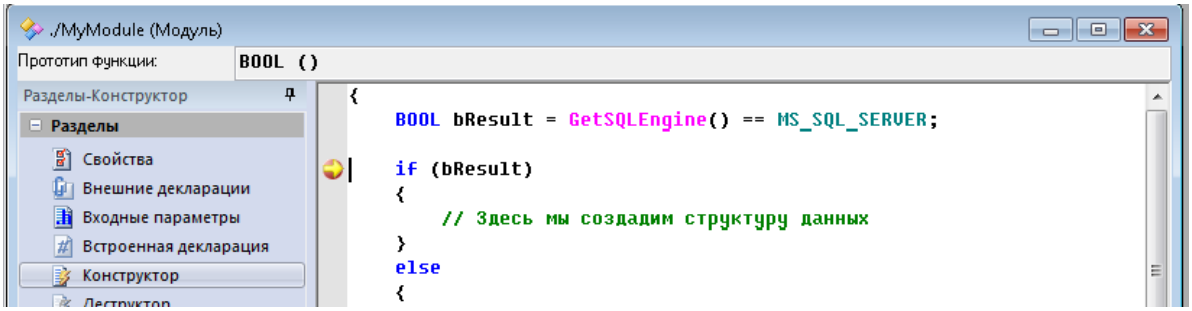
Теперь попробуем отладить наш код и получить сообщение "Поддерживается только Microsoft SQL Server!". Как бы нам это сделать? Неужели придется устанавливать Postgres Pro и пробовать на нем? Конечно, нет. Можно сделать проще.

Для этого воспользуемся возможностями отладчика и изменим значение bResult налету. Установим точку прерывания, как показано ниже.

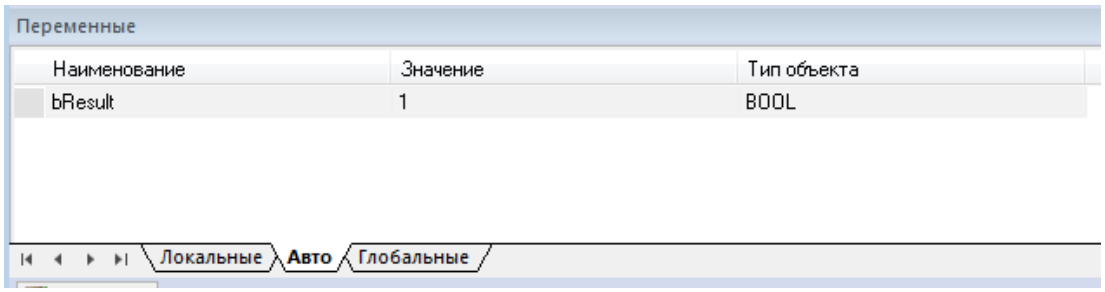


Для этого нужно дважды кликнуть мышкой в этом месте.

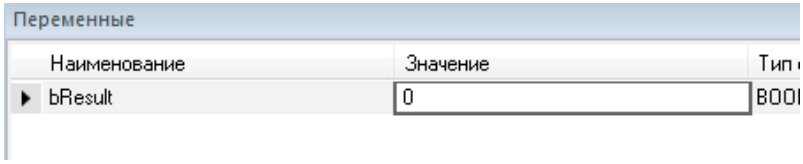
После этого запускаем модуль. Достигнув точки прерывания процесс остановится.



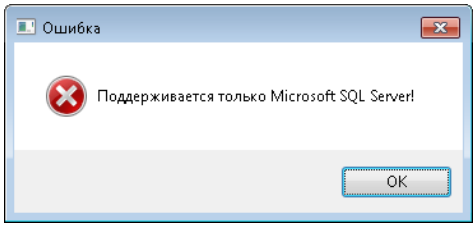
В нижней части экрана мы видим панель «Переменные». Если панель не видна, нужно вызвать ее через меню (Вид \ Панели \ Переменные). Встаем на закладку «Авто». Здесь отображаются автоматические (или стековые) переменные. Т.е. те, которые объявлены в блоке кода.



Встанем на значение переменной и изменим 1 на 0.



Продолжим исполнение, нажав F7, и получим сообщение



После этого модуль завершится. Отлично! Этого мы и хотели. Теперь мы уверены, что наш модуль можно запустить только на СУБД Microsoft SQL Server.

2.6.1.2 Создание структуры данных

Для того чтобы автоматически восстанавливать наши данные в базе, нам нужно при загрузке модуля отрабатывать SQL-скрипт, который будет проверять наличие таблиц в БД и создавать их, если они отсутствуют. А также, он будет обновлять данные в таблицах.

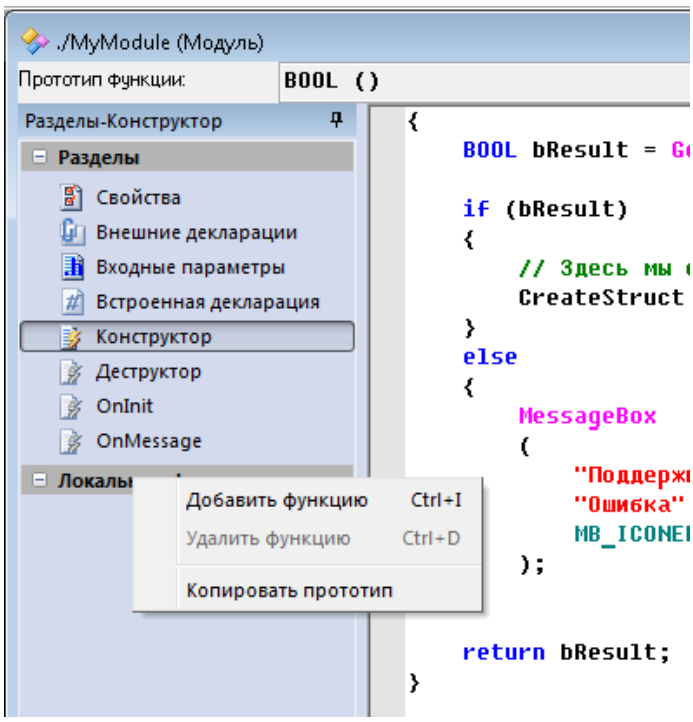
Этот скрипт приведен ниже.

```
IF OBJECT_ID ('Countries', 'U') IS NULL
BEGIN
	CREATE TABLE Countries
	(
		id_Country		int IDENTITY PRIMARY KEY,
		Name			nvarchar (255) NOT NULL,
		PopulationSize	int NOT NULL CHECK (PopulationSize >= 0)
	)

	GRANT INSERT, UPDATE, DELETE ON Countries TO public
END
GO

DELETE FROM Countries
GO

SET IDENTITY_INSERT Countries ON

INSERT INTO Countries (id_Country, Name, PopulationSize) VALUES
(1, N'Россия', 145975300),
(2, N'Индия', 1373701701),
(3, N'Китай', 1406778200),
(4, N'Германия', 83149300),
(5, N'Франция', 67413000),
(6, N'Италия', 60317000),
(7, N'Испания', 47500000),
(8, N'Великобритания', 68562151)

SET IDENTITY_INSERT Countries OFF
GO

IF OBJECT_ID ('Cities', 'U') IS NULL
BEGIN
	CREATE TABLE Cities
	(
		id_City		int IDENTITY PRIMARY KEY,
		Name		nvarchar (255) NOT NULL,
		PopulationSize	int NOT NULL CHECK (PopulationSize >= 0),
		Capital		int NOT NULL CHECK (Capital IN (1,0)),
		id_Country	int FOREIGN KEY (id_Country)
		REFERENCES Countries (id_Country) ON DELETE CASCADE
	)

	GRANT INSERT, UPDATE, DELETE ON Cities TO public
END
ELSE
BEGIN
```

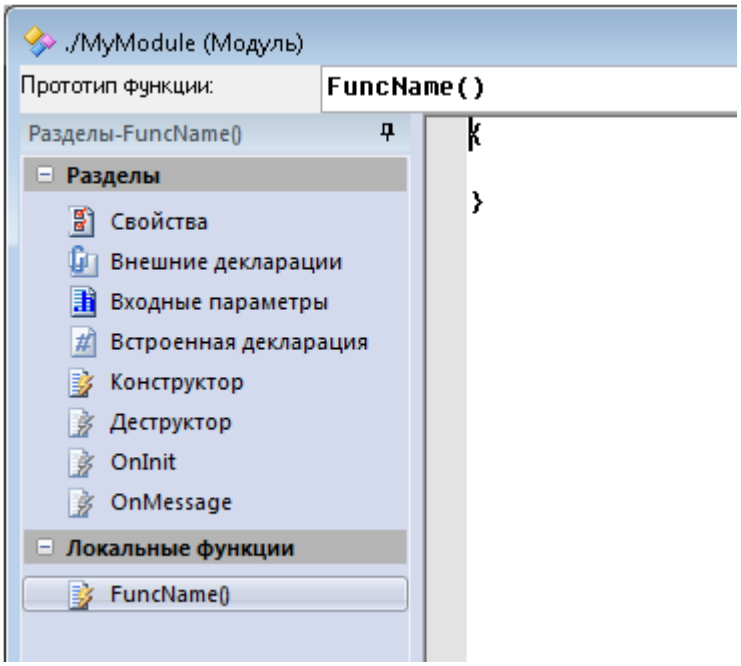
```
        DBCC CHECKIDENT (Cities, RESEED, 1)
END
GO

INSERT INTO Cities (id_Country, Name, PopulationSize, Capital) VALUES
(1, N'Москва', 12655050, 1),
(1, N'Санкт-Петербург', 5388759, 0),
(1, N'Волгоград',  100476, 0),
(1, N'Новосибирск', 1620162, 0),
(2, N'Нью-Дели', 300000, 1),
(2, N'Мумбаи', 15414288, 0),
(2, N'Калькутта', 4496694, 0),
(2, N'Дели', 9879172, 0),
(3, N'Пекин', 21710000, 1),
(3, N'Гонконг', 7500700, 0),
(3, N'Шанхай', 23390000, 0),
(3, N'Харбин', 5015000, 0),
(4, N'Берлин', 3644826, 1),
(4, N'Мюнхен', 1471508, 0),
(4, N'Гамбург', 1845229, 0),
(4, N'Лейпциг', 593197, 0),
(5, N'Париж', 2148327, 1),
(5, N'Ницца', 533554, 0),
(5, N'Марсель', 868277, 0),
(5, N'Бордо', 257068, 0),
(6, N'Рим', 2870500, 1),
(6, N'Венеция', 259939, 0),
(6, N'Милан', 1366180, 0),
(6, N'Неаполь', 956183, 0),
(7, N'Мадрид', 3334730, 1),
(7, N'Барселона', 1664182, 0),
(7, N'Валенсия', 800215, 0),
(7, N'Севилья', 691395, 0),
(8, N'Лондон', 8908081, 1),
(8, N'Манчестер', 552858, 0),
(8, N'Ливерпуль', 498042, 0),
(8, N'Белфаст', 341877, 0)
```

Скрипт получился довольно большой. Чтобы не загромождать наш конструктор, мы создадим отдельную функцию, которая будет исполнять этот скрипт, а из конструктора просто вызовем ее.

Для этого в правой панели редактора модуля кликнем правой клавишей мыши и в контекстном меню выберем пункт «Добавить функцию».



В списке локальных функций появилась новая функция FuncName.

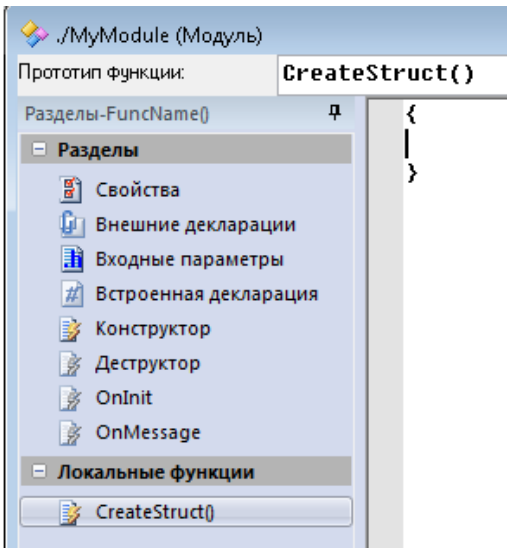


Это имя мы изменим на более осмысленное. Назовем нашу функцию CreateStruct. Здесь же мы можем задать параметры функции и тип возвращаемого значения. Например, если бы мы хотели написать функцию сложения двух целых чисел, то мы бы написали так:

```
int Sum (int x, int y)
```

А тело функции было бы таким:

```
{
        return x+y;
}
```

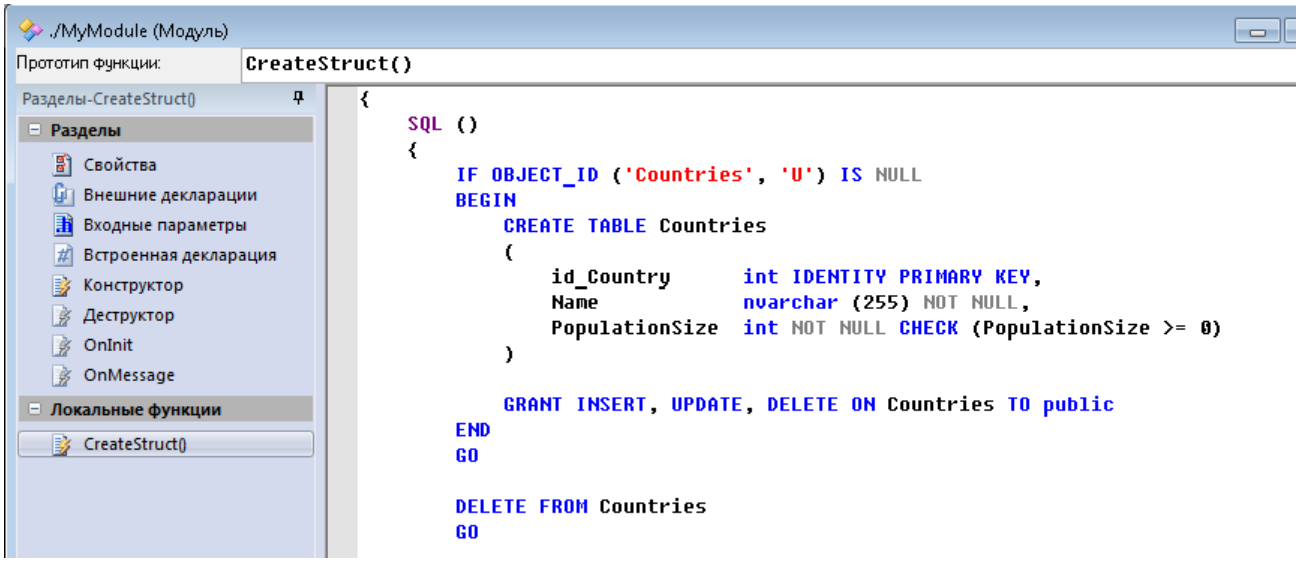


Наша функция не будет принимать входных параметров и возвращать значение, она будет только лишь исполнять SQL-скрипт. Поэтому мы просто изменим ее имя.

Для исполнения SQL-кода в X2 служит оператор «SQL». Форма вызова оператора:

```
SQL (V1, V2, …, VN)
{
        код на SQL
}
```

Здесь V1, V2, …, VN – это необязательные параметры, предназначенные для возврата значений из блока кода на SQL, если он содержит SELECT. В нашем случае они не нужны, т.к. наш скрипт не возвращает значений.



./MyModule (Модуль)

Прототип функции: CreateStruct()

Разделы-CreateStruct()

- Разделы
  - Свойства
  - Внешние декларации
  - Входные параметры
  - Встроенная декларация
  - Конструктор
  - Деструктор
  - OnInit
  - OnMessage
- Локальные функции

```
            (7, N'Мадрид', 3334730, 1),
            (7, N'Барселона', 1664182, 0),
            (7, N'Валенсия', 800215, 0),
            (7, N'Севилья', 691395, 0),
            (8, N'Лондон', 67886004, 1),
            (8, N'Манчестер', 552858, 0),
            (8, N'Ливерпуль', 498042, 0),
            (8, N'Белфаст', 341877, 0)
        } // SQL
    }
```

Теперь вернемся в конструктор модуля и вызовем фуекцию CreateStruct.

./MyModule (Модуль)

Прототип функции: BOOL ()

Разделы-Конструктор

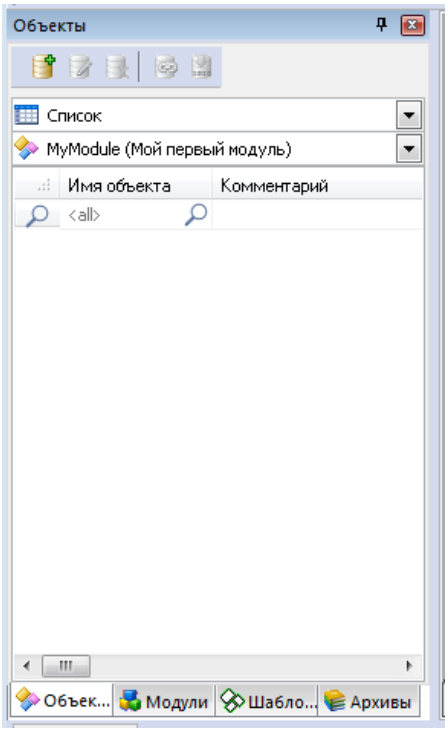
- Разделы
  - Свойства
  - Внешние декларации
  - Входные параметры
  - Встроенная декларация
  - Конструктор
  - Деструктор
  - OnInit
  - OnMessage
- Локальные функции
  - CreateStruct()

```
{
    BOOL bResult = GetSQLEngine() == MS_SQL_SERVER;

    if (bResult)
    {
        // Здесь мы создадим структуру данных
        CreateStruct ();
    }
    else
    {
        MessageBox
        (
            "Поддерживается только Microsoft SQL Server!",
            "Ошибка",
            MB_ICONERROR | MB_OK
        );
    }

    return bResult;
}
```

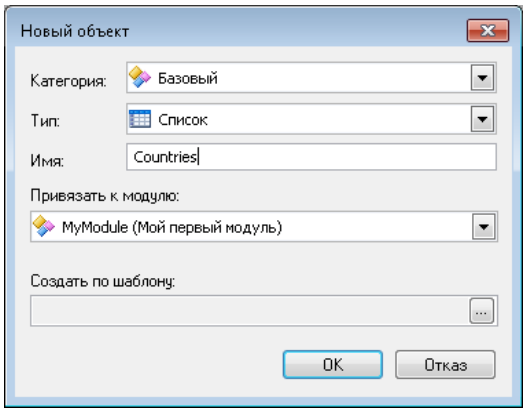
Сохраним изменения в БД. Теперь каждый раз при запуске модуля наши данные будут восстанавливаться. Все готово к началу работ. Можо приступать.

2.7 Создаем список

Вначале мы создадим список стран. Для этого в Диспетчере объектов встанем на закладку «Объекты» и выберем тип объекта «Список».

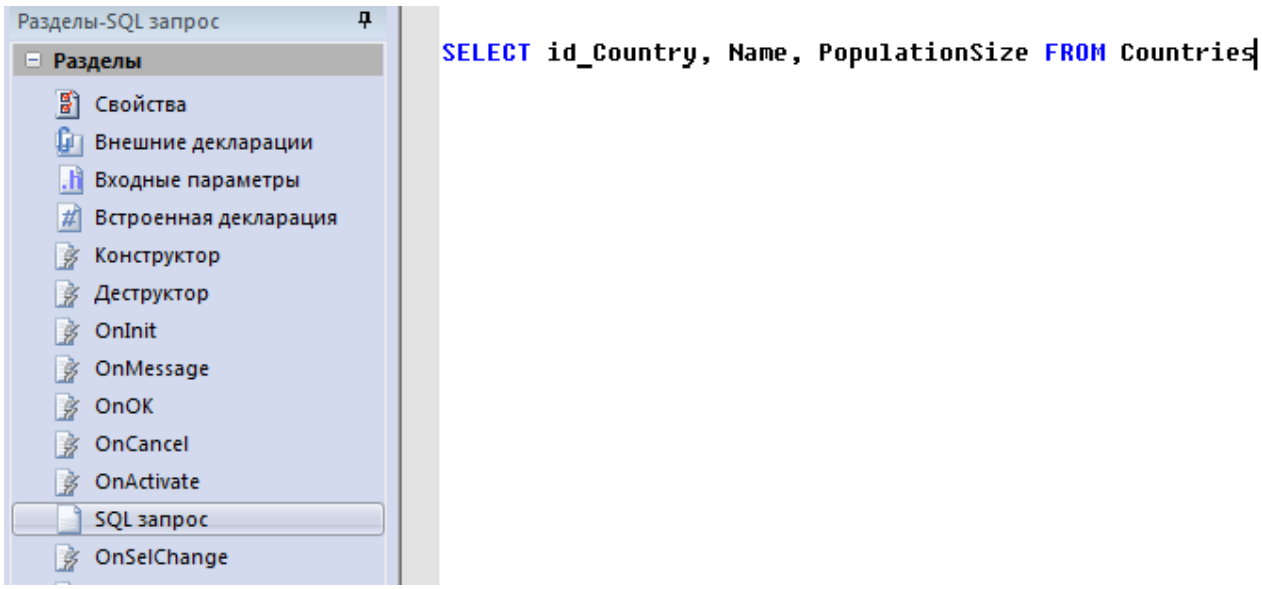


Добавим новый объект (Ins) и в диалоге создания объекта введем имя «Countries»



Далее, в редакторе списка перейдем в раздел «SQL запрос» и введем запрос

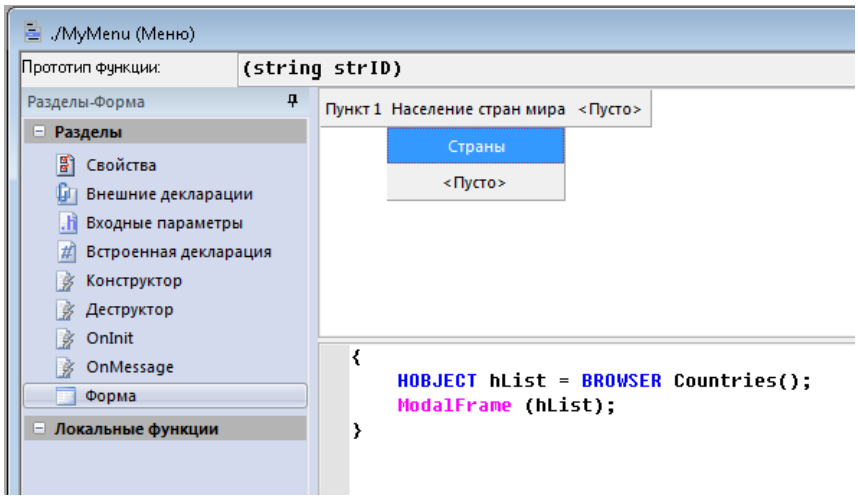
```
SELECT id_Country, Name, PopulationSize FROM Countries
```



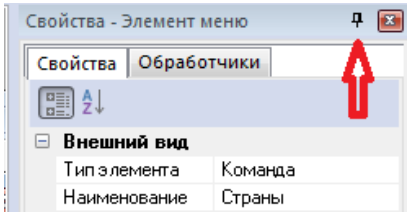
Наш список готов, сохраним его в БД.

Теперь нужно вызвать его из меню. Вернемся в наше меню, создадим в нем группу «Население стран мира» и добавим в нее команду «Страны», а в обработчике напишем такой код:

```
{
        HOBJECT hList = BROWSER Countries ();
        ModalFrame (hList);
}
```



Подсказка. Если Вас раздражает, что панель «Свойства» постоянно исчезает при переключении фокуса, то закрепите ее на экране.



Давайте рассмотрим подробнее код обработчика.

Визуальные объекты исполняются в два этапа:

1. Создание экземпляра объекта
2. Визуализация экземпляра объекта

Визуальные объекты, это те объекты, которые отображаются на экране. Это списки, деревья, диалоги, диаграммы и т.д. (есть еще невизуальные объекты, такие, как процедуры или декларации).

Для каждого типа визуальных объектов существует свой оператор создания экземпляра. Для списков это оператор BROWSER, для диалогов – DIALOG, для деревьев – TREE, и т.д. Оператор создания экземпляра создает в памяти экземпляр объекта, отрабатывает его конструктор и возвращает дескриптор этого экземпляра. Дескриптор – это целочисленная величина, имеющая тип HOBJECT, которая в дальнейшем будет идентифицировать этот экземпляр.

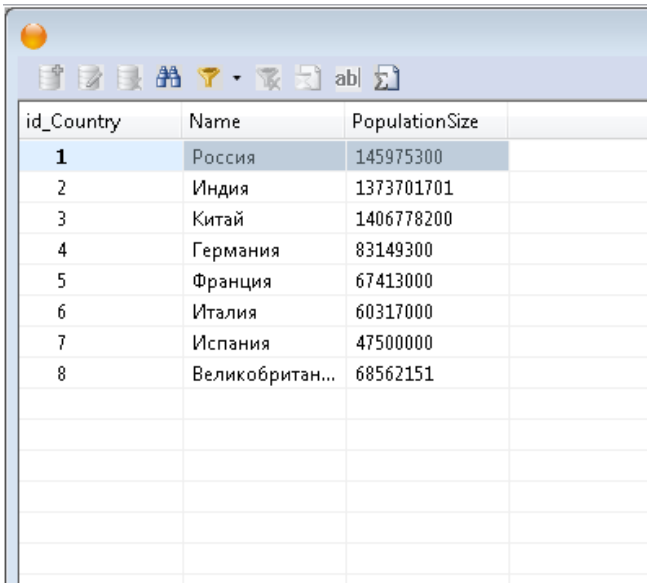
В нашем случае мы объявляем переменную hList, имеющую тип HOBJECT и присваиваем ей значение дескриптора, которое вернул оператор BROWSER Countries ().

После того, как экземпляр создан, мы можем к нему обращаться, используя его дескриптор. Например, мы можем изменить его настройки, заданные в разделе «Свойства». Но он еще не выведен на экран. Чтобы вывести его на экран, используются функции визуализации. Вот некоторые из них:

- ModalFrame – окно в модальном режиме
- ShowFrame – окно в немодальном режиме
- ShowPane – прилипающая панель

Функция визуализации задает категорию окна, в котором будет выведен экземпляр объекта. В нашем случае мы используем функцию ModalFrame (hList), которая отобразит наш экзепляр hList в модальном окне.

Теперь сохраним наше меню в БД, запустим модуль и выберем в меню наш пункт «Страны».



id_Country	Name	PopulationSize
1	Россия	145975300
2	Индия	1373701701
3	Китай	1406778200
4	Германия	83149300
5	Франция	67413000
6	Италия	60317000
7	Испания	47500000
8	Великобритан...	68562151

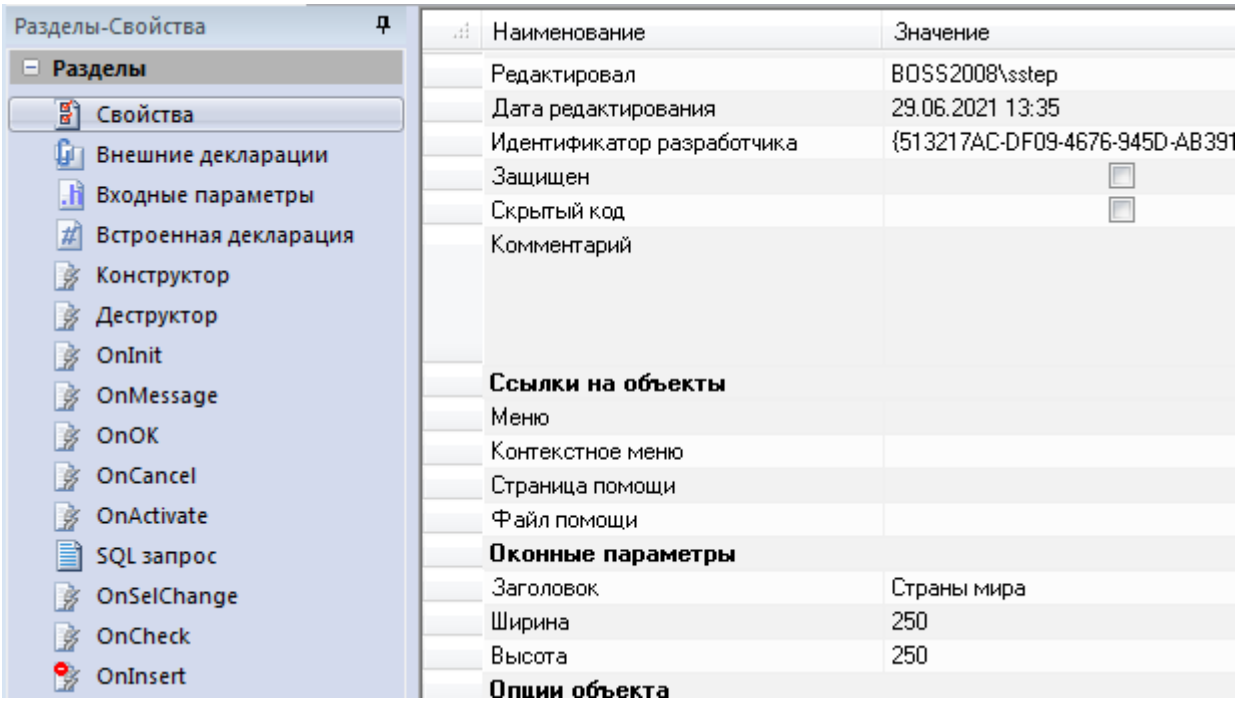
2.7.1 Настройка списка

Несмотря на то, что мы только задали SQL-запрос для нашего списка и еще не написали ни строки кода, мы получили вполне работающий список. Работает сортировка по столбцам, можно перетаскивать их мышкой, можно выполнять поиск и строить фильтр, можно маштабировать список (Ctrl+колесико мышки) и т.д. Полный перечень операций, доступных для списка можно увидеть в контекстном меню (правый клик).

Тем не менее, нас кое-что не устраивает:

1. Нет заголовка окна
2. Окно слишком большое, его не удается уменьшить
3. Колонки имеют имена полей таблицы, а хотелось бы задать свои имена
4. Первая колонка не должна отображаться для пользователя, она нужна нам только для служебных целей
5. Колонку «Страна» надо бы сделать пошире
6. В колонке «Население» неплохо было бы выводить числа с разделителем тысяч и выравнивать по правому краю

Сейчас мы все это проделаем. Закрываем приложение и идем в редактор списка.



В разделе «Свойства» в поле заголовка вводим «Страны мира».

Ширину и высоту устанавливаем в 250 пик. Тут нужно сказать несколько слов о размерах окна объекта. Здесь, в разделе «Свойства», задаются минимальные размеры окна. В период исполнения окно можно будет растянуть, но сжать меньше этих размеров нельзя. Это касается не только списков, но и других визуальных объектов.

После этого переходим в раздел «Колонки». Здесь мы будем задавать свойства колонок. С помощью клавиши «Ins» добавим три колонки.

	Заголовок	Модификатор	Флаги	Формат вывода	Формат ввода	Имя переменной	Ширина
0		Данные	1				100
1		Данные	0				100
2		Данные	0				100

А теперь настроим их следующим образом

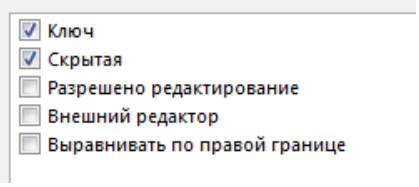
	Заголовок	Модификатор	Флаги	Формат вывода	Формат ввода	Имя переменной	Ширина
0		Данные	3				100
1	Страна	Данные	0				120
2	Население	Данные	16	r			100

Обратим внимание на столбцы «Модфикатор» и «Флаги».

Модификатор имеет два значения: «Данные» и «Пиктограмма». Если модификатор установлен в значение «Данные», то поле SQL-запроса интерпретируется, как данные, полученные из БД. Если он установлен в значение «Пиктограмма», то в поле запроса должно быть имя пиктограммы из таблицы x2Image. В этом случае список отобразит картинку, а не текст. Этот механизм мы рассмотрим позже.

Флаги управляют свойствами колонок.

Самое перве поле запроса (с индексом 0) является ключевым полем, и мы не хотим отображать эту колонку в списке. Так и выставим флаги.

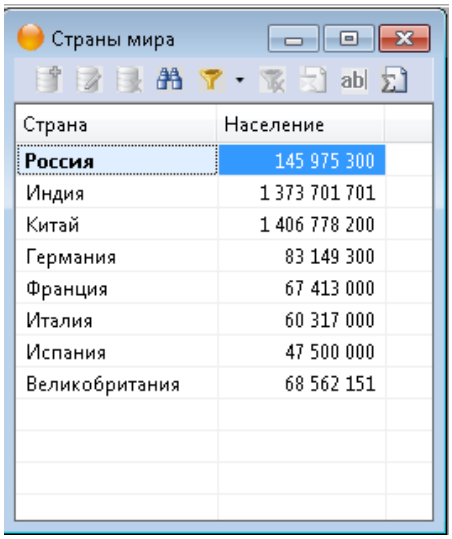


Для последнего поля запроса «Население» выставим флаг «Выравнивать по правой границе». И еще в столбце «Формат вывода» зададим маску «r».

Маска вывода. Для всех встроенных типов данных в языке X2 существует преобразование в строковый тип. Например, дату можно отобразить так: «29.06.21». Или так: «29 июня 2021». То же относится и к чисовым типам. В нашем случае маска «r» означает, что число нужно вывести с учетом региональных настроек, т.е. будет использоваться разделитель тысяч. Более подробно о масках можно узнать в разделе электронной документации «Маски преобразования в строковый тип», либо в описании фукции ToString. Также, сущечтвуют маски ввода, о которых будет сказано ниже при рассмотрении диалогов.

Теперь сохраним изменения, запустим модуль и вызовем список.

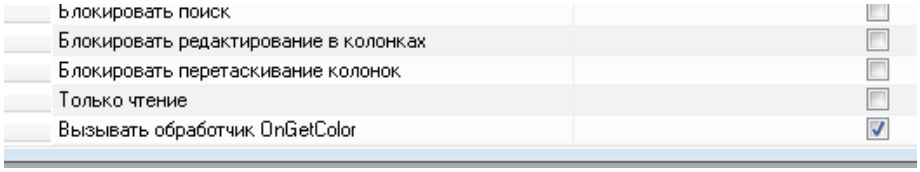
Ну вот, наш список приобрел более приличный вид.



Сразу оговорюсь, что все настройки колонок можно задать и программным путем с помощью функции ListSetFieldOptions, а заголовок окна можно установить с помощью функции SetFrameTitle.

2.7.2 Выделение цветом

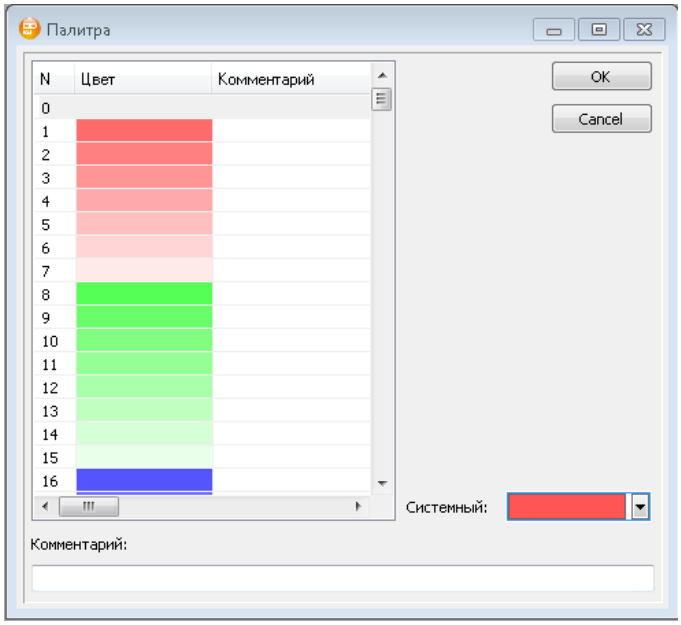
Ячейки в списке можно выделять цветом. Для этого в настройках списка (раздел «Свойства») нужно установить флаг «Вызывать обработчик OnGetColor».



Если этот флаг установлен, то при отображении списка для каждой его ячейки будет вызываться обработчик OnGetColor, который должен вернуть системе индекс цвета в палитре. По умолчанию этот флаг снят, т.к. вызов обработчика замедляет прокрутку списка.

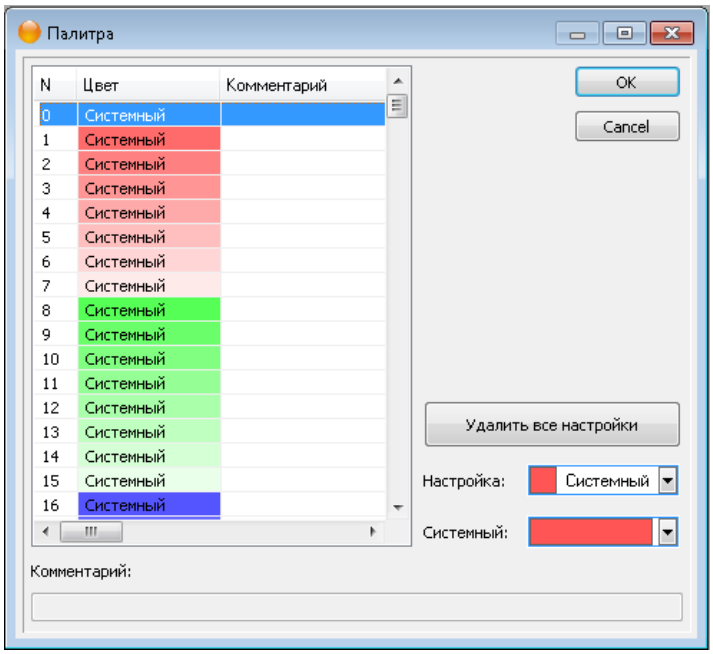
Палитра

В системе существуют две палитры: системная и пользовательская. Системная палитра настраивается разработчиком в среде разработки RPDesigner.exe (Сервис \ Справочники \ Палитра).



Здесь каждому цвету соответствует свой индекс (он отображен в первой колонке). Всего палитра может содержать 255 цветов. Разработчик может настроить цвета палитры так, как ему требуется. Поле «Комментарий» является необязательным. Комментарий может быть полезным, когда разработчики одной команды хотят использовать определенные цвета единым образом. Это придает интерфейсу целостность. Например, во всех списках, где выводятся отрицательные числа, можно для таких ячеек задать единый цвет. Это, также, поможет пользователю, если он захочет перенастроить цвета в списках.

Пользовательская палитра настраивается пользователем в среде исполнения RPExec.exe (Файл \ Настройка \ Палитра).



Пользовательская палитра базируется на системной, но каждый пользователь может перенастроить цвета по своему усмотрению. Эта настройка не меняет системную палитру и действует только для конкретного пользователя. Списки всегда отображаются в соответствии с пользовательской палитрой. Если пользователь ничего не настраивал, то пользовательская палитра будет соответствовать системной.

Обработчик OnGetColor

После того, как мы установили флаг «Вызывать обработчик OnGetColor», этот обработчик стал доступен. Перейдем в него и напишем такой код:

```
{
	int nColor = -1;

	if (ListColToField (nCol) == 2)
	{
		int nPopulationSize;

		ListGetFieldData (nPopulationSize, nRow, 2);

		if (nPopulationSize >= 1000000000)
			nColor = 2;
		else if (nPopulationSize >= 100000000)
			nColor = 4;
		else
			nColor = 6;
	}

	return nColor;
}
```

./Countries (Список)

Прототип функции: `int OnGetColor (int nRow, int nCol)`

Разделы-OnGetColor

Разделы

- Свойства
- Внешние декларации
- Входные параметры
- Встроенная декларация
- Конструктор
- Деструктор
- OnInit
- OnMessage
- OnOK
- OnCancel
- OnActivate
- SQL запрос
- OnSelChange
- OnCheck
- OnInsert
- OnDelete
- OnEdit
- OnEditCell
- OnGetColor
- Колонки

```
{
    int nColor = -1;

    if (ListColToField (nCol) == 2)
    {
        int nPopulationSize;

        ListGetFieldData (nPopulationSize, nRow, 2);

        if (nPopulationSize >= 1000000000)
            nColor = 2;
        else if (nPopulationSize >= 100000000)
            nColor = 4;
        else
            nColor = 6;
    }

    return nColor;
}
```

В этом обработчике мы будем раскрашивать колонку, содержащую численность населения, в три разных цвета в зависимости от значения поля.

Как мы видим, обработчик принимает на входе два параметра: nRow и nCol. Это координаты ячейки списка, для которой обработчик должен вернуть индекс цвета (значение от 0 до 255). Если обработчик вернет -1, то система не будет использовать палитру для задания фона ячейки, т.е. будет использоваться цвет фона окна, заданный в настройках самой операционной системы.

Здесь еще нужно пояснить, что параметр nCol содержит номер видимой колонки списка. Пользователь может поменять местами колонки, поэтому, если мы хотим раскрашивать только колонку с цифрами, то нам нужно завязываться не на номер видимой колонки, а на номер поля в запросе, т.к. последовательность полей в запросе постоянна. Функция ListColToField как раз и возвращает нам номер поля запроса по номеру колонки. Если это поле PopulationSize (номер 2, отсчет от 0), то с помощью функции ListGetFieldData мы достаем из ячейки списка данные в переменную nPopulationSize. А дальше просто анализируем ее значение и возвращаем 2, 4 или 6.

Выглядит это вот так.

Страны мира

Страна	Население
Россия	145 975 300
Индия	1 373 701 701
Китай	1 406 778 200
Германия	83 149 300
Франция	67 413 000
Италия	60 317 000
Испания	47 500 000
Великобритания	68 562 151

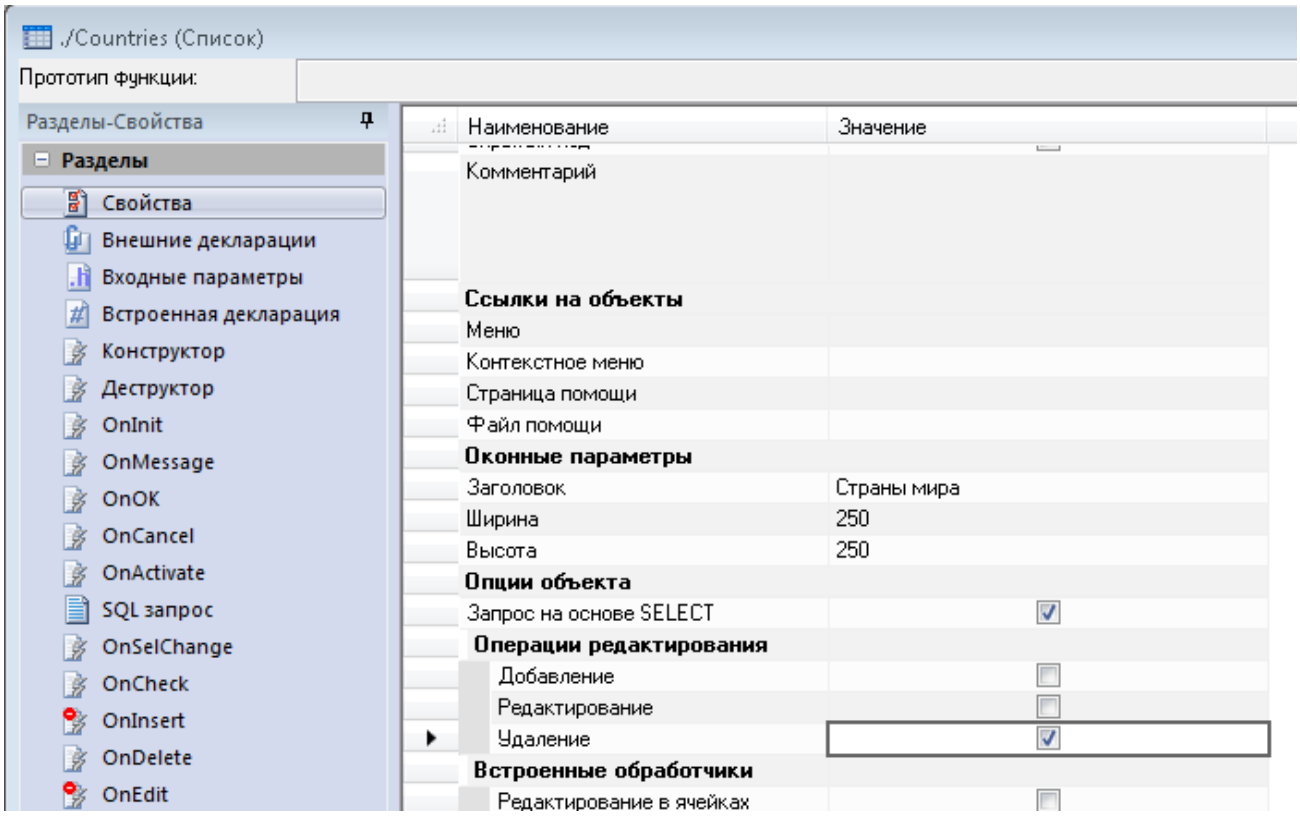
2.7.3 Редактирование в списке

Теперь, когда у нас есть список, научим его операциям редактирования.

По умолчанию операции редактирования в списке заблокированы. Чтобы их разблокировать, нужно установить соответствующие флаги в настройках списка. Это можно сделать на этапе разработки в разделе «Свойства», но также, можно менять состояние этих флагов в период исполнения с помощью функции SetOptions. Это позволяет программисту динамически управлять доступностью операций редактирования.

2.7.3.1 Операция удаления

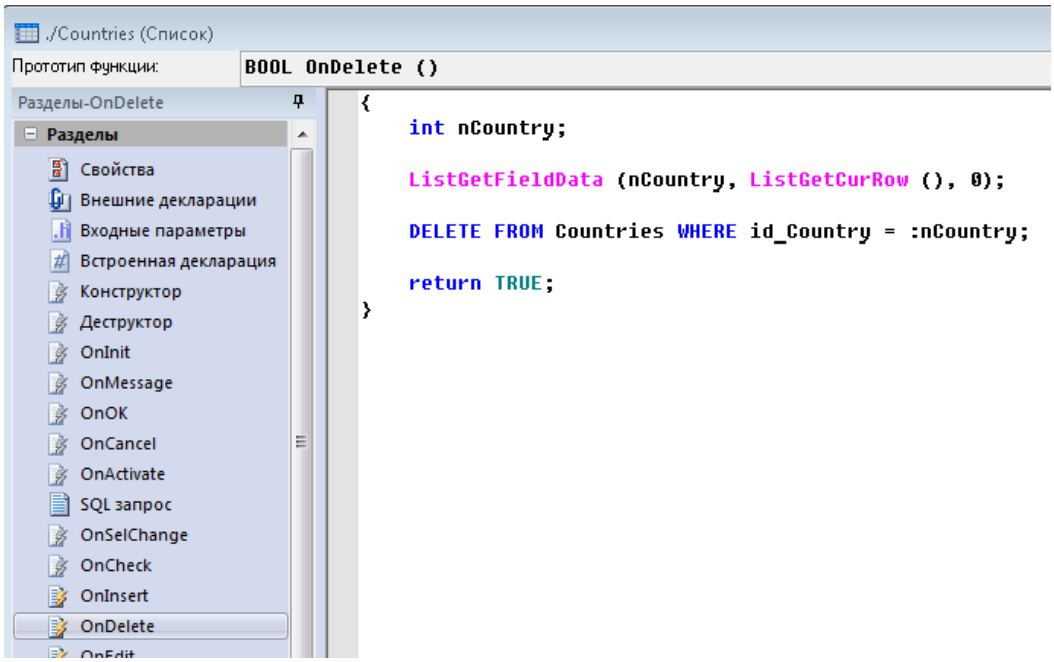
Начнем с удаления, в нашем примере это самая простая операция. Для этого перейдем в раздел «Свойства» и установим флажок «Удаление».



Мы видим, что при установке этого флажка стал доступен обработчик «OnDelete». Он будет вызван на исполнение, когда мы в списке нажмем клавишу «Del» или соответствующую кнопку в линейке инструментов списка.

Перейдем в него и напишем такой код:

```
{
        int nCountry;

        ListGetFieldData (nCountry, ListGetCurRow (), 0);

        DELETE FROM Countries WHERE id_Country = :nCountry;

        return TRUE;
}
```



Давайте рассмотрим этот код подробнее.

Чтобы удалить строку из таблицы, мы должны выполнить оператор DELETE и указать значение ключевого поля удаляемой строки. Но вначале мы должны получить это значение. Для этого мы объявляем переменную nCountry и вызываем функцию ListGetFieldData. Эта функция положит в нашу переменную nCountry значение того поля SQL-запроса, которое задают два последующих параметра (номер строки и номер поля запроса; нумерация начинается с 0). Функция ListGetCurRow вернет нам номер текущей строки в списке, а в качестве номера поля мы передаем 0, поскольку в запросе списка ключевое поле id_Country стоит именно в этой позиции.

```
SELECT id_Country, Name, PopulationSize FROM Countries
```

После того, как мы получили в переменную nCountry ключ удаляемой строки, выполним оператор DELETE. Здесь в ограничение WHERE мы выполнили подстановку нашей переменной. Синтаксически это выделено двоеточием. В коде на SQL мы используем двоеточие для обозначения тех мест, куда нужно подставить значения переменных.

Вы уже знаете, что для исполнения SQL-кода в X2 служит оператор «SQL», но в данном случае мы его не используем, а вызываем DELETE непосредственно. Дело в том, что в X2 одиночные операторы INSERT, UPDATE, DELETE, TRUNCATE можно вызывать непосредственно. В этом случае запрос обязательно должен заканчиваться точкой с запятой.

В нашем случае можно было бы написать так:

```
SQL ()
{
        DELETE FROM Countries WHERE id_Country = :nCountry
}
```

При такой форме записи точка с запятой в конце запроса не нужна. Впрочем, обе формы эквивалентны, так что можно использовать любую по своему усмотрению.

Ну и наконец, последнее.

```
        return TRUE;
```

Обработчик OnDelete должен вернуть значение типа BOOL (TRUE или FALSE). Если обработчик вернул TRUE, то это будет сигналом для системы, что операция удаления данных из БД прошла корректно, и система удалит из списка выделенные строки. Если же он вернул FALSE, значит, что-то пошло не так, и система не будет удалять строки из списка.

Поскольку при удалении мы не выполняем никаких проверок, то возвращаем TRUE. А, если при исполнении DELETE возникнет серверная ошибка, то обработчик будет завершен автоматически, не дойдя до оператора return.

Это поведение системы можно изменить. С помощью функции SetSqlMuteMode можно перевести систему в режим подавления ошибок, функция SqlSuccess возвращает статус исполнения последнего SQL-запроса, а функция GetSqlErrorString возвращает текст серверной ошибки.

Сохраним изменения и запустим модуль. Обратим внимание, что в линейке инструментов списка стала доступна кнопка удаления.

Также, стала доступна команда контекстного меню, и список будет реагировать на клавишу «Del». Нажмем эту клавишу и убедимся, что выделенная строка исчезла.

2.7.3.2 Операция добавления

Теперь реализуем операцию добавления строки.

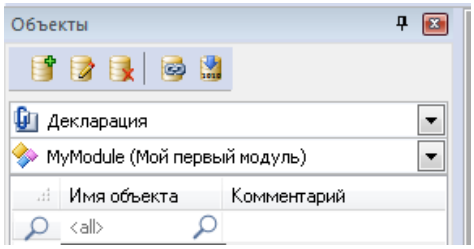
Для этого в списке есть обработчик OnInsert. Чтобы реализовать эту операцию, нам потребуется диалог для ввода данных. Из обработчика мы вызовем этот диалог. В диалоге мы выполним проверку корректности ввода. Если диалог будет закрыт по кнопке «OK», то в обработчике OnInsert мы выполним вставку данных в базу и обновим содержимое списка. И еще нам потребуется структура для обмена данными между списком и диалогом. С этого мы и начнем.

Объект «Декларация»

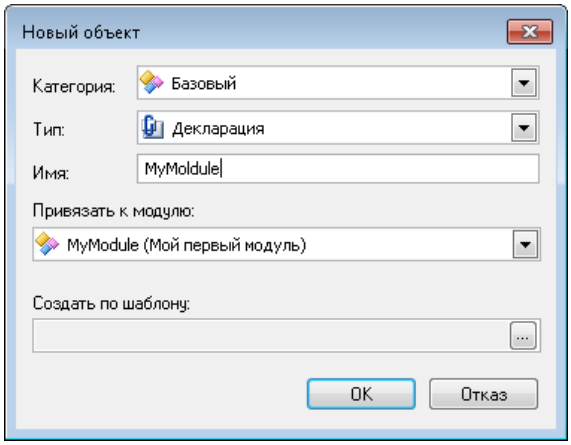
Структура – это тип данных. Мы должны объявить его в таком месте, чтобы он был доступен, как в списке, так и в диалоге. Для этого и служит объект «Декларация». В декларации мы можем объявлять типы, константы и глобальные переменные. Все прочие объекты имеют раздел «Внешние декларации». Если включить в объект декларацию, то все, что объявлено в ней, будет доступно в данном объекте.

Объекты, также, имеют раздел «Встроенная декларация». Встроенная декларация играет ту же роль, что и внешняя, только все, что объявлено в ней, будет доступно лишь внутри данного объекта. Переменные, объявленные во встроенной декларации, имеют локальную область видимости, и являются членами данных объекта. Они не доступны снаружи.

Начнем с создания объекта «Декларация». Для этого в Диспетчере объектов переключимся на закладку «Объекты» и выберем тип «Декларация».

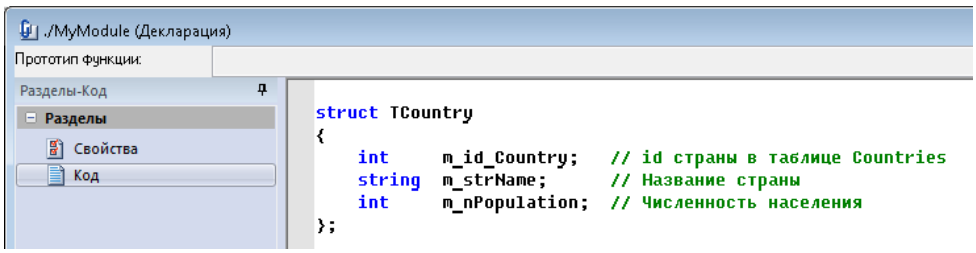


Дальше, как обычно, создадим новый объект и укажем имя. Декларация может иметь любое имя, но, поскольку мы будем объявлять в ней только то, что потребуется нам в нашем модуле, назовем ее «MyModule».

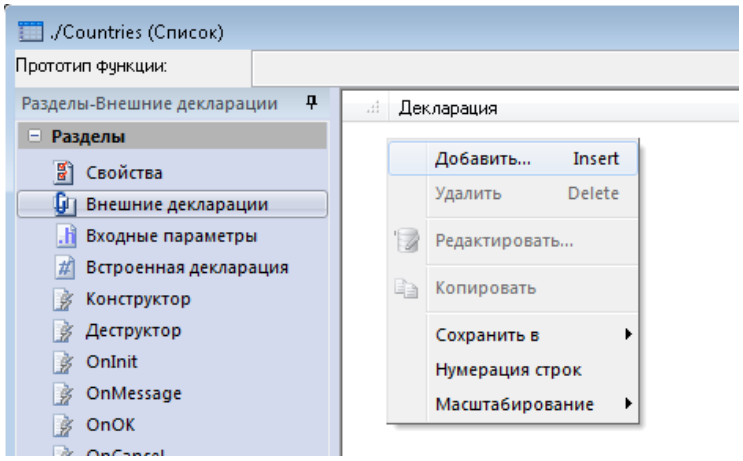


Теперь, когда декларация создана, объявим в ней нашу структуру для обмена данными между списком и диалогом.

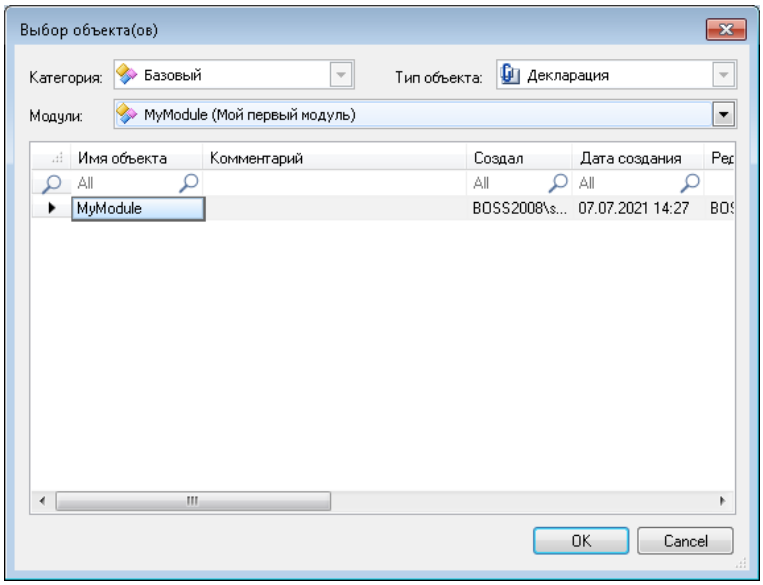
```
struct TCountry
{
        int       m_id_Country;  // id страны в таблице Countries
        string    m_strName;     // Название страны
        int       m_nPopulation; // Численность населения
};
```

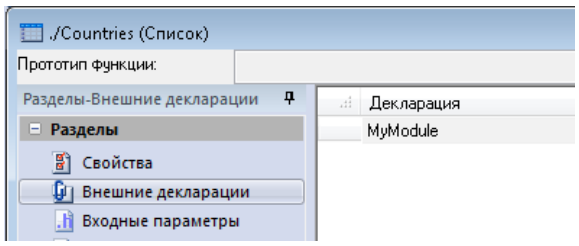


Декларация готова. Сохраним ее в БД и включим в наш список. Для этого в редакторе списка выберем раздел «Внешние декларации» и кликнем правой клавишей мыши в окне справа. В контекстном меню выберем команду «Добавить».



Далее, в диалоге выбора объектов выберем нашу декларацию.





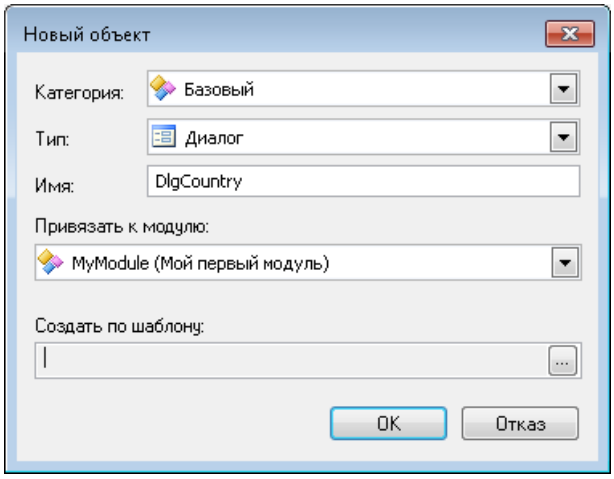
После того, как мы подключили нашу декларацию, структура TCountry стала доступна в списке Countries, и мы можем ее использовать.

Создаем диалог редактирования списка

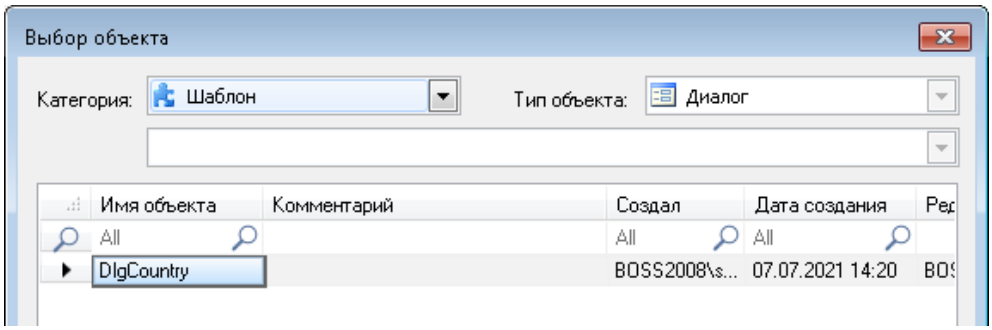
Теперь создадим диалог для ввода данных. Этот диалог мы будем использовать, как для вставки, так и для редактирования. Чтобы не тратить время на рисование формы диалога, можно использовать шаблон. В демонстрационном модуле уже есть готовый шаблон для этого диалога.

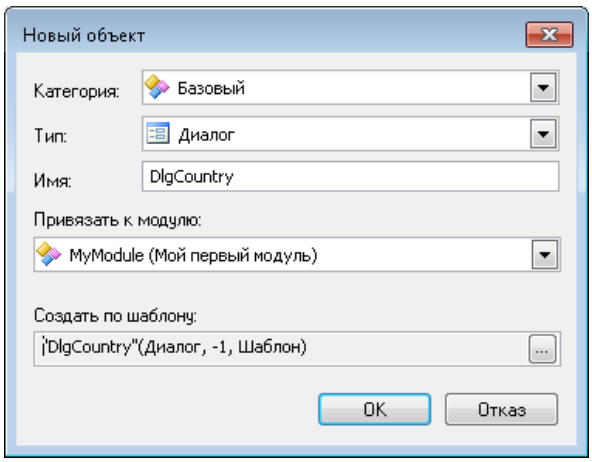
Немного о шаблонах. Шаблон – это просто заготовка объекта. Шаблоны хранятся в таблице x2Templates. При сохранении в БД они не компилируются, как настоящие объекты, сохраняется только их исходный код. Шаблоны используются в качестве основы для создания настоящих объектов. К примеру, если в приложении есть много форм, использующих логотип Компании, то бывает удобно вначале создать шаблон и разместить на нем логотип, а потом создавать объекты по этому шаблону.

В Диспетчере объектов перейдем на закладку «Объекты» и укажем тип объекта «Диалог». Далее, как обычно, создаем новый объект. Нашему новому диалогу дадим имя DlgCountry.

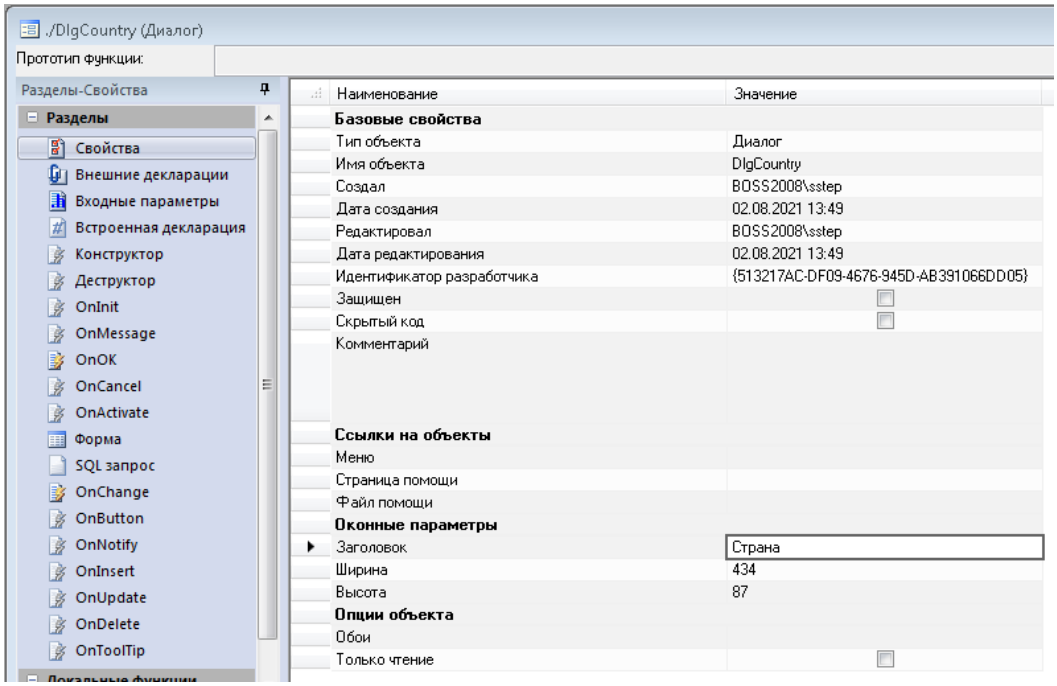


Теперь зададим шаблон. Нажмем на кнопку «…» в поле «Создать по шаблону» и в выпадающем меню укажем «Шаблон…». В диалоге выбора объектов установим фильтр следующим образом:





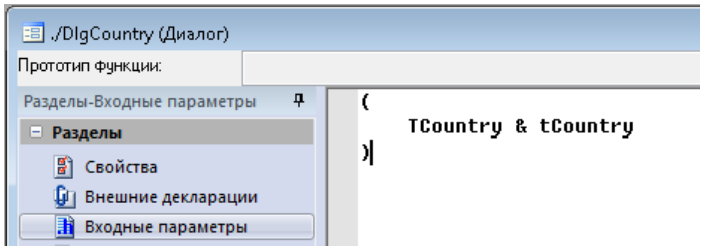
После того, как диалог создан, обратим внимание на одну особенность шаблонов: шаблоны не хранят заголовок окна. Поэтому мы должны ввести заголовок вручную.



Теперь сохраним наш диалог и рассмотрим подробно, как он устроен.

Перейдем в раздел «Внешние декларации». Как и в списке Countries, декларация MyModule уже включена в диалог.

Теперь перейдем в раздел «Входные параметры».

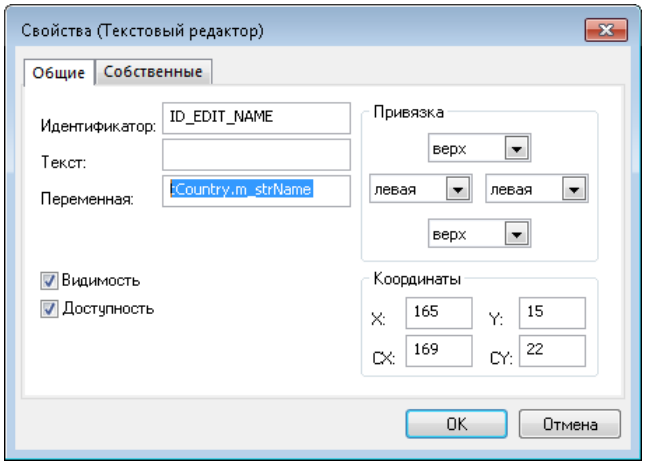


Здесь мы видим, что диалог принимает один входной параметр tCountry, который имеет тип TCountry, объявленный в нашей декларации. Знак «&» означает, что параметр передается по ссылке, а не по значению. Это важно для нас, поскольку через этот параметр диалог вернет списку данные, введенные пользователем.

Параметры могут передаваться по значению или по ссылке. При передаче по значению объект работает с локальной копией передаваемой переменной. При передаче по ссылке объект работает непосредственно с переменной вызывающего кода. Структуры и массивы могут передаваться только по ссылке.

Теперь перейдем в раздел «Форма».

Диалог имеет два поля ввода: «Название» и «Численность населения». Дважды кликнем на верхнем поле ввода, чтобы войти в его свойства.

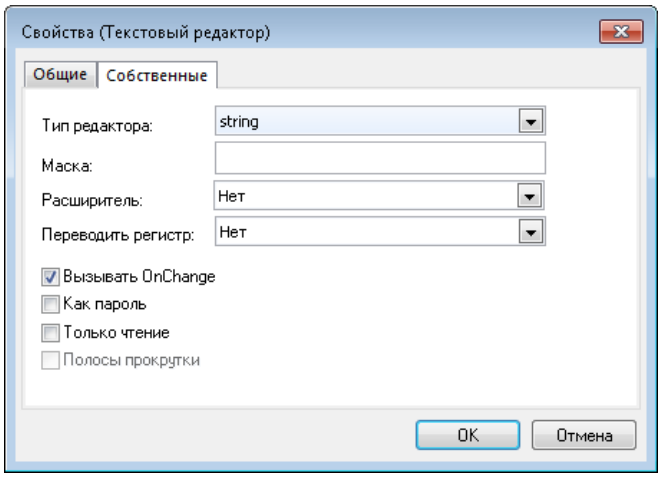


Обратим внимание, что с этим полем ввода связана переменная tCountry.m_strName. Это поле нашей структуры, передаваемой в диалог в качестве параметра. В эту переменную попадет значение поля, когда диалог завершится.

С управляющим элементом диалога можно связать переменную, которая называется буферной переменной. Тип переменной должен соответствовать типу управляющего элемента. При вызове диалога значение буферной переменной проинициализирует управляющий элемент. Далее, если пользователь изменит состояние управляющего

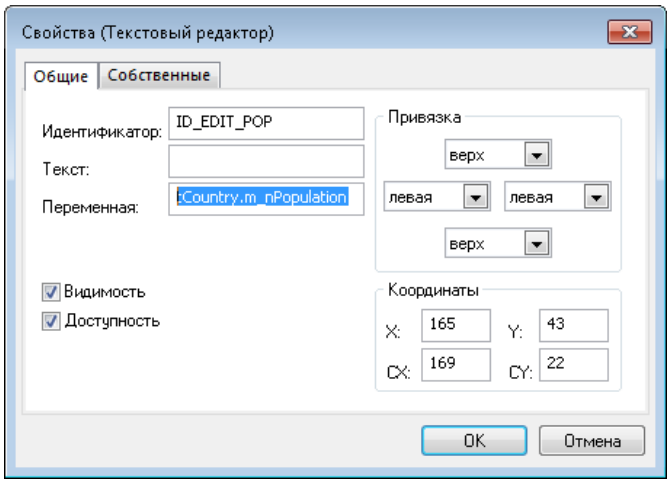
элемента, то значение буферной переменной, также, будет изменено. Это произойдет, когда управляющий элемент потеряет фокус ввода (например, при переключении на другое поле или нажатии на кнопку).

Переключимся на закладку «Собственные».

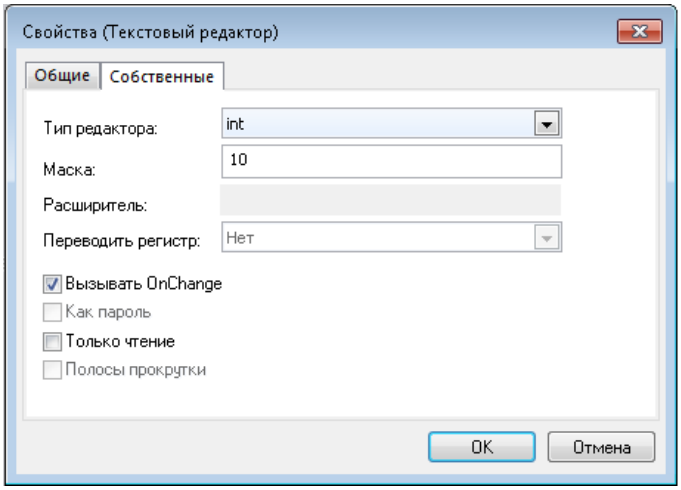


Здесь мы видим, что тип редактора указан, как string, а также, установлен флаг «Вызывать OnChange». OnChange – это обработчик, который будет вызываться при потере фокуса управляющим элементом, если пользователь изменил его состояние. Если снять этот флаг, то обработчик вызываться не будет.

Теперь рассмотрим свойства второго поля.



С ним связано второе поле нашей структуры tCountry.m_nPopulation.

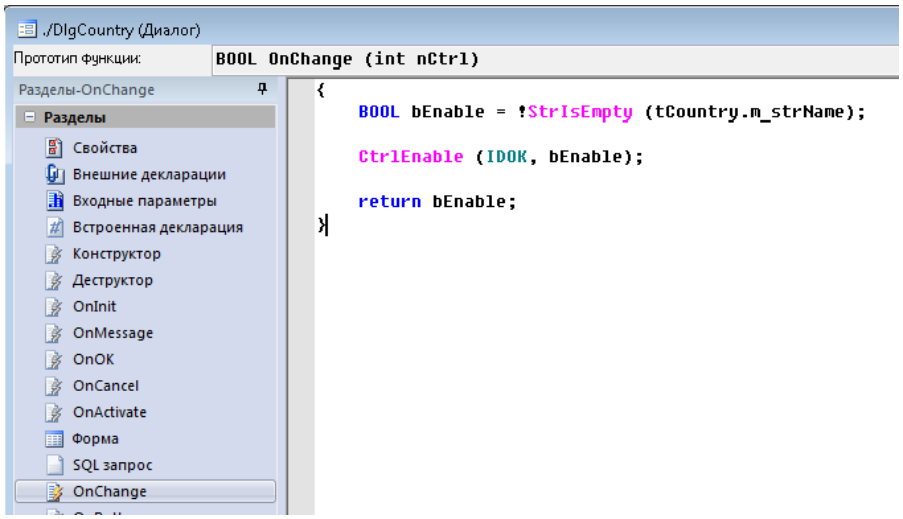


Здесь тип редактора указан, как int. Это позволит пользователю вводить только целые числа. Обратим внимание на поле «Маска».

Для встроенных типов данных существуют маски ввода и маски вывода. Маски – это наборы управляющих символов, определяющие представление данных. Маска ввода задается в редакторе и определяет формат ввода, например, для номеров телефонов, паспортных данных и т.д. Маска вывода определяет то, как будут выглядеть данные при переводе в текстовый формат. Мы уже сталкивались с масками вывода, когда задавали настройки для колонок списка.

В поле «Маска» задана маска ввода для нашего редактора. В данном случае маска указывает, что пользователь сможет вводить только положительные числа, ограниченные 10-ю символами.

Итак, наш диалог получает на входе структуру, два поля которой связаны с двумя полями диалога. Через эту структуру он и вернет данные списку. Осталось только проверить данные на корректность. Мы сделаем это в обработчиках OnChange и OnOK. Перейдем в обработчик OnChange и посмотрим на его код.



Поскольку для численности населения мы уже задали ограничения через маску ввода, то это значение и так будет корректным. А вот для названия страны нужно выполнить проверку, что строка не пуста. В зависимости от этого мы будем делать доступной или недоступной кнопку «OK».

Функция CtrlEnable управляет доступностью элемента. Она принимает на входе идентификатор элемента и признак, будет ли он доступен. IDOK – это и есть идентификатор кнопки «OK». Это можно увидеть, если войти в ее свойства.

Обработчик OnChange принимает на входе параметр (int nCtrl). Это идентификатор элемента, который был изменен. В первый раз обработчик будет вызван при начальной инициализации диалога, и в этом случае nCtrl будет равен -1. Впоследствии, когда он будет вызываться на действия пользователя, nCtrl будет содержать значение идентификатора. В данном случае нам этот параметр не нужен, мы будем выполнять проверку в любом случае.

Обработчик, также, должен вернуть значение типа BOOL. Когда пользователь меняет состояние управляющего элемента, система устанавливает для него внутренний флаг модификации. Если обработчик вернет TRUE, то система сбросит флаг модификации элемента, полагая, что данные были введены корректно. Если обработчик вернет FALSE, то система оставит флаг модификации взведенным, и обработчик будет вызываться каждый раз при потере элементом фокуса ввода.

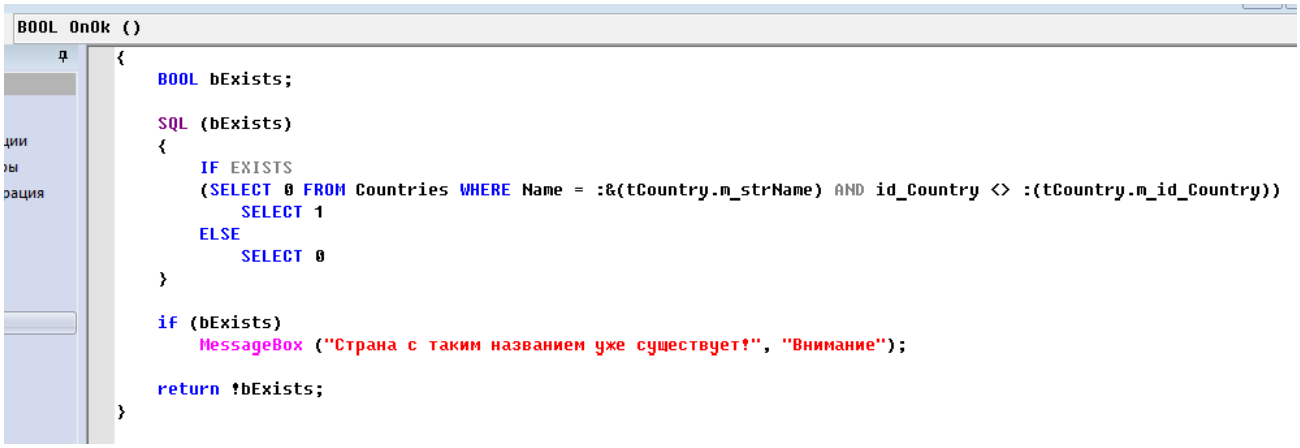
В нашем случае обработчик возвращает значение переменной bEnable, определяющей, пуста строка названия страны или нет.

Но это еще не все.

При вставке новой строки мы должны проверить, что страны с таким названием еще нет в таблице Countries. Мы сделаем это в обработчике OnOK. Этот обработчик будет вызван, когда пользователь нажмет кнопку «OK».

Перейдем в него и рассмотрим его код.

```
{
        BOOL bExists;

        SQL (bExists)
        {
                IF EXISTS
                (SELECT 0 FROM Countries WHERE
                Name = :&(tCountry.m_strName) AND id_Country <> :(tCountry.m_id_Country))
                        SELECT 1
                ELSE
                        SELECT 0
        }

        if (bExists)
                MessageBox ("Страна с таким названием уже существует!", "Внимание");

        return !bExists;
}
```



Начнем с того, что обработчик должен вернуть значение типа BOOL. Мы это видим в верхней строке, где указан его прототип. Если он вернет TRUE, то диалог закроется по кнопке «OK»; если он вернет FALSE, диалог останется на экране.

Выполним проверку, чтобы удостовериться, что в таблице Countries не существует страны с таким названием и другим ключом id_Country.

Для этого мы объявили переменную bExists и передали ее в оператор SQL, как параметр. В эту переменную система положит значение, которое вернет оператор SELECT. Дальше мы выполним проверку и выдадим сообщение, а из обработчика вернем !bExists, т.е. TRUE, если такой строки в таблице не существует.

Мы уже знаем, что в код на SQL можно подставлять значения переменных, и места подстановок обозначены двоеточием. Обратим внимание на две детали.

Во-первых, на круглые скобки. Они используются потому, что мы обращаемся не к скалярной переменной, а к полю структуры. tCountry.m_strName, по сути, является выражением, которое компилятор должен вычислить. Т.е. не просто взять данные по

адресу, скрытому за именем tCountry, но еще и выполнить смещение на указанное поле структуры m_strName. А выражения при подстановке должны окружаться круглыми скобками.

Во-вторых, на знак «&». Дело в том, что синтаксис SQL предполагает, что строковые литералы должны окружаться одинарными кавычками. Но подстановку можно выполнять двумя способами: по ссылке и по значению. Если бы мы выполняли подстановку по значению, то мы должны были бы написать так:

```
:("'" + tCountry.m_strName + "')
```

или так:

```
:(StrQuote (tCountry.m_strName))
```

При подстановке по ссылке кавычки не нужны, т.к. в этом случае запрос будет параметризованным, а подставляемое значение будет передано, как его параметр. При этом нужно понимать, что синтаксис SQL позволяет параметризировать только значения. Имена объектов БД, к примеру, не могут выступать, как параметры.

```
string strCountries = "Countries";

SELECT * FROM :strCountries          -- правильно
SELECT * FROM :&strCountries         -- ошибка
```

Список. Обработчик OnInsert.

Вернемся к нашему списку, перейдем в раздел «Свойства» и установим флажок «Добавление». Как видим, стал доступен обработчик OnInsert. Напишем код обработчика и рассмотрим его.

```
{
	BOOL		bResult = FALSE;
	TCountry	tCountry;

	if (ModalFrame (DIALOG DlgCountry (tCountry), NO_SIZING) == IDOK)
	{
		SQL (tCountry.m_id_Country)
		{
			INSERT INTO Countries (Name, PopulationSize) VALUES
			(
				:&(tCountry.m_strName),
				:(tCountry.m_nPopulation)
			)
			SELECT @@IDENTITY
		}

		bResult = TRUE;
		ListSetFieldData (tCountry.m_id_Country, -1, 0);
		ListRefreshRow ();
	}

	return bResult;
}
```

Мы объявили переменную tCountry, через которую диалог вернет нам данные, введенные пользователем. Оператор

```
	DIALOG DlgCountry (tCountry)
```

создаст экземпляр диалога и вернет его дескриптор. Этот дескриптор мы передаем в функцию ModalFrame, которая визуализирует диалог в модальном фрейме. Второй ее параметр NO_SIZING означает, что окно диалога не будет растягиваться мышкой.

Функция ModalFrame вернет нам управление, когда пользователь закроет диалог, до этого момента исполнение нашего кода будет приостановлено. Пользователь может закрыть диалог двумя способами: нажать «OK» либо нажать «Отказ» (или крестик в верхнем правом углу окна). Функция ModalFrame вернет нам идентификатор кнопки, с помощью которой пользователь закрыл диалог, т.е. IDOK или IDCANCEL. Это условие мы и проверяем в операторе if.

Дальше нам нужно выполнить вставку новой строки в таблицу Countries и получить ключ этой строки. Для этого мы с помощью оператора SQL исполняем два запроса. INSERT выполняет вставку, SELECT @@IDENTITY возвращает текущее значение IDENTITY, т.е. ключ новой строки, в поле структуры tCountry.m_id_Country.

После того, как мы выполнили вставку в таблицу и получили ключ новой строки, нам нужно обновить данные в списке. Перед вызовом обработчика OnInsert система уже добавила в конец списка новую пустую строку и сделала ее текущей. С помощью функции ListSetFieldData мы помещаем значение tCountry.m_id_Country в ключевое поле списка (-1 – текущая строка, 0 колонка). Здесь нужно вспомнить, что в настройках колонок списка мы задали, что нулевая колонка соответствует ключевому полю запроса.

Теперь, когда ключевое поле нашей новой строки задано, мы можем заставить список загрузить из БД эту строку. Для этого служит функция ListRefreshRow. Она исполнит запрос списка, подставив в него ограничение по значению ключевого поля через секцию WHERE. Т.е. в нашем случае она исполнит запрос

```
SELECT id_Country, Name, PopulationSize
FROM Countries
WHERE id_Country = НашеЗначение
```

Полученные значения она положит в поля указанной строки списка (по умолчанию – текущая строка).

Есть и другие способы обновить данные в строке списка. Например, с помощью функции ListSetFieldData положить их в соответствующие поля самому. А можно просто перечитать список полностью с помощью функции ListRequery.

Ну и наконец, рассмотрим переменную bResult. Мы объявили ее в самом начале, и вернули из обработчика ее значение в самом конце. Дело в том, что обработчик OnInsert должен вернуть значение типа BOOL. Если он вернет TRUE, то система будет считать, что все прошло успешно, и не будет предпринимать никаких действий. Если же он вернет FALSE, то система удалит из списка ту новую строку, которую она добавила перед вызовом обработчика OnInsert, и восстановит текущее выделение. Это должно произойти, если пользователь отказался от своих действий, т.е. функция ModalFrame вернула значение, отличное от IDOK.

2.7.3.3 Операция редактирования

Мы уже научили наш список удалять строки и добавлять новые. Теперь реализуем операцию редактирования текущей строки.

Операция редактирования реализуется в обработчике OnEdit. Этот обработчик будет вызван, если пользователь дважды кликнет на строке списка или нажмет клавишу Enter.

Как было сказано выше, мы будем использовать тот же диалог, что и использовали для вставки. Вначале мы вытащим данные из колонок текущей строки, положим их в нашу структуру tCountry и вызовем диалог. Если диалог будет закрыт по кнопке «ОК», то мы выполним UPDATE и положим новые данные в соответствующие колонки списка.

Для начала перейдем в раздел «Свойства» и установим флаг «Редактирование».

Теперь обработчик OnEdit стал доступен. Реализуем его.

```
{
        int             nCurRow = ListGetCurRow ();
        TCountry        tCountry;

        ListGetFieldData (tCountry.m_id_Country,        nCurRow, 0);
        ListGetFieldData (tCountry.m_strName,           nCurRow, 1);
        ListGetFieldData (tCountry.m_nPopulation,       nCurRow, 2);

        if (ModalFrame (DIALOG DlgCountry (tCountry), NO_SIZING) == IDOK)
        {
                UPDATE Countries SET
                        Name            = :&(tCountry.m_strName),
                        PopulationSize  = :(tCountry.m_nPopulation)
                WHERE id_Country = :(tCountry.m_id_Country);

                ListSetFieldData (tCountry.m_strName,           nCurRow, 1);
                ListSetFieldData (tCountry.m_nPopulation,       nCurRow, 2);
        }

        return FALSE;
}
```

С помощью функции ListGetCurRow мы получили номер текущей строки. Дальнейший алгоритм достаточно прозрачен. Полагаю, что он не требует дополнительных комментариев, с учетом того, что мы уже знаем. Следует только пояснить, почему мы возвращаем FALSE.

Если обработчик OnEdit возвращает FALSE, система не выполняет никаких действий. Если обработчик возвращает TRUE, система перечитывает из БД текущую строку списка. Для этого исполняется запрос списка с подстановкой в него ограничения, построенного по ключевым полям.

В нашем случае мы сами обновили данные в строке списка с помощью функции ListSetFieldData, поэтому нет нужды заставлять систему перечитывать текущую строку. Но мы могли бы поступить иначе. Самое главное для нас – это положить данные в базу. Система может и сама перечитать строку.

В этом случае код был бы таким:

```
{
	…

	if (ModalFrame (DIALOG DlgCountry (tCountry), NO_SIZING) == IDOK)
	{
		UPDATE Countries SET
			Name		= :&(tCountry.m_strName),
			PopulationSize	= :(tCountry.m_nPopulation)
		WHERE id_Country = :id_Country;
	}

	return TRUE;
}
```

Здесь только не следует забывать, что в настройках колонок списка обязательно должны быть заданы ключевые поля, иначе у системы не будет возможности идентифицировать строку, чтобы корректно ее перечитать.

2.8 Связанные списки

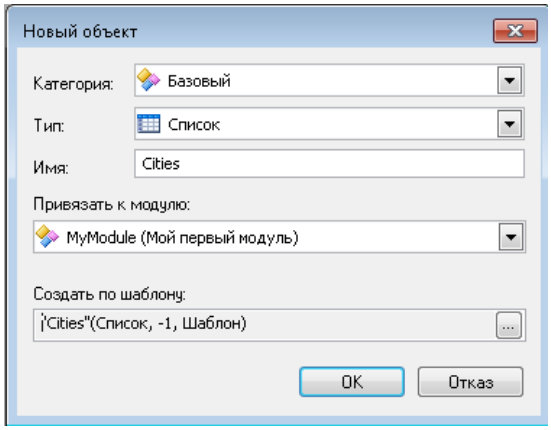
При создании приложений довольно часто возникает задача отображения связанных списков, когда один является ведущим, а другой (или другие) – ведомыми. При движении по ведущему списку ведомые списки должны обновляться. В нашем случае мы продемонстрируем это на списке стран и списке городов. При движении по списку стран список городов будет обновляться, отображая только города выбранной страны.

Для этого нам потребуется список городов и диалог, который будет служить той формой, на которой мы разместим оба этих списка.

Поскольку мы уже умеем создавать списки и диалоги, то не будем тратить время на их создание с нуля, а используем готовые шаблоны.

2.8.1 Список городов

Начнем с создания списка городов. Создадим его по готовому шаблону и назовем Cities.

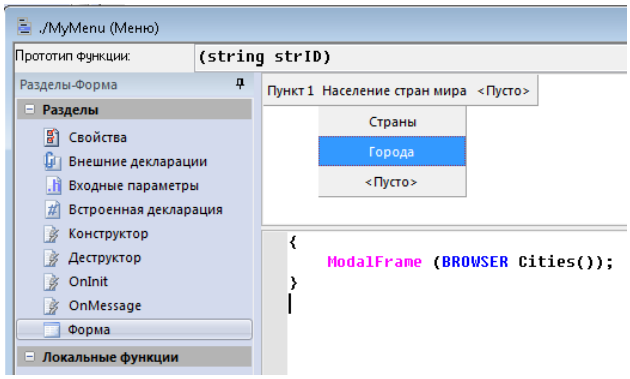


Здесь стоит вспомнить, что шаблоны не хранят заголовок окна, поэтому в разделе «Свойства» введем заголовок «Города» и сохраним список в БД.

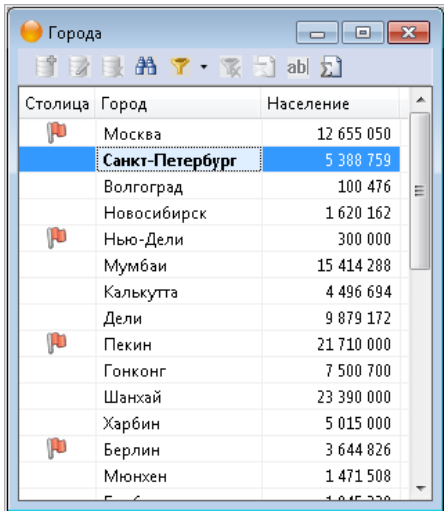
Мы можем увидеть, что декларация «MyModule» уже включена в наш список. Она потребуется нам в дальнейшем, когда мы будем связывать два списка.

Чтобы запустить список на исполнение, добавим в наше меню новый пункт «Города» и напишем обработчик

```
{
        ModalFrame (BROWSER Cities());
}
```



Сохраним меню, запустим модуль и посмотрим, как выглядит наш список городов.



Города

Столица	Город	Население
	Москва	12 655 050
	Санкт-Петербург	5 388 759
	Волгоград	100 476
	Новосибирск	1 620 162
	Нью-Дели	300 000
	Мумбаи	15 414 288
	Калькутта	4 496 694
	Дели	9 879 172
	Пекин	21 710 000
	Гонконг	7 500 700
	Шанхай	23 390 000
	Харбин	5 015 000
	Берлин	3 644 826
	Мюнхен	1 471 508

В этом списке нет ничего примечательного за исключением одной особенности. Мы видим, что в колонке «Столица» напротив столиц отображается пиктограмма флажка. Рассмотрим, как это сделано.

Запрос списка имеет такой вид:

```
SELECT
        CASE Capital WHEN 1 THEN 'flag_16' ELSE '' END AS IconCapital,
        Name,
        PopulationSize
FROM Cities
:m_strWHERE
```

Первое поле запроса возвращает нам имя пиктограммы из таблицы x2Image, если поле Capital таблицы Cities равно 1, или пустую строку в противном случае. А в настройках колонок списка модификатор для этой колонки выставлен в положение «Пиктограмма».

	Заголовок	Модификатор	Флаги	Формат вывода	Формат ввода	Имя переменной	Ширина
0	Столица	Пиктограмма	0				55
1	Город	Данные	0				120
2	Население	Данные	16	r			100

В этом случае система рассматривает значение этого поля не как данные, а как имя пиктограммы, и отображает саму пиктограмму. В случае пустой строки пиктограмма отображаться не будет.

С таблицей x2Image мы уже сталкивались, когда задавали иконку для модуля. Она является хранилищем всех пиктограмм, которые можно использовать в приложении. Например, пиктограмму можно разместить на кнопке или задать в качестве фона диалога.

Обратим внимание, что в запросе это поле поименовано явным образом

```
… AS IconCapital
```

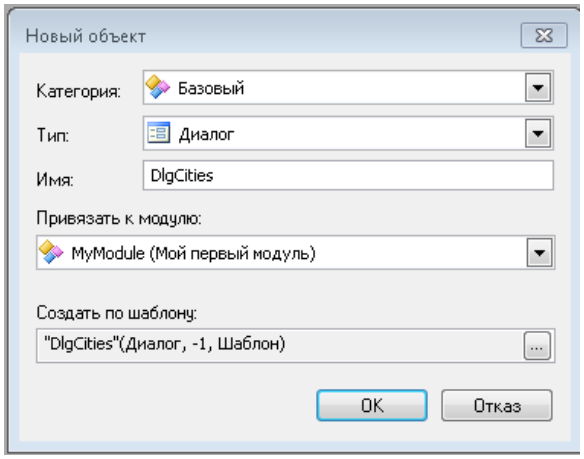
Дело в том, что с точки зрения SQL-сервера это поле является вычисляемым, поэтому сам сервер не дает ему имени автоматически. Но система не сможет построить список, если в его запросе есть безымянные поля, поэтому для всех вычисляемых полей следует задавать имена явно.

И наконец, мы видим, что в запросе выполняется подстановка переменной :m_strWHERE. Эта переменная объявлена во встроенной декларации списка. Он будет задавать нам

фильтр, когда мы будем связывать список городов со списком стран. Мы остановимся на ней чуть позже.

2.8.2 Диалог для связанных списков

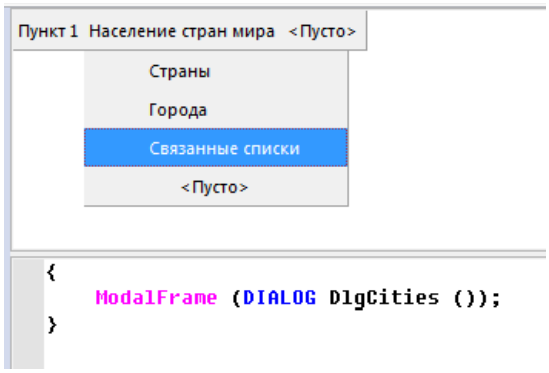
Далее нам потребуется диалог, на котором мы разместим два связанных списка. Как и в прошлый раз, создадим его, используя готовый шаблон и назовем «DlgCities».



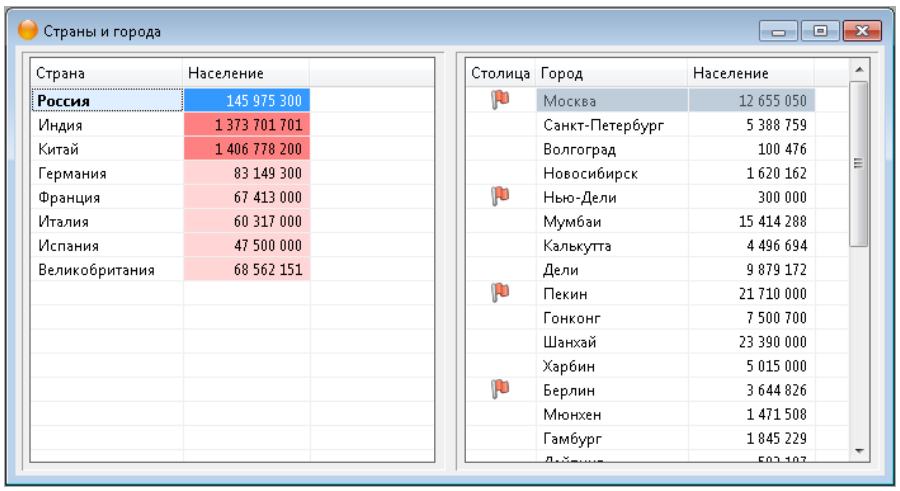
Перейдем в раздел свойств, введем заголовок окна «Страны и города» и сохраним объект в базу данных.

Чтобы вызвать диалог на исполнение, добавим в меню новый пункт «Связанные списки» и напишем его обработчик:

```
{
        ModalFrame (DIALOG DlgCities ());
}
```

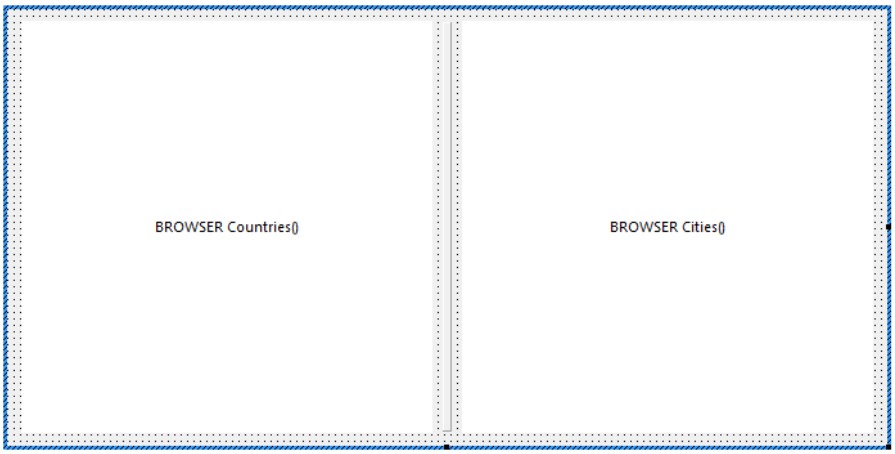


Сохраним меню, запустим модуль и вызовем наш диалог.



Так выглядит наш диалог. На нем расположены два списка, которые пока еще не связаны между собой. Они разделены сплиттером, вертикальной линией, которую можно перемещать мышкой. Если диалог растянуть, то сплиттер можно будет перемещать, и вместе с ним будут меняться размеры списков.

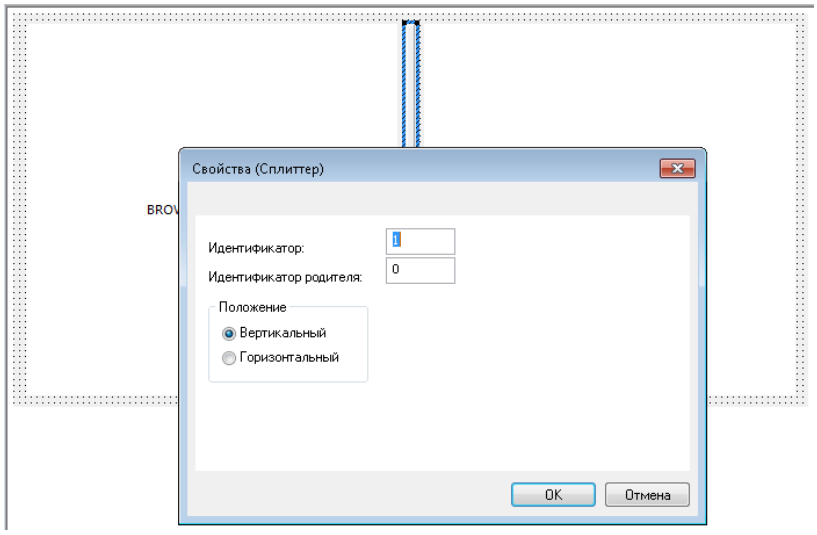
Посмотрим, как устроен наш диалог. Перейдем в раздел «Форма».



Здесь мы видим три элемента: два списка и сплиттер между ними.

Немного о сплиттерах.

Сплиттер может быть горизонтальным или вертикальным. Он делит окно диалога на две панели. Когда пользователь перемещает его мышкой, то одна панель расширяется, а другая сужается. На любой из панелей можно разместить еще один сплиттер, и тогда он разделит эту панель еще на две панели. Таким образом, сплиттеров в окне диалога может быть сколько угодно. Важно лишь правильно задать их иерархию. Это делается с помощью идентификаторов. Если войти в свойства сплиттера (двойной клик мыши), то мы увидим, что у него есть собственный идентификатор и идентификатор родителя. Идентификатор – это просто целое число. У сплиттера верхнего уровня собственный идентификатор равен 1, а идентификатор родителя – 0. Если мы разместим на одной из панелей новый сплиттер, то его идентификатор будет равен 2, а идентификатор родителя – 1, и т.д. Идентификаторы разработчик задает сам, как ему угодно, важно лишь соблюсти иерархию, иначе диалог отобразится неправильно.



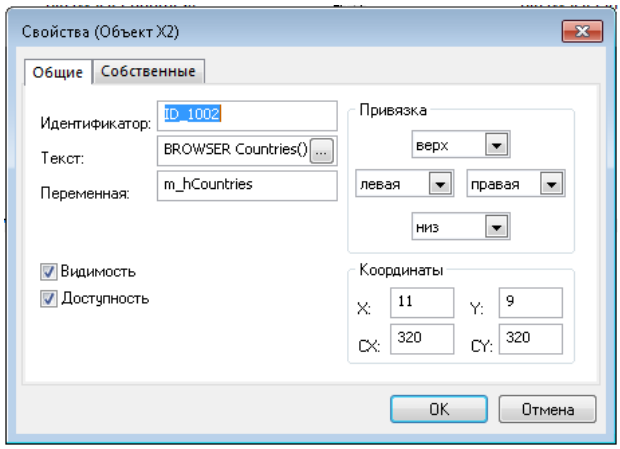
Управляющие элементы, размещенные на панелях, имеют настройку, задающую способ привязки границ элемента к границам панели. Это определяет то, как будет вести себя элемент при изменении размеров панели, т.е. будет ли он растягиваться и сужаться вместе с панелью, или же он будет перемещаться вместе с одной из границ панели.

Если диалог не имеет ни одного сплиттера, то панелью будет считаться окно самого диалога.

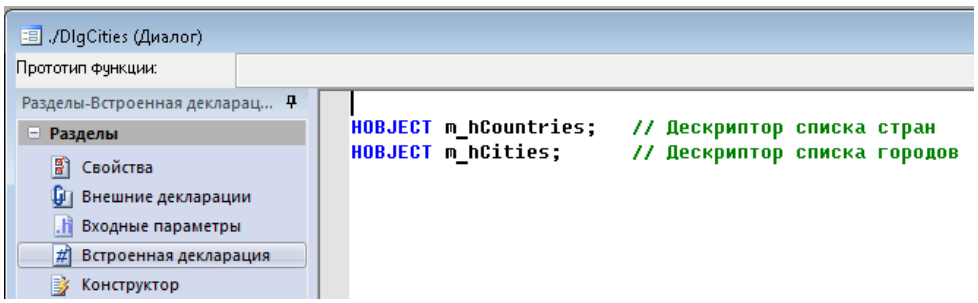
А сейчас перейдем к спискам.

Внутри диалога размещаются управляющие элементы. Это могут быть кнопки, редакторы, флажки, поля, содержащие статический текст и т.д. В том числе, в качестве управляющего элемента может выступать любой визуальный объект, например, список, диаграмма или даже другой диалог. В нашем случае мы разместили на диалоге два списка.

Дважды кликнем на левом списке, чтобы посмотреть его свойства.



В поле «Текст» мы видим «BROWSER Countries()». Это и задает, какой объект X2 будет отображен в качестве управляющего элемента. Именно этот оператор отработает при начальной загрузке диалога. Если бы наш список «Countries» принимал входные параметры, то здесь мы могли бы их передать. Как мы уже знаем, оператор «BROWSER» возвращает дескриптор экземпляра объекта. Этот дескриптор и попадет в буферную переменную m_hCountries. Она объявлена во встроенной декларации диалога.



Как мы уже знаем, переменные, объявленные во встроенной декларации, являются локальными для объекта. Они доступны везде внутри объекта, но недоступны снаружи. Фактически, они являются членами данных объекта.

Вообще-то задавать буферную переменную для управляющего элемента не обязательно. Но нам потребуются дескрипторы двух наших списков для того, чтобы сделать их связанными.

2.8.3 Связываем списки

Итак, у нас есть диалог с двумя списками и их дескрипторы лежат в переменных m_hCountries и m_hCities. Чтобы связать два списка, мы будем использовать механизм сообщений.

Как работает механизм сообщений.

С одной стороны, есть встроенная функция SendMessage, которая служит для отправки сообщения объекту, с другой стороны, каждый объект имеет обработчик OnMessage, который система вызовет в качестве реакции на сообщение.

Функция SendMessage объявлена следующим образом:

```
int SendMessage (HOBJECT hObj, int nMsg, & Param)
```

Здесь hObj – это дескриптор того объекта, которому мы направляем сообщение. Параметр nMsg определяет номер сообщения. Это просто целое число, которое разработчик задает самостоятельно. Оно может быть любым. Важно, чтобы эти числа не дублировались, поэтому сообщения удобно задавать в виде констант во внешней декларации. Третий параметр Param служит для передачи дополнительной информации. Как мы видим, он является ссылочным, и его тип не задан. Это значит, что через этот параметр можно передать любой тип, в том числе структуру. Поскольку параметр передается по ссылке, то обработчик OnMessage может изменить его значение, и вызывающий код это увидит.

Обработчик OnMessage объявлен так:

```
int OnMessage (int nMsg, & Param)
```

Он принимает номер сообщения и тот самый параметр Param, который был отправлен функцией SendMessage. Обработчик возвращает тип int. Это то значение, которое вернет функция SendMessage.

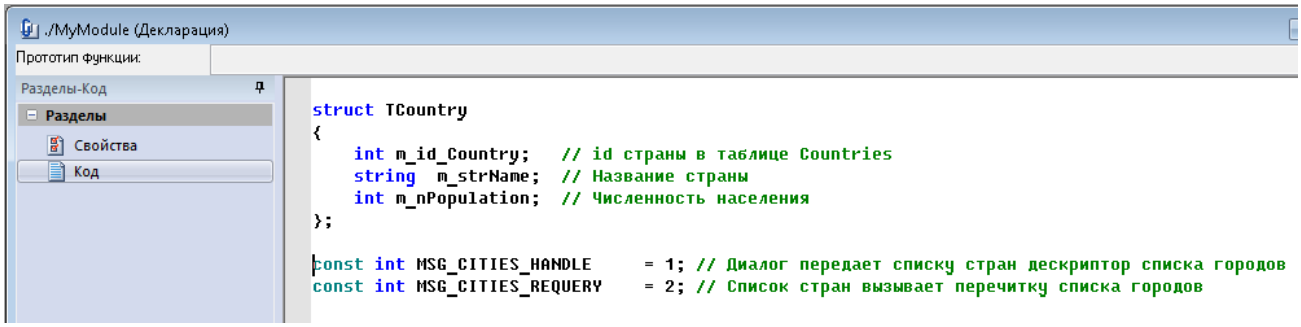
С помощью сообщений объекты могу обмениваться друг с другом произвольной информацией.

Список стран при изменении текущей строки должен сообщить списку городов, что ключ страны id_Country изменился, а список городов должен сформировать новый запрос с ограничением по id_Country и обновить свое содержимое.

Список стран может послать сообщение списку городов, но для этого он должен знать его дескриптор. Оба дескриптора, как мы знаем, хранятся в диалоге. Соответственно, диалог должен передать списку стран дескриптор списка городов.

Отправимся в нашу декларацию и объявим в ней две константы, которые будут служить нам сообщениями.

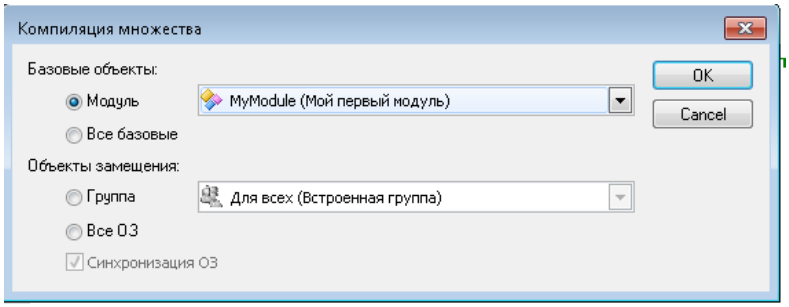
```
const int MSG_CITIES_HANDLE = 1;  // Диалог передает списку стран дескриптор списка городов
const int MSG_CITIES_REQUERY = 2; // Список стран вызывает перечитку списка городов
```



Сохраним декларацию и перекомпилируем наш модуль.

Когда меняется декларация, все объекты, в которые она включена должны быть перекомпилированы, иначе возникнет ошибка периода исполнения.

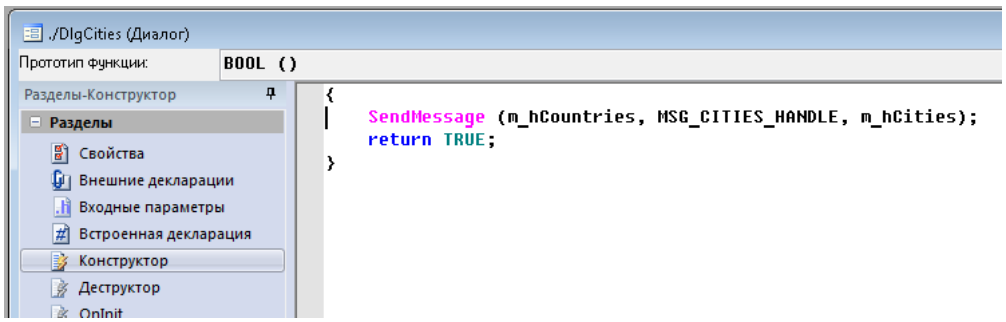
В сущности, перекомпиляция всего модуля не обязательна, достаточно было бы перекомпилировать только те объекты, в которые включена измененная декларация. Но перекомпиляцию всего модуля вызвать проще, чем компилировать каждый объект в отдельности. По сочетанию клавиш «Ctrl+Alt+F7» вызовем диалог компиляции (в главном меню: Разработка \ Компиляция множества).



Здесь нам пригодится тот факт, что все наши объекты привязаны к модулю. Если Вы не уверены, что при создании объектов всегда задавали привязку к модулю, то установите переключатель в положение «Все базовые». Тогда будут перекомпилированы все объекты.

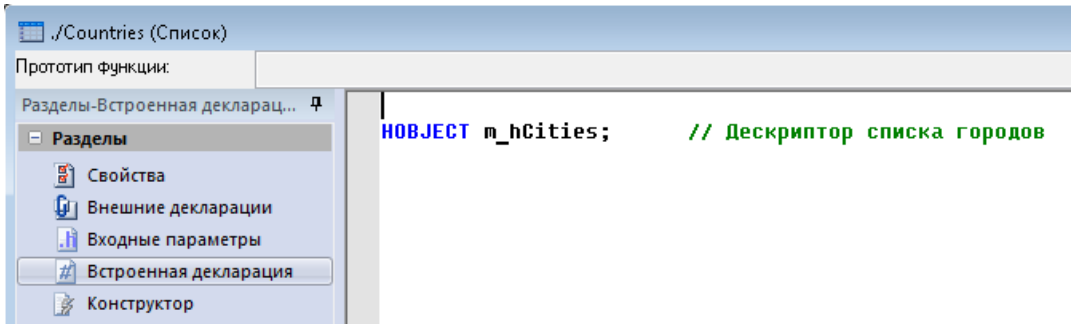
После компиляции изменения в декларации вступили в силу.

Теперь реализуем обмен сообщениями. Для начала войдем в конструктор диалога DlgCities и раскомментируем строку кода с вызовом SendMessage. Конструктор отрабатывает при создании объекта, так что диалог сообщит списку стран дескриптор писка городов, как только диалог будет создан.



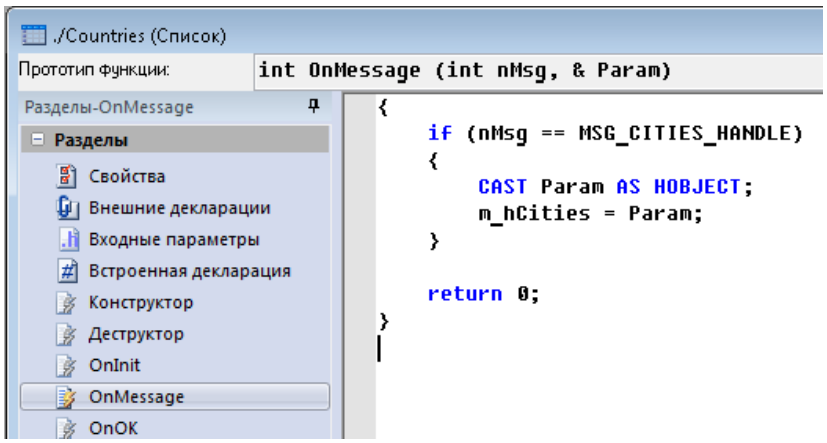
Сохраним изменения и перейдем в список «Countries». Здесь мы должны реализовать реакцию на сообщение. Фактически, нам нужно просто сохранить переданный дескриптор в какой-то внутренней переменной. Для этого перейдем в раздел «Встроенная декларация» и объявим переменную

```
HOBJECT m_hCities;          // Дескриптор списка городов
```



Теперь реализуем обработчик OnMessage:

```
{
        if (nMsg == MSG_CITIES_HANDLE)
        {
                CAST Param AS HOBJECT;
                m_hCities = Param;
        }

        return 0;
}
```



Здесь нужно заострить внимание на одной детали. Поскольку параметр Param является бестиповой ссылкой, то компилятор никак не может знать заранее, какой именно тип будет передан вызывающим кодом. Мы должны сообщить ему об этом, чтобы он знал, как работать с этой прерменой. Для этого служит оператор CAST. В нашем случае мы говорим компилятору, что Param имеет тип HOBJECT.

Оператор CAST работает только с параметрами, переданными по ссылке без указания типа. Его нельзя применить к переменной, тип которой задан статически. Например, такой код даст ошибку.

```
int i;
CAST i AS string;
```

Еще следует отметить, что область действия оператора CAST – блок, т.е. фрагмент кода, окруженный фигурными скобками. При выходе из блока параметр теряет свой тип. Это значит, что в разных блоках мы можем приводить параметр к разным типам. Но в одном блоке нельзя использовать CAST более одного раза. Принцип очень прост: если тип не задан, то его можно задать, но, если он уже задан, то изменить его нельзя.

```
if (nMsg == 1)
{
        CAST Param AS int;
}
else if (nMsg == 2)
{
        CAST Param AS string;

        CAST Param AS string; // Ошибка.
        // Нельзя использовать CAST дважды в одном блоке
}
```

После того, как наш ведущий список получил дескриптор ведомого, он будет слать ведомому списку сообщения при каждом изменении текущей строки. На изменение текущей строки срабатывает обработчик OnSelChange. Он объявлен так:

```
OnSelChange (BOOL bRowChanged, BOOL bColChanged)
```

На самом деле он срабатывает не только на изменение текущей строки, но и на изменение текущей колонки, а также, на изменение количества выделенных строк для списков с множественным выделением. Что именно произошло в списке, и по какой причине вызван обработчик, мы можем понять по значению параметров bRowChanged и bColChanged. Нас интересует только ситуация, когда изменилась текущая строка, т.е. bRowChanged != 0.

Далее мы могли бы с помощью функции ListGetFieldData запросить у списка значение ключевого поля текущей строки и отправить сообщение списку городов. Но мы воспользуемся другим, более простым механизмом. Мы уже сталкивались с понятием буферных переменных, когда работали с диалогом. В списке тоже можно использовать буферные переменные. Это самые обычные локальные переменные, объявленные во встроенной декларации, только мы можем путем настроек сязать их с конкретными колонками списка. Тогда они будут содержать значения соотетствующих полей текущей строки, т.е. система будет автоматически присваивать им эти значения при движении по списку. Этим мы и воспользуемся. Для этого мы объявим во встроенной декларации еще одну переменную

```
int m_id_Country;        // Буферная переменная, связанная с колонкой, содержащей ключ
```

./Countries (Список)

Прототип функции:

Разделы-Встроенная деклараци...

Разделы
- Свойства
- Внешние декларации
- Входные параметры
- Встроенная декларация
- Конструктор

```
HOBJECT m_hCities;      // Дескриптор списка городов
int     m_id_Country;   // Буферная переменная, связанная с колонкой, содержащей ключ
```

После этого мы должны связать нашу переменную с ключевым полем. Перейдем в настройки колонк и сделаем это.

	Заголовок	Модификатор	Флаги	Формат вывода	Формат ввода	Имя переменной	Ширина
0		Данные	3			m_id_Country	100
1	Страна	Данные	0				120
2	Население	Данные	16	r			100

А теперь перейдем в обработчик OnSelChange и введем код:

```
{
        if (m_hCities && bRowChanged)
        {
                SendMessage (m_hCities, MSG_CITIES_REQUERY, m_id_Country);
        }
}
```

./Countries (Список)

Прототип функции: `OnSelChange (BOOL bRowChanged, BOOL bColChanged)`

Разделы-OnSelChange
- Входные параметры
- Встроенная декларация
- Конструктор
- Деструктор
- OnInit
- OnMessage
- OnOK
- OnCancel
- OnActivate
- SQL запрос
- OnSelChange
- OnCheck

```
{
    if (m_hCities && bRowChanged)
    {
        SendMessage (m_hCities, MSG_CITIES_REQUERY, m_id_Country);
    }
}
```

Здесь мы отправляем сообщение MSG_CITIES_REQUERY списку городов, дескриптор которого хранится в переменной m_hCities, а в качестве параметра передаем значение ключа текущей строки m_id_Country. Сохраним наши изменения в БД.

Сейчас нам нужно обработать это сообщение в списке Cities. Откром его, перейдем в обработчик OnMessage и раскомментируем его код.

Чтобы снять комментарии можно выделить закомментированный код и нажать кнопку  *в линейке инструментов.*

./Cities (Список)

Прототип функции: `int OnMessage (int nMsg, & Param)`

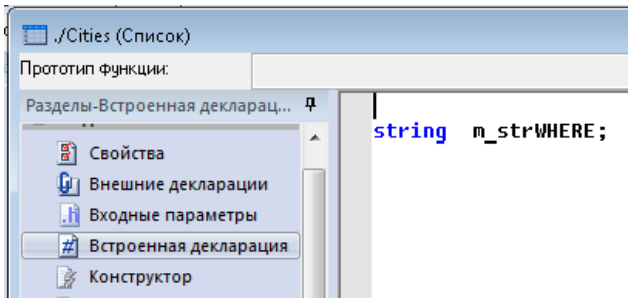
Разделы-OnMessage
- Входные параметры
- Встроенная декларация
- Конструктор
- Деструктор
- OnInit
- OnMessage
- OnOK
- OnCancel
- OnActivate

```
{
    if (nMsg == MSG_CITIES_REQUERY)
    {
        CAST Param AS int;

        m_strWHERE = "WHERE id_Country = " + ToString (Param);
        ListRequery ();
    }

    return 0;
}
```

Здесь мы формируем строку ограничения m_strWHERE для запроса списка и перечитываем его с помощью функции ListRequery. Вспомним, как выглядит запрос списка.

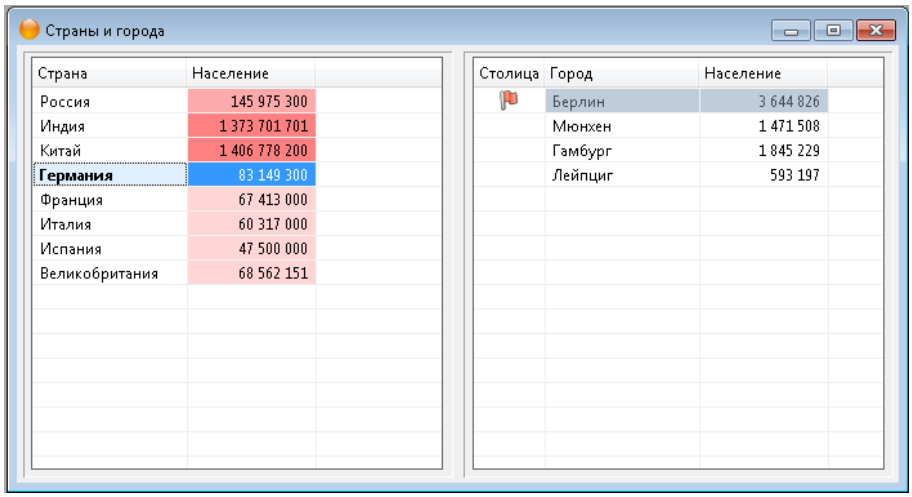
```
SELECT
        CASE Capital WHEN 1 THEN 'flag_16' ELSE '' END AS IconCapital,
        Name,
        PopulationSize
FROM Cities
:m_strWHERE
```

Переменная m_strWHERE объявлена во встроенной декларации списка



Изначально она пуста, поэтому ее подстановка не меняет запрос, и при вызове списка городов через меню мы видим в нем все города. Но, если мы получили сообщение от ведущего списка, то строка будет сформирована и ограничит возвращаемое множество.

Сохраним изменения в базу и запустим модуль.



Теперь при движении по ведущему списку ведомый перечитывается нужным образом.

Мы реализовали связь между списками. Вообще-то, это не единственный способ, как можно было бы связать объекты. Например, можно было бы передать ведомому списку ключ через входной параметр, а можно было бы вообще формировать запрос ведомого

списка на стороне ведущего… Здесь все зависит от нашей фантазии. Да и вообще, сделать связанными можно любые объекты, не только списки. Важно понимать сам принцип:

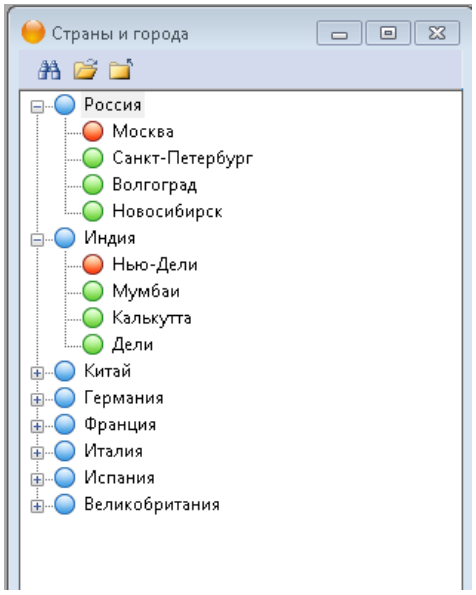
1. Ведущий объект должен уметь ловить интересующие события, для этого нужно задействовать его подходщий обработчик
2. Ведущий объект должен иметь дескриптор ведомого, чтобы отправлять ему сообщения
3. Ведущий объект должен передавать ведомому достаочно информации, чтобы тот смог правильно изменить свое состояние

Вот и вся премудрость. Другие варианты реализации связанных объектов можно найти в демонстрационном модуле. В примере «Списки \ Подмена запроса» с помощью переключателя на прилипающей панели можно менять содержимое списка. В примере «Прочие объекты \ Диаграммы \ Примеры диаграмм» через панель настроек можно менять вид диаграммы. А в примере «Прочие объекты \ Блок-схемы \ Редактор» связь между объектами работает в обе стороны. Через панель настроек можно менять состояние фигур в блок-схеме, а в зависитмоти от типа фигуры, выделенной в редакторе, меняется сама панель настроек.

2.9 Дерево

Дерево довольно часто используется в интерфейсах. Здесь мы попробуем построить дерево средствами X2.

На первом уровне мы будем отображать страны, на втором – города. И еще, каждому узлу дерева мы присвоим соответствующую пиктограмму. Выглядит это так:



Прежде, чем мы приступим к реализации алгоритмов, давайте разберемся, как вообще строится дерево в X2.

Каждый узел дерева содержит:

- *собственный идентификатор*
- *идентификатор родителя*
- *наименование узла, т.е. текст, который будет отображаться*
- *две пиктограммы для свернутого и развернутого состояния (необязательные параметры)*
- *произвольные данные, которые могут потребоваться для построения дочерних узлов (необязательный параметр)*

Для добавления узла служит функция TreeAddItem. Она принимает на входе все данные, которые будет содержать новый узел, и возвращает его идентификатор. Идентификатор представляет собой целое число (тип int), уникальное в рамках дерева. Система автоматически генерирует идентификаторы узлов, обеспечивая их уникальность. Если идентификатор родительского узла равен нулю, то функция добавит узел верхнего уровня, т.е. не имеющий родителя. Если он отличен от нуля, то функция добавит дочерний узел. Стоит, также, отметить, что после добавления узла его параметры можно изменить.

Дерево строится в обработчике OnExpand. Он объявлен так:

```
OnExpand (int nItem, BOOL bExpand)
```

Обработчик OnExpand вызывается при разворачивании или сворачивании узла, и его параметр nItem и есть идентификатор узла, меняющего свое состояние, а параметр bExpand определяет само действие. Если bExpand – TRUE, то узел разворачивается, если FALSE – сворачивается.

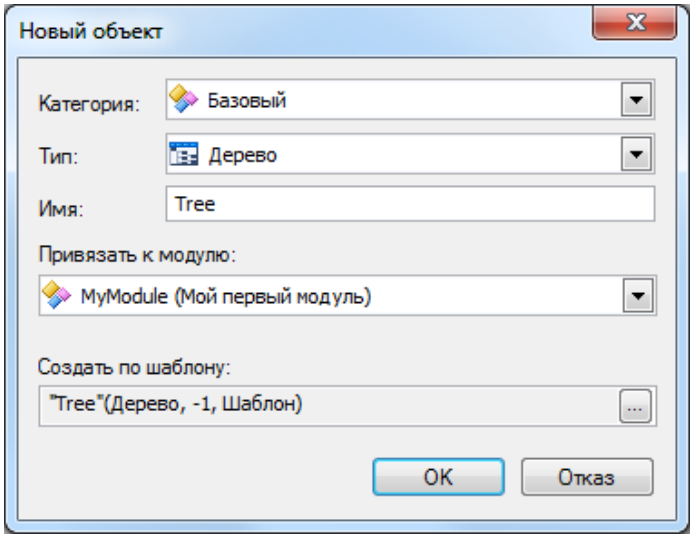
В первый раз система вызывает этот обработчик при визуализации объекта. Поскольку дерево еще не содержит ни одного узла, то nItem == 0. В этом случае мы должны построить верхний уровень дерева. Далее, при разворачивании каждого узла nItem будет содержать идентификатор этого узла, и мы должны построить для него нижележащий уровень. Если обработчик OnExpand при разворачивании узла не добавит для него ни одного дочернего, то этот узел будет считаться листевым, т.е. не будет иметь крестика для разворачивания.

Здесь следует учесть одну деталь. Обработчик OnExpand вызывается только при разворачивании тех узлов, которые не имеют дочерних. Если мы однажды построили нижележащий уровень для какого-то узла, то при повторном его разворачивании обработчик вызываться не будет. Если же мы хотим, чтобы обработчик вызывался каждый раз, то при сворачивании узла (параметр bExpand == FALSE) мы должны удалить его детей. Это можно сделать с помощью функции TreeDropChildren.

Таким образом, алгоритм обработчика OnExpand для построения дерева сводится к следующим операциям:

- *При инициализации, т.е. когда nItem == 0, строим верхний уровень*
- *При разворачивании узла строим его нижележащий уровень*
    - *Сам узел должен содержать достаточно информации для этого*
    - *Если ничего не построили, то этот узел становится листевым*
- *При сворачивании узла, либо удаляем всех его детей, либо ничего не делаем*

Создадим наше дерево, используя готовый шаблон «Tree».



Напомню, что шаблоны не хранят заголовок окна, поэтому перейдем в раздел свойств и введем имя «Страны и города».

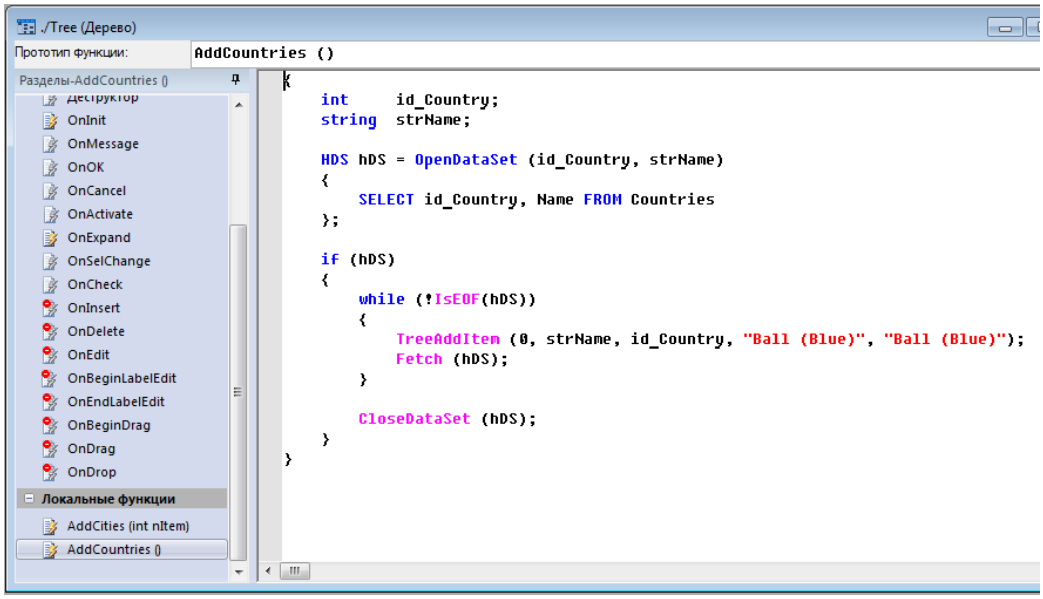
Оконные параметры	
Заголовок	Страны и города
Ширина	300
Высота	600

Сохраним объект в базу данных и посмотрим, как он устроен. Прежде всего, войдем в обработчик OnExpand.

```
{
        if (!bExpand)
                return;

        if (nItem)
                AddCities (nItem);
        else
                AddCountries ();
}
```

Когда узел сворачивается (bExpand == 0), то мы ничего не делаем.

Когда он разворачивается и nItem == 0, строим верхний уровень, т.е. добавляем страны. Если же nItem != 0, то добавляем города для конкретной страны. Функции AddCountries и AddCities – это локальные функции нашего дерева. Посмотрим, как они устроены.




```
AddCountries ()
{
        int     id_Country;
        string  strName;

        HDS hDS = OpenDataSet (id_Country, strName)
        {
                SELECT id_Country, Name FROM Countries
        };

        if (hDS)
        {
                while (!IsEOF(hDS))
                {
                        TreeAddItem (0, strName, id_Country, "Ball (Blue)", "Ball (Blue)");
                        Fetch (hDS);
                }

                CloseDataSet (hDS);
        }
}
```

Здесь основная часть кода выполняет чтение данных из базы, и только одна функция TreeAddItem работает с деревом, добавляя в него новый узел.

Мы уже знакомы с оператором SQL, который может загружать данные из базы. Но он загружает только одну строку. Оператор OpenDataSet открывает набор данных, состоящий из множества строк. Он принимает в качестве параметров имена переменных, в которые будут помещаться значения соответствующих полей запроса при движении по набору, а сам запрос задается в фигурных скобках. Последовательность переменных и их типы должны соответствовать полям запроса.

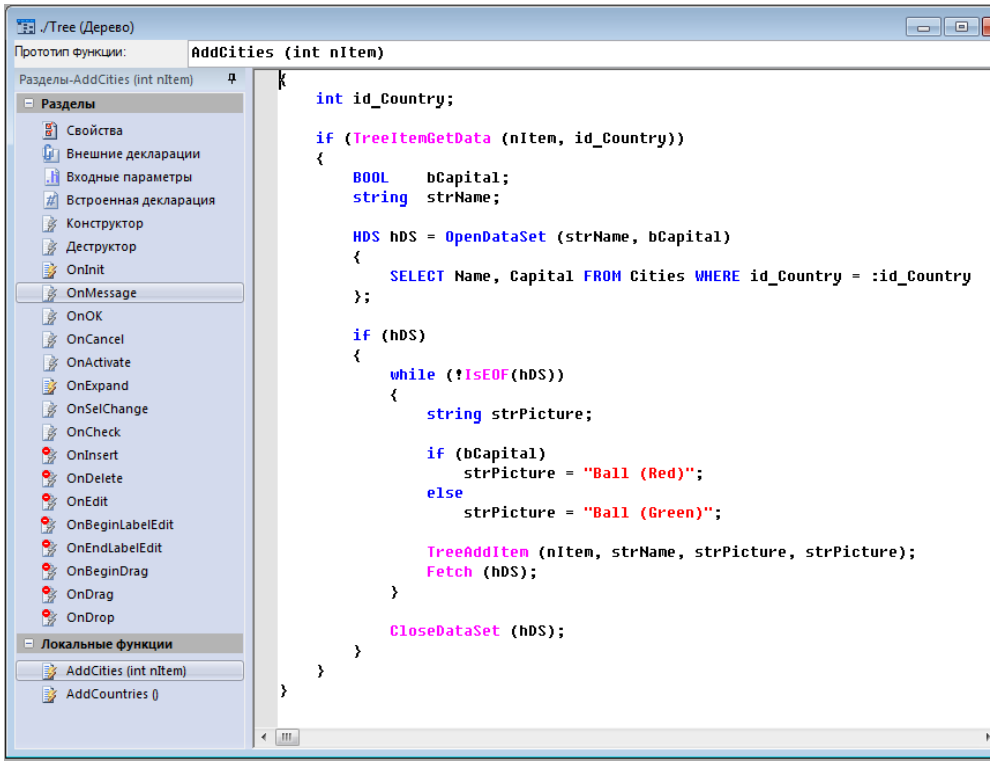
Оператор OpenDataSet возвращает дескриптор набора данных (тип HDS). Далее цикл while выполняет проход по набору данных, пока он не закончится. Функция IsEOF возвращает признак окончания набора, а функция Fetch переходит на следующую строку и загружает значения ее полей в переменные. Когда набор закончился, функция CloseDataSet закрывает его и освобождает ресурсы, выделенные на его обслуживание.

Сама же целевая операция реализуется функцией TreeAddItem. Рассмотрим подробнее, как она объявлена.

```
int TreeAddItem
(
        int       nParent,
        string    strText,
        & data,
        string    strImg1 = “”,
        string    strImg2 = “”,
        HOBJECT hObj = 0
)
```

В качестве идентификатора родительского узла мы передаем 0, поскольку строим верхний уровень. В качестве наименования узла передаем strName, где хранится наименование страны, а в качестве произвольных данных, связанных с узлом, передаем id_Country. Это нам понадобится, когда мы будем строить нижележащий уровень. И, наконец, два последних параметра – это имена пиктограмм узла для развернутого и свернутого состояния узла. В нашем случае для обоих состояний используется одна и та же пиктограмма.

Теперь рассмотрим функцию AddCities.




```
AddCities (int nItem)
{
        int id_Country;

        if (TreeItemGetData (nItem, id_Country))
        {
                BOOL    bCapital;
                string  strName;

                HDS hDS = OpenDataSet (strName, bCapital)
                {
                        SELECT Name, Capital FROM Cities WHERE id_Country = :id_Country
                };

                if (hDS)
                {
                        while (!IsEOF(hDS))
                        {
                                string strPicture;

                                if (bCapital)
                                        strPicture = "Ball (Red)";
                                else
                                        strPicture = "Ball (Green)";

                                TreeAddItem (nItem, strName, strPicture, strPicture);
                                Fetch (hDS);
                        }

                        CloseDataSet (hDS);
                }
        }
}
```

Она выполняет аналогичные действия, только добавляет нижележащий уровень при разворачивании узла, поэтому она принимает на входе идентификатор узла родителя.

Вначале с помощью функции TreeItemGetData мы пытаемся получить данные, связанные с узлом. Если мы получили эти данные, т.е. функция вернула TRUE, то узел nItem соответствует стране, а данные являются ключом id_Country в таблице Cities. Именно его мы связали с узлом при добавлении стран в дерево. Если же этих данных нет, то разворачиваемый узел соответствует городу, и наша функция ничего не должна делать.

Весь последующий код делает то же самое, что и код функции AddCountries. Отметим только три отличия. В SELECT мы подставляем id_Country, строя ограничение по стране. Второе: при добавлении узла в функцию TreeAddItem мы не передаем никаких дополнительных данных. Это и гарантирует нам, что при разворачивании узла, соответствующего городу, мы не пройдем проверку

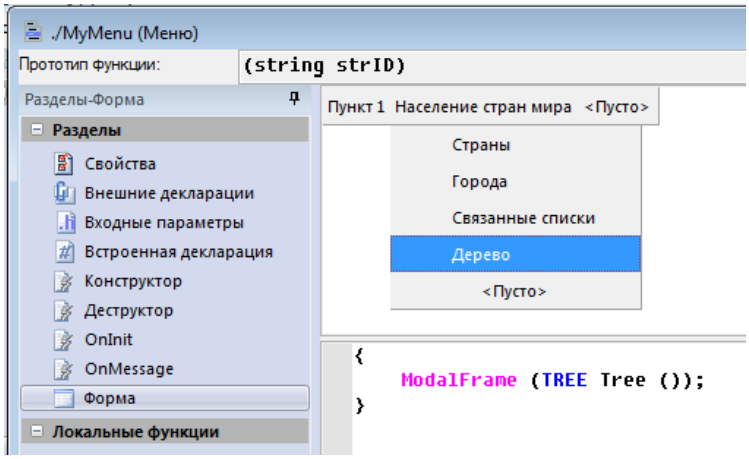
```
if (TreeItemGetData (nItem, id_Country))
```

и выйдем из функции, не добавив ни одного дочернего узла. Следовательно, узел города при попытке разворачивания станет листевым.

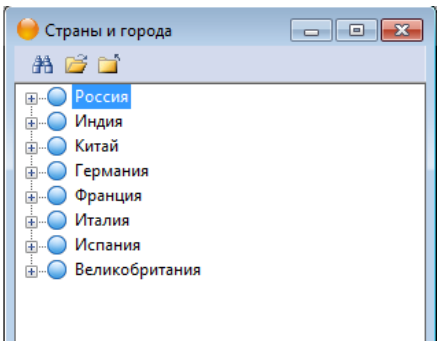
И третье в том, что мы подставляем имя пиктограммы в зависимости от условия, является ли город столицей.

Теперь перейдем в меню и добавим в него новый пункт «Дерево».

```
{
        ModalFrame (TREE Tree ());
}
```



Сохраним изменения, вызовем модуль и посмотрим, как выглядит дерево.



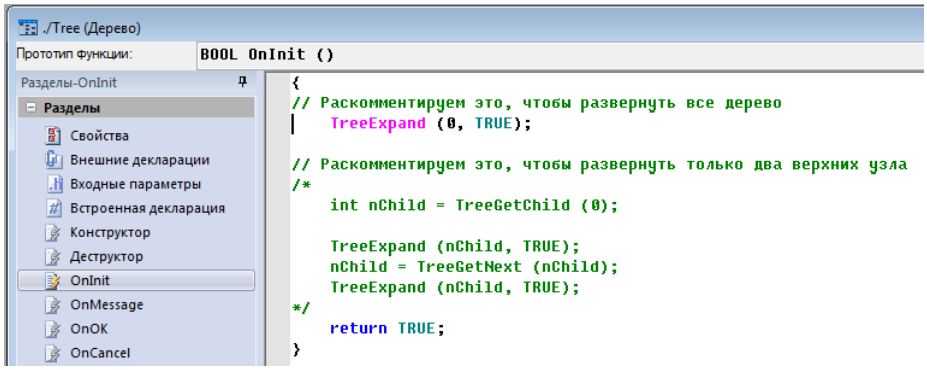
Мы видим, что построен только верхний уровень. Если мы хотим, чтобы дерево разворачивалось при начальном построении, то мы можем сделать это в обработчике OnInit.

Мы уже знаем, что при создании экземпляра объекта отрабатывает конструктор. Но конструктор отрабатывает до визуализации объекта, поэтому в конструкторе мы не можем обращаться к визуальным аспектам. Обработчик OnInit отрабатывает после визуализации. В нашем случае последовательность будет такой:

1. Оператор создания экземпляра TREE Tree () вызовет конструктор
2. Функция визуализации ModalFrame создаст окно объекта и
   2.1. Вызовет OnExpand с параметром nItem == 0, что приведет к построению верхнего уровня дерева
   2.2. Вызовет OnInit

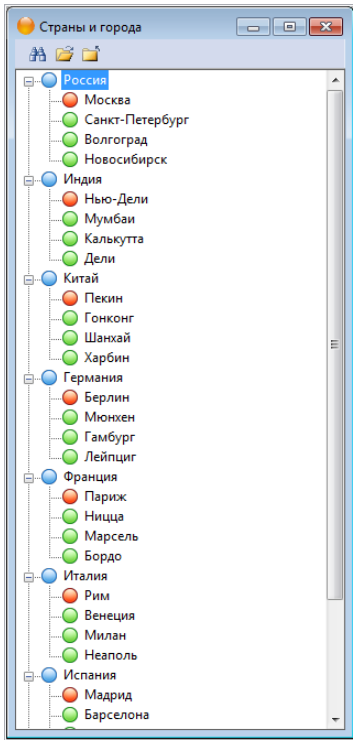
Таким образом, мы попадем в OnInit, когда верхний уровень дерева уже построен. Это означает, что мы можем обращаться к узлам и разворачивать их.

Перейдем в OnInit и раскомментируем код

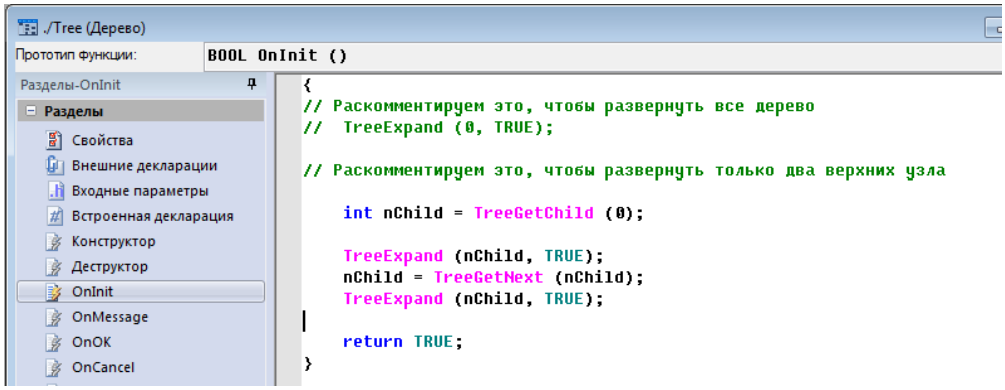


Функция TreeExpand получает первым параметром идентификатор узла, который нужно развернуть, а вторым параметром – признак того, будет ли развернута вся ветка рекурсивно, или же должен быть развернут только один нижележащий уровень.

Вызов TreeExpand (0, TRUE) приведет к разворачиванию всего дерева. Сохраним изменения и посмотрим, как это выглядит.

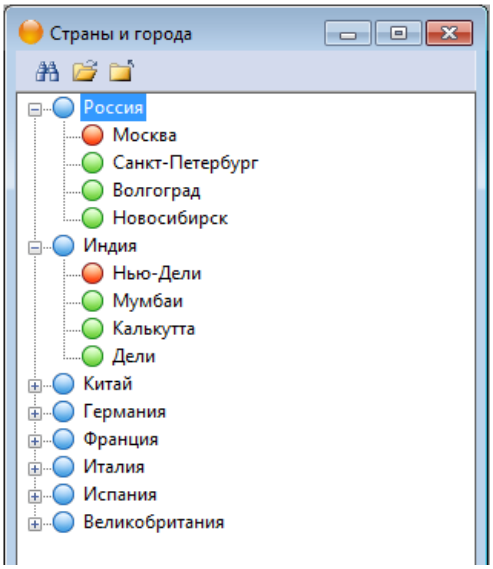


Если мы хотим развернуть только какие-то конкретные узлы, то сделаем следующее.



Функция TreeGetChild принимает идентификатор родителя и возвращает идентификатор его первого дочернего узла. В нашем случае это будет первый узел верхнего уровня дерева. Функция TreeGetNext принимает идентификатор узла и возвращает идентификатор следующего узла того же уровня. В нашем случае это будет первый узел верхнего уровня.

Сохраним изменения, запустим модуль и посмотрим, как это выглядит.



В заключение отметим, что в дереве можно реализовать операции редактирования, а также, перетаскивания веток мышкой. Пример этого можно найти в демонстрационном модуле «Деревья \ Обработчики». Там же есть примеры, иллюстрирующие работу с флажками в дереве и использования обработчика OnSelChange, который вызывается при изменении текущего узла.

2.10 Диаграмма

Платформа X2 предоставляет широкие возможности для создания графиков и диаграмм. Здесь мы построим простую столбчатую диаграмму, которая будет отображать в виде колонок численность населения различных стран.

Для построения диаграммы служит сегмент OnBuild, именно здесь должен быть реализован алгоритм ее построения. Этот сегмент кода вызывается автоматически после OnInit, но может быть вызван явным образом с помощью функции ChartRebuild.

Диаграмма строится на основе серии точек. Вначале создается серия, потом в нее добавляются точки.

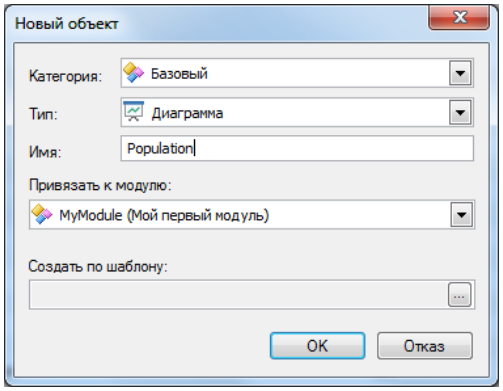
Серия создается с помощью функции ChartCreateSeries и изначально представляет собой пустой массив. Серия имеет категорию и тип. Категория определяет вид диаграммы, например, столбчатая, пузырьковая, тор, пирамида и т.д. Тип определяет то, каким образом будут представлены данные разных серий, независимо друг от друга, общим стеком, стеком с нормализацией и т.д.

Серий можно задать сколько угодно. Обычно, все они имеют одну и ту же категорию, например, график. Но можно совмещать и разные категории, например, график и область. Важно, чтобы категории были логически совместимы.

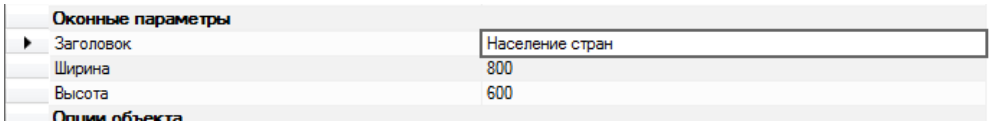
Для различных категорий характерны свои настройки, определяющие вид осей, тип и толщину линий, вид меток и маркеров и т.д. Эти настройки задаются в разделе «Свойства». Именно они будут применяться к серии при ее создании. Если нам требуется создать серию с другим настройками, то вначале мы должны изменить настройки программным путем, а потом создать новую серию. Тогда к ней будут применены измененные настройки.

Для добавления точек используются функции семейства ChartAdd… В зависимости от категории диаграммы используется своя функция добавления точки. Например, для линейного графика, построенного в двух числовых осях, используется функция ChartAddDataXY, а для плоскости в трехмерном пространстве используется функция ChartAddDataXYZ. В большинстве случаев используется функция ChartAddData.

Чтобы создать диаграмму, в закладке «Объекты» установим категорию «Диаграмма» и создадим новый объект. Назовем его «Population».

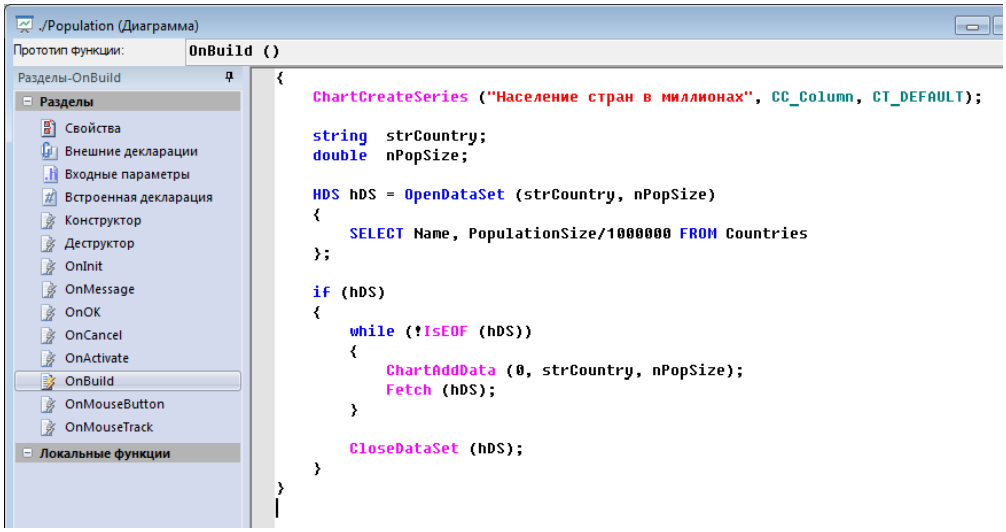


Перейдем в категорию «Свойства» и установим заголовок окна «Население стран».



Теперь перейдем в обработчик OnBuild и введем код:

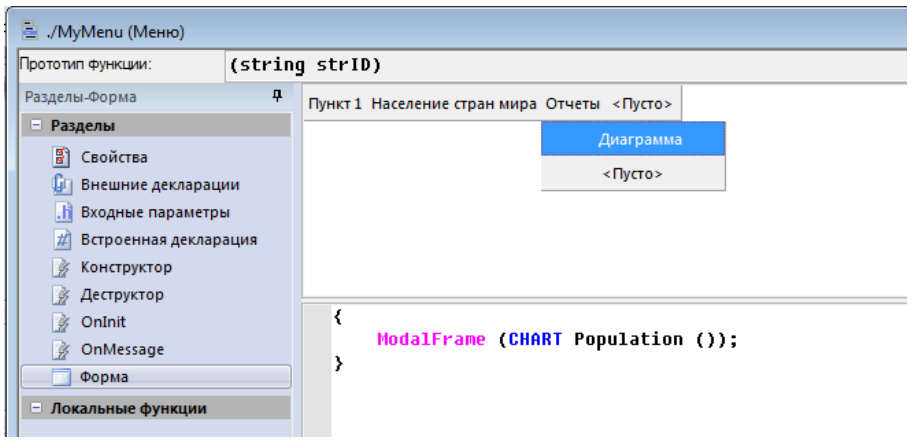
```
{
        ChartCreateSeries ("Население стран в миллионах", CC_Column, CT_DEFAULT);

        string   strCountry;
        double   nPopSize;

        HDS hDS = OpenDataSet (strCountry, nPopSize)
        {
                SELECT Name, PopulationSize/1000000 FROM Countries
        };

        if (hDS)
        {
                while (!IsEOF (hDS))
                {
                        ChartAddData (0, strCountry, nPopSize);
                        Fetch (hDS);
                }

                CloseDataSet (hDS);
        }
}
```



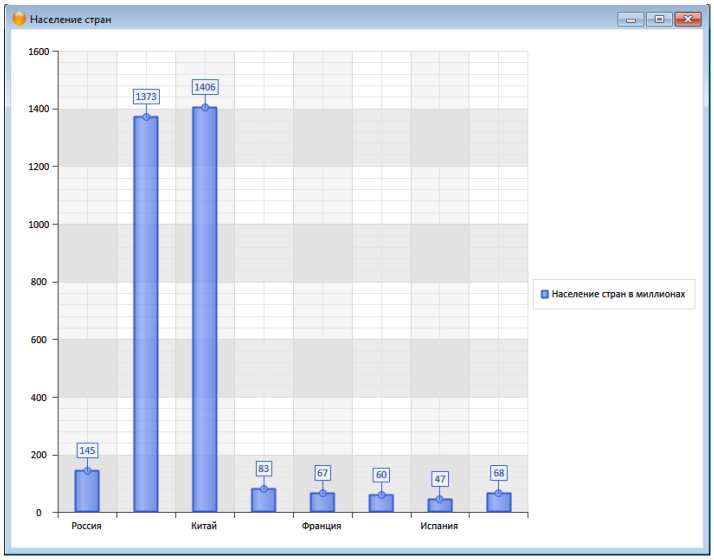
Алгоритм достаточно прост. Первым делом мы создаем серию точек с помощью функции ChartCreateSeries. Мы задаем название серии «Население стран в миллионах», указываем категорию CC_Column (столбчатая диаграмма), и тип CT_DEFAULT (тип по умолчанию для данной категории). Дальше открываем набор данных и, двигаясь по нему в цикле, добавляем точки с помощью функции ChartAddData.

Сохраним объект и доработаем меню, чтобы вызвать его. Для этого добавим в меню группу «Отчеты», создадим в ней команду «Диаграмма» и напишем код вызова.

```
{
        ModalFrame (CHART Population ());
}
```

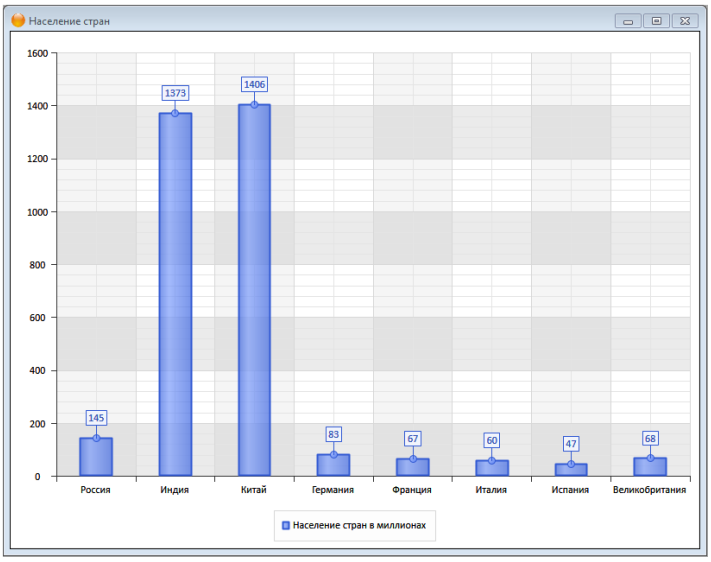


Сохраним меню, запустим модуль и посмотрим, как выглядит наша диаграмма.



Легенда находится справа и занимает значительную часть окна. Хотелось бы перенести ее вниз. Закроем модуль, вернемся в редактор диаграмм и в разделе «Свойства» для расположения легенды укажем «Снизу».

Сохраним изменения и еще раз запустим модуль. Теперь это выглядит так.



Мы построили простую диаграмму с одной серией точек. Примеры более сложных диаграмм с несколькими сериями можно найти в демонстрационном модуле «Прочие объекты \ Диаграммы \ Примеры диаграмм». В этом же примере демонстрируется, как можно динамически перестраивать диаграмму с помощью таймера.

Объект «Диаграмма» имеет обработчики OnMouseButton и OnMouseTrack, которые позволяют создавать интерактивные диаграммы, реагирующие на события мыши. Пример этого есть в демонстрационном модуле «Прочие объекты \ Диаграммы \ Интерактивная диаграмма».

2.11 Отчет

В этом примере мы создадим простой отчет в Excel на основе наших данных. Но вначале познакомимся с тем, как вообще формируются отчеты в X2.

2.11.1 Общие сведения об отчетах

При создании отчета, как правило, используется заранее подготовленный шаблон. Это файл в формате того средства, в котором будет отображаться отчет, например, MS Office или Crystal Reports. Шаблон обычно содержит все статические данные, а код, создающий отчет, заполняет только отдельные недостающие поля. Это существенно упрощает код создания отчета. Хотя, в случае простого отчета шаблон не обязателен.

Для создания отчетов в X2 существует специальный тип объекта «Отчет». Этот объект представляет собой контейнер, содержащий шаблон и код формирования отчета. Вообще-то отчет можно сформировать в коде любого объекта, не используя при этом объект «Отчет», но тогда нужно как-то позаботиться о том, чтобы шаблон, если он требуется, был доступен на той машине, где будет формироваться отчет. В этом смысле объект «Отчет» очень удобен. Он содержит внутри себя все, что нужно для формирования отчета, и его можно экспортировать, как и все прочие объекты X2. В нашем примере мы будем использовать именно объект «Отчет».

Экземпляр этого объекта создается оператором REPORT. При этом система загружает из базы данных ресурс объекта и распаковывает шаблон во временную директорию. Имя файла шаблона система генерирует автоматически, обеспечивая его уникальность в рамках директории. Код формирования отчета располагается в конструкторе объекта. Первым делом нужно получить доступ к файлу шаблона. Для этого служит функция ReportGetTemplate. Она возвращает имя файла и путь. А дальше должен идти код, который дополнит шаблон недостающими данными.

Код формирования отчета можно написать, используя различные технологии. Примеры формирования отчетов можно найти в демонстрационном модуле «Прочие объекты \ Отчеты».

2.11.2 Отчет в формате Crystal Reports

Для отчета в формате Crystal Reports необходимо в свойствах объекта установить флаг «Отчет Crystal Reports». Это нужно для того, чтобы система могла создать экземпляр просмотрщика Crystal Reports.

В коде отчета нет необходимости получать доступ к файлу шаблона, поскольку системе оно известно. Достаточно лишь установить значения параметров и вывести отчет на экран или на печать. Для этого служат функции ReportSetParam, ReportShow и ReportPrint. Эти три функции работают только с отчетами Crystal Reports.

2.11.3 Отчет в формате doc или xls

Для формирования таких отчетов можно использовать VBScript. Типичная схема такова:

- Открываем сессию VBScript (функция OpenVBS)
- Получаем имя шаблона (функция ReportGetTemplate)
- Исполняем код на VBScript (оператор VBS). В коде на VBScript
    - Создаем экземпляр приложения Word или Excel
    - Открываем шаблон
    - Выполняем необходимые действия
- Закрываем сессию VBScript (функция CloseVBS)

Для работы таких отчетов необходимо, чтобы на машине, где формируется отчет, был установлен пакет MS Office.

2.11.4 Отчет в формате docx или xlsx

Документы *.docx и *.xlsx представляют собой zip-архивы, внутри которых содержатся xml-файлы, описывающие структуру и содержание документа в стандарте Office Open XML. Платформа X2 предоставляет набор функций, которые позволяют работать с такими документами напрямую без запуска приложения Word или Excel. Есть два семейства таких функций. Семейство функций с префиксом «Docx» работает с документами Word, семейство «Xlsx» – с документами Excel. Соответственно, шаблон отчета должен быть создан в одном из этих форматов.

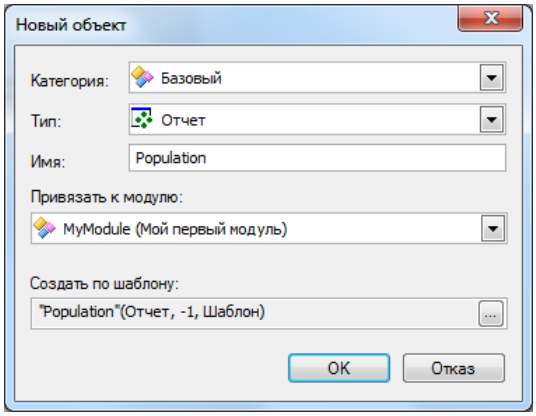
Типичная схема работы такова:

- Получаем имя шаблона (функция ReportGetTemplate)
- Открываем файл шаблона docx или xlsx (функция DocxOpen или XlsxOpen)
- Вносим изменения (набор функций Docx… или Xlsx…)
- Сохраняем файл (функция DocxSave или XlsxSave)
- Закрываем файл (функция DocxClose или XlsxClose)
- Выводим отчет на экран, если это требуется (функция ShellExecute)

Для работы таких отчетов не обязательно иметь пакет MS Office на машине пользователя. Документы *.docx и *.xlsx можно просматривать и с помощью других пакетов офисных программ, таких, как OpenOffice или LibreOffice. Тем не менее, сам шаблон должен быть в формате MS Office, т.к. все функции X2 для работы с такими файлами поддерживают только стандарт Office Open XML.

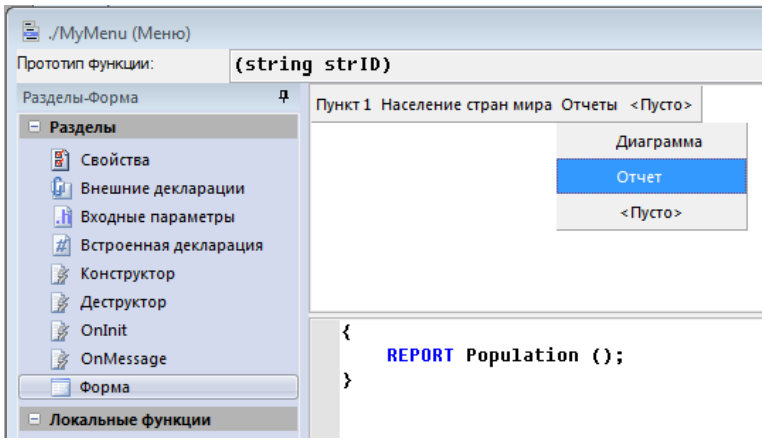
2.11.5 Создаем отчет

В демонстрационном модуле уже есть готовый шаблон объекта «Отчет». Он называется «Population». Создадим наш объект на основе этого шаблона и назовем тем же именем.

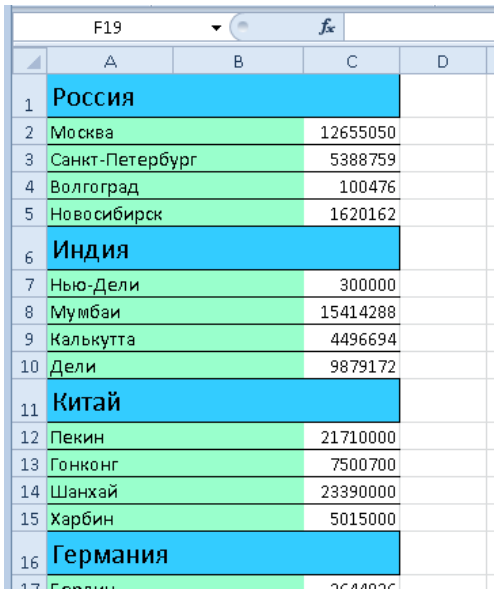


Сохраним объект в базу и напишем код его вызова через меню. Для этого в меню добавим новый пункт «Отчет» и напишем его обработчик

```
{
        REPORT Population ();
}
```

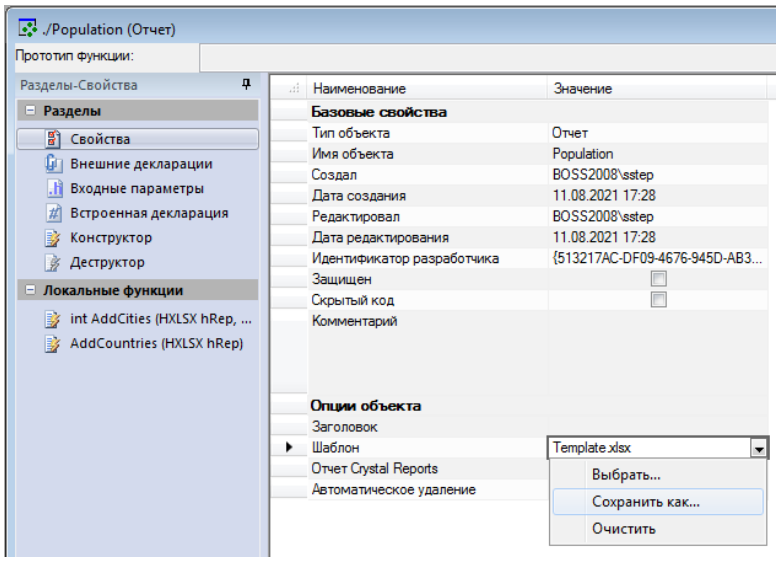


Сохраним изменения, запустим модуль и вызовем наш отчет. Он имеет такой вид.

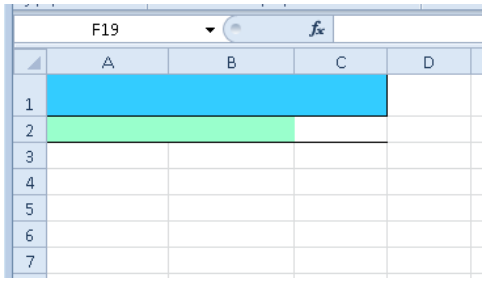


	A	B	C	D
1	Россия			
2	Москва		12655050	
3	Санкт-Петербург		5388759	
4	Волгоград		100476	
5	Новосибирск		1620162	
6	Индия			
7	Нью-Дели		300000	
8	Мумбаи		15414288	
9	Калькутта		4496694	
10	Дели		9879172	
11	Китай			
12	Пекин		21710000	
13	Гонконг		7500700	
14	Шанхай		23390000	
15	Харбин		5015000	
16	Германия			

Чтобы понять, как он работает, нам нужно посмотреть на его шаблон. Для этого войдем в раздел «Свойства» и сохраним файл шаблона на диск.

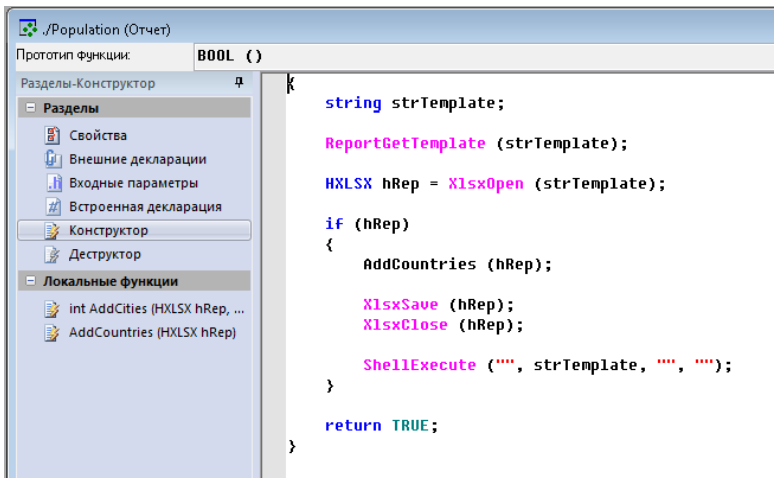


Открыв файл, мы увидим, что в нем заданы форматы для двух строк. Первая строка служит образцом для вывода стран, а вторая – для вывода городов.



Теперь перейдем в конструктор и рассмотрим его код.

```
{
        string strTemplate;

        ReportGetTemplate (strTemplate);

        HXLSX hRep = XlsxOpen (strTemplate);

        if (hRep)
        {
                AddCountries (hRep);

                XlsxSave (hRep);
                XlsxClose (hRep);

                ShellExecute ("", strTemplate, "", "");
        }

        return TRUE;
}
```



Мы видим, что для создания отчета используется типичная схема:

- ReportGetTemplate
- XlsxOpen
- XlsxSave
- XlsxClose
- ShellExecute

Здесь интерес представляет только локальная функция AddCountries, которая и вносит данные в шаблон. Рассмотрим ее код.

```
{
        int     id_Country;
        string  strName;

        HDS hDS = OpenDataSet (id_Country, strName)
        {
                SELECT id_Country, Name FROM Countries
        };

        if (hDS)
        {
                int nRow = 0;

                while (!IsEOF(hDS))
                {
                        if (nRow > 0)
                                XlsxRowInsert (hRep, nRow, 0);

                        XlsxCellSetData (hRep, nRow, 0, strName);
                        nRow = AddCities (hRep, id_Country, nRow++);

                        Fetch (hDS);
                }

                CloseDataSet (hDS);
        }
}
```

./Population (Отчет)

Прототип функции: AddCountries (HXLSX hRep)

Разделы-AddCountries (HXLSX ...

Разделы
- Свойства
- Внешние декларации
- Входные параметры
- Встроенная декларация
- Конструктор
- Деструктор

Локальные функции
- int AddCities (HXLSX hRep, ...
- AddCountries (HXLSX hRep)

```
{
    int     id_Country;
    string  strName;

    HDS hDS = OpenDataSet (id_Country, strName)
    {
        SELECT id_Country, Name FROM Countries
    };

    if (hDS)
    {
        int nRow = 0;

        while (!IsEOF(hDS))
        {
            if (nRow > 0)
                XlsxRowInsert (hRep, nRow, 0);

            XlsxCellSetData (hRep, nRow, 0, strName);
            nRow = AddCities (hRep, id_Country, nRow++);

            Fetch (hDS);
        }

        CloseDataSet (hDS);
    }
}
```

Как видим, здесь открывается набор данных с запросом «SELECT id_Country, Name FROM Countries» и выполняется проход по нему в цикле.

Функция XlsxRowInsert добавляет новую строку в документ. Параметр nRow задает позицию для вставки, а третий параметр 0 задает номер строки, которая будет использоваться в качестве образца. Новая строка унаследует формат строки образца. Как мы помним, в шаблоне нулевая строка уже задана, поэтому вставлять ее не нужно. Для этого выполняется проверка if (nRow > 0). После этого в ячейку с координатами (nRow, 0) вставляется название страны strName. Это делает функция XlsxCellSetData. А потом вызывется локальная функция AddCities, которая и добаляет список городов для страны по ее id_Country.

Обратим внимание на то, что последним параметром ей передается nRow++. Оператор «++» увеличит значение nRow на 1, и выполнится он до того, как будет вызвана сама функция AddCities. Т.е. функция получит уже увеличенное значение nRow.

Функция AddCities устроена аналогичным образом.

./Population (Отчет)

Прототип функции: int AddCities (HXLSX hRep, int id_Country, int nRow)

Разделы-int AddCities (HXLSX h...

Разделы
- Свойства
- Внешние декларации
- Входные параметры
- Встроенная декларация
- Конструктор
- Деструктор

Локальные функции
- int AddCities (HXLSX hRep, ...
- AddCountries (HXLSX hRep)

```
{
    string  strName;
    int     nPopSize;

    HDS hDS = OpenDataSet (strName, nPopSize)
    {
        SELECT Name, PopulationSize FROM Cities WHERE id_Country = :id_Country
    };

    if (hDS)
    {
        while (!IsEOF(hDS))
        {
            if (nRow > 1)
                XlsxRowInsert (hRep, nRow, 1);

            XlsxCellSetData (hRep, nRow, 0, strName);
            XlsxCellSetData (hRep, nRow, 2, nPopSize);
            nRow++;

            Fetch(hDS);
        }

        CloseDataSet(hDS);
    }

    return nRow;
}
```

```
{
	string	strName;
	int	nPopSize;

	HDS hDS = OpenDataSet (strName, nPopSize)
	{
		SELECT Name, PopulationSize FROM Cities WHERE id_Country = :id_Country
	};

	if (hDS)
	{
		while (!IsEOF(hDS))
		{
			if (nRow > 1)
				XlsxRowInsert (hRep, nRow, 1);

			XlsxCellSetData (hRep, nRow, 0, strName);
			XlsxCellSetData (hRep, nRow, 2, nPopSize);
			nRow++;

			Fetch(hDS);
		}

		CloseDataSet(hDS);
	}

	return nRow;
}
```

На входе она принимает id_Country и текущую позицию nRow для вставки следующей строки, а возвращает текущую позицию nRow после вставки.

Как и в функции AddCountries, здесь открывается набор с запросом «SELECT Name, PopulationSize FROM Cities WHERE id_Country = :id_Country» и выполняется проход по нему. В цикле выполняется вставка строки, только в качестве образца используется не нулевая строка, а первая. А потом в две ячейки вставляются название города strName и численность его населения nPopSize. Ну и, конечно же, накручивается счетчик строк (nRow++), чтобы функция AddCountries получила правильную позицию для своей вставки.

2.12 Заключение

В заключение отметим, что в этом кратком практикуме мы познакомились только с некоторыми возможностями платформы X2. Значительная часть возможностей осталась за кадром. Но нашей задачей было лишь получить начальные знания, чтобы упростить дальнейшее самостоятельное изучение платформы. В какой-то части источником информации может служить эта книга, но основным источником является электронная документация, которая поставляется в дистрибутиве платформы. Платформа развивается, появляется новый функционал, вносятся дополнения в существующий, и документация содержит сведения, актуальные для той версии платформы, с которой она выпущена.

3 Среда разработки – полезные советы

Полное описание среды разработки RPDesigner.exe можно найти в электронной документации. Здесь мы лишь обратим внимание на некоторые детали, которые могут оказаться полезными в работе и сделают инструмент более удобным.

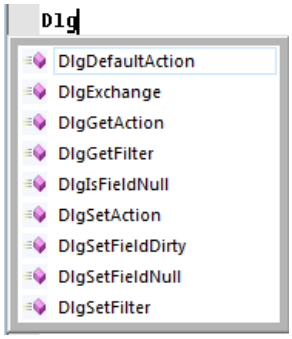
3.1 Настройка среды разработки

Пункт главного меню «Вид\Панели\Настройки программы» выводит на экран панель настроек. Здесь можно задать шрифт редактора кодов, а также, изменить некоторые настройки визуального интерфейса.

3.2 Разработка

3.2.1 Префиксы имен функций

Многие встроенные функции имеют префиксы. Например, функции, относящиеся к спискам, имеют префикс List, а функции, относящиеся к диаграммам, имеют префикс Chart. Это сделано для того, чтобы облегчить программисту поиск нужной функции. Никто не может запомнить имена всех функций, но, если Вы знаете, что нужная Вам функция относится к диалогам, то просто наберите в редакторе «Dlg» и всплывающая подсказка выведет на экран все функции с этим префиксом.



3.2.2 Поддержка IntelliSense

При вводе символов кодовый редактор выводит на экран список известных компилятору имен, который ограничивается по мере ввода. Когда имя встроенной функции введено полностью, появляется подсказка с описанием формальных параметров функции. Эта подсказка появляется, также при наведении курсора на имя функции.

Список IntelliSense, также, можно вызвать с помощью сочетания клавиш Ctrl+пробел. Также, по клавише F1 можно вызвать окно электронной документации.

Во время отладки, когда среда исполнения остановлена и ожидает отладочной команды, можно навести курсор на имя переменной, чтобы посмотреть ее значение. Это работает только для скалярных переменных, имеющих простой встроенный тип. Для структур и массивов их значения выводятся в панель «Переменные».

3.2.3 Горячие клавиши

Чтобы найти парную скобку блока кода, установите каретку перед символом скобки («{» или «}») и нажмите «Ctrl+{». Каретка перескочит на парную скобку, если она существует. Если же применить «Shift+Ctrl+{», то система не только найдет парную скобку, но и выделит весь блок.

Если Вы хотите сдвинуть блок кода вправо или влево, выделите его и используйте клавиши «Tab» или «Shift+Tab».

Если Вы хотите создать экземпляр объекта, но не помните точно его имени, введите оператор создания экземпляра, к примеру, «BROWSER», и нажмите «Shift+Ctrl+down». Система выведет диалог выбора объекта с фильтром по типу. Выберите нужный объект, и система подставит его имя в редактор кода.

Чтобы вызвать объект на редактирование прямо из редактора кода, установите каретку на его имя и нажмите «Ctrl+Enter». Например, для строки кода

```
HOBJECT hList = BROWSER ListColor();
```

система вызовет на редактирование список ListColor.

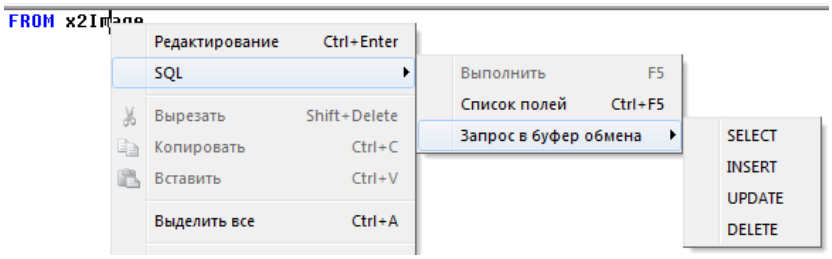
3.2.3.1 Работа с кодом на SQL

SQL-запрос можно вызвать на исполнение непосредственно из редактора кода. Для этого нужно выделить его и нажать F5.

```
SELECT FileName AS Pic, FileName AS KeyFld, FileExt FROM x2Image
:strWHERE
```

Система выведет результат запроса в окно вывода. При этом запрос не должен содержать подстановочных переменных, т.к. на данный момент их еще не существует.

Если установить каретку на имени таблицы, то можно получить список ее полей или автоматически сформировать запрос в буфере обмена.



3.2.4 Комментирование кода

Выведите на экран инструментальную панель редактора кодов. Для этого нужно щелкнуть правой клавишей в области инструментальной панели (тулбара), и в появившемся списке выбрать пункт «Редактор». Панель выглядит так:

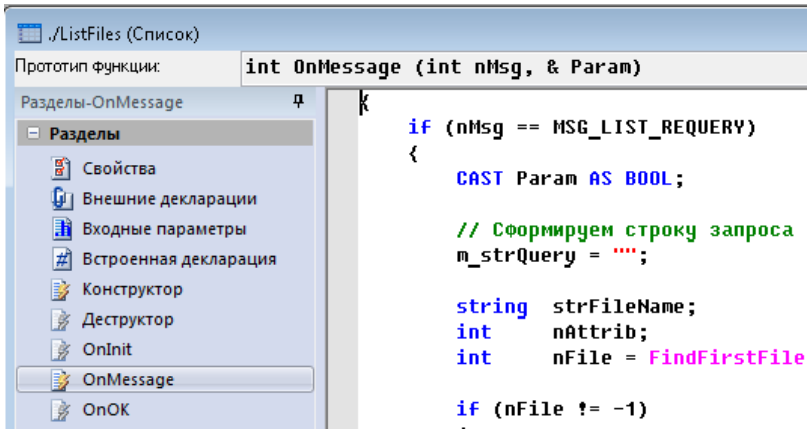
Обратите внимание на кнопки  . Первая комментирует выделенный блок кода в стиле X2. Вторая делает то же самое, но в стиле комментариев SQL. Третья снимает комментарий.

3.2.5 Прототипы функций

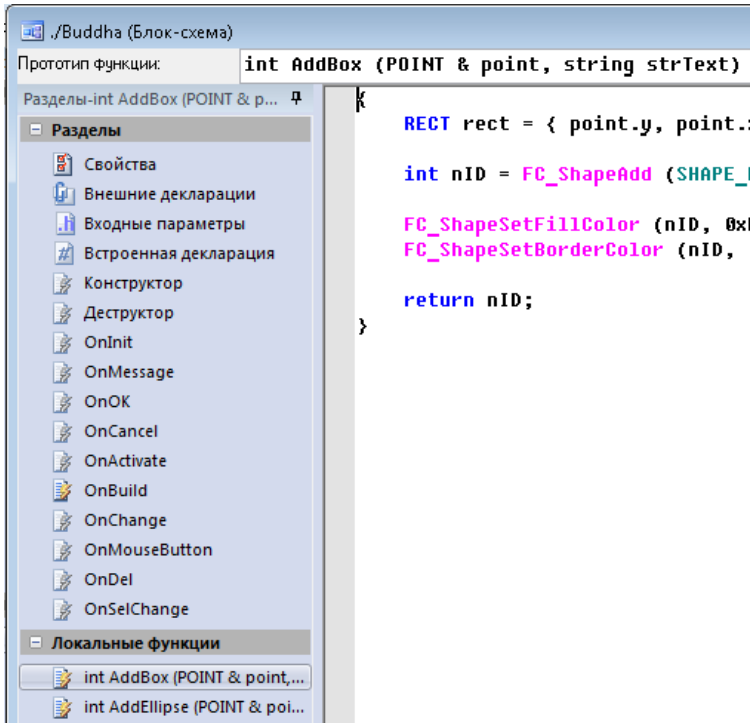
При разработке стоит обратить внимание на строку «Прототип функции». Она служит подсказкой и сообщает о том, какие параметры принимает обработчик или локальная функция объекта, и значение какого типа следует вернуть.

Например, обработчик OnMessage возвращает значение типа int и принимает два параметра:

- nMsg – тип int
- Param – бестиповая ссылка (требует приведения типа с помощью оператора CAST)

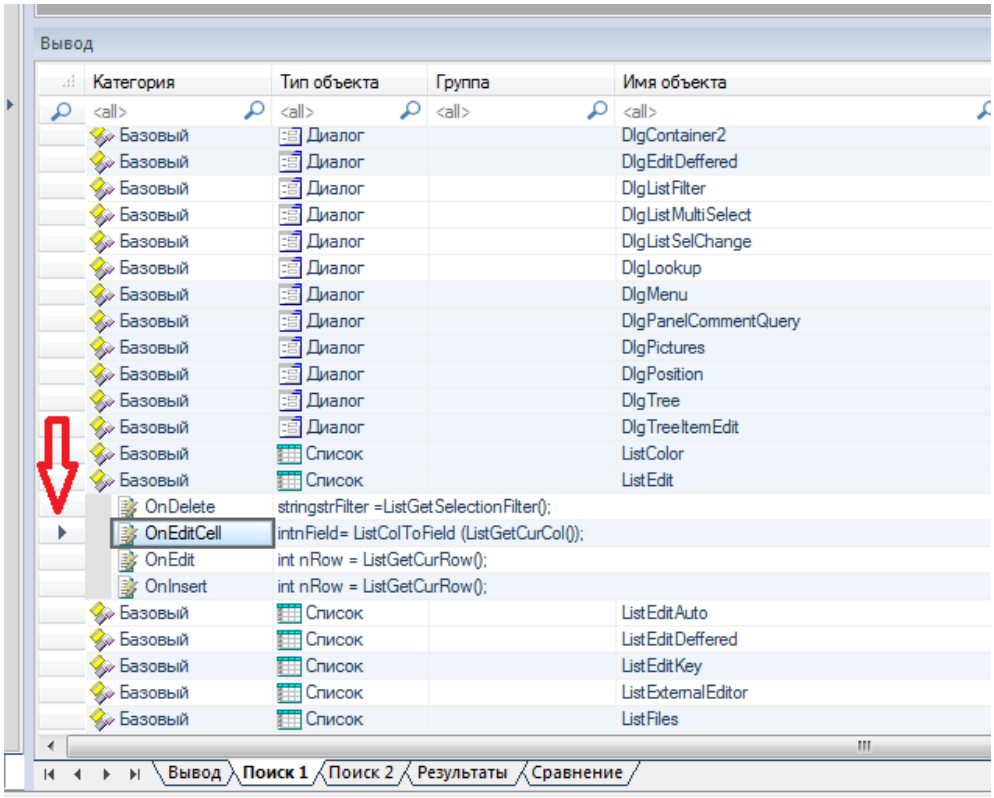


Для обработчиков эта строка является нередактируемой, т.к. их параметры жестко заданы. Для локальных функций разработчик сам может задать параметры в этой строке.



3.2.6 Окно поиска

В окне поиска, чтобы вызвать на редактирование найденный объект, нужно дважды кликнуть на самой первой колонке, которая не имеет наименования. Двойной клик на другой колонке приводит к разворачиванию дерева.



3.3 Компиляция

Каждый раз при сохранении объекта вызывается его компиляция. Компилятор возвращает протокол компиляции, который выводится в окно вывода. Окно вывода представляет собой всплывающую панель, которая появляется в нижней части главного окна.

Компиляция единичного объекта может быть вызвана из списка объектов через контекстное меню, либо по сочетанию клавиш Ctrl+F7.

Любой объект X2 может принимать параметры. Параметры описываются в разделе «Входные параметры». Если один объект X2 вызывает из своего кода другой, то при компиляции транслятор проверит наличие в БД вызываемого объекта и корректность передаваемых параметров. Их число и типы должны соответствовать формальному описанию.

В ходе разработки описание параметров объекта может измениться, но другие объекты, его вызывающие, уже откомпилированы со старым описанием. В этом случае возникнет ошибка периода исполнения. Чтобы этого не происходило, перед выпуском модуля настоятельно рекомендуется выполнять компиляцию всех объектов модуля. Это можно сделать по сочетанию клавиш Ctrl+Alt+F7, либо через пункт главного меню «Разработка / Компилировать множество».

При любом изменении объекта «Декларация» требуется перекомпиляция всех объектов, в которые она включена.

3.3.1 Позиционирование на ошибке

Протокол компиляции может содержать сообщения об ошибках. Текст сообщения может быть, примерно, таким:

```
    Компиляция объекта: TYPE: PROCEDURE, NAME: CollectionFind ...
    Компиляция объекта: TYPE: PROCEDURE, NAME: CollectionFind. Шаг 2.
    Компиляция сегмента Procedure ...
    ERROR: 45; Pos:(12, 14) rrr : Неопознанный идентификатор!

    RESULT: ERROR
    ERRORS: 1, WARNINGS 0
Предупреждение : Объект сохранен без компиляции!
```

Если дважды кликнуть на строке с надписью ERROR, то система поднимет на редактирование объект, который вызвал данную ошибку, и установит каретку в соответствующую позицию.

3.4 Отладка и отладочный режим

Среда исполнения RPExec.exe имеет два режима: режим эксплуатации и режим отладки.

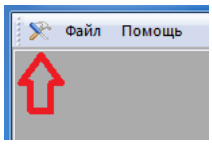
Режим эксплуатации возникает, когда RPExec запускается из командной строки. В этом режиме система кэширует ресурсы объектов.

Отладочный режим возникает, когда среда исполнения запускается из Дизайнера. В обычном и в отладочном режиме среда исполнения использует разные процессоры, поэтому переключиться из одного режима в другой нельзя.

Отладочный режим существенно медленнее режима эксплуатации, но позволяет выполнять отладочные операции:

- Остановка в точках прерывания
- Пошаговая отладка
- Непосредственный доступ к переменным с возможностью редактирования их значений

Кроме того, отладочный режим отключает кэширование ресурсов в среде исполнения, а также, в главном меню появляется дополнительная кнопка, позволяющая вызвать на редактирование активный объект.



При пошаговой отладке каждый раз, когда среда исполнения прерывает работу и передает управление среде разработки, она, также, передает ей значения всех переменных, которые находятся в данной области видимости. Поскольку глобальные переменные всегда находятся в области видимости, то будут передаваться все их значения. Поэтому следует избегать большого количества глобальных переменных. Это может замедлить отладку. То же относится к большим структурам или массивам.

3.4.1 Точки прерывания и пошаговая отладка

Чтобы установить точку прерывания в редакторе кода переместите каретку на нужную строку кода и нажмите F9 или дважды щелкните на серой полосе слева. Теперь исполнение будет прерываться в этом месте.

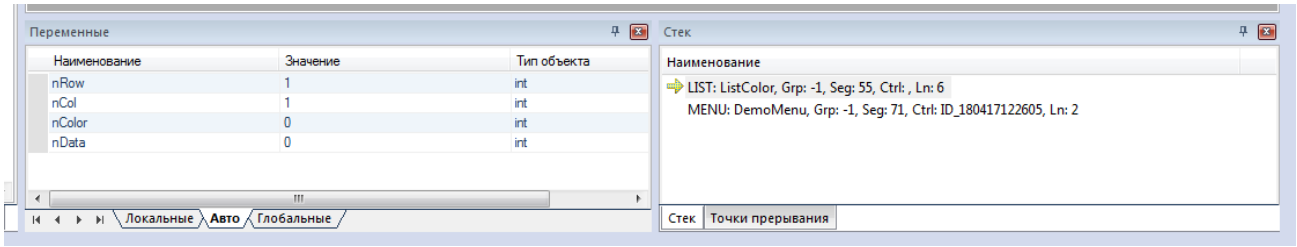
```
// Достаем четвертое поле буфера списка.
ListGetFieldData (nData, nRow, 4);

if (ListColToField (nCol) == 2)
    nColor = 21;
else if (nData)
    nColor = 5;
else
    nColor = 13;
```

Далее используйте клавиши:

- F7 – продолжить исполнение до следующей точки прерывания
- F10 – шаг в текущем сегменте
- F11 – шаг с погружением внутрь вызываемого сегмента
- Shift+F10 – выход из исполняемого сегмента

Для удобства можно вывести в линейку инструментов панель отладчика (Вид\Панели\Отладчик).

В отладочном режиме в Дизайнере появляются дополнительные панели, которые становятся доступными, когда среда исполнения остановлена в ожидании отладочной команды.



3.4.2 Панель «Переменные»

Панель «Переменные» позволяет просматривать и редактировать значения переменных. Она имеет три закладки:

- Закладка «Локальные» отображает локальные переменные объекта, объявленные в его встроенной декларации. Эти переменные доступны в любом сегменте объекта.
- Закладка «Авто» отображает автоматические (или стековые) переменные, объявленные в блоке кода, ограниченном фигурными скобками. Они доступны внутри того блока, в котором объявлены.
- Закладка «Глобальные» отображает глобальные переменные, объявленные во внешних декларациях. Эти переменные доступны в тех объектах, в которые включена соответствующая декларация.

Значения переменных можно менять путем непосредственного редактирования. Для этого кликните в панели на ячейке, содержащей значение переменной.

Изменение значений переменных в ходе отладки бывает очень полезно при тестировании алгоритма, например, когда нужно загнать исполнение в другую ветку оператора switch, быстро накрутить счетчик цикла for, или изменить значение входного параметра.

3.4.3 Панель «Стек»

Панель «Стек» позволяет просматривать стек вызовов и переключать контекст исполнения на любой из исполняемых сегментов. Для переключения контекста исполнения нужно дважды щёлкнуть на строке контекста, система поднимет редактор соответствующего объекта и установит каретку на исполняемой строке кода.

3.4.4 Панель «Точки прерывания»

Панель «Точки прерывания» отображает все точки прерывания. Точку прерывания можно сделать временно неактивной, если снять флажок. Отладчик не будет останавливаться на неактивной точке. Можно, также удалить ненужные точки прерывания. Используйте для этого контекстное меню.

3.5 Диагностика на стороне пользователя

В ряде случаев ошибка может проявляться на стороне пользователя, но на стороне разработчика ее воспроизвести не удается. В этих случаях диагностику приходится проводить на стороне пользователя. Если пользователь имеет достаточную квалификацию, и его ключ дает ему право на запуск Дизайнера (среды разработки), то он может попытаться самостоятельно отладить код. Но даже в этом случае отладка может оказаться затруднительной, если ошибка носит «плавающий» характер.

В таких ситуациях можно использовать встроенные средства диагностики, либо написать собственный алгоритм вывода трассировки.

Модифицированные объекты можно передать пользователю через файл экспорта.

3.5.1 Функция GetLastError

Многие встроенные функции X2 при завершении выставляют код ошибки, который можно получить с помощью функции GetLastError. Для конкретной функции характерен конкретный набор кодов ошибок, а некоторые функции не выставляют этот код. Это нужно уточнять в документации.

Функция GetLastError может работать в режиме вывода сообщений на экран или без вывода в зависимости от параметров. Например:

```
if (GetLastError () == ERR_BAD_HANDLE)
{
	// Делаем что-то...
}
```

При такой форме вызова функция не будет выводить сообщений на экран. А при такой форме вызова:

```
GetLastError (TRUE, FALSE);
```

функция будет выводить на экран код ошибки, если он не равен ERR_OK.

3.5.2 Трассировка и функция TRACE

Режим трассировки можно включить при запуске среды исполнения RPExec.exe с помощью ключа командной строки «-T:файл».

В файл трассировки попадают все запросы, которые отправляются на SQL-сервер. С помощью функции TRACE в него можно вывести дополнительную информацию из прикладного кода.

Но включенный режим трассировки может заметно повлиять на производительность, поэтому трассировка всех SQL-запросов не во всех случаях может оказаться приемлемой.

В этом случае можно написать собственный алгоритм трассировки и использовать его только в конкретных местах, которые вызывают подозрения.

3.5.3 Собственный алгоритм трассировки

Алгоритм может выглядеть так:

```
int       i = 1;
double    d = 3.14;
string    str = "My string";


HFILE hFile = FileOpen ("D:\\Trace.txt", FOF_WRITE | FOF_CREATE | FOF_NOTRUNCATE);

if (hFile)
{
        // Выводим значения переменных
        FileWrite (hFile, "i = " + ToString (i) + "\r\n");
        FileWrite (hFile, "d = " + ToString (d) + "\r\n");
        FileWrite (hFile, "str = " + str + "\r\n");
}

FileClose (hFile);
```

Файл D:\Trace.txt будет содержать:

```
i = 1
d = 3.14
str = My string
```

3.6 Графы

При анализе кода очень полезной может оказаться функция построения графов. Вызвать построение графов можно через контекстное меню в списке объектов (пункт «Построить граф…»). Графы отображают взаимосвязи между объектами в древовидном представлении.

Граф зависимостей объекта отображает те объекты, которые из него вызываются.

Граф вызовов (или прямой граф вызовов) отображает цепочки вызовов, которые приводят к вызову данного объекта.

Обратный граф вызовов просто является транспонированной версией прямого графа вызовов.

Если в графе навести курсор на имя объекта, то можно получить дополнительную информацию о том, из каких сегментов вызывается данный объект. Можно, также, вызвать объект на редактирование двойным кликом.

3.7 Архив

Каждый раз при сохранении объекта его предыдущая копия помещается в архив. Копию объекта из архива можно восстановить или открыть на просмотр.

Можно, также, удалять копии объектов из архива либо защитить какие-то конкретные копии от случайного удаления.

Доступ к архиву реализуется через панель «Архивы». Путь по меню: «Вид \ Панели \ Архивы».

3.8 Шаблоны объектов

Шаблоны – это просто заготовки объектов. Они хранятся отдельно от объектов и не привязываются к модулям. При сохранении они не компилируются, поэтому код объекта может быть не дописан. Любой объект можно сохранить, как шаблон, и на основе шаблона можно создать новый объект.

Шаблоны могут быть удобны при командной разработке, например, для поддержания единства интерфейсов или единых методов обработки ошибок.

Еще одним важным свойством шаблона является то, что объект, созданный на его основе, наследует идентификатор разработчика шаблона. Это можно использовать для того, чтобы предоставить возможность партнерской компании создавать защищенные модули, которые будут контролироваться Вашими ключами лицензионной защиты.

3.9 Экспорт базовых объектов

Перенести базовые объекты из одной БД в другую можно с помощью экспортного файла. Файл формируется командой «Экспорт» (Сервис \ Сопровождение \ Базовые объекты \ Экспорт) и загружается командой «Импорт». Файл, также, можно вложить в инсталляционный пакет, предназначенный для инсталлятора баз данных RPSetupDB.exe.

Все, что сказано ниже, относится только к экспорту базовых объектов. Для объектов замещения используется иной алгоритм.

В директории инсталляции клиентской части присутствуют три файла:

- InstApp_CreateStruct.sql
- InstApp_Scenary.sql
- InstApp_DropStruct.sql

Эти три файла содержат SQL-скрипты, которые реализуют алгоритм импорта для MS SQL Server. Для Postgres Pro используются аналогичные файлы с постфиксом _PG.

Файл InstApp_CreateStruct.sql создает набор временных таблиц в БД, куда будет выполняться импорт.

Функция экспорта формирует еще один файл InstApp_Insert.sql, который содержит операторы INSERT для вставки объектов в эти таблицы.

Файл InstApp_Scenary.sql содержит собственно сценарий переноса объектов из временных таблиц в постоянные системные таблицы. При этом создаются записи в таблицах истории импорта, а также, в таблице протокола #rptmp_Protocol.

И, наконец, файл InstApp_DropStruct.sql удаляет из БД временные таблицы, созданные в InstApp_CreateStruct.sql.

Все эти файлы включаются в экспортный файл. К нему, также, можно прикрепить еще два пользовательских скрипта, которые будут исполнены до и после импорта.

Функция импорта выполняет следующе операции:

1. Исполняет скрипты
   1.1. User_before.sql – если присутствует
   1.2. InstApp_CreateStruct.sql
   1.3. InstApp_Insert.sql
   1.4. InstApp_Scenary.sql
2. Из таблицы #rptmp_Protocol извлекает данные и формирует строку протокола
3. Исполняет скрипты
   3.1. InstApp_DropStruct.sql
   3.2. User_after.sql – если присутствует

Разработчик при желании может внести свои изменения в эти файлы скриптов. Конечно, не следует изменять имена и структуру существующих таблиц, но можно добавить новые или вывести дополнительную информацию в таблицу протокола.

4 Обзор языка программирования X2

Полное описание языка программирования X2 можно найти в электронной документации, входящей в комплект поставки платформы.

В данном разделе приводятся только общие сведения о языке и рассматриваются некоторые конструкции.

4.1 Основные свойства языка X2

1. Синтаксически язык X2 близок к языку «C»
2. Компилируемый
3. Регистрочувствительный
4. Использует статическую типизацию (тип переменной задается при ее объявлении и не может быть изменен)
5. Кроме встроенных типов данных поддерживаются пользовательские типы – структуры
6. Поддерживаются:
   - 6.1. Три категории переменных: глобальные, локальные, автоматические. Они имеют разную область видимости и время жизни.
   - 6.2. Константы
   - 6.3. Статические массивы, а также, возможность передачи массива в качестве параметра
   - 6.4. Динамические массивы (коллекции)
   - 6.5. Передача параметров по ссылке
     - 6.5.1. Проверка типов параметров осуществляется, как на этапе компиляции, так и в период исполнения
     - 6.5.2. Поддерживаются бестиповые ссылки с возможностью приведения к известному типу. Корректность приведения контролируется в период исполнения.
7. Встроенная поддержка SQL
   - 7.1. Блоки кода на SQL отделяются от блоков кода на языке X2 посредством явных синтаксических конструкций. Исключения составляют одиночные операторы INSERT, UPDATE, DELETE. Эти операторы можно использовать в коде на X2 непосредственно.
   - 7.2. В код на SQL можно подставлять переменные X2. Места подстановки переменных помечаются явным образом.
   - 7.3. В качестве подстановки можно использовать не только переменные, но и выражения любой сложности, возвращающие скалярные значения
8. Работа с БД. Поддерживаются возможности:
   - 8.1. Открытия новых соединений с источниками данных ODBC, а также, возможность наследования объектами X2 открытого соединения

8.2. Работы с наборами данных

8.3. Исполнения произвольного блока кода на SQL с возможностью возврата данных в вызывающий код

9. Поддерживается возможность работы с сессиями VBScript и JScript, т.е. в коде на X2 можно использовать вставки кода на этих языках.

10. Есть возможность задействовать библиотеки (dll), написанные на C++ или C#

4.2 Типы данных

Типы данных могут быть встроенными и пользовательскими.

Встроенные типы данных поддерживаются непосредственно на уровне языка. Для них определено большинство встроенных операторов и функций языка.

Пользовательские типы данных задаются разработчиком с помощью ключевого слова struct.

4.2.1 Встроенные типы данных

Встроенные типы данных делятся на основные (или базовые) и производные. Базовых типов всего пять:

Тип	Диапазон значений	Значение по умолчанию	Преобразование в строку
int	от –2147483648 до 2147483647 в десятичной системе исчисления; разрядность – 32 bit	0	Маска по умолчанию: n
double	1.7E+/-308 (15 цифр) в десятичной системе исчисления; разрядность – 64 bit	0	По умолчанию используется наиболее короткая форма вывода, либо десятичная, либо экспоненциальная, и число выводится с точностью до 6 значащих цифр
string	Длина строки ограничена доступным объемом оперативной памяти	Пустая строка	
datetime	от 01.01.100 до 31.12.9999 по Григорианскому календарю без учета перехода на летнее время	0 (30.12.1899 полночь)	Маска по умолчанию: d %Y-%m-%d
datedif	Разница в датах от минимального до максимального значения типа datetime	0	Маска по умолчанию: f %D, %H, %M, %S

Для справки:

Тип datetime в SQL имеет диапазон 01.01.1753 – 31.12.9999 и значение по умолчанию 01.01.1900 00:00:00

4.2.1.1 Производные типы данных

Производные типы данных, как и основные типы, тоже являются встроенными, но, по своей сути, не являются самостоятельными типами. Они представляют собой лишь псевдонимы существующих основных типов, только с более узкой специализацией.

Производные типы существуют только на уровне транслятора, на уровне физического представления они реализованы как основные встроенные типы.

Смысл использования таких типов состоит в том, чтобы задействовать механизм проверки типов транслятора для каких-то групп специфических операций. Например, функции, работающие с набором данных, используют дескрипторы наборов. А функции работы с файловой системой используют дескрипторы файлов. И те, и другие дескрипторы на уровне внутреннего представления является целыми числами (тип int), но по смыслу они различаются, и было бы ошибкой передавать дескриптор файла в функцию, обслуживающую набор данных, полученных из БД. Поэтому введены два новых типа: HDS (дескриптор объекта DataSet) и HFILE (дескриптор файла). Хотя оба они и являются производными от встроенного типа int, но для каждого из них определён собственный набор операций. Таким образом, ошибочная передача целого числа в качестве дескриптора отсекается на уровне трансляции.

Производный тип автоматически приводится к базовому типу, от которого он произошел. Это означает, что производный тип можно передать в функцию, принимающую базовый тип. В частности, для типа HDS доступны все арифметические операции, определенные для типа int. Но обратное преобразование невозможно, т.е. передать тип int вместо HDS нельзя.

Преобразование в строку для производного типа осуществляется по тем же правилам, что и для его базового типа.

В таблице, приведенной ниже, дан пример некоторых производных типов. Полное описание производных типов приводится в электронной документации, соответствующей версии платформы.

Производный тип	Базовый тип	По умолчанию	Описание
BOOL	int	0 (FALSE)	Логический тип данных. Для него определены две встроенные константы: TRUE = 1 и FALSE = 0
HDS	int	0	Дескриптор объекта DataSet
HFILE	int	0	Дескриптор файла
HDB	int	0	Дескриптор соединения с сервером
HCOL	int	0	Дескриптор объекта коллекция
HVBS	int	0	Дескриптор сессии VBScript
HJS	int	0	Дескриптор сессии JScript
HBIN	int	0	Дескриптор объекта Binary
HOBJECT	int	0	Дескриптор объекта X2

4.2.2 Пользовательские типы данных – структуры

Структуры – типы данных, представляющие собой совокупность полей различных типов. Поле структуры может иметь, как встроенный тип данных, так и пользовательский, а также, может быть массивом.

Пример:

```
struct TFIO
{
        string    m_strFName;    // имя
        string    m_strMName;    // отчество
        string    m_strLName;    // фамилия
};

struct TPerson
{
        int           m_nID;          // id в базе данных
        TFIO          m_tFIO;         // ФИО
        datetime      m_dtBirthday;   // день рождения
        HBIN          m_hPhoto;       // фотография
};

struct TCommand
{
        int           m_nID;                  // id в базе данных
        string        m_strName;              // имя команды
        TPerson       m_arrMembers[10];       // члены команды (10 чел)
};
```

В данном примере все имена полей структур имеют префиксы. Это не является обязательным, имена могут быть любыми, но использование префиксов целесообразно, т.к. повышает читаемость кода. Особенно это полезно в больших проектах, где программистам часто приходится работать с чужим кодом. Префиксы помогают быстро понять, каков тип переменной, является ли она массивом или членом данных другой структуры или объекта.

4.2.3 Литералы

Все числовые литералы задаются без пробелов и разделителей разрядов. В качестве разделителя целой и дробной части используется точка (символ «.»).

Литералы типа int могут быть заданы, как в десятичной, так и в шестнадцатеричной форме. В шестнадцатеричной форме литерал должен начинаться с последовательности «0x». В его теле могут использоваться буквы латинского алфавита от A до F в любом регистре.

Литералы типа double могут быть заданы в форме с десятичной точкой (n.m), либо в экспоненциальном представлении (n.kE[+/-]m).

Литералы типа datetime обрамляются одинарными кавычками (символ «’»), например,

‘13-08-2013 13:31:47’.

Литералы типа datedif, обрамляются одинарными кавычками (символ «’») и имеют формат ‘Дни, Часы, Минуты, Секунды’, например, ’10, 12, 33, 25’.

Строковые литералы обрамляются двойными кавычками (символ «”»). Поддерживаются Esc-последовательности:

- \r – Возврат каретки
- \n – Перевод строки
- \t – Табуляция
- \” – Двойная кавычка
- \\ – Обратный слеш
- \Код – Код символа UTF 16 в десятичном или шестнадцатеричном (0x) представлении

4.2.4 Приведение типов

Компилятор языка X2 не поддерживает явного приведения типов, но может выполнять неявное приведение типов.

Правила неявного приведения типов действуют только для простых встроенных типов:

- все производные типы автоматически приводятся к своим базовым типам
- тип int автоматически приводится к типу BOOL

Производный тип автоматически приводится к базовому типу, от которого он произошел. Это означает, что производный тип можно передать в функцию, принимающую базовый тип. В частности, для типа HDS доступны все арифметические операции, определенные для типа int. Но обратное преобразование невозможно, т.е. передать тип int вместо HDS нельзя.

Для всех встроенных базовых типов существует преобразование в строковый формат. Для этого служит функция ToString. Преобразование в строку для производного типа осуществляется по тем же правилам, что и для его базового типа.

При преобразовании в строковый тип какого-либо значения можно указать способ его представления. Это делается посредством маски, которая является последовательностью управляющих символов. Для каждого типа данных предусмотрен свой набор масок. Если маска не указана, используется маска по умолчанию.

Для обратного преобразования служат функции ToInt, ToDouble, ToDatetime.

4.3 Переменные и константы

4.3.1 Объявление переменных

Синтаксис объявления переменной:

type VarName1 [= value | { value1, … valueN}] [, VarName2 [= value | { value1, … valueN}]];

где

- type – тип переменной
- VarName1, VarName2 – имена переменных
- value – значение, если type – простой встроенный тип
- value1, … valueN – значения полей переменной, если type – структурированный тип

Первый символ имени переменной должен быть буквой.

Одним оператором может быть объявлено несколько переменных одного типа.

При объявлении переменная может быть проинициализирована явным образом. Если этого не сделано, она примет значение по умолчанию в зависимости от типа.

Для инициализации структурированных типов (структуры, массивы) используются фигурные скобки, в которых перечисляются инициализаторы полей, разделенные запятой.

При инициализации структур число значений и их типы должны соответствовать числу полей структуры и их типам, а также, должны следовать в том порядке, как они объявлены в структуре.

При инициализации массивов число инициализаторов должно соответствовать числу элементов массива, а их тип должен соответствовать типу массива. Инициализация проводится от нулевого элемента к (N – 1)-му, где N – общее число элементов массива.

Пример:

// Три переменные типа string и две переменные типа int

```
string   strName, strPic, strWHERE = "WHERE ";
int      nColor;
int      nData = 0;
```

4.3.2 Категории переменных

Существуют следующие категории переменных:

- Глобальные
- Локальные
- Автоматические (или стековые)

Категория	Область видимости	Время жизни
Глобальные	Все приложение	С момента объявления до завершения сессии
Локальные	Экземпляр объекта	С момента создания экземпляра объекта до момента его уничтожения
Автоматические	Блок	Время исполнения блока кода, в котором объявлена переменная

4.3.3 Глобальные переменные

Глобальные переменные могут быть объявлены только в объекте «Декларация». Чтобы глобальная переменная стала видна в конкретном объекте, в него явным образом должна быть включена та декларация, в которой эта переменная объявлена.

1. Все переменные, объявленные в декларации, имеют глобальную область видимости
   1.1. Декларация – единственный объект, в котором можно объявить глобальные переменные
2. Декларация может содержать объявление переменных, но не может содержать кода их инициализации
   2.1. Код инициализации переменных может быть включен в любой **исполняемый** объект (целесообразно использовать для этого конструктор модуля, либо процедуру, вызываемую из конструктора модуля)
3. Глобальная переменная создается в тот момент, когда впервые будет обработана декларация, содержащая ее объявление
   3.1. Если одна декларация включена в несколько объектов, то она будет обработана при вызове первого из этих объектов. При вызове следующего, декларация не будет обрабатываться вторично.

4.3.4 Локальные переменные и встроенная декларация

Каждый объект X2 имеет специальный сегмент «Встроенная декларация». Встроенная декларация аналогична внешней, только не представляет собой самостоятельного объекта. Внешняя декларация, являясь самостоятельным объектом, может быть включена в любой другой объект. Встроенная декларация не может быть использована в каком-либо ином объекте и действует только внутри того объекта, в котором объявлена.

Все типы, переменные и константы, объявленные во встроенной декларации, имеют локальную область видимости, т.е. экземпляр объекта. Соответственно, все, что объявлено в ней, доступно в любом сегменте экземпляра объекта, но недоступно за его пределами.

4.3.5 Автоматические переменные

Переменная, объявленная в блоке кода, называется автоматической (или стековой).
Область ее видимости – блок, в котором она объявлена, а также, все вложенные блоки.
При выходе из блока, в котором она объявлена, она становится недоступной.

4.3.6 Массивы

Массив – переменная, представляющая собой набор элементов одного типа. Тип элемента массива может быть, как встроенным, так и пользовательским. Размерности массива должны задаваться константными значениями целочисленного типа. Число размерностей массива не ограничено.

Обращение к элементам массива производится по индексу, т.е. порядковому номеру элемента в массиве. Нумерация начинается с 0. При обращении к элементам массива автоматически осуществляется проверка индексов на выход за пределы заданных размерностей.

Объявление массива:

```
type VarName [m][n]… [= { value1, … valueN }];
```

Где:

- type – тип элемента массива
- VarName – имя массива
- [m], [n] – константы типа int, определяющие размерности массива. Значения должны быть больше 0. В качестве размерности может быть задано константное выражение, т.е. выражение, которое может быть посчитано компилятором.
- value1, … valueN – значения инициализации элементов массива. Являются необязательными параметрами.

При объявлении массива, его можно проинициализировать начальными значениями. Тип инициализатора должен соответствовать типу элемента массива, либо быть приводимым к

нему. Число значений должно быть равно произведению всех размерностей массива. Последовательность инициализаторов – от младшего (правого) значения размерности к старшему (левому).

При обращении к элементам массива размерности m индекс задается от 0 до m – 1.

Пример:

```
// Одномерный массив из 10-х элементов типа string
string arrStr [10];

// Трехмерный массив из 8-ми элементов типа MyStruct
MyStruct arrStruct [2][2][2];

// Двумерный массив из 9-и элементов типа int и его инициализация
BOOL b = 0;

int arrInt [3][3] =
{
	0, 1, 2,
	3, 4, 5,
	6, 7, b // тип BOOL автоматически приводится к типу int
};

/*
	Элемент arrInt[0][0] получит значение 0
	arrInt[0][1] == 1
	arrInt[1][0] == 3
	arrInt[2][2] == b
*/
```

4.3.7 Константы

В целом, синтаксис объявления константы аналогичен синтаксису объявления переменной, но имеет некоторые особенности:

- при объявлении используется ключевое слово **const**
- константа должна быть проинициализирована при объявлении, в дальнейшем изменить значение константы нельзя
- константа может иметь только простой встроенный тип, т.е. не может быть массивом или структурой

Константы, объявленные в объекте «Декларация», имеют глобальную область видимости, т.е. доступны в любом объекте, в который включена данная декларация.

Константы, объявленные в сегменте «Встроенная декларация», имеют локальную область видимости.

Константы, объявленные в блоке кода, аналогично автоматическим переменным, доступны только в данном блоке.

Синтаксис объявления константы:

const type Name1 = value1 [, Name2 = value2 … NameN = valueN];

где

- const – ключевое слово
- type – тип константы
- Name1-NameN – одно или несколько имен однотипных констант
- value1-valueN – значения соответствующих констант

Первый символ имени должен быть буквой.

В одном операторе может быть объявлено несколько констант одного типа.

В качестве значения константы может быть передано константное выражение.

Пример:

```
const string strName = “Имя”;
const int nColor = 127;
```

4.3.8 Функции VarClear и VarCopy

Переменная может иметь довольно сложную структуру, а также, может быть массивом большой размерности. Иногда возникает необходимость сделать копию такой переменной, т.е. скопировать значения всех ее полей в другую переменную того же типа. Можно вручную написать алгоритм на X2, который будет выполнять простое присвоение для каждого поля переменной, но этот алгоритм может оказаться довольно громоздким, если полей очень много. Функция VarCopy выполняет такое копирование. Единственное условие в том, что типы переменных источника и приемника должны совпадать.

Функция VarClear работает аналогично, только каждому полю присваивает значение по умолчанию, соответствующее его типу. Это бывает полезно, когда переменная уже использовалась в алгоритме, и ее поля содержат какие-то данные, но в какой-то момент возникла необходимость проинициализировать ее заново.

4.4 Передача параметров

При вызове объекта на исполнение ему могут быть переданы параметры. Количество параметров, их последовательность и типы должны быть явным образом описаны разработчиком в структуре объекта, который их принимает. Для этого служит сегмент «Входные параметры».

Параметры могут передаваться по значению или по ссылке

- При передаче по значению объект работает с локальной копией передаваемой переменной
- При передаче по ссылке объект работает непосредственно с переменной вызывающего кода

При объявлении ссылочного параметра можно не указывать его тип (бестиповая ссылка). В этом случае вызывающий код может передать через параметр переменную любого типа, но в вызываемом коде нужно будет явным образом указать тип посредством оператора CAST.

При описании параметров могут быть заданы их значения по умолчанию

- Значения по умолчанию могут иметь только параметры, передаваемые по значению. Параметры, передаваемые по ссылке, не могут иметь значений по умолчанию.
- Если параметр имеет значение по умолчанию, то все параметры справа от него тоже должны иметь значения по умолчанию

4.4.1 Передача параметров по значению

При передаче параметра по значению производится копирование значения из вызывающего кода в локальную переменную вызываемого объекта. Таким образом, вызываемый объект будет работать с копией передаваемой переменной.

Передача по значению возможна только для простых встроенных типов. Массивы и структуры могут передаваться только по ссылке, и не могут быть переданы по значению.

4.4.2 Передача параметров по ссылке

Параметры, передаваемые по ссылке, не могут иметь значений по умолчанию. Они всегда должны передаваться явным образом.

- Ссылка может быть инициализирована (связана с конкретной переменной) только в вызывающем коде
- Если ссылка инициализирована вызывающим кодом, то перенаправить ее на другую переменную невозможно
- Нет возможности объявить ссылочную переменную, можно получить ссылку только как передаваемый параметр

Обращение к параметру, переданному по ссылке, синтаксически не отличается от обращения к любой другой локальной переменной, т.е. обращение к ссылочному параметру транслируется в обращение к значению той переменной, на которую он ссылается.

Нельзя передавать по ссылке

- Константы
- Результаты промежуточных вычислений

Если параметр передан по ссылке, а вызывающий объект, являющийся владельцем переменной, перестал существовать, то ссылка становится некорректной. В системе нет возможности проверить достоверность такой ссылки, и при разработке следует учитывать это обстоятельство, т.к. обращение к несуществующей переменной может привести к аварийному завершению программы.

4.4.3 Бестиповая ссылка и оператор CAST

При описании параметра, передаваемого по ссылке, можно не указывать его тип. В этом случае ссылка будет бестиповой.

Для бестиповых ссылок определен один единственный оператор – CAST, никаких иных операций для таких переменных не определено. Оператор CAST сообщает компилятору, какой именно тип имеет данная переменная.

При передаче параметров транслятор передает вызываемому коду и их типы (это осуществляется через скрытые параметры). Таким образом, в период исполнения можно проверить тип передаваемого параметра. Но в период трансляции этот тип не известен. Оператор CAST при трансляции сообщает компилятору, с каким типом он должен работать, а в период исполнения проверяет реально переданный тип на соответствие приводимому.

Оператор CAST задает только тип, но не влияет на размерности массивов. Если параметр передан как массив, то после применения оператора CAST его размерности сохранятся.

На этапе трансляции оператор CAST указывает транслятору тип переменной:

1. Оператор CAST применим только к бестиповым ссылкам
2. Область действия оператора CAST – блок
   2.1. В области действия одного оператора CAST вызов другого оператора CAST даст ошибку периода компиляции, т.к. тип переменной уже установлен
   2.2. При выходе из блока ссылка опять превращается в бестиповую, поэтому правомерно применение нового оператора CAST
3. Оператор транслируется в функцию проверки типа на совпадение

На этапе исполнения:

1. Вызываемый экземпляр объекта получает не только значения передаваемых параметров, но и их типы, которые были сохранены в вызывающем коде компилятором
2. Оператор CAST проверяет, является ли указанный тип приводимым к типу фактического параметра. Если является, исполнение сегмента кода продолжается, иначе – прерывается с выдачей сообщения об ошибке.

Пример:

```
// Пусть процедура Proc принимает два параметра: (int nReason, & Data)
{
        if (nReason == 1)
        {
                CAST Data AS int;
                …
        }
        else if (nReason == 2)
        {
                CAST Data AS string;
                …
        }
}
```

4.4.4 Передача массивов

1. При описании параметра, являющегося массивом, размерность данного массива указывается явно
2. При передаче массива в качестве параметра, размерность не указывается (транслятор получает ее из декларации переменной)
   2.1. Размерность реально передаваемого массива должна соответствовать размерности, декларированной в описании параметра, иначе возникнет ошибка периода компиляции
3. Массив может быть передан, только по ссылке
4. В период исполнения для массивов, переданных через параметры, осуществляется проверка на выход за границы массива

4.4.5 Синтаксис

При описании объекта параметры, которые он принимает, перечисляются через запятую в круглых скобках.

[Тип_возврата] (Тип Имя [= Значение_по_умолчанию] [, Следующий параметр …])

или

[Тип_возврата] (Тип & Имя [Размерности массива] [, Следующий параметр …])

или

[Тип_возврата] (& Имя [Размерности массива] [, Следующий параметр …])

где

Тип_Возврата	Тип, значения, которое вернет объект. Если не указан, объект не возвращает значения. Тип возврата может быть только простым встроенный типом. Нельзя вернуть массив или структуру.
Тип	Тип параметра. Для бестиповых ссылок тип не указывается.
&	Если указан, то передача параметра осуществляется по ссылке, если не указан, то по значению
Значение_по_умолчанию	Любая константа указанного типа. Параметры, передаваемые по ссылке, не могут иметь значений по умолчанию.
Имя	Имя параметра.

При вызове объекта передаваемые параметры перечисляются через запятую в круглых скобках.

Пример:

```
// При объявлении процедуры Proc

int (int Var1, datetime & Var2[2][4], int & Var3, string Var4 = “Text”)


// При вызове

datetime        Arr[2][4] = {1,1,1,1,1,1,1,1};
int             nVar = 123;
int             ret = Proc (1, Arr, nVar);

ret = Proc (1, Arr, nVar, “My Text”);

// Пример передачи массива по бестиповой ссылке
// При объявлении процедуры

(&par[2][2], int nType)
{
        if (nType == 0)
        {
                CAST par AS int;

                par [0][0] = 1;
                par [0][1] = 2;
                par [1][0] = 3;
                par [1][1] = 4;
        }
        else if (nType == 1)
        {
                CAST par AS string;

                par [0][0] = "10";
                par [0][1] = "20";
                par [1][0] = "30";
                par [1][1] = "40";
        }
        else
        {
                CAST par AS datetime;
        }
}

// При вызове

        int     i[2][2];

        ProcTest (i, 0);
        MessageBox (ToString (i [0][0]) +  ToString (i [0][1]) +    ToString (i [1][0]) + ToString (i [1][1]));

        string s[2][2];

        ProcTest (s, 1);
        MessageBox (s [0][0] + s [0][1] + s [1][0] + s [1][1]);

        ProcTest (s, 3); // Вернет ошибку преобразования типов
```

4.4.6 Проверка передаваемых параметров

При компиляции объекта осуществляется проверка параметров для всех вызываемых объектов. При этом

1. Общее число фактических параметров не должно быть меньше числа обязательных (не имеющих значений по умолчанию) и не должно превышать общего числа формальных параметров
2. Типы всех фактических параметров должны соответствовать типам формальных параметров, либо быть приводимыми к ним

На этапе исполнения при вызове объекта, также, осуществляется проверка переданных параметров, поскольку нет гарантии, что вызываемый объект не был изменен. При несовпадении числа параметров или их типов возникает ошибка периода исполнения.

4.5 Обзор операторов языка

4.5.1 Все операторы что-либо возвращают

Для числовых типов реализованы арифметические, логические, битовые операторы, а также, операторы сравнения. Для всех типов реализованы операторы присвоения. Следует понимать, что все операторы что-либо возвращают. Это может быть результат промежуточных вычислений, а может быть ссылка на один из операндов. Что именно возвращает каждый конкретный оператор, описано в электронной документации, а здесь мы рассмотрим некоторые примеры.

К примеру, оператор «+» (сложение) возвращает результат промежуточных вычислений. Его тип зависит от типов операндов, но хранится он только в стеке. Если его не положить в какую-либо переменную, он исчезнет, когда свернется стек. Именно поэтому результаты промежуточных вычислений нельзя передавать по ссылке. А вот оператор «++» (инкремент) возвращает ссылку на свой единственный операнд.

```
… // Пусть функция F принимает параметр int по ссылке
int a = 0;

F (a++);        // Корректный код
F (a + 1);      // ошибка
F (123);        // ошибка
```

Рассмотрим другой пример. Оператор присвоения «=» возвращает ссылку на свой левый операнд. Попробуем этим воспользоваться.

```
nItem = TreeGetChild (0);
… // Что-то делаем

while (nItem = TreeGetNext (nItem))
{
        … // Что-то делаем
}
```

В данном примере цикл перебирает все узлы дерева одного уровня и выполняет с ними какие-то операции. Здесь нам не важно, что он с ними делает, интересно то выражение, которое передается оператору цикла while в качестве логического условия. Интересно то, что в нем используется оператор присвоения. Рассмотрим подробнее, как это работает.

Функция TreeGetNext принимает значение переменной nItem, ищет узел дерева, следующий за nItem на том же уровне, и возвращает его идентификатор. Она вернет 0, если не найдет следующего узла. Далее, оператор «=» присваивает переменной nItem то значение, которое вернула функция, и возвращает ссылку на свой левый операнд, т.е. на переменную nItem. Таким образом, после вычисления выражения в переменной nItem будет лежать новое значение, а цикл while проверит то, что лежит в nItem. Впрочем, этот код можно было бы написать так:

```
nItem = TreeGetChild (0);
… // Что-то делаем

while (nItem)
{
        nItem = TreeGetNext (nItem);
        … // Что-то делаем
}
```

4.5.2 Операторы if и switch

Строить ветвления в коде можно с помощью операторов if и switch. Здесь стоит отметить, что все операторы, проверяющие логическое условие, проверяют его на 0. Если выражение, задающее условие, возвращает 0, то условие считается не выполненным. Если же возвращаемое значение не равно 0, то условие выполнено.

Пример:

```
int a = 32;

if (a) // Условие выполнено
{
}

if (a == 32) // Условие выполнено
{
}

if (a == 33) // Условие не выполнено
{
}

if (a - 32) // Условие не выполнено
{
}

if (a - 33) // Условие выполнено
{
}
```

При выполнении логических операций можно использовать встроенные константы TRUE и FALSE. заданы они следующим образом:

```
const int TRUE = 1;
const int FALSE = 0;
```

Следует понимать, что логическое условие может быть выполнено, хотя его результат не всегда может быть равен TRUE.

Пример:

```
int a = 32;

if (a + 1) // Условие выполнено
{
}

if (a + 1 == TRUE) // Условие не выполнено
{
}

if (a + 1 != FALSE) // Условие выполнено
{
}
```

По поводу оператора switch важно запомнить, что выйти из оператора можно только в двух случаях: если оператор switch исполнен полностью и закончился, либо по оператору break.

Пример:

```
int a = 1;

switch (a)
{
	case 1:
		a += 2;
	case 2:
		a += 3;
	break;

	case 3:
		a += 4;
	case 4:
		a += 5;
}
```

В этом примере, если переменная «a» изначально равна 1, то после работы оператора switch она примет значение 6. Если бы она была равна 2, то приняла бы значение 5. А если 3, то – 12.

4.5.3 Операторы циклов

Для построения циклов реализованы операторы for и while.

Синтаксис оператора while достаточно очевиден. Синтаксис оператора for требует некоторых пояснений:

for ([Выражение1] ; Условие ; [Выражение2] [;]) Блок

Блок кода будет исполняться циклически до тех пор, пока выполняется Условие.

Условие представляет собой любое арифметическое или логическое выражение, возвращающее тип BOOL, int или double. Условие считается выполненным, если возвращаемое значение не равно 0.

Выражение1 исполняется один раз непосредственно перед входом в цикл. Как правило, оно предназначено для объявления или инициализации счетчика итераций.

Выражение2 исполняется на каждой итерации цикла в конце блока. Как правило, оно предназначено для изменения счетчика итераций.

```
for (int i = 0; i < 10; i++)
{
// какой-либо код;
}
```

Здесь нужно отметить, что переменная i, объявленная в цикле, не будет доступна за его переделами. Если требуется доступ к ней, следует объявить ее выше:

```
int i = 0;

for (; i < 10; i++)
{
// какой-либо код;
}

i++; // здесь i доступна
```

4.5.3.1 Операторы break и continue

Для изменения последовательности исполнения внутри цикла служат операторы break и continue. Оператор break вызывает выход из цикла, оператор continue – возврат к началу цикла.

```
while (TRUE)
{
	…
	if (Условие 1)
		break;
	else if (Условие 2)
		continue;
	…
}
```

В примере объявлен бесконечный цикл, выход из которого произойдет, только если будет выполнено условие 1, и отработает break. Если же будет выполнено условие 2, то отработает оператор continue, и исполнение перейдет к началу цикла, после чего будет вновь выполнена проверка условия выхода (в нашем примере оно всегда TRUE).

Операторы break и continue можно применять, также, в цикле for.

4.5.4 Оператор return

Оператор предназначен для немедленного завершения процедуры или сегмента кода, т.е. передает управление вызывающему коду. Он, также, может быть использован для возврата значения из процедуры или сегмента.

С помощью оператора return можно вернуть только значение, имеющее простой встроенный тип, т.е. нельзя вернуть структуру, массив или ссылку.

4.6 Поддержка SQL

По умолчанию, если при выполнении SQL-кода возникает серверная ошибка, исполнение текущего сегмента прерывается с выдачей сообщения на экран. Изменить это поведение можно с помощью функции SetSqlMuteMode.

Успешность выполнения последнего SQL-оператора можно проверить с помощью функции SqlSuccess. Формулировку ошибки, возвращаемую SQL-сервером, можно получить с помощью функции GetSqlErrorString.

Ограничения:

- максимальное число переменных, подставляемых в SQL-запрос – 50
- максимальное число полей, которые может вернуть SQL-запрос – 50

При чтении бинарных данных из БД система выделяет буфер. По умолчанию его размер составляет 0x1FFFFF (2 МБ). Это значение можно изменить с помощью функции SetBufferSize.

4.6.1 Работа с источниками данных

При запуске среды исполнения всегда осуществляется соединение с источником данных. Его использует загрузчик ресурсов для загрузки кодов исполняемого приложения. Это соединение не доступно программисту и не может быть изменено.

Используя ту же авторизацию, среда исполнения автоматически создает второе соединение с тем же источником данных. Это соединение называется **главным соединением** и предназначено для обращения к данным БД. Именно на этом соединении будет отрабатывать весь SQL-код внутри объекта. Это соединение закрывается автоматически при завершении программы, и не может быть подменено или закрыто из прикладного кода. Его дескриптор (тип HDB) можно получить посредством функции GetMainConnect.

При активации роли приложения (функция AppRole) на главном соединении загрузчик ресурсов, также, перенастраивается на новое имя пользователя БД. Это необходимо для корректной работы механизма замещения объектов. При этом выполняется сброс кэша ресурсов загрузчика.

Существует возможность создавать дополнительные соединения: функции OpenConnect, CloseConnect. Функция OpenConnect создает новое соединение и возвращает его дескриптор, а функция CloseConnect закрывает соединение с указанным дескриптором.

Для работы с данными БД каждый объект прикладного кода использует собственную копию дескриптора соединения. Когда один объект вызывает другой, то система передает ему дескриптор соединения вызывающего объекта, и в структуре вызываемого объекта создается копия этого дескриптора. Функция GetConnect возвращает текущую копию дескриптора соединения объекта. Функция SetConnect устанавливает ее в новое значение. После этого объект начинает использовать новое соединение для всех операторов SQL, а также, все вызываемые объекты будут наследовать этот дескриптор. Чтобы восстановить дескриптор соединения, нужно повторно вызвать SetConnect со старым значением.

Работа приложения начинается с вызова объекта типа модуль. При вызове модуля, среда исполнения передает ему дескриптор главного соединения. Копия этого дескриптора сохраняется в структуре объекта и будет передаваться всем вызываемым объектам. Таким образом, если не вызывать функцию SetConnect то все приложение для работы с данными будет использовать главное соединение. Если же на каком-то этапе вызвать функцию SetConnect, то с этого момента сам объект начинает работать с другим соединением, и все вызываемые объекты будут его наследовать. Из этого следует, к примеру, что в одном объекте можно использовать другие объекты, которые могут работать с разными соединениями. Например, в диалоге можно разместить два списка, каждый из которых будет работать с собственным соединением.

4.6.2 Оператор вызова SQL

Оператор предназначен для вызова на исполнение произвольного кода на SQL.

```
SQL (выражение1, выражение2, …, выражениеN)
{
        Код на SQL
}
```

Параметры:

- выражение1, выражение2, …, выражениеN – необязательные параметры, предназначенные для возврата значений из блока кода на SQL
    - Выражение – любое выражение на языке X2, возвращающее ссылку на существующую переменную (в простейшем случае – имя переменной)
        - Выражение не должно возвращать константу или результат промежуточных вычислений
        - Переменная должна иметь простой встроенный тип
    - Если параметры не указаны, никаких значений из блока кода на SQL возвращено не будет
    - Число и типы параметров должны соответствовать числу и типам возвращаемых значений, заданных в SQL-запросе
    - Общее число параметров не должно превышать 50
    - Если запрос возвращает NULL в каком-либо поле, то переменная, связанная с этим полем получает значение по умолчанию, соответствующее типу переменной. Нет способа выявить эту ситуацию, поэтому рекомендуется писать SQL-запрос так, чтобы он не возвращал значений NULL.
- Код на SQL – произвольный код на SQL
    - Перед отправкой на SQL-сервер в код на SQL может быть выполнена подстановка значений переменных
    - Код на SQL может быть разбит на пакеты с помощью ключевого слова GO
    - Если код на SQL разбит на пакеты, то возврат значений производится только из последнего пакета
    - Если последний пакет возвращает несколько наборов данных, то будут использованы данные из первого набора
    - Если последний набор данных имеет более одной строки, то будет использована первая строка набора

4.6.2.1 Подстановка значений переменных в код на SQL

Перед отправкой на SQL-сервер в код на SQL может быть выполнена подстановка переменных. Подстановка может выполняться в любой точке кода на SQL, но не выполняется в строковые литералы SQL (окруженные одинарными кавычками). Места подстановки должны быть помечены явным образом посредством символа «:». При подстановке могут выполняться вычисления, т.е. в качестве подстановочных значений можно использовать выражения на языке X2. В этом случае выражение должно быть окружено круглыми скобками.

Подстановка может выполняться двумя способами:

- Строковая подстановка
- Ссылочная подстановка (binding)

В случае строковой подстановки значение указанной переменной преобразуется в строку, и полученная строка подставляется в код на SQL. Для преобразования может быть указана маска, задающая формат преобразования, либо использована функция ToString. Если маска не указана явно, используется маска по умолчанию.

В случае ссылочной подстановки (binding) значение переменной не преобразуется в строку. Вместо этого драйверу SQL-сервера передается ссылка на буфер, связанный с именем переменной. Этот способ служит для параметризации SQL-запросов, и в ряде случаев позволяет повысить эффективность их отработки за счет сокращения серверного КЭШ-а запросов и, соответственно, времени на компиляцию запроса. В большинстве случаев это дает эффект при подстановке в секцию WHERE запроса.

Общее число подставляемых значений не должно превышать 50.

Синтаксис:

```
    Код на SQL … :VarName[Маска] … Код на SQL
либо
    Код на SQL … :&VarName … Код на SQL
либо
    Код на SQL … :(Выражение) … Код на SQL
либо
    Код на SQL … :&(Выражение) … Код на SQL
```

- Символ «:» указывает на место подстановки
- VarName – Имя существующей переменной
    - Переменная должна иметь простой встроенный тип. Нельзя ссылаться на структуры, массивы или их поля.
    - Если требуется сослаться на поле структуры или элемент массива, следует использовать конструкцию :(Выражение) или :&(Выражение)
- [Маска] – Маска преобразования в строку
    - Необязательный параметр. Если параметр не указан, используется маска по умолчанию для данного типа.
- Символ «&» указывает, что должна быть выполнена ссылочная подстановка

- Выражение – произвольное выражение на языке X2, возвращающее значение простого встроенного типа
    - Если указан символ «&» (ссылочная подстановка), то выражение должно вернуть ссылку на существующую переменную

Пример 1:

```
string strIn = "Текст";
string strOut;

// Неправильно. В строковые литералы подстановка не выполняется.
SQL (strOut)
{
	SELECT ':strIn'
}

// Правильно. Используем ссылочную подстановку,
// одинарные кавычки не нужны, поскольку сервер сам определит тип данных.
SQL (strOut)
{
	SELECT :&strIn
}
```

Пример 2:

```
struct TMyType
{
	int m_nInt;
};

TMyType		mt;
int		i;

mt.m_nInt = 10;

// Неправильно. Обращение к полю структуры или элементу массива
// является выражением. Выражения нужно окружать круглыми скобками
SQL (i)
{
	SELECT :mt.m_nInt
}

// Правильно
SQL (i)
{
	SELECT :(mt.m_nInt)
}
```

4.6.2.2 Комментарии в коде на SQL

В коде на SQL могут использоваться комментарии.

Закомментированные строки при исполнении не выкусываются из кода, а отправляются на сервер.

В закомментированные строки подстановка переменных не выполняется, т.е. транслятор распознает такие комментарии.

Блочные комментарии (/**/) могут использоваться в любом месте кода на SQL. Для строковых комментариев (--) действует такое правило: символ, завершающий блок кода на SQL не должен находиться в закомментированной строке.

Пример:

```
// Неправильно. Символ, завершающий блок, находится в закомментированной строке

SQL ()
{ select 1 -- where 1 }

INSERT INTO TblName (FldName) VALUES (:MyVar) -- VALUES (:MyVar1);

// Правильно

SQL ()
{
        select 1 -- where 1
}

INSERT INTO TblName (FldName) VALUES (:MyVar) /*VALUES (:MyVar1)*/;
```

4.6.2.3 Встроенные операторы SQL

Некоторые операторы SQL являются встроенными. Это:

- INSERT
- UPDATE
- DELETE
- TRUNCATE

Язык позволяет использовать их непосредственно, т.е. без вызова конструкции SQL () {…}. Например, вызов

```
INSERT INTO TblName (FldName) VALUES (:MyVar);
```

эквивалентен вызову

```
SQL ()
{
	INSERT INTO TblName (FldName) VALUES (:MyVar)
};
```

Внутри встроенных операторов SQL работает подстановка переменных по тем же правилам, что и внутри конструкции SQL () {…}.

Ограничения:

- Первое слово оператора должно быть либо полностью в верхнем (INSERT), либо полностью в нижнем (insert) регистре
- Оператор должен заканчиваться символом «;»

4.6.3 Работа с наборами данных

Для работы с наборами данных служит встроенный объект – DataSet. С помощью данного объекта можно получить в контекст исполнения программы произвольный набор данных, возвращаемый оператором select.

Экземпляр объекта DataSet идентифицируется дескриптором. Для дескриптора определен специальный тип данных – HDS.

Для работы с объектом определен следующий набор функций:

- оператор OpenDataSet
- функция CloseDataSet
- функция IsEOF
- функция Fetch
- функция IsFieldNull

Общая схема работы с набором данных представлена ниже.

```
string   strName;
int      nType;

HDS hDS = OpenDataSet (strName, nType)
{
        SELECT TOP 3 [ObjectName], [id_ObjectType] FROM [x2Objects]
};

if (hDS)
{
        while (!IsEOF(hDS))
        {
                MessageBox(strName + " " + ToString(nType));
                Fetch (hDS);
        }
        CloseDataSet(hDS);
}
```

4.6.4 Перехват ошибок SQL

По умолчанию при возникновении ошибки SQL система выводит сообщение об ошибке и прерывает исполнение текущего сегмента кода. Это поведение можно изменить с помощью функции SetSqlMuteMode. Если установить режим вывода сообщений в FALSE, то система не будет выводить сообщений об ошибках, и будет продолжать исполнение. Это позволяет программисту самостоятельно обрабатывать ошибки SQL.

Пример

В конструкторе модуля создаем структуру данных с помощью прикладной процедуры DemoCreateStruct. В случае ошибки сервера процедура вернет FALSE.

```
{
	// Будем перехватывать сообщений об ошибках SQL Server
	SetSqlMuteMode(TRUE);

	// Создадим структуру данных
	BOOL bResult = DemoCreateStruct();

	if (!bResult)
	{
		// Ошибка SQL
		string strError = GetSqlErrorString ();

		MessageBox
		(
		"При подготовке структуры данных произошла ошибка\r\n\r\n" + strError + "\r\n\r\n" +
		"Приложение будет завершено.",
		"Ошибка",
		MB_ICONERROR | MB_OK
		);
	}
	else
	{
		SetSqlMuteMode(FALSE);
	}

	// Если конструктор модуля вернет FALSE, приложение будет завершено
	return bResult;
}
```

4.7 Коллекции (динамические массивы)

Коллекция, как и массив, представляет собой объект, предназначенный для хранения данных, но в отличие от массива:

- Элементы коллекции могут иметь разные типы данных
- Коллекция одномерна, т.е. не может иметь несколько измерений, как массив
- Размер коллекции может меняться динамически по мере добавления в нее новых элементов

В частном случае, когда все элементы коллекции имеют один и тот же тип, она может быть представлена, как одномерный динамический массив. Это может быть удобно для хранения данных, загруженных из БД, если элементом коллекции является структура, соответствующая строке набора данных.

Коллекция идентифицируется дескриптором, уникальным в рамках сессии. Для дескриптора определён тип данных – HCOL.

Доступ к элементу коллекции осуществляется по его порядковому номеру в коллекции (индексу). Отсчет начинается с нуля.

Для работы с коллекциями определён следующий набор функций:

- функция CreateCollection – создает коллекцию
- функция DropCollection – уничтожает коллекцию
- функция CollectionGetSize – возвращает количество элементов в коллекции
- функция CollectionGetAt – возвращает элемент коллекции
- функция CollectionSetAt – перезаписывает элемент коллекции
- функция CollectionAdd – добавляет новый элемент в коллекцию
- функция CollectionRemoveAt – удаляет элемент коллекции
- функция CollectionClear – удаляет все элементы коллекции

При добавлении нового элемента в коллекцию (функция CollectionAdd) данные из переменной копируются во внутренний буфер элемента. При извлечении данных (функция CollectionGetAt) они из внутреннего буфера копируются в переменную.

Основной принцип работы с коллекцией: какой тип положил в коллекцию, такой тип и извлек.

Пример работы с коллекцией можно найти в демонстрационном модуле. Путь по меню: Списки\Редактирование\Отложенное сохранение.

Ниже приводятся еще два примера.

Пример 1: // Используем коллекцию для хранения данных разных типов

```
HCOL hCol = CreateCollection ();

for (int i = 0; i < 3; i++)
{
	string strData = "str_" + ToString(i);

	CollectionAdd (hCol, i);
	CollectionAdd (hCol, strData);
}

for (int i = 0; i < CollectionGetSize(hCol); i++)
{
	int	n;
	string	str;

	CollectionGetAt(hCol, i, n);
	CollectionGetAt(hCol, i++, str);
	MessageBox("string: " + str + "\r\nint: " + ToString(n));
}

DropCollection (hCol);
```

Пример 2: // Используем коллекцию для загрузки данных из таблицы

```
struct TObject
{
	int	m_nType;
	string	m_strName;
};

TObject		tObj;
HCOL		hCol = CreateCollection ();

HDS	hDS = OpenDataSet (tObj.m_nType, tObj.m_strName)
{
	SELECT TOP 3 [id_ObjectType], [ObjectName] FROM [x2Objects]
};

if (hDS)
{
	while (!IsEOF(hDS))
	{
		CollectionAdd(hCol, tObj);
		Fetch (hDS);
	}
	CloseDataSet(hDS);
}

for (int i = 0; i < CollectionGetSize(hCol); i++)
{
	CollectionGetAt(hCol, i, tObj);
	MessageBox(tObj.m_strName + " " + ToString(tObj.m_nType));
}

DropCollection (hCol);
```

4.8 Файлы

В системе определены два набора функций, работающих с файлами: функции работы с файлом и функции доступа к файловой системе.

4.8.1 Функции доступа к файловой системе

Функции доступа к файловой системе позволяют работать с одиночным файлом или множеством файлов, имена которых соответствуют заданной маске.

- FileCopy – копирование файлов
- FileMove – перемещение файлов
- FileRename – переименование файлов
- FileDelete – удаление файлов
- FileGetAttributes – получение атрибутов
- FileSetAttributes – установка атрибутов
- FileClearAttributes – снятие атрибутов
- FileCreatePath – создание пути

Пример:

```
FileRename("d:\\*.txt", "Boss_*.txt", FALSE, FALSE);
```

Функции, вызывающие стандартные диалоги ОС:

- GetOpenFileName – диалог открытия файла
- GetSaveFileName – диалог сохранения файла
- BrowseForFolder – диалог выбора директории

Пример вызова диалога открытия файла можно найти в демонстрационном модуле. Путь по меню: Диалоги\Управляющие элементы\Изображения.

Функции перечисления файлов FindFirstFile, FindNextFile и FindClose предназначены для перебора всех файлов, соответствующих заданной маске.

Общая схема использования функций перечисления такова. Вначале вызывается функция FindFirstFile, которой передается маска. Функция находит первый файл, соответствующий маске и возвращает его имя, атрибуты и некий целочисленный идентификатор. Далее в цикле вызывается функция FindNextFile, которой передается идентификатор, возвращенный функцией FindFirstFile. Функция вернет TRUE, если найдет следующий файл, соответствующий маске, и FALSE, если такого файла нет. По окончании цикла вызывается функция FindClose, которая завершает перечисление файлов.

Пример использования функций перечисления файлов можно найти в демонстрационном модуле. Путь по меню: Деревья\Файлы.

4.8.2 Функции работы с файлом

Система поддерживает работу с двумя видами файлов: дисковым файлом и файлом в памяти. Файл в памяти аналогичен дисковому файлу, только расположен в оперативной памяти. Ввиду этого, для файлов в памяти неприменимы некоторые функции, такие, как FileGetName, FileGetPath, FileFlush и пр. (см. описание соответствующих функций). В остальном техника работы с файлами в памяти не отличается от техники работы с дисковыми файлами, поэтому файл в памяти может быть полезен в качестве временного буфера, с которым можно работать так же, как и с обычным файлом.

Для открытия файла служит функция FileOpen. Она создает в системе объект, обеспечивающий работу с файлом, и возвращает его дескриптор. Функция имеет две реализации: одна открывает дисковый файл, другая – файл в памяти.

Файл, открытый функцией FileOpen, следует закрыть функцией FileClose, иначе он будет открыт до завершения сессии.

Максимальный размер файла не должен превышать 2047 Мб.

4.8.2.1 Пример 1. Работа с текстовым файлом

В оперативной памяти система хранит строки в кодировке UTF16. Если требуется работать с файлом в другой кодировке, то нужно указать желаемую кодировку, тогда в ходе операции строка будет перекодирована. Следует помнить, что для перекодирования система выделяет в оперативной памяти дополнительный буфер, размер которого может превышать размера исходного буфера.

Функция FileWrite для строки имеет параметр BOOL bWriteSize. Если его установить в TRUE, то перед записью в файл самой строки, функция сохранит ее длину. При работе с текстовым файлом этот параметр должен быть установлен в FALSE. Аналогично для функции FileRead параметр nSize должен быть установлен в 0.

В примере, приведенном ниже, мы работаем с файлом в кодировке 1251.

```
HFILE hFile = FileOpen ("d:\\MyFile.txt", FOF_CREATE | FOF_EXCLUSIVE | FOF_WRITE);

if (hFile)
{
	FileWrite (hFile, "Это текстовый файл!", FALSE, 1251);
	FileClose (hFile);
}

hFile = FileOpen ("d:\\MyFile.txt", FOF_READ);

if (hFile)
{
	string strText;

	FileRead (hFile, strText, 0, 1251);
	FileClose (hFile);
	MessageBox (strText);
}
```

4.8.2.2 Пример 2. Сохранение в файл набора данных

Иногда требуется сбросить в файл какие-то произвольные данные, для их дальнейшего использования. Например, это могут быть настройки программы или протокол работы какого-то алгоритма. Важно то, что у нас есть какие-то переменные, и мы хотим сбросить их значения в файл, а потом восстановить их из файла.

Принципы работы с таким файлом таковы:

- в какой последовательности мы пишем, в такой последовательности и читаем
- при сохранении строк сохраняем и размер строки

```
string      strUser;
int         nUserID;
string      strApp;
datetime    dtDate;
string      strSessionUser;

SQL (strUser, nUserID, strApp, dtDate, strSessionUser)
{
        SELECT SUSER_SNAME(), SUSER_ID(), APP_NAME(), GETDATE(), SESSION_USER
}

HFILE hFile = FileOpen ("d:\\MyFile.dat", FOF_CREATE | FOF_EXCLUSIVE | FOF_WRITE);

if (hFile)
{
        FileWrite (hFile, strUser, TRUE);
        FileWrite (hFile, nUserID);
        FileWrite (hFile, strApp, TRUE);
        FileWrite (hFile, dtDate);
        FileWrite (hFile, strSessionUser, TRUE);
        FileClose (hFile);
}

strUser = strApp = strSessionUser = "";
nUserID = -1;
dtDate = '01-01-1900';

hFile = FileOpen ("d:\\MyFile.dat", FOF_READ);

if (hFile)
{
        FileRead (hFile, strUser);
        FileRead (hFile, nUserID);
        FileRead (hFile, strApp);
        FileRead (hFile, dtDate);
        FileRead (hFile, strSessionUser);
        FileClose (hFile);
        MessageBox
        (
                "User = " + strUser + "\r\n" +
                "UserID = " + ToString (nUserID) + "\r\n" +
                "App = " + strApp + "\r\n" +
                "Date = " + ToString (dtDate) + "\r\n" +
                "SessionUser = " + strSessionUser
        );
}
```

4.8.2.3 Пример 3. Работа с бинарным форматом

Система позволяет загружать из файла данные двоичного формата и сохранять их в файл. Это могут быть, к примеру, файлы пакета MS Office, изображения или любые другие данные. Для временного хранения в памяти двоичных данных используется двоичный буфер (см. раздел «Двоичный буфер (объект Binary)»).

Пример, демонстрирующий загрузку пиктограммы из файла и ее дальнейшее использование можно найти в демонстрационном модуле по пути «Диалоги\ Управляющие элементы\Изображения».

4.9 Двоичный буфер (объект Binary)

Объект Binary представляет собой буфер для хранения данных, полученных из полей БД типа varbinary, а также, дисковых файлов. Размер буфера зависит от объема загруженных в него данных и не может превышать значения 2047 Мб.

Функции работы с объектом:

- BinaryCreate – создает экземпляр объекта и возвращает его дескриптор
- BinaryDrop – удаляет экземпляр объекта
- BinaryGetSize – возвращает текущий размер объекта
- BinaryClear – удаляет все данные из буфера и обнуляет его размер

Функций непосредственного доступа к содержимому буфера объекта Binary не предусмотрено, но есть возможность:

- загрузить в буфер данные из файла или БД
- записать данные из буфера в файл или БД
- поместить в буфер значения переменных X2 и извлечь их назад
- связать буфер с управляющим элементом «Пиктограмма», если данные в буфере являются пиктограммой

После создания экземпляр объекта существует до момента его явного удаления, либо до завершения сессии.

Общий сценарий работы с двоичным буфером таков:

- Создаем буфер и сохраняем его дескриптор в переменной hBin
- Загружаем в буфер данные
    - Из таблицы – с помощью оператора SQL (hBin) { SELECT …}
    - Либо из файла – с помощью функции FileRead
- Используем буфер
    - Связываем с управляющим элементом «Пиктограмма», если данные в буфере являются пиктограммой
    - Либо сохраняем в таблицу с помощью оператора INSERT … VALUES (:&hBin). Обратите внимание, дескриптор в оператор вставки должен передаваться по ссылке!
    - Либо сохраняем в файл с помощью функции FileWrite
- Удаляем буфер, когда он больше не нужен

Если мы хотим сохранить какой-то набор переменных в поле таблицы типа varbinary, то нам нужно:

- Создать файл в памяти (FileOpen без параметров)
- Сбросить в него значения переменных (функция FileWrite)
- Перегрузить его содержимое в двоичный буфер (функция FileRead)

В примере, приведенном ниже, из БД загружается изображение в формате jpg и сохраняется в дисковый файл. Аналогично можно провести обратную операцию, т.е. загрузить данные из файла и сохранить их в поле таблицы.

Пример:

```
HBIN hBin = BinaryCreate ();

// Загружаем из БД картинку в формате jpg
SQL (hBin)
{
	SELECT TOP 0 [Data] FROM [dbo].[x2Image] WHERE [FileExt] = 'jpg'
}

// Сохранение в дисковый файл
if (BinaryGetSize (hBin) > 0)
{
	string strName;

	if (GetSaveFileName(strName))
	{
		HFILE hFile = FileOpen (strName, FOF_CREATE | FOF_WRITE);

		if (hFile)
		{
			FileWrite(hFile, hBin);
			FileClose(hFile);
		}
	}
}
else
{
	MessageBox
	(
		"В таблице x2Image не найдено изображения формата jpg!", "Внимание"
	);
}

BinaryDrop(hBin);
```

4.10 Поддержка ActiveX Scripting

Для работы со скриптовыми языками предусмотрены следующие функции и операторы:

- Функции OpenVBS и OpenJS – создание новой сессии VBScript и JScript соответственно
- Функции CloseVBS и CloseJS – завершение существующей сессии VBScript и JScript соответственно
- Операторы VBS и JS – исполнение блока кода на VBScript и JScript соответственно
- Функция GetScriptVar – получение значения переменной VBScript или JScript в контекст вызова X2

Для исполнения сценария скрипта необходимо вначале открыть сессию и получить ее дескриптор. В дальнейшем он будет использоваться в качестве идентификатора сессии. Для дескрипторов сессии введены типы HVBS (для сессии VBScript) и HJS (для сессии JScript).

После открытия сессии можно исполнить скриптовый сценарий с помощью операторов VBS и JS. Оператор можно вызывать для одной открытой сессии многократно. В сценарий можно выполнять подстановку переменных X2. Подстановка выполняется только по значению.

После отработки сценария можно запросить значения переменных, объявленных в сценарии, с помощью функции GetScriptVar.

Таким образом, в скриптовый сценарий можно передавать параметры путем подстановки переменных и возвращать значения с помощью функции GetScriptVar.

Если открытая сессия больше не нужна, ее следует завершить с помощью функции CloseVBS или CloseJS. Если этого не сделать, открытая сессия будет существовать до момента завершения всего приложения.

4.10.1 Типы данных HVBS и HJS

Типы HVBS и HJS предназначены для хранения дескрипторов сессий VBScript и JScript соответственно. Эти типы являются производными от типа int и автоматически к нему приводятся.

4.10.2 Функции OpenVBS и OpenJS

Функции предназначены для создания новой сессии VBScript и JScript соответственно.

Прототип:

HVBS OpenVBS ()

HJS OpenJS ()

Входные параметры:

Нет входных параметров.

Возвращаемое значение:

В случае успеха – дескриптор сессии VBScript или JScript соответственно. В противном случае – 0.

4.10.3 Операторы VBS и JS

Операторы предназначены для исполнения кода на VBScript или JScript соответственно.

Формат вызова:

VBS (HVBS hVBS) { Код на VBScript }VBS

или

VBS (HVBS hVBS) { Код на VBScript }

JS (HJS hJS) { Код на JScript }JS

Входные параметры:

- hVBS – дескриптор сессии VBScript (возвращается функцией OpenVBS)
- hJS – дескриптор сессии JScript (возвращается функцией OpenJS)
- Код – произвольный фрагмент кода на VBScript или JScript соответственно. В код выполняется подстановка переменных X2.

4.10.3.1 Подстановка переменных

Перед исполнением скрипта в код может быть выполнена подстановка переменных. Подстановка может выполняться в любой точке кода. Места подстановки должны быть помечены явным образом посредством конструкции «:(Выражение)».

Выполняется только строковая подстановка. Подстановка по ссылке невозможна.

Общее число подставляемых значений не должно превышать 50.

Синтаксис:

```
Код … :(Выражение) … Код
```

Подстановка:

- Символ «:» указывает на место подстановки
- Выражение – произвольное выражение на языке X2, возвращающее значение простого встроенного типа
    - В качестве выражения может выступать имя переменной

Примечания:

- *Недопустима подстановка по схеме «Код … :VarName … Код»*
- *Если за символом «:» следует символ, отличный от «(», то «:» не считается меткой подстановки и игнорируется. В частности, конструкция «:&(» будет проигнорирована и не вызовет ошибки трансляции.*

4.11 Работа с буфером обмена Clipboard

Для работы с буфером обмена Windows определен набор функций:

- ClipboardGetPicture – получить изображение из буфера обмена
- ClipboardGetText – получить текст из буфера обмена
- ClipboardSetPicture – поместить изображение в буфер обмена
- ClipboardSetText – поместить текст в буфер обмена
- ClipboardClear – удалить все данные из буфера обмена

5 Общие принципы разработки

Приложение на X2 представляет собой набор взаимосвязанных объектов. Самый первый объект, который вызывается на исполнение – это модуль. Из модуля вызываются другие объекты. Это можно сделать через меню, связанное с модулем, либо непосредственно из кода модуля. Вызываемый объект также может вызывать другие объекты и т.д. Объекты имеют обработчики событий. Изначально они пусты. Программист настраивает поведение объекта, реализуя нужные ему обработчики. Таким образом и строится приложение.

Объекты X2 делятся на визуальные и не визуальные. Визуальные объекты отображаются на экране. Не визуальные – не отображаются. Примером не визуальных объектов могут служить процедуры и декларации. Примером визуальных – списки, диалоги, меню.

Жизненный цикл экземпляра визуального объекта состоит из нескольких фаз:

- Создание экземпляра
- Отображение на экране его визуальной части
- Уничтожение экземпляра

Создается экземпляр объекта с помощью оператора создания экземпляра. Оператор возвращает дескриптор экземпляра. После того, как экземпляр объекта создан, к нему можно обращаться, используя полученный дескриптор. Например, создав экземпляр объекта, можно, используя полученный дескриптор, изменить какие-либо его свойства, а потом отобразить его на экране. Либо можно отправить ему сообщение, посредством которого инициировать какой-либо алгоритм, реализованный внутри объекта (см. раздел «Обработка сообщений».)

Для отображения созданного экземпляра объекта на экране используются функции, задающие режим отображения:

- Функции общего назначения:
    - ModalFrame – модальный фрейм
    - ShowFrame – немодальный фрейм
    - ShowPane – прилипающая панель
- Специализированные функции:
    - LookupFrame – создает модальный фрейм с предопределенным набором опций и позиционирует его на экране под заданным управляющим элементом. Функция предназначена для отображения выпадающего списка управляющего элемента Lookup, построенного на обработчиках событий.
    - ListEditCurCell – визуализирует указанный диалог и позиционирует его в текущую ячейку списка. Функция предназначена для отображения диалога редактирования ячейки списка, если для колонки установлен флаг LFF_EXTERNAL_EDITOR (использовать внешний редактор).

Функции отображения создают окно объекта, после чего вызывают его сегмент инициализации OnInit.

После создания окна экземпляр объекта начинает откликаться на действия пользователя и события системы. Окно существует до момента его закрытия. Закрытие окна может быть инициировано пользователем, либо вызвано программным путем.

При уничтожении объекта вызывается его деструктор. После этого дескриптор экземпляра объекта становится недействительным (см. функцию IsObject).

5.1 Модуль, главное меню и главное окно приложения

Один экземпляр среды исполнения RPExec.exe может исполнять только один модуль. Чтобы запустить второй модуль, нужно запустить второй экземпляр RPExec.exe.

Модуль запускается только через командную строку (ключ –M(m)). Его нельзя запустить программным путем из кода на X2. Модуль может принимать параметры.

При запуске модуль создает окно. Это окно называется главным окном приложения. Если закрыть главное окно, работа модуля завершится.

Главное меню приложения – это меню, связанное с текущим модулем. Оно появляется при запуске модуля в линейке меню главного окна. При активации какого-либо объекта X2 данное меню может быть, либо полностью заменено, либо достроено справа пунктами меню активного объекта (см. раздел «Активный объект»). То, как именно будет вести себя меню (достраиваться или замещаться), зависит от режима визуализации объекта.

При запуске модуля события происходят в такой последовательности:

- загружается модуль и отрабатывает его конструктор
- загружается меню, связанное с модулем (главное меню), и отрабатывает его конструктор
- появляется главное окно приложения
- отрабатывает OnInit меню
- отрабатывает OnInit модуля

Следует отметить, что OnInit меню будет отрабатывать каждый раз, когда меню активируется. Это может происходить при смене активного объекта.

Если требуется активировать роль приложения на сервере (application role), то это следует сделать в конструкторе модуля. Поскольку он отрабатывает до загрузки главного меню, то для главного меню отработает механизм замещения объектов.

Если конструктор модуля вернет FALSE, то экземпляр объекта не будет создан, и приложение завершится. В этом случае главное окно не появится на экране.

Если OnInit модуля вернет FALSE, то приложение, также, завершится. Но OnInit модуля отрабатывает после создания главного окна. Соответственно, в OnInit можно вызывать визуальные объекты, например, список или диалог. Если этот объект вызван в модальном режиме, то исполнение OnInit модуля будет приостановлено до закрытия окна этого объекта.

Деструктор модуля вызывается при закрытии приложения после уничтожения главного окна.

5.1.1 Параметры модуля

Модуль принимает один строковый параметр, который может быть передан через командную строку (ключ –I(i)), и не возвращает значения.

Прототип:

(string strInput)

Строка strInput, переданная в модуль, доступна во всех сегментах модуля.

5.2 Идентификация экземпляров объектов

Для идентификации экземпляров объектов используются дескрипторы. Дескриптор имеет тип данных HOBJECT и возвращается оператором создания экземпляра объекта. Каждый экземпляр имеет собственный дескриптор. Например, два экземпляра списка List1 будут иметь два разных дескриптора. Уникальность дескрипторов обеспечивается в рамках сессии среды исполнения. В рамках одной сессии не может возникнуть двух одинаковых дескрипторов, относящихся к разным экземплярам. В разных сессиях дескрипторы могут дублироваться, поэтому дескриптор, сохраненный в файл или в БД в одной сессии, в другой сессии может оказаться недействительным или же задействованным под экземпляр объекта другого типа.

5.2.1 Тип данных HOBJECT

Тип данных HOBJECT является производным от типа int. Для него определены операции, определенные для типа int, но имеют смысл только две из них: «==» и «!=».

Тип int не приводится к типу HOBJECT.

Получить дескриптор текущего экземпляра объекта можно с помощью функции GetHandle.

5.3 Создание экземпляра объекта

Для разных типов объектов определены разные способы создания экземпляра.

- **Модуль.** Имя модуля передается среде исполнения через командную строку, либо посредством указания в диалоге соединения с сервером. Экземпляр модуля создается системным кодом среды исполнения и существует до завершения сессии. Другого способа создания экземпляра модуля нет. В среде исполнения одновременно может существовать только один экземпляр модуля. Получить дескриптор модуля можно с помощью функций GetHandle и GetModuleHandle.
- **Процедура.** Нет специального оператора создания экземпляра процедуры. Процедура вызывается из кода X2 просто путем указания ее имени и параметров. Экземпляр процедуры создается системным кодом среды исполнения и существует до момента завершения процедуры. Как и все прочие объекты, каждый экземпляр процедуры тоже имеет свой собственный дескриптор. Он может оказаться полезным при рекурсивных вызовах. Получить его можно с помощью функции GetHandle.
- **Визуальные объекты.** Для каждого типа объекта существует собственный оператор создания: BROWSER – для списков, DIALOG – для диалогов и т.д. Каждый из них описывается в соответствующем разделе документации.

Каким бы способом ни создавался экземпляр объекта, автоматически или посредством оператора создания, при этом выполняются следующие операции:

- Загрузка ресурса объекта. Первичная загрузка производится из БД, далее – в зависимости от того, включен ли режим кеширования ресурсов.
- Выделяется память для внутреннего представления экземпляра объекта
- Создаются локальные переменные экземпляра
- Копируются значения передаваемых параметров в локальные переменные
- Вызывается на исполнение конструктор (таким образом, конструктор отрабатывает до создания окна объекта)

5.4 Визуализация экземпляра объекта

5.4.1 Общие сведения

Объекты могут визуализироваться, двумя способами:

- как дочерние в составе родительских объектов
- в собственном фрейме

Дочерние объекты визуализируются автоматически в ходе визуализации родительского объекта. Например, если список включен в диалог, как управляющий элемент, то его визуализация происходит при визуализации самого диалога, и для этого не требуется вызывать какие-либо операторы визуализации. Дочерние объекты не имеют полосы заголовка, и координаты их окон привязаны к координатам окна родительского объекта. Они перемещаются только вместе с родительским окном и не могут быть закрыты без закрытия окна родителя.

Объект типа «Меню» всегда визуализируется, как дочерний. Меню визуализируется при активизации объекта, с которым оно связано.

Объект можно, также, визуализировать в отдельном фрейме. Фрейм представляет собой служебное окно, являющееся контейнером для отображения окна объекта. При закрытии фрейма уничтожается и экземпляр объекта, который визуализирован в данном фрейме. Можно создать фрейм двух видов: обычный фрейм и панель.

Обычный фрейм имеет заголовок, он способен перемещаться в рамках главного окна, а также, его можно растягивать и сжимать, минимизировать и максимизировать.

Основное отличие обычного фрейма от панели в том, что панель можно не только перемещать в рамках главного окна, но и прикреплять к его границам. Каким именно будет поведение панели, зависит от параметров, переданных функции создания панели.

Обычный фрейм может быть вызван в модальном или немодальном режиме. Панель всегда создается в немодальном режиме.

5.4.2 Модальный и немодальный режим

Модальный фрейм при создании блокирует все окна, существующие на момент его появления. Это относится только к окнам, отображающим объекты X2, служебные окна не блокируются. В заблокированном состоянии окно перестает реагировать на пользовательский ввод, но продолжает реагировать на события системы и вызовы функций X2. При закрытии модальный фрейм возвращает все заблокированные окна в их исходное состояние блокировки. В отличие от модального, немодальный фрейм не блокирует окна.

При создании, как модальный, так и немодальный фрейм активизирует объект, вызванный в данном фрейме. Это влияет на полоску меню (см. раздел «Активный объект»). При закрытии модальный фрейм активирует тот объект, который был активен до его создания. Немодальный фрейм при закрытии не управляет активизацией объектов. В этом случае активным станет тот объект, который активизирует ОС путем передачи фокуса ввода его окну.

5.4.3 Активный объект

Экземпляр объекта считается активным, если его окно имеет фокус ввода, либо, в случае диалога, фокус ввода имеет один из его управляющих элементов. В системе может существовать только один активный объект. Дескриптор экземпляра активного объекта возвращает функция GetActiveObject. Каждый раз при активации или деактивации объекта срабатывает его обработчик OnActivate.

Активный объект управляет полоской меню.

Если данный объект вызван в модальном фрейме, его меню полностью замещает существующее. Если в немодальном – его меню достраивает справа главное меню приложения (см. раздел «Модуль, главное меню и главное окно приложения»).

Если активным становится вложенный объект, то он всегда достраивает справа существующее меню. Например, в диалог D1 вложен (через закладку или сплиттер) диалог D2, а в диалог D2 вложен список L. Тогда при активации списка получаем меню:

- Если D1 в модальном фрейме – меню D1 + меню D2 + меню L
- Если D1 в немодальном фрейме – главное меню + меню D2 + меню L

5.5 Обработка сообщений

Многие объекты X2 способны обрабатывать пользовательские сообщения.

Пользовательское сообщение имеет идентификатор (id) и дополнительный параметр. Идентификатор сообщения – это целое число (тип int). Дополнительный параметр – это бестиповая ссылка, т.е. через него можно передать ссылку на любой тип. Доп. параметр не является обязательным.

Отправить сообщение можно с помощью функции SendMessage, которая принимает дескриптор экземпляра объекта-адресата, id сообщения и указанный параметр.

Каждый объект, способный обрабатывать сообщения, имеет сегмент OnMessage, который может содержать произвольный код на языке X2. Функция SendMessage находит по дескриптору экземпляр объекта-адресата и вызывает его обработчик OnMessage, передавая ему id сообщения и указанный дополнительный параметр.

Поскольку доп. параметр передается по ссылке без указания типа, то он, во-первых, может иметь произвольный тип, и во-вторых, изменение его значения будет доступно в вызывающем коде. Таким образом, механизм пользовательских сообщений выполняет две функции:

- Позволяет экземплярам объектов обмениваться данными, передавая через доп. параметр ссылку на буфер (это может быть переменная встроенного типа или структура). Данный буфер может служить, как для отправки данных, так и для получения.
- Дает возможность из одного объекта вызвать какой-либо алгоритм, реализованный в другом объекте.

Например, у списка работников можно запросить табельный номер текущего работника. Для этого нужно задействовать какое-либо целое число под сообщение. Пусть это будет число 100. Для наглядности в декларации объявим константу и включим эту декларацию в список

```
const int MSG_GET_TAB_NUM = 100;
```

Через доп. параметр будем получать наш табельный номер. Табельный номер имеет строковый тип данных. В обработчике списка OnMessage (int nMsg, & Param) напишем код:

```
if (nMsg == MSG_GET_TAB_NUM)
{
	// Не забываем, что Param передан, как бестиповая ссылка
	CAST Param AS string;

	Param = // здесь получаем таб. номер из текущей строки;

	// Вернем 1. Это будет означать, что сообщение обработано.
	return 1;
}
return 0;
```

Теперь любой объект может послать списку работников сообщение MSG_GET_TAB_NUM, и список работников вернет ему табельный номер выбранного работника. Для этого нужно написать код:

```
string    strTabNum;

if (SendMessage (hObj, MSG_GET_TAB_NUM, strTabNum) == 1)
{
        // Получили табельный номер, обрабатываем.
}
```

В нашем случае табельный номер имеет простой тип. Если нам требуется вернуть какие-либо структурированные данные, в той же декларации можно объявить структуру. Вызывающий код создаст переменную и передаст ее через & Param, а вызываемый код приведет Param к нужному типу и наполнит данными.

Аналогично можно заставить объект выполнять какие-либо типовые действия, если зарезервировать под это специальное сообщение, а в обработчике реализовать алгоритм.

Чтобы номера сообщений не пересекались, удобно задействовать под них отдельную декларацию, в которой можно хранить все константы с номерами сообщений, а также, все структуры, описывающие их доп. параметры. Эту декларацию можно включать во все объекты, которые будут использовать механизм обмена пользовательскими сообщениями.

5.6 Общие свойства объектов

Для каждого типа объекта X2 определен собственный набор свойств, но есть свойства, которыми обладают все объекты:

- Имя объекта
- Тип объекта
- Меню, связанное с объектом (для большинства визуальных объектов)
- ID разработчика
- Признак защищенности

5.6.1 Идентификатор разработчика и признак защищенности

Идентификатор разработчика (ИР) зашит в лицензионный ключ платформы. Ключ платформы поставляется вендором платформы. Вендор платформы обеспечивает уникальность ИР. При создании нового объекта ИР прописывается в бинарную структуру объекта и не может быть изменен.

Объект X2 может быть защищен от нелицензионного использования. Для этого служит флаг в настройках «Признак защищенности». Признак защищенности может быть изменен только владельцем объекта. Владелец объекта – это тот, кто его создал, т.е. ИР объекта соответствует ИР ключа платформы. В этом случае среда разработки дает возможность изменить признак защищенности, т.е. флаг в настройках становится доступным. Кроме того, в меню Дизайнера есть команда «Сервис\Изменение признака защищенности», которая позволяет менять этот признак для группы объектов.

Если объект защищен, т.е. флаг «Признак защищенности» установлен, то для использования объекта требуется наличие ключа, который может выдать только сам разработчик. Это проверяется средой исполнения на этапе загрузки объекта из базы данных. Если объект не защищен, т.е. флаг снят, среда исполнения позволяет использовать объект беспрепятственно. Кроме того, среда исполнения позволяет беспрепятственно использовать собственные объекты, для которых ИР объекта совпадает с ИР ключа платформы.

Для использования механизма лицензионной защиты и генерации собственных ключей необходимо приобрести у венодора платформы генератор ключей.

5.6.2 Общие сегменты кода

Каждый тип объекта имеет собственный набор сегментов кода. В данном разделе описаны только те сегменты, которые представлены для всех типов объектов. Исключение составляют процедуры, у которых отсутствуют конструктор и деструктор, а также, прочие сегменты.

Перечень общих сегментов кода:

1. Список внешних деклараций
2. Встроенная декларация
3. Конструктор
4. Деструктор
5. Обработчики событий
   5.1. OnInit – инициализация
   5.2. OnMessage – обработка пользовательских сообщений
   5.3. OnOK – фрейм закрыт кнопкой «OK»
   5.4. OnCancel – фрейм закрыт кнопкой «Отказ»
   5.5. OnActivate – вызывается системой при смене активного объекта

5.6.3 Список внешних деклараций

Список содержит все внешние декларации, включенные в объект. При включении декларации в объект ему становятся доступны все типы, константы и глобальные переменные, объявленные в данной декларации.

5.6.4 Встроенная декларация

Встроенная декларация предназначена для объявления типов, констант и переменных с локальной областью видимости. Все, что объявлено во встроенной декларации доступно в любой точке экземпляра объекта, но не доступно за его пределами.

К сведению, типы, константы и переменные, объявленные внутри сегмента кода, имеют область видимости – блок. Они становятся недоступными при выходе из блока, в котором объявлены.

5.6.5 Конструктор

Конструктор вызывается сразу после создания экземпляра объекта, но до создания окна объекта.

Конструктор не принимает параметров и возвращает значение типа BOOL.

Прототип:

BOOL ()

Если конструктор вернет TRUE (или значение != 0), исполнение объекта продолжается.

Если конструктор вернет FALSE (== 0), то экземпляр объекта будет уничтожен. В этом случае деструктор объекта вызываться не будет.

В конструкторе могут быть произведены любые действия, связанные с инициализацией объекта, но, поскольку окна объекта еще нет, обращаться к нему нельзя.

Если сегмент не реализован, значение по умолчанию – TRUE.

5.6.6 Деструктор

Деструктор объекта вызывается непосредственно перед уничтожением экземпляра объекта. Он не принимает параметров и не возвращает значений.

Деструктор предназначен для освобождения ресурсов, занятых в ходе исполнения приложения, например, для удаления временно созданных объектов БД, записи протокола и т.д. Деструктор вызывается, когда окна объекта уже не существует, поэтому обращаться к нему нельзя.

5.6.7 Обработчик OnInit

Данный обработчик вызывается при инициализации объекта сразу после создания его визуальной части.

Прототип:

BOOL OnInit ()

Не имеет входных параметров.

Если возвращает TRUE (или значение != 0), исполнение объекта продолжается.

Если вернет FALSE (== 0), то экземпляр объекта будет уничтожен. В этом случае будет вызван деструктор объекта.

Если сегмент не реализован, значение по умолчанию – TRUE.

5.6.8 Обработчик OnMessage

Обработчик вызывается, когда приходит пользовательское сообщение, отправленное командой SendMessage.

Прототип:

int OnMessage (int nMsg, & Param)

Параметры:

- nMsg – Номер сообщения
- Param – Дополнительный параметр. Поскольку параметр передается по ссылке без указания типа, в обработчике он должен быть приведен к типу, переданному через команду SendMessage. Приведение бестиповой ссылки к конкретному типу выполняется оператором CAST.

Обработчик возвращает целое число, которое будет возвращено в вызывающий код командой SendMessage. Смысл этого числа определяет сам разработчик.

Если сегмент не реализован, значение по умолчанию – 0.

5.6.9 Обработчик OnOK

Обработчик вызывается, когда фрейм закрывается кнопкой «OK» (идентификатор IDOK).

Прототип:

BOOL OnOK ()

Не имеет входных параметров.

Примечания.

Если обработчик возвращает TRUE (!= 0), фрейм будет закрыт. В случае модального фрейма функция ModalFrame, посредством которой он был создан, вернет IDOK. Будет ли при этом уничтожен экземпляр объекта, определяется флагом KEEP_OBJECT, переданным в функцию визуализации.

Если обработчик возвращает FALSE (0), фрейм не будет закрыт, и экземпляр объекта продолжит работу.

Если сегмент не реализован, значение по умолчанию – TRUE.

5.6.10 Обработчик OnCancel

Обработчик вызывается, когда фрейм закрывается кнопкой «Отказ» (имеющей идентификатор IDCANCEL), клавишей Esc или служебной кнопкой с крестиком в правой части строки заголовка.

Также, данный обработчик будет вызван в ходе закрытия окна с помощью функций CloseFrame или Quite.

Прототип:

OnCancel ()

Не имеет входных параметров и не возвращает значений.

Примечания.

В отличие от деструктора, обработчик OnCancel вызывается до уничтожения окна.

5.6.11 Обработчик OnActivate

Обработчик вызывается системой при смене активного объекта.

Прототип:

OnActivate (BOOL bActivate)

Параметры:

- bActivate –TRUE, если объект активируется, FALSE, если деактивируется.

Обработчик не возвращает значения.

5.7 Общие функции

Для каждого типа объекта существует набор специфических функций, работающих с объектом только данного типа. Каждый такой набор описывается в соответствующем разделе, посвященном конкретному типу объекта.

В данном разделе описывается набор функций, которые могут работать с объектами различных типов.

Функция идентифицирует экземпляр объекта по его дескриптору. Все функции, получающие в качестве параметра дескриптор экземпляра объекта, интерпретируют 0, как дескриптор собственного экземпляра.

Перечень общих функций:

1. Функции модуля
   - 1.1. GetModuleHandle – возвращает дескриптор текущего модуля
   - 1.2. GetModuleTitle – возвращает заголовок модуля, отображаемый в главном окне приложения
   - 1.3. SetModuleTitle – устанавливает заголовок модуля, отображаемый в главном окне приложения
   - 1.4. Quite – завершает исполнение модуля
2. Информация об экземпляре
   - 2.1. GetHandle – возвращает дескриптор текущего исполняемого объекта
   - 2.2. IsObject – проверка существования объекта
3. Информация об объекте
   - 3.1. GetObjectName – возвращает имя объекта
   - 3.2. GetObjectType – возвращает тип объекта
   - 3.3. GetGroupId – возвращает id функциональной групп (для объектов замещения)
   - 3.4. GetGroupName – возвращает имя функциональной групп (для объектов замещения)
4. Сообщения
   - 4.1. SendMessage – отправка пользовательского сообщения
5. Настройки объекта
   - 5.1. GetOptions – возвращает флаги настроек объекта
   - 5.2. SetOptions – устанавливает флаги настроек объекта
   - 5.3. GetWinSize – возвращает размер окна, заданный разработчиком
   - 5.4. SetWinSize – изменяет размер окна, заданный разработчиком
6. Фрейм объекта
   - 6.1. GetFrameType – для объектов, вызванных в собственном фрейме, возвращает тип фрейма
   - 6.2. CanCloseFrame – можно ли закрыть окно объекта
   - 6.3. CloseFrame – закрытие окна объекта

6.4. GetFrameTitle – возвращает заголовок фрейма

6.5. SetFrameTitle – устанавливает заголовок фрейма

7. Прочие функции

7.1. DestroyObject – уничтожает экземпляр объекта

7.2. GetOwner – возвращает дескриптор экземпляра родительского объекта

7.3. GetMenu – возвращает дескриптор экземпляра меню указанного объекта

7.4. GetActiveObject – возвращает дескриптор экземпляра текущего объекта

7.5. OnOK – вызывает обработчик OnOK объекта

7.6. OnCancel – вызывает обработчик OnCancel объекта

7.7. OnInit – вызывает обработчик OnInit объекта

6 Списки

Некоторые приемы программирования списков представлены в демонстрационном модуле. Путь по меню: «Списки».

6.1 Оператор создания экземпляра – BROWSER

HOBJECT BROWSER name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта «Список» и возвращает дескриптор экземпляра.

6.2 Палитра

Палитра используется при отображении списков для выделения определенных строк цветом. Она представляет собой массив из 256 элементов (от 0 до 255). Каждый элемент массива представляет собой RGB-тройку, задающую цвет. Порядковый номер элемента в массиве является номером цвета в палитре.

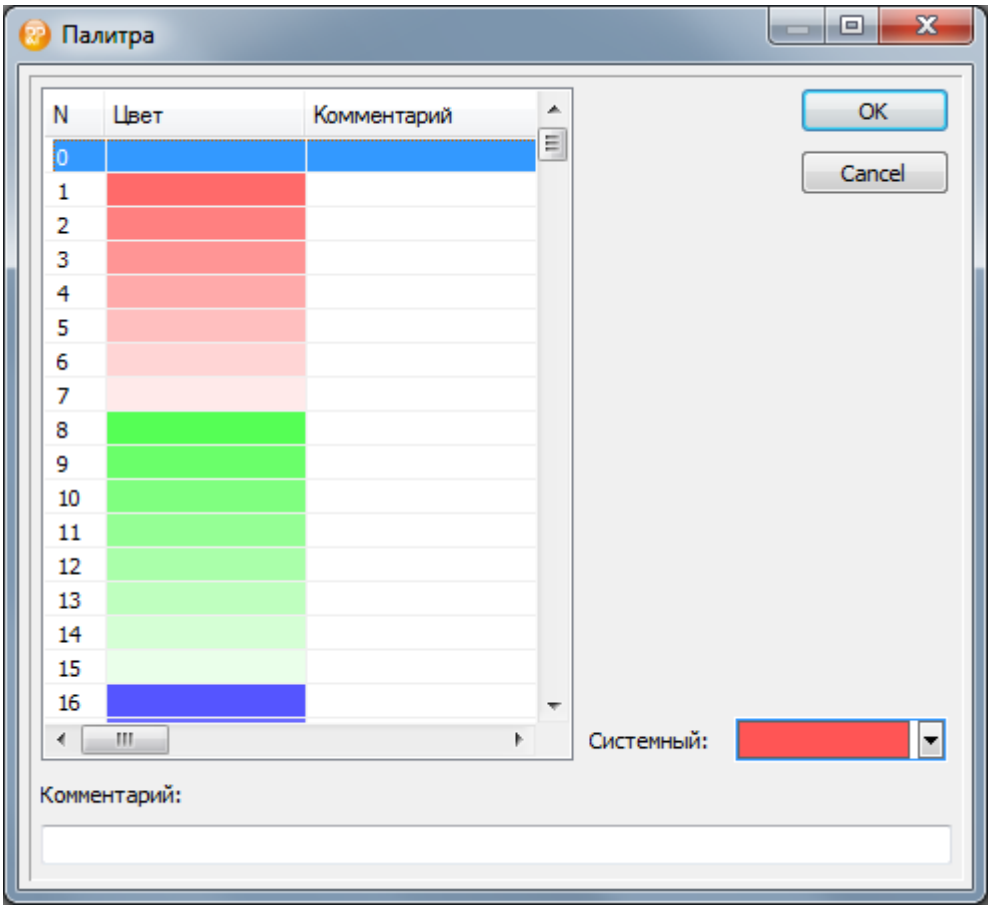
Существует системная палитра и пользовательские палитры. Системная палитра одна. Она создается при создании репозитория RP и может редактироваться с помощью Дизайнера разработчиком прикладного кода.

Пользовательских палитр может быть сколько угодно. Пользовательскую палитру настраивает конечный пользователь в среде исполнения. Каждый пользователь может настроить собственную палитру.

Палитра хранится в репозитории RP в таблице x2Palette.

6.2.1 Системная палитра

Системная палитра настраивается разработчиком в Дизайнере. Путь по меню: Сервис\Справочники\Палитра.



При создании репозитория в таблицу выполняется вставка 208 цветов системной палитры. Остальные цвета не настроены, т.е. при их использовании в списке строки будут иметь цвет фона окна.

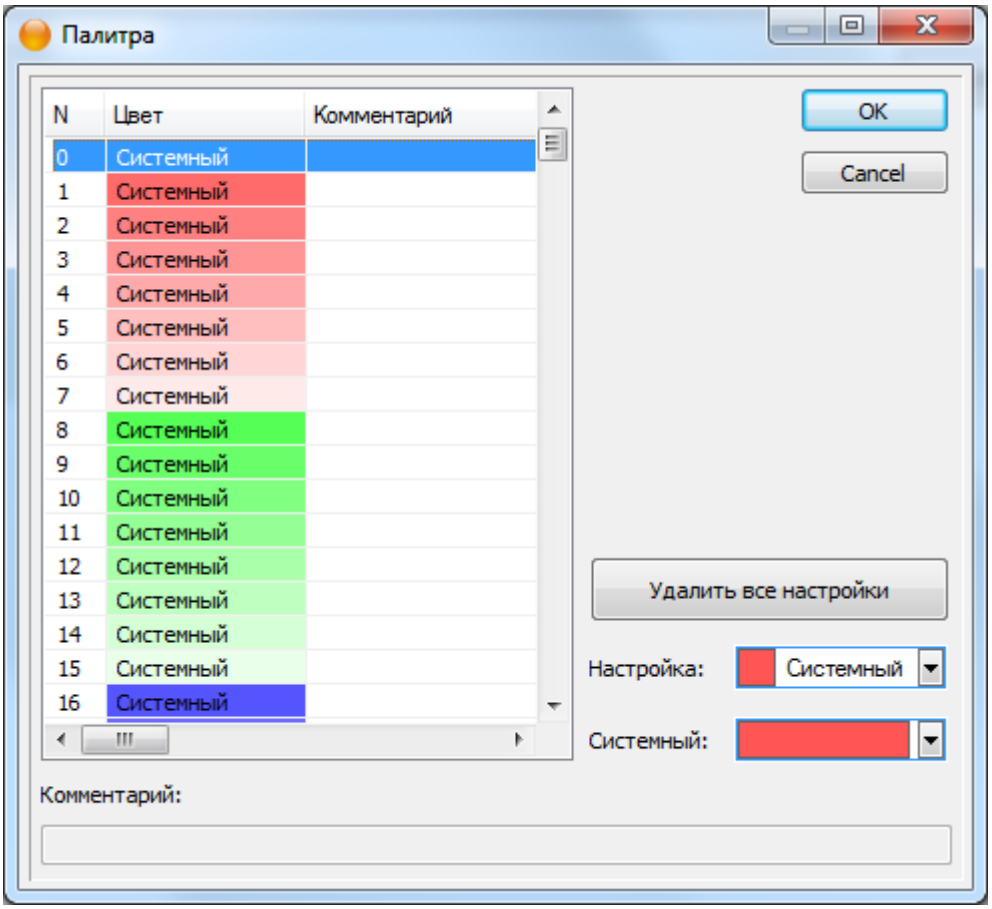
Разработчик может перенастроить любой из цветов системной палитры.

Предполагается, что разработка в организационном порядке придаст каким-то конкретным цветам смысловую нагрузку и будет использовать их во всех интерфейсах единым образом. В этом случае таким цветам следует дать комментарии. Это поможет пользователю понять, какие цвета ему следует перенастроить.

При создании репозитория комментарии цветам не даются.

6.2.2 Пользовательская палитра

Пользовательская палитра настраивается конечным пользователем в среде исполнения. Путь по меню: Файл\Настройка\Палитра.



Если пользователя удовлетворяет расцветка списков, он может не настраивать собственную палитру. Если же его не удовлетворяет какой-то цвет, он может его перенастроить. Для каждого настроенного цвета в таблице x2Palette будет присутствовать одна строка. Результирующая пользовательская палитра будет формироваться, как множество цветов, настроенных пользователем, дополненное недостающими цветами из системной палитры.

Для тех цветов, которые пользователь перенастроил, в колонке «Цвет» выводится текст «Настройка». Для остальных, т.е. не настроенных, цветов выводится текст «Системный».

Пользователь может только перенастраивать сами цвета, но не может редактировать комментарии к ним. Комментарии всегда берутся из системной палитры.

При настройке цвета пользователь может выбрать любой цвет, в том числе, и системный. В этом случае из пользовательской палитры удаляется цвет с таким индексом, и вместо него будет использоваться цвет из системной палитры.

По кнопке «Удалить все настройки» из таблицы x2Palette удаляются все строки с логином данного пользователя, т.е. в списках будут использоваться цвета системной палитры.

6.3 Категории списков

В зависимости от того, каким способом формируется набор данных для списка, списки делятся на две категории:

- Запрос списка построен на основе оператора SELECT
  - Оператор SELECT должен быть только один
  - Не должен содержать предложения WITH
- Запрос списка представляет собой любой фрагмент кода на SQL, возвращающий набор данных
  - Если код SQL возвращает несколько наборов данных, для построения списка будет использован первый набор

Категория списка задается явным образом на этапе разработки (в свойствах списка флаг «Запрос на основе SELECT»), но может быть изменена в период исполнения вызовом функции SetOptions (флаг LO_QUERY_SELECT).

6.4 Многострочное выделение

Чтобы список поддерживал многострочное выделение, нужно в настройках установить флаг «Многострочное выделение». Изменить эту настройку можно в период исполнения с помощью функции SetOptions (флаг LO_MULTI_SELECT), но сделать это можно только до визуализации объекта.

Чтобы найти все выделенные строки, нужно использовать функцию ListGetSelectedRow.

Пример:

```
// Получаем первую выделенную строку
int nRow = ListGetSelectedRow(-1);

if (nRow != -1)
{
	…

	// В цикле проходим по всем выделенным строкам
	while (nRow = ListGetSelectedRow(nRow))
	{
	…
	}
}
```

6.5 Редактирование списков

6.5.1 Удаление строк

Для удаления строк списка используется обработчик OnDelete. Он срабатывает при нажатии на клавишу Del. Чтобы задействовать обработчик, в настройках списка нужно установить флаг «Удаление» в категории «Редактирование». Если список поддерживает многострочное выделение, то в обработчике это должно быть учтено.

Для удаления всего множества выделенных строк нужно сформировать строку ограничения и подставить ее в запрос. Это можно сделать двумя способами.

- Использовать функцию ListGetSelectionFilter. Чтобы функция корректно отработала, в настройках колонок должны быть обозначены все ключевые поля.

```
string strFilter = ListGetSelectionFilter();
DELETE FROM [MyTable] WHERE :strFilter;
```

- Пройти по выделенным строкам и самостоятельно сформировать ограничение.

```
string    strIN;          // Здесь будем формировать строку
int       id_Key;         // Это ключевое поле, оно лежит в нулевой колонке

// Получаем первую выделенную строку
int nRow = ListGetSelectedRow(-1);

if (nRow != -1)
{
        ListGetFieldData (id_Key, nRow, 0);     // Получаем ключевое поле
        strIN = ToString (id_Key);              // Переводим его в строку

        // В цикле проходим по всем выделенным строкам
        while (nRow = ListGetSelectedRow(nRow))
        {
                ListGetFieldData (id_Key, nRow, 0);
                strIN += "," + ToString (id_Key);
        }

        DELETE FROM [MyTable] WHERE [id_Key] IN (:strIN);
}
```

6.5.1.1 Встроенный обработчик OnDelete

Для простых запросов, которые строятся по одной таблице, можно использовать встроенный обработчик. Чтобы задействовать встроенный обработчик, в настройках списка нужно установить флаг «Удаление» в категории «Встроенные обработчики».

Для использования встроенного обработчика необходимо, чтобы выполнялись следующие требования:

- Список должен быть построен на основе оператора SELECT
- В настройках колонок должны быть указаны все ключевые поля, по которым будет идентифицироваться строка
- Запрос не должен иметь подстановок по ссылке; подстановка переменных по значению допустима

Если в списке выделено несколько строк, встроенный обработчик удалит все выделенные строки.

Код встроенного обработчика, генерируемый компилятором:

```
{
	BOOL	bResult  = FALSE;
	string	strFilter  = ListGetSelectionFilter();

	if (StrIsEmpty(strFilter))
	{
		MessageBox ("Не найдены ключевые поля!");
	}
	else
	{
		SQL ()
		{
			WITH T AS (:(ListGetQuery())) DELETE T WHERE (:strFilter)
		}

		bResult = SqlSuccess();
	}

	return bResult;
}
```

6.5.2 Вставка и редактирование

Вставка и редактирование в списке может осуществляться двумя способами: путем самостоятельной реализации обработчиков OnInsert и OnEdit, либо путем использования встроенных обработчиков. В настройках списка необходимо явно задать, какой из способов будет использоваться. Если используются встроенные обработчики, в настройках списка необходимо указать диалог, который будет использоваться при вставке и обновлении.

6.5.2.1 Самостоятельная реализация обработчиков

Самостоятельная реализация обработчиков предполагает, что вся логика вставки и обновления реализуется прикладным программистом, и система не выполняет никаких автоматических действий. Данный подход предоставляет программисту максимальные возможности. В частности, можно строить редактируемые списки на основе произвольного запроса (не обязательно SELECT), для идентификации строк можно использовать составной ключ (не обязательно int IDENTITY), можно строить списки с отложенным сохранением, когда все операции модификации проводятся во внутреннем буфере, а в БД переносятся только при закрытии списка, и т.д.

При самостоятельной реализации обработчиков можно выделить два основных подхода:

1. Логика работы с БД реализуется в обработчиках списка. Список предоставляет диалогу редактирования буфер, в который диалог складывает данные пользователя, а по закрытию диалога список выполняет вставку или обновление в БД. В демонстрационном модуле: Списки\Редактирование\Список на основе хранимой процедуры.
2. Логика работы с БД реализуется в обработчиках диалога. Диалог получает от списка только ключ редактируемой строки. Данные он загружает из БД самостоятельно, и при закрытии выполняет вставку или обновление, а список только перечитывает строку по имеющемуся ключу. В демонстрационном модуле: Списки\Редактирование\Составной ключ.

Оба подхода могут варьироваться в деталях реализации, и возможно, также, их совмещение, когда часть логики реализуется в списке, а часть в диалоге.

Пример списка с отложенным сохранением представлен в демонстрационном модуле: Списки\Редактирование\Отложенное сохранение.

6.5.2.2 Использование встроенных обработчиков

Чтобы задействовать встроенные обработчики, в настройках списка нужно установить флаг «Добавление и редактирование» в категории «Встроенные обработчики».

Использование встроенных обработчиков не предполагает реализации какого-либо кода. В этом случае система на этапе компиляции списка автоматически генерирует обработчики, что налагает ряд своих ограничений:

1. Список должен быть построен на основе оператора SELECT
2. Запрос не должен иметь подстановок по ссылке; подстановка переменных по значению допустима
3. Запрос списка может иметь только одно поле, идентифицирующее строку в наборе
   3.1. Это поле должно быть первым в запросе
   3.2. Оно должно иметь тип int и признак IDENTITY
   3.3. Оно должно быть помечено, как ключевое в настройках колонок
4. И для вставки, и для обновления используется один и тот же диалог редактирования

Диалог редактирования, также, должен соответствовать определенным требованиям:

1. Диалог обязательно должен иметь SQL-запрос
   1.1. Запрос должен представлять собой одиночный оператор SELECT
   1.2. Запрос должен быть построен по тому же набору, что и список
   1.3. Ключевое поле в запросе должно быть одно и должно совпадать по имени с ключевым полем запроса списка
2. Вся логика работы с БД должна быть реализована в диалоге, т.е. диалог должен выполнить INSERT или UPDATE

При вызове диалога встроенный обработчик списка будет устанавливать режим вызова диалога, с помощью функции DlgSetAction, передавая ей соответствующий параметр, DA_INSERT или DA_UPDATE (см. раздел «Код встроенных обработчиков»). Этот признак будет в дальнейшем определять, какой из обработчиков диалога будет исполняться по кнопке «OK», OnInsert или OnUpdate.

В диалоге непременно должен быть реализован SQL-запрос. На основе этого запроса диалог формирует запросы на вставку и обновление и исполняет их. Эту операцию можно, также, вызвать программно с помощью функции DlgDefaultAction. Эта функция выполняет вставку, обновление или удаление в зависимости от того, в каком режиме вызван диалог.

Если обработчик OnInsert или OnUpdate не реализован, то вместо него система будет вызывать функцию DlgDefaultAction автоматически.

Если же обработчик реализован, то система вызовет его. Внутри обработчика программист может в любом месте вызвать функцию DlgDefaultAction, но может не вызывать ее, а выполнить операцию вставки или обновления вручную.

Для справки:

Диалог имеет встроенную переменную DlgAction. К данной переменной нельзя обращаться непосредственно, для работы с ней есть пара функций DlgGetAction и DlgSetAction.

По умолчанию DlgAction имеет значение 0, но ее значение может быть изменено функцией DlgSetAction. Изменить значение DlgAction можно только до визуализации диалога, т.е. из конструктора или из внешнего кода до вызова функции визуализации. После визуализации диалога функция DlgSetAction будет вызывать состояние ошибки.

Переменная DlgAction создана для автоматизации редактирования в списке, но может быть использована программистом и для каких-либо иных целей. Система реагирует на значение DlgAction следующим образом:

- *Значение DA_INSERT – диалог вызван на вставку*
    - *Функция DlgDefaultAction будет выполнять INSERT*
    - *На IDOK будет вызываться обработчик OnInsert*
- *Значение DA_UPDATE – диалог вызван на редактирование*
    - *Функция DlgDefaultAction будет выполнять UPDATE*
    - *На IDOK будет вызываться обработчик OnUpdate*
- *Значение DA_DELETE – диалог вызван на удалениен*
    - *Диалог открывается в режиме «Только чтение»*
    - *Функция DlgDefaultAction будет выполнять DELETE*
    - *На IDOK будет вызываться обработчик OnDelete*
- *Любое иное значение – программист сам определяет смысл этого значения*
    - *Функция DlgDefaultAction будет возвращать ошибку при вызове*
    - *На IDOK будет вызываться OnOK*

6.5.2.2.1 Код встроенных обработчиков

```
BOOL OnInsert()
{
	BOOL	bResult = FALSE;
	HOBJECT hObj = DIALOG %1;

	DlgSetFilter ("1 = 0", hObj);
	DlgSetAction (DA_INSERT, hObj);

	if (ModalFrame (hObj) == IDOK)
	{
		int	nIdentity = 0;

		SQL (nIdentity)
		{
			SELECT @@IDENTITY;
		}

		ListSetFieldData (nIdentity, ListGetCurRow(), 0);
		bResult = TRUE;
	}

	return bResult;
}

BOOL OnEdit()
{
	BOOL	bResult		= FALSE;
	string	strFilter	= ListGetRowFilter();

	if (StrIsEmpty(strFilter))
	{
		MessageBox ("Не найдены ключевые поля!");
	}
	else
	{
		HOBJECT hObj = DIALOG %1;

		DlgSetFilter (strFilter, hObj);
		DlgSetAction (DA_UPDATE, hObj);
		bResult = (ModalFrame (hObj) == IDOK);
	}

	return bResult;
}
```

В тело обработчика выполняется подстановка:

- %1 – в этом месте подставляется вызов диалога, указанного в настройках списка (строка вида “MyDialog (Parameters)”)

После подстановки обработчик компилируется и добавляется в ресурс списка.

6.5.3 Редактирование в ячейках

Для редактирования в ячейках списка служит обработчик OnEditCell. Обработчик вызывается на событие редактирования в текущей ячейке списка, если выполняются следующие условия:

1. В настройках данной колонки установлен флаг «Разрешено редактирование» (LFF_EDIT)
2. В период исполнения в списке установлен режим редактирования в колонках (см. контекстное меню списка)
3. Пользователь нажал клавишу Enter, любую символьную клавишу или выполнил двойной клик левой клавишей мыши на ячейке списка

Обработчик используется по-разному, в зависимости от того, установлен ли в настройках данной колонки флаг «Внешний редактор» (LFF_EXTERNAL_EDITOR).

Состояние флага определяет, будет ли система перед вызовом обработчика вызывать встроенный редактор, или же вызовет обработчик непосредственно, предоставляя разработчику самостоятельно вызвать редактор.

6.5.3.1 Встроенный редактор

Пример использования встроенного редактора можно найти в демонстрационном модуле по пути: Списки\Редактирование\Список на основе хранимой процедуры.

Если флаг «Внешний редактор» снят, то система вызовет вначале встроенный редактор, а обработчик после того, как пользователь ввел данные и нажал Enter.

Встроенный редактор представляет собой обычный текстовый редактор. Его тип данных система определяет автоматически на основании типа данных, который вернул запрос списка. Маска ввода встроенного редактора задается в настройках колонки в поле «Формат ввода». Если маска ввода не указана (пустая строка), то используется маска по умолчанию для данного типа редактора.

Обработчик должен выполнить два обязательных действия:

- обновить данные в БД (UPDATE)
- обновить данные в окне списка

Обработчик вызывается на событие редактирования в любой колонке, для которой установлен флаг «Разрешено редактирование». Чтобы определить, в какой колонке возникло событие, нужно вызвать функцию ListGetCurCol или функцию ListGetCurCell, т.к. редактирование всегда производится только в текущей ячейке списка.

Новое значение поля передается в обработчик через входной параметр, являющийся бестиповой ссылкой, поэтому его следует привести к нужному типу посредством оператора CAST.

Если с редактируемым полем связана буферная переменная, то на момент вызова обработчика в ней будет храниться старое значение. Также, старое значение вернет функция ListGetFieldData.

1. Если вход в режим редактирования осуществляется нажатием символьной или цифровой клавиши, то
   1.1. Если данный символ соответствует маске, то он попадает в окно редактора
   1.2. Если не соответствует, в окне редактора будет пустая строка
2. Если вход в режим редактирования осуществляется нажатием клавиши Enter или по двойному клику мыши, то
   2.1. Данные, содержащиеся в буфере запроса, попадают в окно редактора и отображаются в соответствии с маской ввода
   2.2. Вся строка ввода выделяется

Выход из режима редактирования осуществляется нажатием клавиш Enter или Esc. Клавиша Esc отменяет предыдущие действия. В этом случае окно редактора просто закрывается. Клавиша Enter подтверждает ввод. В этом случае:

1. Система вызывает обработчик OnEditCell
2. Если обработчик вернул
   2.1. TRUE (!= 0) – Окно редактора закрывается
   2.2. FALSE (0) – Окно редактора остается на экране

6.5.3.2 Внешний редактор

Пример использования внешнего редактора можно найти в демонстрационном модуле по пути: Списки\Редактирование\Редактирование в ячейке – внешний редактор.

Если флаг «Внешний редактор» установлен, то система проверяет наличие обработчика и просто вызывает его, не выполняя никаких иных действий.

Разработчик должен выполнить все действия по редактированию самостоятельно, в том числе, вызвать свой редактор. В качестве редактора может выступать диалог, либо любой другой объект, вызванный в модальном фрейме. Диалог использовать удобно потому, что он обрабатывает клавиши Enter и Esc. Чтобы объект другого типа реагировал на них, нужно в функцию ModalFrame передать флаги STATUSLINE и OKCANCEL.

Можно, также, использовать функцию ListEditCurCell. Функция работает только с диалогами. Она создает окно указанного диалога в модальном режиме и позиционирует его в координаты текущей ячейки списка.

Например, нередко возникает потребность в качестве редактора использовать выпадающий список (lookup). Для этого:

- Создайте диалог и разместите на его форме выпадающий список
  - Левую верхнюю координату установите в (0,0), а ширину и высоту задайте небольшими, например, 60 и 17. Исходите из того, что управляющий элемент должен поместиться в ячейку списка, а система может растянуть его, но не может сжать менее, чем задано разработчиком.
  - Размер диалога можно оставить произвольным, а в его конструкторе вызвать функцию SetWinSize (60, 17) и подогнать размер диалога под размер управляющего элемента. Это необходимо сделать в конструкторе, т.е. до визуализации окна.

- В обработчике списка OnEditCell с помощью оператора DIALOG создайте экземпляр диалога, передав ему необходимые параметры. Если вход в режим редактирования выполнен по нажатию символьной клавиши, то через параметр Data будет передан символ, введенный пользователем.
- Вызовите функцию ListEditCurCell и передайте ей дескриптор созданного диалога
- Проверьте код возврата функции. Если он равен IDOK, обновите данные в БД и в ячейке списка

6.5.3.3 Обновление данных в списке

Обновить данные в окне списка можно тремя способами:

- Вызвать функцию ListSetFieldData. Это самый быстрый метод, т.к. при этом не строится никакого запроса к БД, а просто новое значение отображается в ячейке списка.
- Вызвать функцию ListRefreshRow. В этом случае будет перечитана вся строка. Это может быть полезно, когда в запросе есть вычисляемые поля, значения которых могли измениться при редактировании данного поля. Но это возможно только в том случае, когда список построен на основе оператора SELECT.
- Вызвать функцию ListRequery, т.е. перечитать весь список. Это стоит делать только в том случае, если после редактирования могли измениться данные и в других строках списка (например, серверный триггер обновил данные в связанных таблицах).

Обновление данных в окне приведет, также, к обновлению значения буферной переменной, если таковая имеется. При этом обработчик может выполнить любой код, например, проверить данные на корректность и выдать сообщение пользователю, сдвинуть курсор на следующую строку и т.д. В частности, встроенный обработчик всегда выполняет сдвиг на следующую строку.

6.5.3.4 Встроенный обработчик OnEditCell

Встроенный обработчик генерируется компилятором, если в настройках списка установлен флаг LO_DEFHANDLER_EDITCELL.

Для работы встроенного обработчика необходимо следующее:

- Список должен быть построен на основе оператора SELECT
- В списке обязательно должны быть указаны все ключевые поля, по которым будет идентифицироваться строка
- Запрос списка должен быть таким, чтобы созданное по нему представление (view) было редактируемым
- В настройках колонки не должен быть установлен флаг «Внешний редактор»

```
BOOL OnEditCell(& Data)
{
	BOOL	bResult = TRUE;
	string	strSQL  = "WITH T AS (" + ListGetQuery() + ") UPDATE T SET " +
		ListGetFieldName() + " =";
	string	strFilter = "WHERE (" + ListGetRowFilter() + ")";

	switch (ListGetFieldType())
	{
		case TYPE_INT:
		{
			CAST Data AS int;
			SQL () { :strSQL :&Data :strFilter }
		}
		break;

		case TYPE_DOUBLE:
		{
			CAST Data AS double;
			SQL () { :strSQL :&Data :strFilter }
		}
		break;

		case TYPE_DATETIME:
		{
			CAST Data AS datetime;
			SQL () { :strSQL :&Data :strFilter }
		}
		break;

		case TYPE_STRING:
		{
			CAST Data AS string;
			SQL () { :strSQL :&Data :strFilter }
		}
		break;

		default:
		{
			MessageBox
			(
				"Тип данных не поддерживается!",
				"Ошибка сохранения данных",
				MB_OK | MB_ICONERROR
			);
			bResult = FALSE;
		}
	}

	if (bResult && (bResult = SqlSuccess()))
	{
		int nRow, nCol;

		ListGetCurCell(nRow, nCol);
		ListRefreshRow();
		ListSetCurCell(nRow + 1, nCol);
	}

	return bResult;
}
```

7 Диалоги

7.1 Примеры построения диалогов

Примеры построения диалогов можно найти в демонстрационном модуле. Путь по меню: «Диалоги». Примеры демонстрируют различные аспекты программирования диалогов и управляющих элементов.

Также, примеры построения диалогов редактирования можно найти по пути: «Списки» и «Деревья».

7.2 Оператор создания экземпляра – DIALOG

HOBJECT DIALOG name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта «Диалог» и возвращает дескриптор экземпляра.

7.3 Концепция

Диалог предназначен для получения данных от пользователя, и предоставляет для этого визуальный интерфейс. Диалог может содержать произвольное количество управляющих элементов, таких, как поля статического текста, редакторы различных типов, кнопки и т.д. В качестве управляющих элементов могут, также, выступать и визуальные объекты X2, такие, как диалоги, списки, деревья.

Границы элементов можно связывать с границами диалога таким образом, чтобы при изменении размеров диалогового окна элементы перемещались или изменяли свой размер (см. раздел «Привязка к границам формы»).

Управляющие элементы можно связать с локальными переменными диалога. Это могут быть переменные, объявленные во встроенной декларации, либо параметры, переданные из внешнего кода. Переменные, связанные с управляющими элементами, называются буферными. Обмен данных между управляющими элементами и буферными переменными осуществляется посредством функции DlgExchange. Система, также, автоматически выполняет обмен данными в следующих случаях:

- при инициализации диалога данные из переменных попадают в управляющие элементы
- при закрытии диалога данные из управляющих элементов попадают в переменные

- при потере фокуса управляющим элементом его данные попадают в переменную, если этот элемент получал пользовательский ввод

Неважно, откуда получат данные буферные переменные. Эти данные могут быть переданы диалогу через параметры, либо запрошены из БД самим диалогом. Если параметры передать по ссылке, то пользовательский ввод будет доступен вызывающему коду после закрытия диалога. Таким образом, диалог не обязательно должен обращаться к БД. Вызывающий код (например, обработчик списка) может передать диалогу входные данные, а диалог вернет ему пользовательский ввод. Далее вызывающий код может выполнить целевую операцию (например, вставку в БД). Если данных много, их можно объединить в структуру.

Если же диалогу требуется работать с БД, то он может запросить данные и сохранить их в буферных переменных, а после того, как пользователь нажмет кнопку OK, записать их значения в БД. Таким образом, буферные переменные выполняют функцию промежуточного буфера между БД и управляющими элементами. Для работы с БД можно использовать операторы OpenDataSet или SQL.

Диалог, как и прочие визуальные объекты, имеет ряд обработчиков. Основные:

- OnOK и OnCancel – вызываются при нажатии кнопок с предопределенными идентификаторами IDOK (кнопка «OK») и IDCANCEL (кнопка «Отказ»)
- OnChange – вызывается при изменении пользователем состояния одного из управляющих элементов
- OnButton – вызывается при нажатии кнопки
- OnNotify – вызывается при различных ситуациях в зависимости от типа управляющего элемента

В простейшем случае диалог можно использовать следующим образом:

- В вызывающем коде объявляем набор переменных, через которые будет осуществляться обмен данными с диалогом, и передаем их в диалог в качестве параметров по ссылке. Если параметров много, их можно объединить в структуру.
- В диалоге эти параметры связываем с соответствующими управляющими элементами, таким образом, в них будет попадать пользовательский ввод. Поскольку параметры переданы по ссылке, а не по значению, то после завершения диалога их значения будут доступны в вызывающем коде.
- В обработчике OnOK диалога проверяем корректность ввода. Если ввод некорректен, возвращаем FALSE, тогда диалог не закроется. Для передачи фокуса управляющему элементу можно использовать функцию CtrlSetFocus.
- В вызывающем коде вызываем диалог в модальном фрейме и проверяем код его возврата. Если он равен IDOK, значит, диалог закрыт по кнопке OK, и в наших переменных лежат новые значения. Обрабатываем их.

Пример такого использования диалога можно найти в демонстрационном модуле. Путь по меню: Списки\Редактирование\Отложенное сохранение. В списке нужно встать на любую строку и нажать Enter, появится диалог редактирования.

7.4 Вложенные объекты – каскадный вызов сегментов

Если в качестве управляющего элемента диалога используется объект X2, например, другой диалог, то мы получаем дерево зависимых объектов. В этом случае инициализация экземпляра, а также, его завершение приводит к каскадному вызову соответствующих сегментов кода. Это относится к следующим сегментам:

- Конструктор
- OnInit
- OnOk (OnInsert, OnUpdate, OnDelete только для диалогов)
- OnCancel
- Деструктор

Последовательность вызовов для всех сегментов, кроме конструкторов, строится от самых вложенных объектов к родительским в порядке их обхода по клавише TAB. Например, диалог A включает в себя диалоги B, C и H, диалог B включает в себя диалоги D и E, диалог C включает в себя диалог F.

- A
  - B
    - D
    - E
  - C
    - F
  - H

Тогда последовательность вызовов будет: D-E-B-F-C-H-A. При этом, если какой-либо из сегментов возвращает FALSE, операция прерывается.

Последовательность вызовов конструкторов строится в обратном порядке, т.е. от родительских объектов к дочерним. Это вызвано тем, что иначе родительский объект не сможет проинициализировать параметры, передаваемые дочерним объектам. Таким образом, последовательность будет такой: A-B-D-E-C-F-H.

Последовательность вызовов конструкторов и деструкторов изменить нельзя, но программист может изменить последовательность вызовов сегментов OnInit, OnOk и OnCancel. Для этого нужно в настройках вложенного объекта установить флаг CO_NO_CASCADE. Тогда данный управляющий элемент будет исключен из цепочки каскадного вызова, и его сегменты OnInit, OnOk и OnCancel не будут вызываться автоматически. В этом случае программист должен вызвать их явно. Для этого служат функции:

- BOOL OnInit (HOBJECT hObj)
- BOOL OnOk (HOBJECT hObj)
- OnCancel (HOBJECT hObj)

Эти функции запускают каскадный вызов соответствующих сегментов для объекта и всех его детей. Например, если требуется, чтобы сегмент диалога H нашего примера отработал до сегмента C, то в настройках для H и C устанавливаем флаг CO_NO_CASCADE. Тогда автоматически будет вызвана такая последовательность: D-E-B-A. В сегменте OnInit диалога A нужно вызвать OnInit для диалогов H и C в нужной последовательности. Вызов

функции OnInit приведет к каскадному вызову OnInit для всех дочерних объектов по той же схеме, как описано выше. В частности, для C будет такая цепочка: F-C.

Следует помнить, что флаг CO_NO_CASCADE влияет на алгоритм каскадного вызова в целом, а не для какого-то конкретного типа сегмента. Это означает, что программист должен вызывать явно не только OnInit, но так же OnOk и OnCancel.

В демонстрационном модуле есть пример, иллюстрирующий последовательность каскадного вызова сегментов. Путь по меню: «Диалоги\Последовательность вызова сегментов».

7.5 Общие сведения об управляющих элементах

Каждый управляющий элемент диалога имеет уникальный идентификатор (тип int). Идентификатор задается на этапе разработки и служит для доступа к управляющему элементу из кода на языке X2.

С управляющими элементами, которые служат для ввода данных, может быть связана буферная переменная. Также, они способны отправлять родительскому диалогу уведомления об изменении своего состояния. Это приводит к вызову обработчика OnChange. Вызов обработчика можно заблокировать, если в настройках управляющего элемента снять флаг «Вызывать OnChange»

Все управляющие элементы имеют набор функций доступа в зависимости от типа элемента. Функции, общие для всех управляющих элементов, имеют префикс «Ctrl…», например: CtrlSetFocus, CtrlSetText и т.д. Функции, специфические для конкретного типа элемента, имеют собственные префиксы, например: TabGetCount и TabGetCurSel – для закладок, или FlagGetCheck и FlagSetCheck – для переключателей.

7.5.1 Типы управляющих элементов

Управляющие элементы делятся на несколько категорий в зависимости от их свойств и назначения.

Элементы организации интерфейса предназначены для оформления внешнего вида формы диалога. Пользователь не может вводить данные через эти элементы, поэтому они не имеют обработчиков в диалоге и с ними нельзя связать буферные переменные.

Элементы ввода данных, напротив, предназначены именно для пользовательского ввода. С ними можно связывать буферные переменные. Для них, также, определен обработчик OnChange.

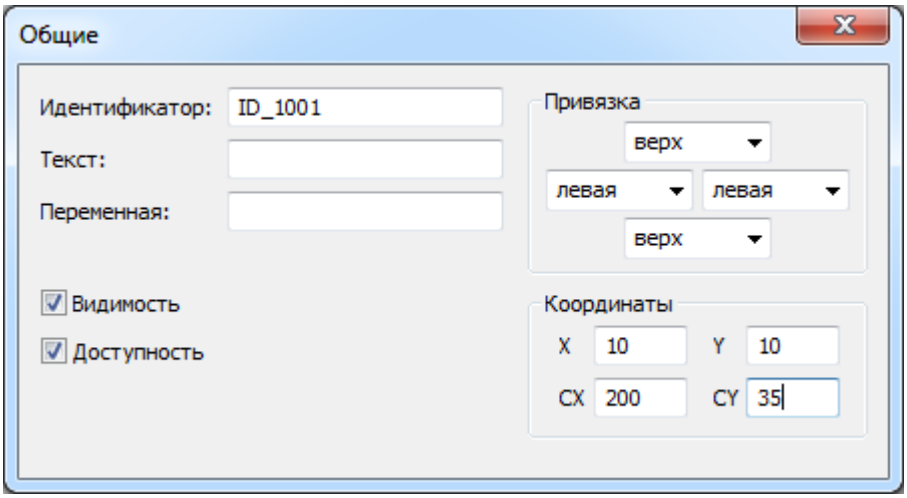
Конопки служат для получения команд от пользователя. С ними нельзя связывать переменные, но для них существует обработчик OnButton.

В качестве управляющего элемента, также, может выступать объект X2. Это может быть только визуальный объект, такой, как дерево, список, диалог и т.д. Нельзя использовать такие объекты, как меню, модуль или процедуру. Для объекта X2 не определено обработчиков в диалоге, но с ним можно связать переменную типа HOBJECT, тогда при инициализации диалога в ней будет сохранен дескриптор данного экземпляра объекта. Имея дескриптор, можно манипулировать объектом с помощью его специфических функций.

1. Элементы организации интерфейса
   1.1. Статический текст
   1.2. Рамка
   1.3. Ссылка
   1.4. Пиктограмма (картинка)
   1.5. Окно закладок
   1.6. Сплиттер
   1.7. Прогресс-бар
   1.8. Анимация
   1.9. Web-browser
2. Элементы ввода данных
   2.1. Флаг
   2.2. Переключатель
   2.3. Редакторы
   2.4. Выпадающий список
   2.5. Слайдер (ползунок)
   2.6. ColorPicker (элемент выбора цвета)
3. Кнопки
4. Объекты X2

7.5.2 Общие свойства управляющих элементов

Вид диалога настройки общих свойств



Флаги настроек и их имена:

1. Видимость
   1.1. Определяет, будет ли управляющий элемент видимым при инициализации диалога
   1.2. Имя флага для функций CtrlGetOptions и CtrlSetOptions – CO_VISIBLE
2. Доступность
   2.1. Определяет, будет ли управляющий элемент доступным при инициализации диалога
   2.2. Имя флага для функций GetCtrlOptions и SetCtrlOptions – CO_DISABLED

7.5.3 Идентификатор

Идентификатор формируется автоматически на этапе разработки при добавлении управляющего элемента на форму диалога. В соответствующее поле свойств управляющего элемента записывается имя константы, которое формируется средой разработки. Имена формируются по правилу «ID_XXXX», где XXXX – четырехзначное число, начиная с 1000. Это имя может быть изменено программистом на этапе разработки на более осмысленное имя. Имена констант должны буть уникальны в рамках одного диалога, иначе возникнет ошибка компиляции.

В период разработки для управляющего элемента задается только имя константы, которая будет служить идентификатором управляющего элемента. Значения же констант формируются компилятором. При компиляции диалога он добавляет во встроенную декларацию все константы, связанные с управляющими элементами, и присваивает им конкретные целочисленные значения. Эти значения могут меняться при перекомпиляции объекта, поэтому, обращаясь к управляющим элементам, не следует использовать числовые литералы. Вместо этого следует использовать имена констант.

В качестве идентификатора управляющего элемента может, также, использоваться константа, объявленная во внешней декларации. Это может быть удобно, если требуется обратиться к управляющему элементу данного диалога из другого объекта. В этом случае разработчик должен сам позаботиться об уникальности значений констант. Например, можно использовать значения, которые заведомо больше числа элементов на форме диалога.

7.5.4 Буферная переменная

Буферная переменная – это переменная, которая служит буфером обмена между окном управляющего элемента и прикладным кодом на языке X2. Обмен данных между управляющими элементами и буферными переменными осуществляется посредством функции DlgExchange.

В качестве буферной переменной может выступать локальная переменная, либо параметр диалога. Тип буферной переменной зависит от конкретного управляющего элемента. Если в качестве буферной переменной выступает параметр, переданный по ссылке, то изменения в окне управляющего элемента диалога будут доступны объекту, вызвавшему этот диалог. Нельзя использовать глобальную переменную в качестве буферной.

Не все управляющие элементы могут иметь буферные переменные.

7.5.5 Координаты и размер

Все оконные управляющие элементы имеют собственные координаты и размер по вертикали и горизонтали. Координаты отсчитываются от левого верхнего угла формы диалога. Они задаются путем перемещения элемента на форме в период разработки. В период исполнения программист не может изменять координаты и размер управляющих элементов, но эти два параметра могут меняться при изменении размеров диалога пользователем. Характер этих изменений зависит от привязки границ управляющего элемента к границам формы.

7.5.6 Привязка к границам формы

Все оконные управляющие элементы имеют свойство привязки своих границ к границам формы диалога. Это свойство определяет, каким образом будет изменять размеры или перемещаться управляющий элемент при изменении размеров диалога. Если диалог имеет сплиттер, то данное свойство определяет привязку управляющего элемента к границам соответствующей панели сплиттера.

Свойство имеет четыре поля: top, left, bottom, right. Поля означают границы управляющего элемента. Каждое из них может содержать одну из пяти констант: top, left, bottom, right и center. Константа означает, к какой границе формы привязывается данное поле.

Например, вариант (top = top, left = left, bottom = bottom, right = right) означает, что верхняя граница элемента привязана к верхней границе формы, левая – к левой, нижняя – к нижней, правая – к правой. Т.е. при растяжении диалога управляющий элемент будет соответственно растягиваться в двух направлениях.

Вариант (top = top, left = left, bottom = top, right = left) означает, что управляющий элемент не будет перемещаться.

Вариант (top = top, left = left, bottom = top, right = right) означает, что управляющий элемент будет растягиваться только в горизонтальном направлении.

7.5.7 Функции общего назначения

Для каждого управляющего элемента определен собственный набор функций доступа. Все функции доступа принимают в качестве параметра идентификатор управляющего элемента. Для большинства элементов определены следующие функции, которые мы будем называть функциями общего назначения:

- CtrlShow – сделать элемент видимым или невидимым
- CtrlEnable – сделать элемент доступным или недоступным
- CtrlGetText – получить текст
- CtrlSetText – установить текст
- CtrlSetFocus – передать фокус ввода управляющему элементу
- CtrlGetOptions – получить флаги настроек
- CtrlSetOptions – установить флаги настроек

7.6 Окно закладок

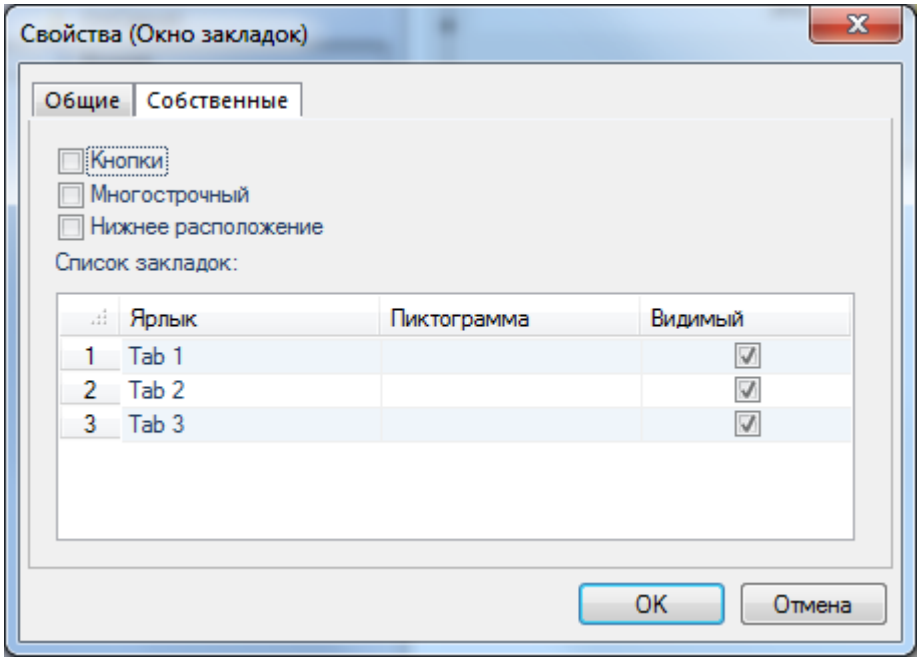
Окно закладок представляет собой прямоугольную область на форме диалога, имеющую один или более ярлыков. Каждый такой ярлык и является закладкой. Каждая из закладок может содержать произвольное количество управляющих элементов. При переключении закладок управляющие элементы текущей закладки становятся видимыми, а управляющие элементы остальных закладок – невидимыми.

На одной форме диалога можно разместить произвольное количество окон закладок.

Закладка может содержать управляющие элементы любого типа, кроме сплиттера и другого окна закладок, в том числе, и объекты X2. Переключение закладок не влияет на признак видимости управляющего элемента. В частности, если с помощью функции CtrlShow сделать видимым какой-либо управляющий элемент, лежащий на одной из закладок, то он появится на экране только тогда, когда эта закладка станет текущей.

Все закладки окна закладок должны быть заданы в период разработки. В период исполнения нельзя создать новую закладку или удалить существующую, но можно управлять их видимостью. Если закладку сделать невидимой, то из окна закладок исчезает ее ярлык и, соответственно, пользователь не может ее активизировать. Таким образом, если требуется по некоторому условию иметь закладки, либо «A» и «B», либо «A» и «C», то на этапе разработки следует создать закладки «A», «B» и «C», а в период исполнения делать видимыми только нужные.

7.6.1 Свойства



Флаги настроек и их имена:

1. Кнопки (флаг CO_PUSHLIKE)
   1.1. Если установлен, закладки будут выглядеть, как кнопки
2. Многострочный (флаг CO_MULTILINE)
   2.1. Если установлен, закладки будут выстраиваться в несколько строк, чтобы все уместились на экране. Если флаг снят, то все закладки будут располагаться в одну строку с возможностью прокрутки.
3. Нижнее расположение (флаг CO_BOTTOM)
   3.1. Если установлен, закладки будут выстраиваться в нижней части окна

Список закладок содержит наименования закладок. Он поддерживает функции, доступные через контекстное меню:

1. Добавить
2. Удалить
3. Переместить (вверх-вниз)

Настройки закладки:

4. Ярлык – содержит наименование закладки
5. Пиктограмма – содержит наименование пиктограммы в таблице x2Image
6. Видимость – если флаг установлен, при инициализации закладка будет видимой

7.6.2 Функции доступа

Поддерживает функции общего доступа:

- CtrlShow
- CtrlEnable
- CtrlGetOptions
- CtrlSetOptions

Собственные функции:

- TabGetCount
- TabGetCurSel
- TabSetCurSel
- TabGetText
- TabSetText
- TabGetImage
- TabSetImage
- TabIsVisible
- TabShow

7.7 Сплиттер

Сплиттер представляет собой вертикальную или горизонтальную полосу, разделяющую некоторую область на две панели. В период исполнения пользователь может перемещать сплиттер, изменяя тем самым размер его панелей.

Сплиттеров на форме диалога может быть сколько угодно, но их иерархия должна быть древовидной. Сплиттер имеет собственный идентификатор и идентификатор родителя. У корневого сплиттера родителя нет (ид. родителя = 0) и он делит на две панели саму форму диалога. Следующий сплиттер ссылается на корневой и делит надвое одну из панелей родительского. И т.д.

На этапе разработки сплиттер выглядит, как вертикальная или горизонтальная полоса. Разработчик просто размещает ее на форме диалога в том месте, где он предполагает увидеть ее в период исполнения. При этом для каждого сплиттера он задает его ID и ID его родителя, выстраивая иерархию таким образом, чтобы на каждой панели каждого сплиттера присутствовало не более одного дочернего сплиттера. Можно, также, задать ширину сплиттера (по умолчанию – 6 пикс.).

В период исполнения построитель окон анализирует положение сплиттеров, их ширину и иерархию, и строит реальный объект периода исполнения. В результате получается некоторый набор панелей с древовидной иерархией. С каждой панелью построитель окон связывает управляющие элементы, которые попали в нее, т.е. верхний левый угол управляющего элемента находится внутри прямоугольника панели. После этого настройки привязки управляющего элемента начинают действовать относительно границ панели, а не границ формы диалога.

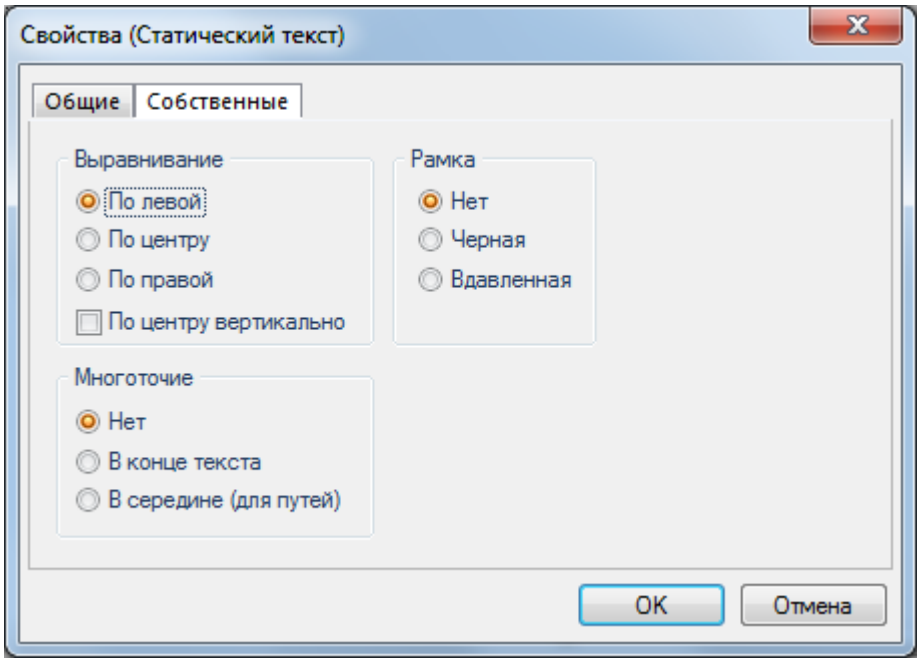
7.7.1 Функции доступа

Сплиттер не имеет собственных функций управления. Общие функции управляющих элементов, также, для него не поддерживаются и возвращают код ошибки ERR_NOT_SUPPORTED.

7.8 Статический текст

Статический текст – это поле, содержащее произвольный текст, и не предназначенное для пользовательского ввода.

7.8.1 Свойства



1. Выравнивание (переключатель)
   1.1. Состояние «По левой» (флаг CO_LEFT)
   1.2. Состояние «По центру» (флаг CO_CENTER)
   1.3. Состояние «По правой» (флаг CO_RIGHT)
   1.4. Состояние «По центру вертикально» (флаг CO_CENTERIMAGE)
2. Рамка (переключатель)
   2.1. Состояние «Черная» (флаг CO_BORDER)
   2.2. Состояние «Вдавленная» (флаг CO_SUNKEN)
3. Многоточие (переключатель)
   3.1. Назначение: если строка не помещается в окне, она будет усечена и дополнена многоточием в соответствии с состоянием переключателя. Для текста обычно усекается конец строки. Для путей – средняя часть до последнего символа «\».
   3.2. Состояние «В конце текста» (флаг CO_ENDELLIPSIS)
   3.3. Состояние «В середине (для путей)» (флаг CO_PATHELLIPSIS)

7.8.2 Функции доступа

Поддерживает только общие функции доступа.

Функции CtrlEnable и CtrlSetFocus не поддерживаются.

7.9 Рамка

Рамка представляет собой прямоугольник, предназначенный для визуального разделения других управляющих элементов. Рамка может содержать текст, который отображается в ее верхнем левом углу.

Рамка не предназначена для пользовательского ввода, не вызывает обработчиков и не может получать фокус ввода. Рамка, также, не имеет буферной переменной.

7.9.1 Функции доступа

Поддерживает только общие функции доступа.

Функции CtrlEnable и CtrlSetFocus не поддерживаются.

7.10 Пиктограмма

Изображение управляющего элемента «Пиктограмма» определяется значением буферной переменной. Переменная может иметь тип string или тип HBIN. В случае типа string, переменная должна содержать имя файла из таблицы x2Image (поле FileName). В случае типа HBIN, объект Binary должен содержать само изображение. Изображение может быть загружено в объект Binary любым способом, как из дискового файла, так и из произвольной таблицы БД.

Поддерживаемые форматы: BMP, GIF, JPEG, PNG, TIFF.

Изображение может выводиться на экран тремя способами.

1. Размер окна урпавляющего элемента подгоняется под размер изображения. Это может быть удобно при отображении маленьких пиктограмм, например, логотипов, чтобы границы окна не выступали за границы изображения.
2. Изображение масштабируется по размерам окна. При этом пропорции изображения сохраняются. Это может быть удобно при отображении больших изображений, которые не помещаются в окне, но их нужно показать полностью.
3. Изображение отображается в реальном размере, но, если оно не помещается в окне полностью, его можно сдвигать с помощью мышки или полос прокрутки.

Способ отображения регулируется набором флагов настроек. Флаги можно установить заранее на этапе разработки через интерфейс настроек, но можно и поменять в ходе исполнения программы. В этом случае, чтобы новые настройки вступили в силу, нужно вызвать функцию DlgExchange с параметром FALSE. Пример этого можно найти в демонстрационном модуле. Путь по меню: «Диалоги \ Управляющие элементы \ Настройки изображения».

Пиктограмма может обрабатывать сообщения мыши (флаг «Вызывать OnNotify»).

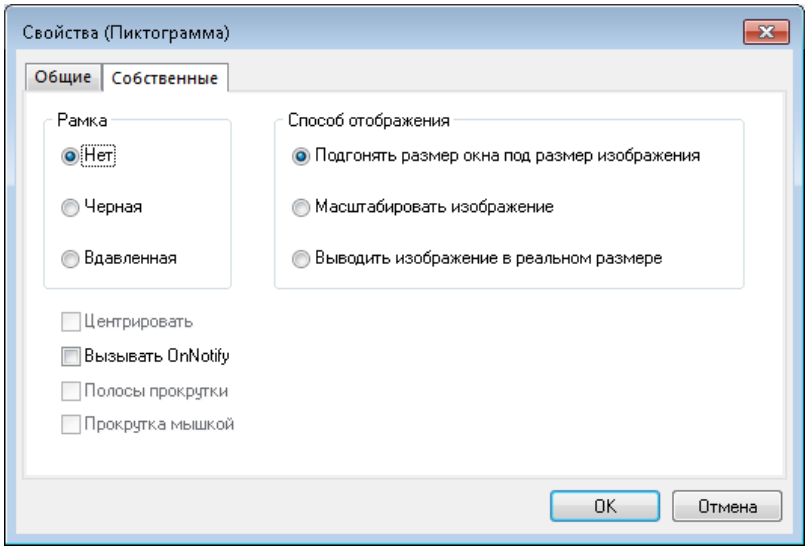
7.10.1 Обработка сообщений мыши

Если установлен флаг «Вызывать OnNotify» (CO_CALLHANDLER) пиктограмма будет обрабатывать сообщения мыши: левый клик и правый клик. При этом вызывается обработчик OnNotify с параметрами:

- idCtrl – идентификатор управляющего элемента
- nCode – код события (одна из констант)
    - NOTIFY_LCLICK – левый клик
    - NOTIFY_RCLICK – правый клик
- POINT & Param1 – содержит координаты курсора в момент события
- int & Param2 – содержит дополнительные флаги события
    - MK_SHIFT – нажата клавиша Shift
    - MK_CONTROL – нажата клавиша Ctrl

Пример обработки клавиш мыши можно найти в демонстрационном модуле: «Диалоги \ Управляющие элементы \ Карта».

7.10.2 Свойства



1. Рамка (переключатель)
   1.1. Состояние «Нет»
   1.2. Состояние «Черная» (флаг CO_BORDER)
   1.3. Состояние «Вдавленная» (флаг CO_SUNKEN)
2. Группа «Способ отображения»
   2.1. Подгонять размер окна под размер изображения (флаги CO_SCALE_PICTURE и CO_REAL_SIZE должны быть сняты)
      2.1.1. В этом случае окно управляющего элемента будет иметь те же размеры, что и изображение. Размеры окна не будут меняться при изменении размеров диалога, даже если этого требует способ привязки границ.
   2.2. Масштабировать изображение (флаг CO_SCALE_PICTURE)
      2.2.1. В этом случае изображение будет растянуто или сжато до размеров окна управляющего элемента. Пропорции изображения сохраняются. При изменении размеров диалога размер окна может меняться в соответствии со способом привязки границ.
   2.3. Выводить изображение в реальном размере (флаг CO_REAL_SIZE)
      2.3.1. В этом случае изображение будет выведено в своем реальном размере. Если изображение не умещается в окне, оно будет обрезано справа и (или) снизу. При изменении размеров диалога размер окна может меняться в соответствии со способом привязки границ.
3. Центрировать (флаг CO_CENTERIMAGE)
   3.1. Изображение будет отцентрировано по той оси, по которой ее размер меньше размера окна.
4. Вызывать OnNotify (флаг CO_CALLHANDLER)
   4.1. Система будет вызывать обработчик OnNotify на события мыши

5. Полосы прокрутки (флаг CO_SCROLL)
   5.1. Изображение будет иметь полосы прокрутки по той оси, по которой его размер превышает размер окна.
6. Прокрутка мышкой (флаг CO_MOUSE_SCROLL)
   6.1. Изображение можно перетягивать с помощью левой клавиши мыши, выполняя тем самым прокрутку. Это возможно только в том направлении, в котором размер изображения превышает размер окна.

Флаг CO_SUNKEN имеет приоритет над флагом CO_BORDER.

Флаг CO_CALLHANDLER не зависит от других флагов.

Если установлен флаг CO_SCALE_PICTURE, то флаги CO_SCROLL и CO_MOUSE_SCROLL игнорируются.

Если установлен флаг CO_REAL_SIZE, то флаги CO_SCROLL и CO_MOUSE_SCROLL учитываются только в том случае, когда флаг CO_CENTERIMAGE снят. Если флаг CO_CENTERIMAGE установлен, то флаги CO_SCROLL и CO_MOUSE_SCROLL игнорируются.

Если оба флага CO_SCALE_PICTURE и CO_REAL_SIZE сняты, то учитываются только флаги CO_SUNKEN, CO_BORDER и CO_CALLHANDLER. Остальные флаги игнорируются.

7.10.3 Функции доступа

Общие функции: поддерживаются все, кроме CtrlGetText и CtrlSetText и CtrlSetFocus.

7.11 Флаг (checkbox)

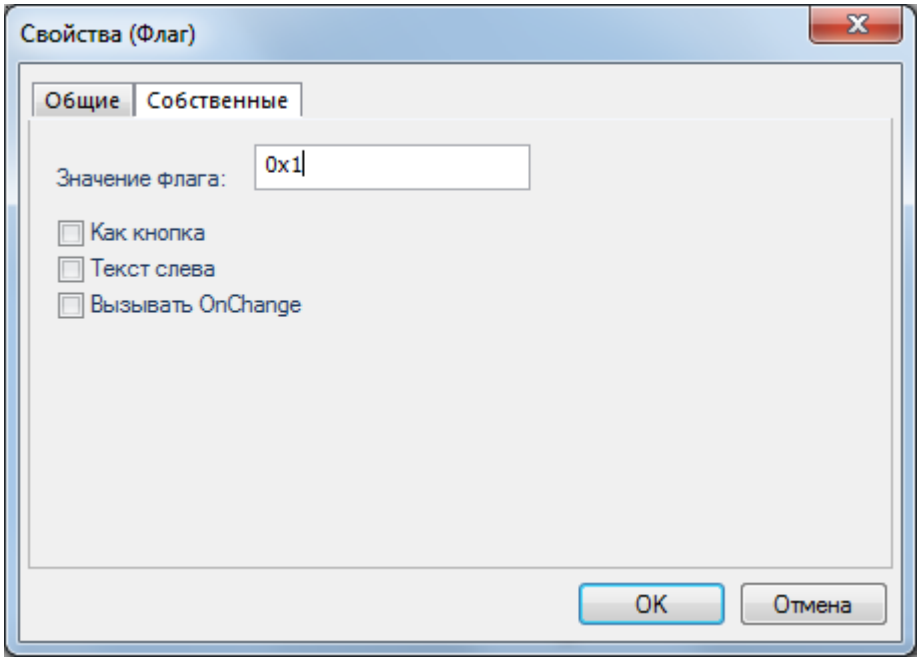
Управляющий элемент этого типа имеет два состояния: отмеченное (флаг установлен) и неотмеченное (флаг снят).

С управляющим элементом можно связать буферную переменную типа int, а также, явно указать значение флага, которое будет контролировать состояние данного элемента. В этом случае при инициализации диалога переменная будет управлять начальным состоянием элемента, а при изменении его состояния пользователем система будет автоматически устанавливать или снимать заданный флаг.

С каждым элементом можно связать отдельную переменную. В этом случае не обязательно явно задавать значение флага (если значение флага не задано, то по умолчанию используется 1). Можно, также, связать несколько элементов с одной переменной. В этом случае для каждого элемента следует задать собственное значение флага. В качестве значений флага рекомендуется использовать однобитные значения (2 в степени n), но можно задать любое число. В этом случае элемент будет управлять сразу несколькими флагами.

Можно вообще не связывать элемент с переменной. В этом случае его состоянием можно управлять с помощью функций FlagGetCheck и FlagSetCheck.

7.11.1 Свойства



1. Значение флага – позволяет ввести значение флага в явном виде
2. Как кнопка (флаг CO_PUSHLIKE)
   2.1. Если установлен, управляющий элемент будет выглядеть, как кнопка
3. Текст слева (флаг CO_LEFT)
   3.1. Если установлен, текст будет слева от изображения, иначе текст будет справа
4. Вызывать OnChange (флаг CO_CALLHANDLER)
   4.1. Если установлен, при изменении состояния управляющего элемента система будет вызывать обработчик OnChange

7.11.2 Функции доступа

Поддерживает все функции общего доступа.

Собственные функции:

- FlagGetCheck
- FlagSetCheck

7.12 Переключатель (radiobutton)

Как и флаг, переключатель имеет два состояния: отмеченное и неотмеченное. Но, в отличие от флага, переключатель имеет смысл только в группе с другими переключателями. Если один переключатель в группе включен, то все остальные выключены. Таким образом, если из флагов можно создать комбинацию, то с помощью переключателя можно выбрать один вариант из нескольких.

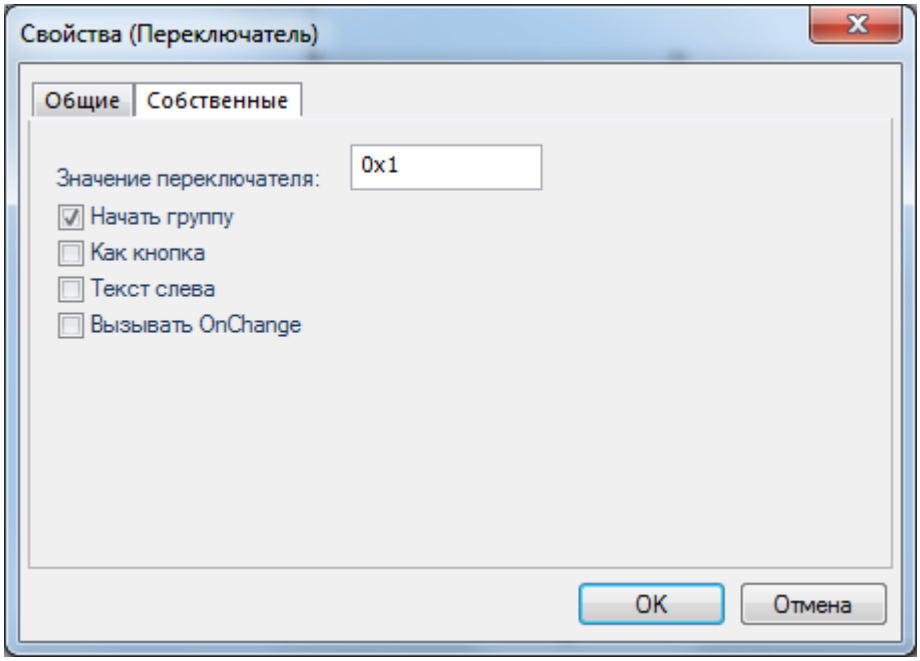
На форме может быть несколько групп переключателей. В группы переключатели объединяются с помощью признака «Начать группу» в настройках элемента. Если в настройках установлен этот флаг, то все последующие переключатели без этого признака в порядке обхода элементов будут составлять его группу. Как только в порядке обхода встречается переключатель с установленным признаком, он начинает новую группу.

С переключателем можно связать буферную переменную типа int, а также, явно указать целочисленное значение, которое переключатель будет устанавливать в связанную переменную.

С каждым элементом можно связать отдельную переменную. В этом случае не обязательно явно задавать значение переключателя (если значение не задано, то по умолчанию используется 1). Можно, также, связать несколько элементов с одной переменной. В этом случае для каждого элемента следует задать собственное значение.

Можно вообще не связывать элемент с переменной. В этом случае его состоянием можно управлять с помощью функций FlagGetCheck и FlagSetCheck.

7.12.1 Свойства



1. Значение переключателя – позволяет ввести значение в явном виде
2. Начать группу (флаг CO_GROUP)
   2.1. Если установлен, данный элемент начинает новую группу переключателей
3. Как кнопка (флаг CO_PUSHLIKE)
   3.1. Если установлен, управляющий элемент будет выглядеть, как кнопка
4. Текст слева (флаг CO_LEFT)
   4.1. Если установлен, текст будет слева от изображения, иначе текст будет справа
5. Вызывать OnChange (флаг CO_CALLHANDLER)
   5.1. Если установлен, при изменении состояния управляющего элемента система будет вызывать обработчик OnChange

7.12.2 Функции доступа

Поддерживает все функции общего доступа.

Собственные функции:

- FlagGetCheck
- FlagSetCheck

7.13 Кнопка

Кнопка, как и прочие управляющие элементы, имеет идентификатор. При нажатии на кнопку система вызывает соответствующий обработчик. Существуют предопределенные идентификаторы, для которых определены собственные обработчики (например, для кнопки с идентификатором IDOK определен обработчик OnOK). Именно они будут вызываться при нажатии на такие кнопки. Для прочих идентификаторов определен один общий обработчик OnButton, который получает идентификатор нажатой кнопки в качестве параметра.

Кнопка может содержать текст, пиктограмму и комментарий.

Пиктограмма относительно текста может размещаться справа, слева и сверху. Можно задать две пиктограммы: для доступного состояния кнопки и для недоступного. В этом случае обе пиктограммы должны иметь одинаковый размер и глубину цвета. Пиктограмму для недоступного состояния можно задать, только если задана пиктограмма для доступного.

Если комментарий не задан, то текст отображается шрифтом диалога. Если комментарий задан, то текст отображается крупным шрифтом, а комментарий – мелким.

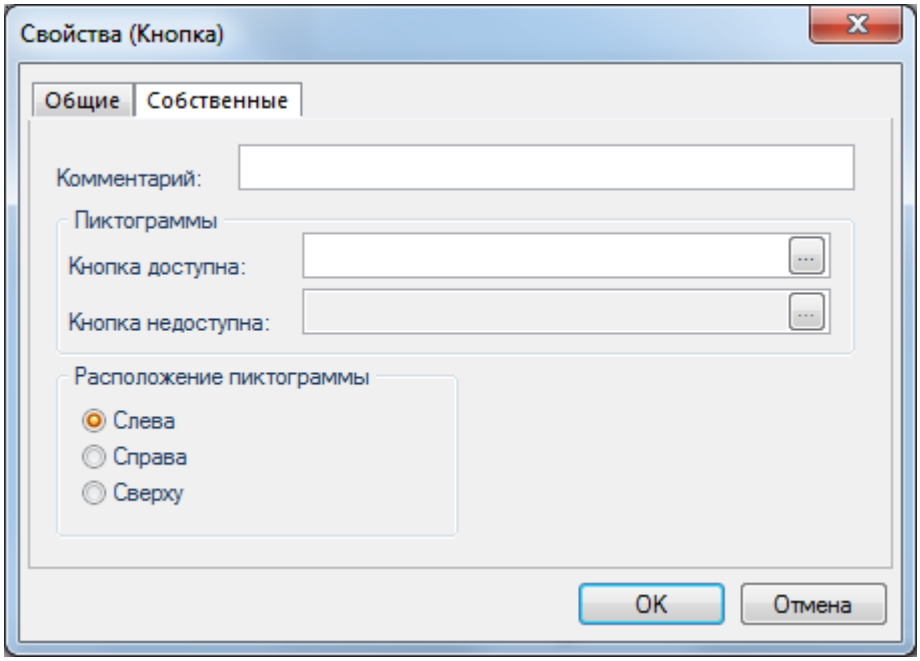
С кнопкой нельзя связать переменную.

7.13.1 Предопределенные идентификаторы кнопок

Для данных идентификаторов существуют собственные обработчики.

- IDOK – кнопка «ОК» (обработчик OnOK)
- IDCANCEL – кнопка «Отказ» (обработчик OnCancel)

7.13.2 Свойства



1. Комментарий – позволяет ввести текст комментария. Может быть пустым, если комментарий отсутствует.
2. Кнопка доступна – позволяет ввести имя пиктограммы из таблицы x2Image для кнопки в доступном состоянии. Может быть пустым, если пиктограмма отсутствует.
3. Кнопка недоступна – позволяет ввести имя пиктограммы из таблицы x2Image для кнопки в недоступном состоянии. Пиктограмма должна иметь тот же размер и глубину цвета, как и пиктограмма для доступного состояния кнопки. Может быть пустым, если пиктограмма отсутствует.
4. Расположение пиктограммы (переключатель)

   4.1. «Слева» - пиктограмма слева от текста

   4.2. «Справа» - пиктограмма справа от текста (флаг CO_RIGHT)

   4.3. «Сверху» - пиктограмма сверху от текста (флаг CO_TOP)

7.13.3 Функции доступа

Поддерживает все функции общего доступа. Функции CtrlGetText и CtrlGetText работают со строкой наименования кнопки.

Собственные функции:

- BtnGetComment
- BtnSetComment
- BtnGetImage
- BtnSetImage

7.14 Текстовый редактор

Редактор предназначен для ввода символьных данных. Редактор имеет настройку, задающую, с каким типом данных он будет работать: string, int, double или datetime. От этой настройки зависит тип буферной переменной, которую можно связать с данным редактором.

Для каждого из перечисленных типов можно задать маску ввода. Маска ввода будет соответствующим образом ограничивать пользовательский ввод. Если маска не задана, используется маска по умолчанию для данного типа. Маски ввода описаны в электронной документации.

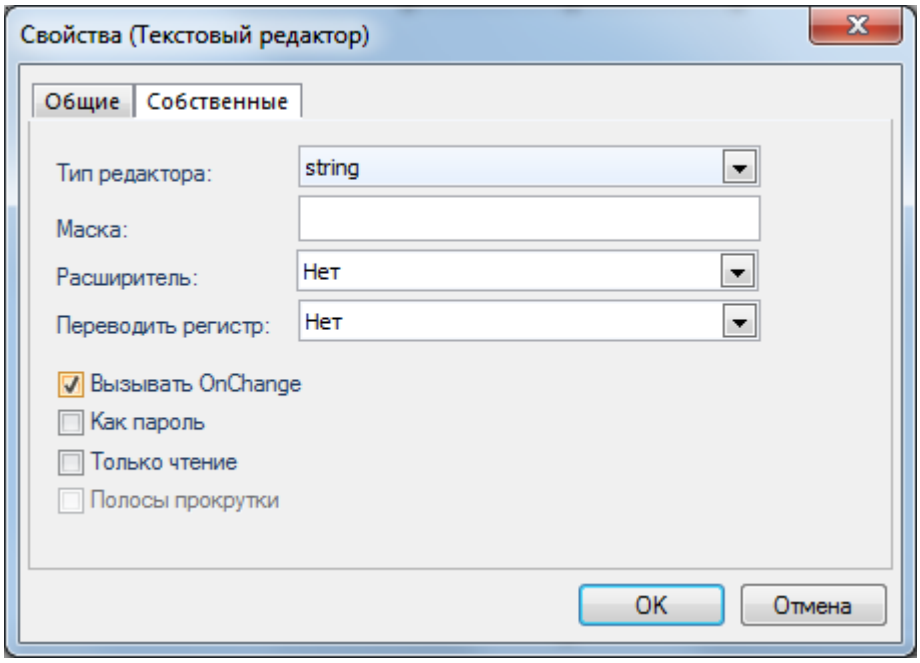
Редактор может (опционально) содержать кнопку расширителя:

- int и double – калькулятор или счетчик (spin button)
- datetime – календарь

При потере фокуса ввода редактор может вызывать обработчик OnChange, это задается в настройках поля (флаг CO_CALLHANDLER). Если флаг установлен, то обработчик будет вызываться, если данные были изменены пользователем. Если данные были изменены программистом путем прямого обращения к управляющему элементу, или путем вызова функции DlgExchange, обработчик вызываться не будет.

Если с редактором связана переменная, то при потере фокуса система будет обновлять ее значение независимо от флага CO_CALLHANDLER.

7.14.1 Свойства



1. Тип редактора – задает тип данных, с которым будет работать редактор. Он должен соответствовать типу буферной переменной, связанной с редактором, если она есть.

2. Маска – предназначено для строки, определяющей маску ввода. Если маска не задана, будет использована маска по умолчанию для данного типа. Для типа «string multiline» поле недоступно.
3. Расширитель – задает кнопку, появляющуюся справа от поля ввода
   3.1. Календарь – доступно только для типа datetime
   3.2. Калькулятор – доступно только для типов int и double
4. Переводить регистр (переключатель) – поле доступно только для типов string и string multiline
   4.1. В верхний (флаг CO_UPPERCASE)
   4.2. В нижний (флаг CO_LOWERCASE)
5. Вызывать OnChange (флаг CO_CALLHANDLER)
   5.1. Если установлен, при потере управляющим элементом фокуса ввода система будет вызывать обработчик OnChange
6. Как пароль (флаг CO_PASSWORD)
   6.1. Если установлен, ввод будет замаскирован звездочками. Не рекомендуется использовать совместно с маской ввода.
   6.2. Поле доступно, если переключатель «Тип редактора» установлен в значение «string»
7. Только чтение (флаг CO_READ_ONLY)
   7.1. Если установлен, ввод в редактор будет заблокирован, но копирование возможно
8. Полосы прокрутки (флаг CO_SCROLL)
   8.1. Если установлен, окно редактора будет содержать полосы вертикальной и горизонтальной прокрутки
   8.2. Поле доступно, если переключатель «Тип редактора» установлен в значение «string multiline»

7.14.2 Горячие клавиши

- Ctrl+Y или Ctrl+Del – очистить поле ввода
  - Очищается поле ввода
  - Если с элементом связана буферная переменная, ее значение устанавливается в значение по умолчанию, соответствующее типу переменной
  - Вызывается обработчик диалога OnChange

Примечания:

Обработчик OnChange вызывается только в том случае, если в настройках элемента установлен флаг «Вызывать OnChange» (CO_CALLHANDLER).

7.14.3 Функции доступа

Поддерживаются все общие функции доступа.

Собственные функции:

- EditGetData
- EditSetData
- EditSetMask
- EditSetEmpty
- EditBrowse

7.15 Объект X2

В качестве управляющих элементов могут выступать оконные объекты X2, такие, как списки, деревья, диалоги.

С объектом X2 можно связать буферную переменную типа HOBJECT, которая получит дескриптор созданного экземпляра объекта. Используя этот дескриптор, можно управлять объектом посредством набора его собственных функций.

Если вложенным объектом является диалог, то его кнопки с предопределенными идентификаторами (IDOK, IDCANCEL) будут скрыты.

7.15.1 Свойства

Общие свойства:

- Поле «Текст» предназначено для ввода строки вызова объекта, например, BROWSER Name(Parameters). Эта же строка отображается в период разработки в условном окне управляющего элемента.

Собственные свойства:

1. Игнорировать при каскадном вызове (флаг CO_NO_CASCADE)

   1.1. Если установлен, то при каскадном вызове обработчики OnInit, OnOK, OnCancel данного объекта вызываться не будут. Этот флаг следует установить, если программист хочет изменить стандартную цепочку вызова обработчиков и вызывать их самостоятельно.

7.15.2 Функции доступа

Поддерживаются все общие функции доступа кроме CtrlGetText и CtrlSetText.

7.16 Выпадающий список (lookup)

Выпадающий список представляет собой однострочный текстовый редактор, имеющий кнопку расширителя, при нажатии на которую выпадает список или иной визуальный объект, позволяющий выбрать строку, которая будет помещена в поле ввода редактора.

Данный управляющий элемент манипулирует несколькими значениями. Одно из них выводится в поле редактора, с ним работает пользователь. Другие (одно или более), представляют собой ключ первого, с ними работает программист. Например, если некоторая сущность идентифицируется в БД по id, то в поле редактора будет выводиться имя сущности, а id будет выступать в роли ключа.

Ввиду этого выпадающий список должен обрабатывать два события:

- Обмен данными (для конкретного ключа нужно получить значение отображения и наоборот, для значения, введенного в поле редактора нужно получить ключ)
- Нажатие на кнопку расширителя, вызывающее визуализацию интерфейса выбора значения

Как и обычный редактор, выпадающий список имеет настройку, задающую, какой тип данных будет выводиться в поле ввода: string, int, double или datetime. В зависимости от этого типа для поля ввода может быть задана маска ввода.

Существует три способа построения выпадающих списков:

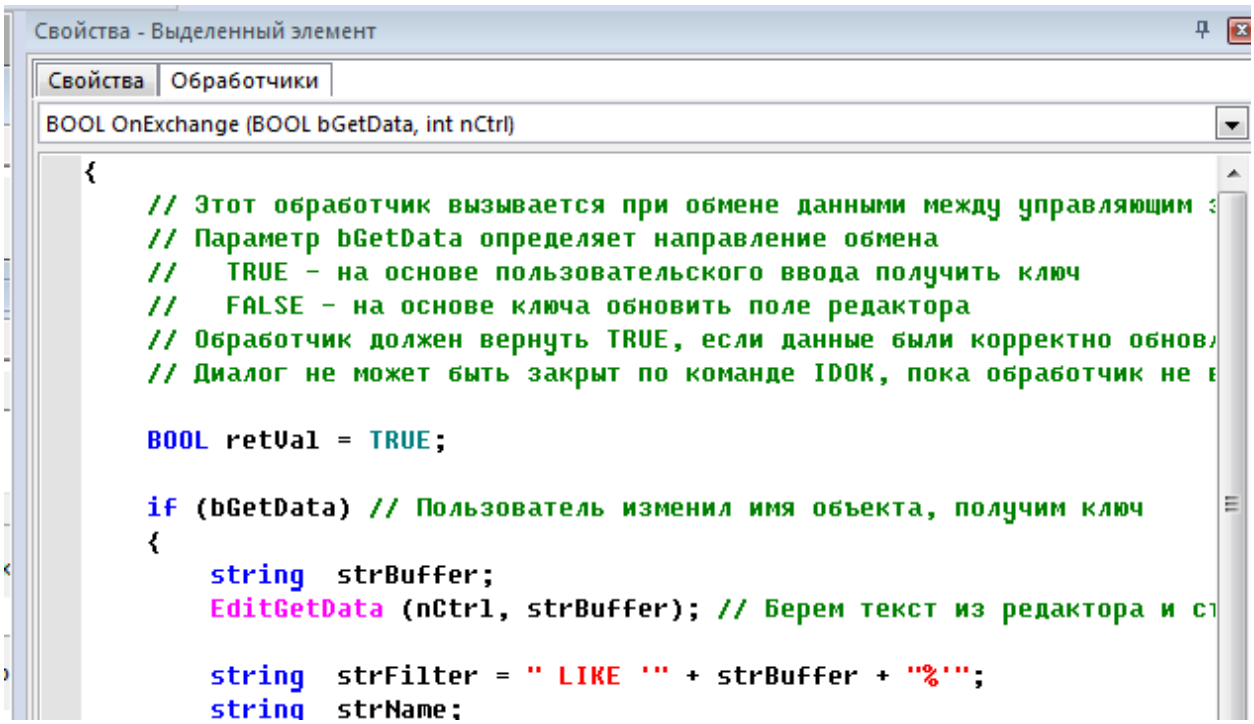
- На основе обработчиков
- На основе списка
- На основе SQL-запроса

7.16.1 На основе обработчиков

Выпадающий список на основе обработчиков не имеет буферной переменной и не выполняет автоматических действий. Вместо этого он имеет два обработчика:

- OnExchange
- OnBrowse

Код обработчиков OnExchange и OnBrowse можно ввести в панели «Свойства – выделенный элемент», закладка «Обработчики».



Обработчик OnExchange вызывается на событие обмена данных. Это происходит в следующих случаях:

- при начальной инициализации
- при потере фокуса управляющим элементом, если поле отображения было изменено пользователем
- при завершении диалога по команде IDOK
- при явном вызове команды DlgExchange

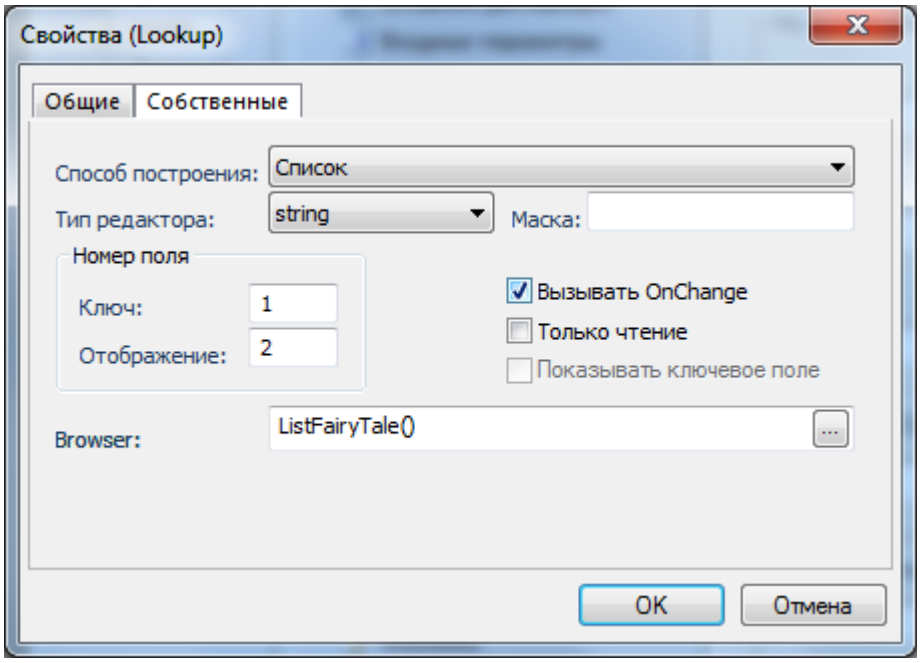
Обработчик принимает параметр, указывающий, в какую сторону должен идти обмен. Задача обработчика состоит в том, чтобы для имеющегося ключа получить поле отображения, либо для нового поля отображения получить значение ключа. Поскольку программист сам отвечает за получение данных (с сервера или иного источника), то он сам же их и помещает в нужные переменные. Ввиду этого, выпадающий список, построенный на основе обработчиков, не может иметь буферной переменной. Зато он позволяет работать с составным ключом, когда сущность идентифицируется несколькими полями таблицы.

Обработчик OnBrowse вызывается при нажатии на кнопку расширителя. Здесь программист должен вызвать требуемый объект (список, диалог, дерево и т.д.), позволяющий пользователю выбрать нужное значение. Программист сам должен связать выбранное значение с ключом.

7.16.2 На основе списка

Управляющий элемент может быть построен на основе существующего списка. Для этого при разработке нужно задать имя списка и указать номера полей его запроса, которые будут использоваться для отображения и в качестве ключа. Это задается в настройках управляющего элемента.

Номера полей задаются в полях «Ключ» и «Отображение». Имя списка задается в поле «Browser» в формате: ИмяСписка (Параметры).



При нажатии на кнопку расширителя система будет визуализировать указанный список, а при обмене данными будет автоматически получать ключ или поле отображения, используя его запрос.

В качестве ключа может выступать только одно значение (составной ключ не поддерживается). Поля ключа должно иметь тип, приводимый к одному из следующих:

- int
- double
- datetime
- string

С полем ключа может быть связана буферная переменная, которая должна иметь соответствующий тип.

Если поле отображения и поле ключа совпадают, система выполняет обмен данными так же, как и для текстового редактора, т.е. в буферную переменную попадает значение из поля редактирования.

Если поля не совпадают, то при обмене данными система обращается к SQL-серверу. Список в этом случае должен быть построен на основе оператора SELECT. Система формирует запрос вида

SELECT %1 FROM (%2) T WHERE %3

где:

- %1 – имя поля ключа либо отображения (в зависимости от направления обмена)
- %2 – запрос списка, полученный после подстановки в него переменных
- %3 – ограничение по полю ключа или отображения (в зависимости от направления обмена)

В том случае, когда поле отображения имеет строковый тип, при получении ключа система вначале формирует ограничение на равенство. Если при таком ограничении возвратился пустой набор, система пытается сформировать ограничение через оператор

LIKE Данные%

Если получена единственная строка, она инициализирует ключ и записывается в буферную переменную. Если получен набор из нескольких строк, система выводит на экран список с наложенным фильтром (LIKE Данные%). Если получен пустой набор, система выводит на экран список без фильтра.

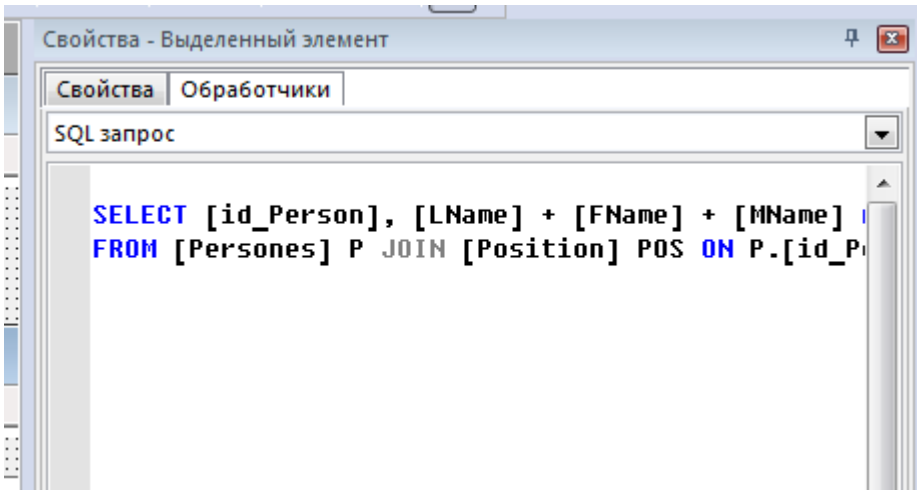
7.16.3 На основе запроса

Управляющий элемент, построенный на основе SQL-запроса, действует аналогично элементу, построенному на основе списка, только при его разработке вместо оператора визуализации списка нужно указать SQL-запрос.

Система формирует псевдо-список, который имеет настройки по умолчанию и указанный SQL-запрос.

Запрос должен представлять собой одиночный оператор SELECT. В запросе могут использоваться переменные диалога в качестве подстановки. Подстановка может выполняться только по значению, нельзя выполнять подстановку по ссылке.

Тело запроса можно ввести в панели «Свойства – выделенный элемент», закладка «Обработчики», категория «SQL-запрос».



7.16.4 Горячие клавиши

- Стрелка вниз – имитирует нажатие на кнопку расширителя
  - Для элементов на основе запроса или списка вызывает браузер с фильтром по введенному значению
  - Для элементов на основе обработчиков вызывает обработчик OnBrowse
- Shift + (Стрелка вниз или нажатие кнопки расширителя)
  - Для элементов на основе запроса или списка вызывает браузер без фильтра
  - Для элементов на основе обработчиков вызывает обработчик OnBrowse
- Ctrl+Y или Ctrl+Del – очистить поле ввода
  - Для элементов на основе запроса или списка
    - Очищается поле ввода

        - Если с элементом связана буферная переменная, ее значение устанавливается в значение по умолчанию, соответствующее типу переменной
        - Вызывается обработчик диалога OnChange
    - Для элементов на основе обработчиков
        - Очищается поле ввода
        - Вызывается обработчик OnExchange, задача которого – установить корректное значение ключа.
        - Если обработчик возвращает FALSE, флаг модификации остается поднятым
        - Если обработчик возвращает TRUE, вызывается обработчик диалога OnChange

Примечания:

Обработчик OnChange вызывается только в том случае, если в настройках элемента установлен флаг «Вызывать OnChange» (CO_CALLHANDLER).

7.16.5 Обработчик OnBrowse

Обработчик вызывается при нажатии на кнопку расширителя.

Прототип

BOOL OnBrowse (BOOL bShift, int nCtrl)

Входные параметры

- bShift – нажата ли клавиша Shift
- nCtrl – идентификатор управляющего элемента

Возвращаемое значение

Обработчик должен вернуть TRUE, если данные были изменены, и FALSE в противном случае. Этот признак система использует для вызова обработчика диалога OnChange.

7.16.6 Обработчик OnExchange

Обработчик вызывается для обмена данными.

Прототип

BOOL OnExchange (BOOL bGetData, int nCtrl)

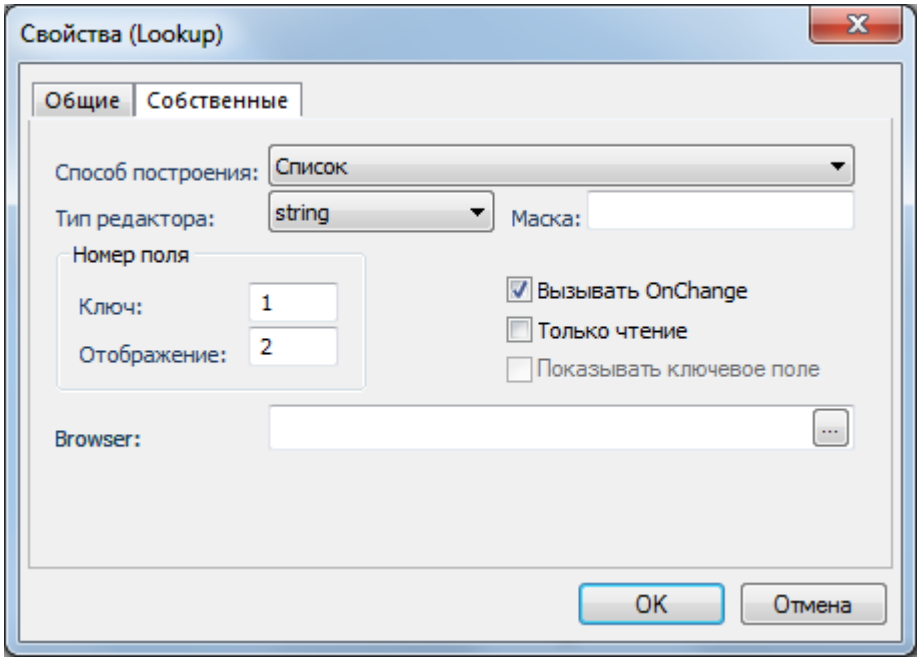
Входные параметры

- bGetData – направление обмена
  - TRUE – на основе пользовательского ввода получить ключ
  - FALSE – на основе ключа обновить поле редактора
- nCtrl – идентификатор управляющего элемента

Возвращаемое значение

Обработчик должен вернуть TRUE, если данные были корректно обновлены, и FALSE в противном случае. Диалог не может быть закрыт по команде IDOK, пока обработчик не вернет TRUE.

7.16.7 Свойства



Общие свойства:

- Поле «Текст» недоступно
- Поле «Переменная» недоступно, если переключатель «Способ построения» в состоянии «Обработчики»

Собственные свойства:

1. Способ построения – задает способ построения выпадающего списка

   1.1. Список

   1.2. SQL-запрос

   1.3. Обработчики

2. Тип редактора – задает тип данных, с которым будет работать редактор. Аналогично соответствующей настройке управляющего элемента «Текстовый редактор».

3. Маска – поле предназначено для строки, определяющей маску ввода. Аналогично соответствующей настройке управляющего элемента «Текстовый редактор».

4. Номер поля ключа – задает порядковый номер ключевого поля в запросе

   4.1. Поле доступно, если переключатель «Способ построения» в состоянии «Список» или «SQL-запрос»

   4.2. Нумерация от 1

5. Номер поля отображения – задает порядковый номер поля в запросе, которое должно отображаться в редакторе

   5.1. Поле доступно, если переключатель «Способ построения» в состоянии «Список» или «SQL-запрос»

   5.2. Нумерация от 1

6. Browser – содержит строку вызова списка в формате: ИмяСписка (Параметры)
   6.1. Поле доступно, если переключатель «Способ построения» в состоянии «Список»
7. Вызывать OnChange (флаг CO_CALLHANDLER)
   7.1. Если установлен, при потере управляющим элементом фокуса ввода система будет вызывать обработчик OnChange
8. Только чтение (флаг CO_READ_ONLY)
   8.1. Если установлен, ввод в редактор будет заблокирован, но возможен выбор значения по кнопке расширителя

7.16.8 Функции доступа

Поддерживаются все функции элемента «Текстовый редактор».

Собственные функции:

- LookupGetBrowser

7.16.9 Пример управляющего элемента на обработчиках

Путь по меню: «Диалоги\ Управляющие элементы\ Выпадающий список (lookup)»

7.17 Прогресс-бар

Управляющий элемент этого типа предназначен для отображения полосы прокрутки при продолжительных операциях.

С этим управляющим элементом нельзя связать буферную переменную, манипулировать им можно с помощью функций доступа.

7.17.1 Функции доступа

Поддерживает следующие функции общего доступа:

- CtrlShow – сделать элемент видимым или невидимым
- CtrlGetOptions – получить флаги настроек
- CtrlSetOptions – установить флаги настроек

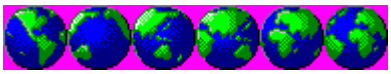
Собственные функции:

- int ProgressGetRange (int idCtrl, HOBJECT hObj = 0)
- int ProgressGetPos (int idCtrl, HOBJECT hObj = 0)
- ProgressSetRange (int idCtrl, int nRange, HOBJECT hObj = 0)
- ProgressSetPos (int idCtrl, int nPos, HOBJECT hObj = 0)
- ProgressStepIt (int idCtrl, HOBJECT hObj = 0)

7.18 Анимация

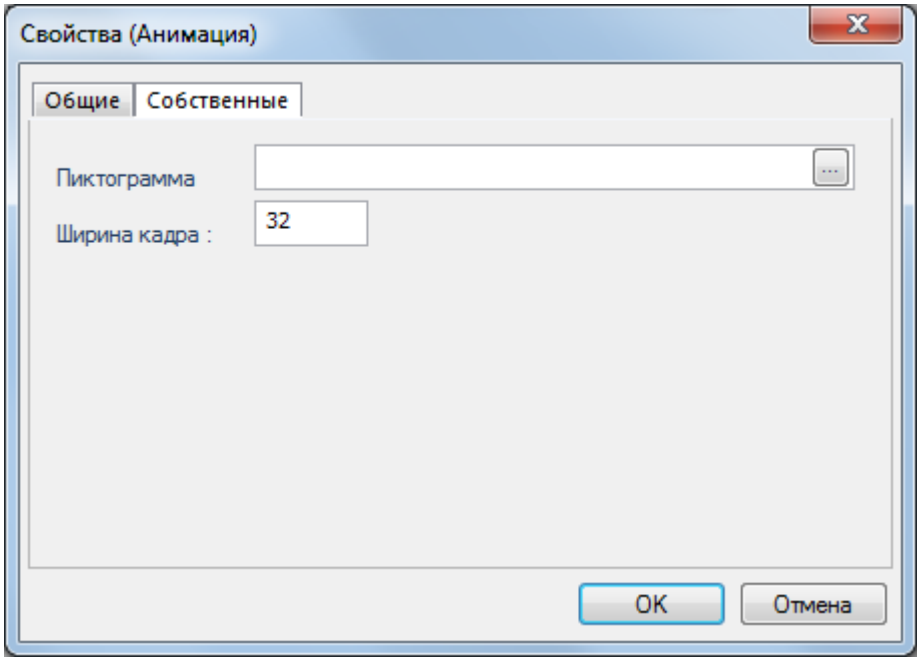
Управляющий элемент этого типа предназначен для отображения анимации. Анимация задается, как серия кадров в одной пиктограмме. Все кадры должны иметь одну и ту же ширину. Прозрачный фон задается цветом (255, 0, 255). Пиктограмма должна иметь формат bmp и храниться в таблице x2Image.

Пример пиктограммы, содержащей шесть кадров, приведен ниже:



При отображении размер управляющего элемента будет приведен к размеру кадра.

7.18.1 Свойства



1. Пиктограмма – поле предназначено для указания имени пиктограммы из таблицы x2Image, связанной с управляющим элементом
2. Ширина кадра – поле задает ширину кадра в пикселях

7.18.2 Функции доступа

Поддерживает следующие функции общего доступа:

- CtrlShow – сделать элемент видимым или невидимым
- CtrlGetOptions – получить флаги настроек
- CtrlSetOptions – установить флаги настроек

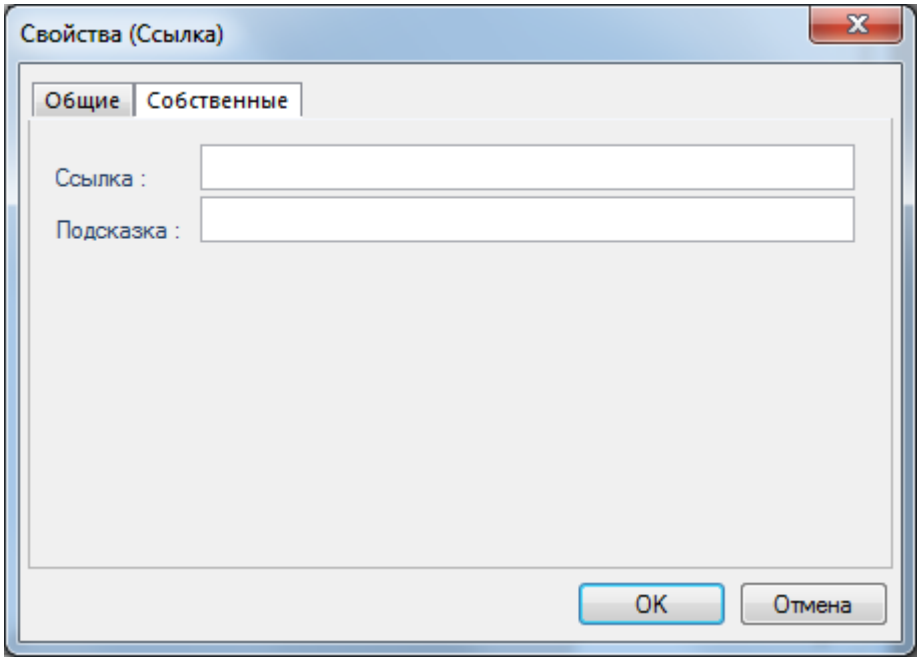
Собственные функции:

- AnimationStart (int idCtrl, int nFrameRate = 500, HOBJECT hObj = 0)
- AnimationStop (int idCtrl, HOBJECT hObj = 0)
- BOOL AnimationIsRunning (int idCtrl, HOBJECT hObj = 0)

7.19 Ссылка

Данный управляющий элемент реализует URL-ссылку. При нажатии на ссылку вызывается команда ShellExecute, которой передается строка ссылки в качестве параметра.

7.19.1 Свойства



1. Ссылка – поле предназначено для указания ссылки URL, которая будет отображаться на экране
2. Подсказка – поле предназначено для указания всплывающей подсказки

7.19.2 Функции доступа

Поддерживает следующие функции общего доступа:

- CtrlShow – сделать элемент видимым или невидимым
- CtrlEnable – сделать элемент доступным или недоступным
- CtrlSetFocus – передать фокус ввода управляющему элементу
- CtrlGetOptions – получить флаги настроек
- CtrlSetOptions – установить флаги настроек

Собственные функции:

- string LinkGetURL (int idCtrl, HOBJECT hObj = 0)

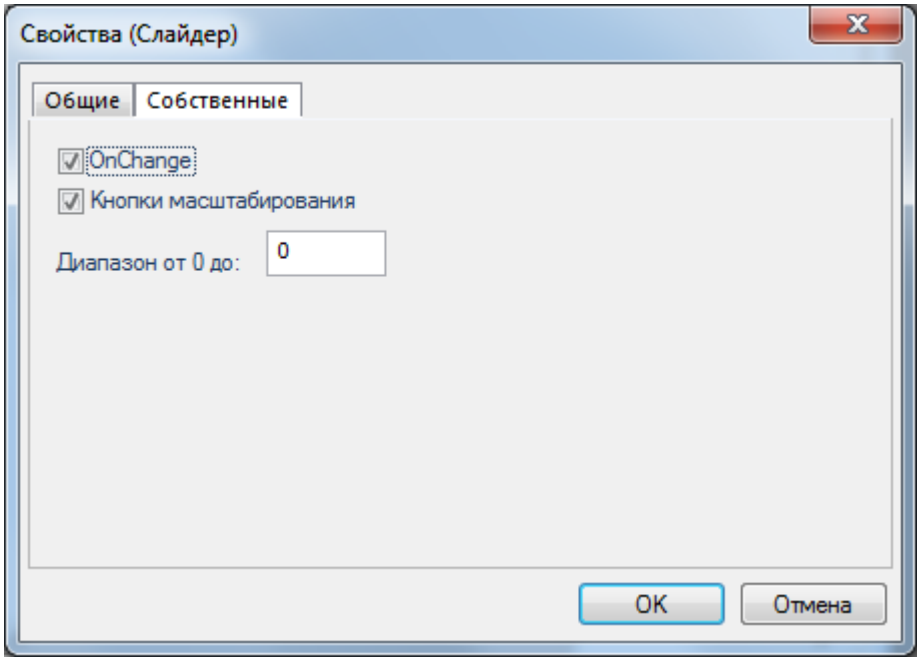
- string LinkGetTooltip (int idCtrl, HOBJECT hObj = 0)
- LinkSetURL (int idCtrl, string strURL, HOBJECT hObj = 0)
- LinkSetTooltip (int idCtrl, string strTooltip, HOBJECT hObj = 0)
- LinkSizeToContent (int idCtrl, HOBJECT hObj = 0)

7.20 Слайдер

Управляющий элемент этого типа предназначен для выбора дискретного значения из набора последовательных значений в диапазоне. Он представляет собой окно, содержащее ползунок, и (необязательно) кнопки увеличения или уменьшения значения.

С этим управляющим элементом можно связать буферную переменную типа int. В этом случае при инициализации диалога переменная будет управлять начальным положением ползунка в диапазоне значений, при изменении положения ползунка в окне слайдера переменная будет автоматически принимать значение равное позиции ползунка. Левая граница диапазона значений, которые может принимать переменная, равна нулю, а правая граница устанавливается прикладным программистом. Если начальное значение связанной переменной будет выходить за рамки диапазона последовательных значений слайдера, то оно будет приведено к значению ближайшей границы диапазона автоматически. Так же этим элементом управления можно манипулировать с помощью специальных функций доступа.

7.20.1 Свойства



1. Флаги:

   1.1. OnChange (флаг CO_CALLHANDLER)

      1.1.1. Если установлен, при изменении позиции ползунка в окне слайдера система будет вызывать обработчик OnChange

   1.2. Кнопки масштабирования

      1.2.1. Если флаг установлен, управляющий элемент будет дополнен кнопками масштабирования по краям слайдера

2. Диапазон – поле предназначено для указания правой границы диапазона значений слайдера

7.20.2 Функции доступа

Поддерживает следующие функции общего доступа:

- CtrlShow – сделать элемент видимым или невидимым
- CtrlEnable – сделать элемент доступным или недоступным
- CtrlSetFocus – передать фокус ввода управляющему элементу
- CtrlGetOptions – получить флаги настроек
- CtrlSetOptions – установить флаги настроек

Собственные функции:

- int SliderGetRange(int idCtrl, HOBJECT hObj = 0)
- int SliderGetPos(int idCtrl, HOBJECT hObj = 0)
- SliderSetRange (int idCtrl, int nRange, HOBJECT hObj = 0)
- SliderSetPos(int idCtrl, int nPos, HOBJECT hObj = 0)

7.21 Web-браузер

Объект этого типа предназначен для отображения содержимого web-страниц. Адрес web-страницы может указываться в виде строки в буферной переменной. Если переменная есть и в ней указан какой-то верный адрес web-страницы, то при инициализации элемента будет осуществлён переход по этому адресу. При переходе на другую страницу внутри этого объекта вызывается обработчик OnChange, в котором происходит изменение значения связанной переменной на текущее значение URL-ссылки страницы. Если URL-ссылка содержит инструкцию, указывающую необходимость её открытия в новом окне, то для этой web-страницы значение связанной переменной web-браузера не будет обновляться, так как это уже самостоятельный объект, неконтролируемый X2.

Web-браузер посылает диалогу два вида уведомлений:

1. NOTIFY_BEFORE_NAVIGATION – когда пользователь перешёл по URL-ссылке, но страница ещё не прогрузилась.
2. NOTIFY_DOCUMENT_COMPLETE – когда страница полностью прогрузилась.

7.21.1 Свойства

Общие свойства:

- Поле «Текст» недоступно
- Поле «Переменная» имеет тип string

Собственные свойства:

3. Флаги:

3.1. OnChange (CheckBox)

3.1.1. Если установлен, при изменении текущего URL открытой страницы система будет вызывать обработчик OnChange

3.1.2. Значение для функций CtrlGetOptions и CtrlSetOptions – CO_CALLHANDLER

3.1.3. По умолчанию флаг установлен

3.1.4. Поле доступно всегда

7.21.2 Функции доступа

Поддерживает следующие функции общего доступа:

- CtrlShow – сделать элемент видимым или невидимым
- CtrlEnable – сделать элемент доступным или недоступным
- CtrlSetFocus – передать фокус ввода управляющему элементу
- CtrlGetOptions – получить флаги настроек
- CtrlSetOptions – установить флаги настроек

Собственные функции:

- string WebBrowserLocationName(int idCtrl, HOBJECT hObj = 0)
- string WebBrowserLocationUrl(int idCtrl, HOBJECT hObj = 0)
- WebBrowserNavigate (int idCtrl, HOBJECT hObj = 0)
- WebBrowserNavigate (int idCtrl, string strUrl, HOBJECT hObj = 0)
- WebBrowserGoBack (int idCtrl, HOBJECT hObj = 0)
- WebBrowserGoForward (int idCtrl, HOBJECT hObj = 0)
- WebBrowserRefresh (int idCtrl, HOBJECT hObj = 0)
- WebBrowserStop (int idCtrl, HOBJECT hObj = 0)

7.22 ColorPicker (элемент выбора цвета)

Управляющий элемент Color picker предназначен для выбора цвета. Он имеет окно, отображающее выбранный цвет, и кнопку расширителя для выбора цвета (опционально).

С данным управляющим элементом может быть связана переменная типа COLORREF. Значение переменной соответствует выбранному цвету.

Состояние элемента по умолчанию – RGB (0, 0, 0). При изменении состояния управляющего элемента вызывается обработчик диалога OnChange.

Пример использования элемента можно найти в демонстрационном модуле по пути: «Прочие объекты \ Блок-схемы \ Редактор». См. диалог DlgGetFillColor.

7.22.1 Свойства

Общие свойства:

- Поле «Текст» недоступно
- Поле «Переменная» имеет тип COLORREF

Собственные свойства:

5. Флаг:
   5.1. Кнопка выбора цвета (CheckBox)
      5.1.1. Если установлен, элемент имеет кнопку расширителя для выбора цвета.
6. Флаг:
   6.1. Вызывать OnChange (CheckBox)
      6.1.1. Если установлен, при выборе цвета из палитры система будет вызывать обработчик OnChange
      6.1.2. Значение для функций CtrlGetOptions и CtrlSetOptions – CO_CALLHANDLER
7. Флаг:
   7.1. Вызывать OnNotify (CheckBox)
      7.1.1. Если установлен, при нажатии на кнопку элемента с изображением стрелочки вниз система будет вызывать обработчик OnNotify
      7.1.2. Значение для функций CtrlGetOptions и CtrlSetOptions – CPO_ON_NOTIFY

7.22.2 Функции доступа

Поддерживает все функции общего доступа кроме CtrlGetText и CtrlSetText.

Собственные функции:

- COLORREF PickerGetColor (int idCtrl, HOBJECT hObj = 0)
- BOOL PickerSetColor (int idCtrl, COLORREF color, HOBJECT hObj = 0)

8 Деревья

8.1 Оператор создания экземпляра – TREE

HOBJECT TREE name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта «Дерево» и возвращает дескриптор экземпляра.

8.2 Механизм построения дерева

Каждый узел дерева имеет уникальный идентификатор (тип int), который генерируется системой автоматически при добавлении нового узла. Этот идентификатор используется функциями API для идентификации узлов. API позволяет управлять узлами дерева в ходе исполнения: добавлять, удалять, менять свойства узлов.

За каждым узлом дерева можно закрепить некие произвольные данные. Данные могут иметь, как простой встроенный тип (int, string и т.д.), так и структурированный, заданный разработчиком.

Дерево строится в обработчике OnExpand. Данный обработчик вызывается для корневых узлов дерева:

- при разворачивании узла, если его нижележащий уровень еще не построен, т.е. узел не имеет ни одного дочернего узла
- при сворачивании узла

Для листевых узлов, а также, для корневых узлов, имеющих нижележащий уровень, обработчик не вызывается.

Обработчик принимает на входе параметры:

- nItem – идентификатор узла дерева, который должен быть развернут/свернут. Если nItem = 0, то это означает коревой уровень дерева.
- bExpand – если TRUE, обработчик вызывается на разворачивание узла, если FALSE – на сворачивание.

При разворачивании узла задача обработчика – построить нижележащий уровень для указанного узла. Для добавления нового узла используется функция TreeAddItem, которой в качестве параметра можно передать данные, необходимые для разворачивания этого узла в дальнейшем. Это может быть id строки в БД или целая структура, заданная разработчиком.

Например, если дерево в БД представлено в виде

```
CREATE TABLE [Tree]
(
        [id_Item]     int PRIMARY KEY,
        [id_Parent]   int FOREIGN KEY REFERENCES [Tree] ([id_Item]) NULL,
        [Name]        nvarchar (200) NOT NULL
        …
        …
)
```

то таким данными может служить id_Item. Тогда при разворачивании этого узла в обработчике может быть код, подобный следующему:

```
// Здесь nItem, это параметр, переданный в обработчик системой.
// Он содержит идентификатор разворачиваемого узла в дереве.
// Он был сгенерирован системой при создании узла и не равен id строки в БД.
// id строки мы храним в данных, связанных с узлом.

        string   strName;
        int      id_Item, id_ItemNew;

        // Получим данные родительского узла
        TreeItemGetData (nItem, id_Item);

        // Загрузим всех детей
        HDS hDS = OpenDataSet (id_ItemNew, strName)
        {
                SELECT [id_Item],[Name] FROM [Tree] WHERE [id_Parent] = :id_Item
        };

        if (hDS)
        {
                // Добавим детей в цикле
                while (!IsEOF(hDS))
                {
                        TreeAddItem (nItem, strName, id_ItemNew);
                        Fetch(hDS);
                }

                CloseDataSet(hDS);
        }
```

Здесь следует уточнить, что функция TreeAddItem, создавая новый узел дерева, копирует данные из переменной id_ItemNew во внутренний буфер этого узла. Соответственно, функция TreeItemGetData копирует их из внутреннего буфера в переменную id_Item.

При сворачивании узла обработчик может удалить дочерние узлы, чтобы освободить занятые ресурсы, но это не является обязательным. Удалить дочерние узлы можно с помощью функции TreeDropChildren. Этот код в обработчике может выглядеть так:

```
        if (!bExpand)
        {
                TreeDropChildren (nItem);
                return;
        }
```

8.3 Примеры построения дерева

Примеры построения дерева приводятся в демонстрационном модуле (путь по меню «Объкты\Деревья»).

1. Пример «Обработчики» демонстрирует, как можно реализовать обработчики, отвечающие за конкретные функции дерева, такие, как редактирование или перетаскивание веток мышкой.
2. Пример «Дерево со списком» реализует один из вариантов организации взаимодействия между деревом и списком внутри одного диалога, а также, демонстрирует работу обработчика OnSelChange, реагирующего на изменение текущей позиции в дереве.

Кром этого можно найти пример дерева по пути в меню «Объкты\Диалоги\Управляющие элементы\Выпадающий список». Там дерево используется в одном из выпадающих списков.

9 Меню

Меню не может быть отображено, как самостоятельный объект, и может быть только привязано к другому визуальному объекту. При создании экземпляра визуального объекта вместе с ним создается и экземпляр его меню. При отображении визуального объекта отображается и его меню. Способ отображения меню зависит от режима отображения визуального объекта.

Меню может быть связано:

- с модулем
- с визуальным объектом (список, диалог и прочие)
- с другим меню в качестве вложенного подменю

9.1 Оператор создания экземпляра – MENU

HOBJECT MENU name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта меню и возвращает дескриптор экземпляра.

Примечания:

Если в настройках объекта указано меню, то оно будет создаваться и уничтожаться автоматически вместе с объектом. Оператор создания меню следует использовать только в тех случаях, когда требуется динамически подменить меню, связанное с объектом.

Оператор MENU создает экземпляр меню, но не отображает его на экране. Для отображения меню его нужно связать с другим визуальным объектом, например, со списком, с помощью функций SetMenu и SetContextMenu. Если объект уже визуализирован, то после подмены его меню следует вызвать функцию RefreshMenu, чтобы обновить полоску главного меню.

При уничтожении объекта система автоматически удаляет связанные с ним меню. Если выполнялась подмена меню, то связанным с объектом становится новое меню, а старое становится свободным. Соответственно, система не будет удалять старое а удалит новое.

9.2 Экземпляр меню

Для каждого экземпляра объекта, имеющего связанное меню, создается отдельный экземпляр меню. Таким образом, одно и то же меню может существовать в системе в нескольких экземплярах. Как и прочие экземпляры объектов в системе, экземпляр меню идентифицируется посредством дескриптора (тип HOBJECT).

Экземпляр меню создается и уничтожается вместе с экземпляром родительского объекта.

Пункт меню идентифицируется посредством строкового идентификатора, уникального в рамках данного меню. Наличие и уникальность идентификаторов пунктов меню контролируется транслятором.

9.3 Вложенные меню

На этапе разработки в состав одного меню может быть включено другое меню. В зависимости от настроек вложенное меню может отображаться двумя способами:

- в виде всплывающего меню
- пункты вложенного меню будут встраиваться между пунктами родительского меню в указанном месте

Количество вложенных меню, как и уровень их вложенности, технически не ограничены.

При создании экземпляра родительского меню автоматически создается и экземпляр вложенного меню, который имеет собственный дескриптор. Если данное вложенное меню тоже имеет вложенное, то будет создан и его экземпляр с собственным дескриптором.

Функция GetOwner для вложенного меню возвращает дескриптор экземпляра родительского меню.

9.4 Отображение меню

Меню, связанное с модулем отображается при входе в модуль.

Меню, связанное с визуальным объектом, отображается, когда объект активизируется, т.е. одно из дочерних окон его фрейма получает фокус ввода. Если объект отображен в модальном фрейме, то его меню замещает существующее, если в немодальном, то его меню дополняет меню модуля справа. Если объект содержит вложенный объект, то, при позиционировании в нем его меню отображается вместо меню родительского объекта.

В частности, это означает, следующее. При входе в модуль отобразилось меню M, связанное с модулем. Далее вызывается диалог в модальном фрейме, и его меню M1 замещает M. Этот диалог имеет вложенный список с меню M11. При переключении на список получаем меню M11. При переключении на диалог получаем M1. Далее вызывается диалог в немодальном фрейме, имеющий меню M2. Отображается M+M2. Данный диалог тоже имеет вложенный список с меню M21. Переводим фокус ввода в данный список, получаем M+M21.

Вложенное меню отображается вместе с родительским.

Если у вложенного объекта нет собственного меню, то вместо него будет отображено меню родительского объекта.

9.5 Доступ к меню

Функции работы с меню позволяют программисту обращаться к экземпляру меню, используя его дескриптор. При этом изменения в одном экземпляре не затрагивают других экземпляров.

Линейка меню на экране может быть составлена из нескольких различных экземпляров меню, каждый из которых имеет собственный дескриптор:

- Меню модуля, либо меню модального фрейма
    - Его вложенные меню
- Меню активного немодального фрейма
    - Его вложенные меню

Дескриптор меню, связанного с визуальным объектом либо модулем, можно получить посредством функции GetMenu. Доступ экземпляру вложенного меню осуществляется посредством функции MenuGetSubMenu.

9.6 Обработчики меню

Объект меню имеет ряд сегментов кода, свойственных всем объектам языка X2. Это:

6. Конструктор
7. Деструктор
8. Обработчики событий

    8.1. OnInit – инициализация

    8.2. OnMessage – обработка пользовательских сообщений

Конструктор вызывается при создании экземпляра меню до отображения его на экране. Конструктор меню вызывается до конструктора родительского объекта. Деструктор меню вызывается при удалении экземпляра меню до вызова деструктора родительского объекта.

Обработчик OnInit вызывается каждый раз при визуализации экземпляра меню.

Обработчик OnMessage будет вызван при отправке экземпляру меню пользовательского сообщения.

Кроме этого, каждый пункт меню, имеющий тип «Команда» (см раздел «Свойства пунктов меню») имеет собственный обработчик, который вызывается при нажатии на данный пункт. Обработчик представляет собой произвольный фрагмент кода на языке X2.

9.6.1 Обработчик команды меню

Обработчик привязан к конкретному пункту меню и срабатывает при выборе этого пункта.

Прототип:

(string strID)

Входные параметры:

- strID – идентификатор данного пункта меню

Обработчик не возвращает значения.

9.7 Примеры работы с меню

Меню нельзя отобразить на экране непосредственно, но можно привязать его к какому-либо визуальному объекту (диалогу, списку, дереву). Как правило, меню привязывают к объекту на этапе разработки. Для этого нужно войти в свойства объекта и указать имя меню и параметры вызова. Меню, как и прочие объекты X2, может принимать входные параметры. В качестве параметров могут выступать локальные переменные объекта, либо константы. Если меню не имеет параметров, нужно указать пустые скобки «()». При компиляции объекта транслятор загрузит его меню и проверит параметры на соответствие по числу и типам.

9.7.1 Подмена меню

Меню, привязанное к объекту, считается дочерним по отношению к нему. Оно будет отображаться на экране вместе с окном родительского объекта, и будет уничтожаться при уничтожении родителя. Если меню привязано к объекту на этапе разработки, его экземпляр будет создаваться системой автоматически. Но можно привязать меню к объекту с помощью программного кода в период исполнения. Для этого нужно вначале создать экземпляр меню с помощью оператора MENU, а потом связать с объектом посредством функции SetMenu. С этого момента новое меню станет дочерним для данного объекта. Если объект уже имел дочернее меню, то оно станет свободным, и не будет уничтожено объектом при закрытии. Если не планируется его дальнейшее использование, то его следует уничтожить вручную с помощью функции DestroyObject. Стоит отметить, что функция SetMenu не перестраивает меню на экране, а только подменяет дескриптор, связанный с объектом. Если меню уже на экране, его нужно перестроить с помощью функции RefreshMenu. Если же мы подменяем меню в конструкторе объекта, то вызывать RefreshMenu не нужно, т.к. оно еще не построено.

Этот код может выглядеть так:

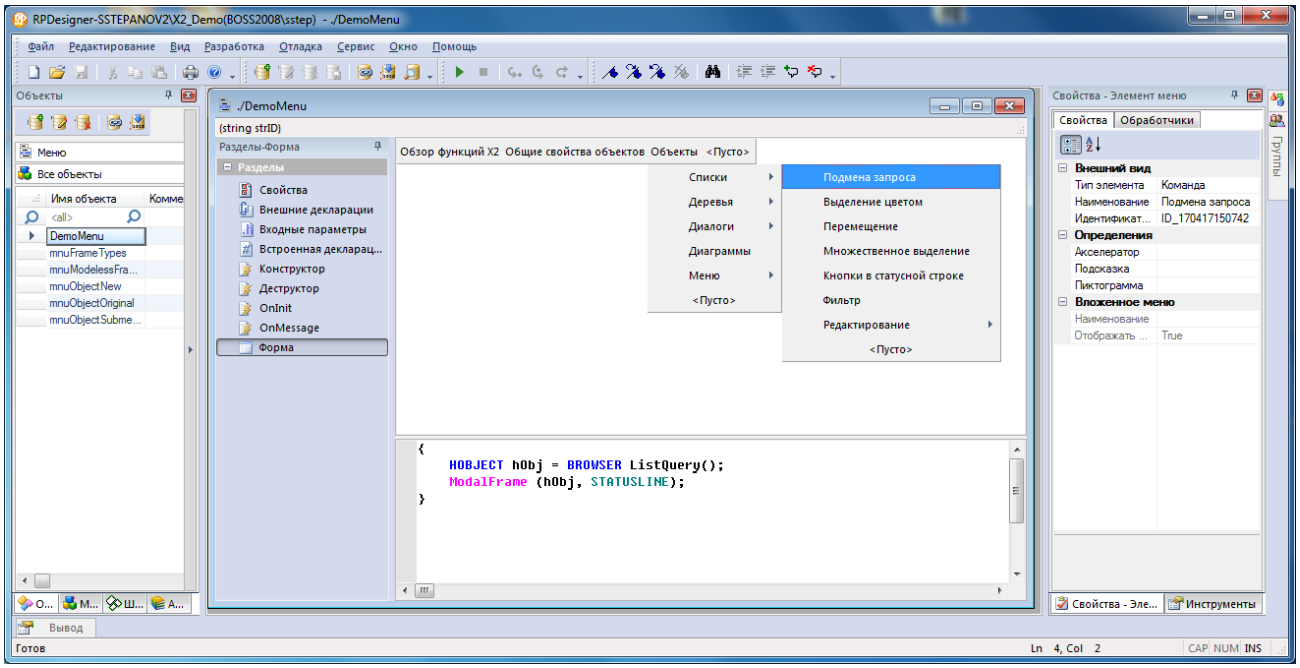
```
HOBJECT hMenu = MENU MyMenu ();

hMenu = SetMenu (hMenu);
RefreshMenu ();

if (IsObject(hMenu))
        DestroyObject(hMenu);
```

В демонстрационном модуле есть пример, иллюстрирующий подмену меню в период исполнения (путь «Меню\Подмена меню»).

9.7.2 Обработчики пунктов меню

Объект меню может содержать пункты следующих видов: команда, разделитель, группа, вложенное меню. Команды имеют обработчики, другие пункты обработчиков не имеют.

В редакторе формы меню обработчик расположен в нижней части под сплиттером.



Обработчик пункта меню может содержать любой код на языке X2. Как правило, он содержит код вызова какого-либо визуального объекта. Обработчик будет исполнен, когда пользователь выберет этот пункт.

9.7.3 Управление пунктами меню

В период исполнения можно менять свойства пунктов меню с помощью программного кода. Можно изменить:

- Наименование пункта
- Подсказку, которая выводится в статусной строке
- Пиктограмму, связанную с пунктом меню
- Состояние доступности (можно делать недоступными целые группы, а также, пункты верхнего уровня)
- Признак того, что пункт отмечен. Если пункт отмечен, то слева от него в области пиктограмм выводится галочка. Это возможно только для пунктов, не имеющих пиктограмм.

Пример управления пунктами меню в демонстрационном модуле: «Меню\ Управление пунктами меню». В этом примере, также, показано, как по дескриптору экземпляра меню вывести все его пункты в линейный список и запросить их свойства.

Обращаясь к пункту меню, мы используем его идентификатор. При создании нового пункта система автоматически генерирует идентификатор этого пункта. Он выглядит, примерно, так: ID_230517124122. Можно оставить его, как есть, но, если предполагается обращаться к этому пункту (например, чтобы управлять его доступностью), то идентификатор стоит задать самостоятельно, придав ему более читаемый вид. Например, если обработчик этого пункта выводит на экран дерево, то можно присвоить ему такой идентификатор: IDM_SHOW_TREE. Идентификатор, это всего лишь строка. Она может быть любой. Важно только, чтобы она была уникальной в рамках одного объекта.

```
MenuEnableItem (ID_230517124122, FALSE, hMenu);
MenuEnableItem (IDM_SHOW_TREE, FALSE, hMenu);
```

Сравните эти два вызова. Второй выглядит более читаемым.

10 Диаграммы

Вид диаграммы определяется двумя признаками: категория диаграммы и ее тип.

Категория диаграммы определяет ее собственный способ отображения. Например: «Круговая», «Пузырьковая», «Гистограмма», и т.д.

Тип определяет то, как будут взаимодействовать между собой различные серии точек внутри одного объекта. Например, серии могут отображаться независимо друг от друга, складываться в стек, или же складываться в стек с нормализацией к 100%.

Существует, также, ряд настраиваемых сущностей, используемых при отображении диаграммы:

- Оси координат (будет ли отображаться сама ось, ее название, какими будут шкала и диапазон и т.д.)
- Линия (прямая, кривая, ступенька, сплошная, штриховая и т.д.)
- Маркеры (небольшие визуальные объекты, которые обозначают точки, задающие диаграмму или график, и используются исключительно для наглядности)
- Метки (небольшие текстовые поля, содержащие значения точек, используемые, как правило, вместе с маркерами)
- Таблица значений (таблица, располагающаяся, как правило, под диаграммой, в которой содержатся значения точек серии)
- Легенда (объект прямоугольной формы, содержащий метку цвета серии, ее название и, в некоторых случаях, дополнительную информацию)

10.1 Категории диаграмм

Группа	Пояснение	Категория	
		Константа	Аналог 3D
Линейные	График	CC_Line	CC_Line3D
	Область	CC_Area	CC_Area3D
	Колонки	CC_Column	CC_Column3D
	Бар	CC_Bar	CC_Bar3D
Фигурные	Круговая	CC_Pie	CC_Pie3D
	Пирамида	CC_Pyramid	CC_Pyramid3D
	Воронка	CC_Funnel	CC_Funnel3D
	Кольцо	CC_Doughnut	CC_Doughnut3D
	Вложенные кольца	CC_DoughnutNested	
	Тор		CC_Torus3D
Прочие	Полярная	CC_Polar	
	Поверхность		CC_Surface3D
	Биржевая		CC_Stock
	Гистограмма	CC_Histogram	
	Пузырьковая	CC_Bubble	CC_Bubble

10.2 Типы диаграмм

Константа	Пояснение
CT_DEFAULT	Используется тип по умолчанию для данной категории
CT_SIMPLE	Серии отображаются независимо друг от друга
CT_STACKED	Серии отображаются общим стеком
CT_100STACKED	Серии отображаются стеком с нормализацией к 100%
CT_RANGE	Серия имеет два значения по оси Y (позиция и размер). Применимо к линейным диаграммам.

10.3 Оператор создания экземпляра – CHART

HOBJECT CHART name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта «Диаграмма» и возвращает дескриптор экземпляра.

10.4 Общая методика построения диаграмм

Как и прочие объекты X2, объект «Диаграмма» имеет стандартный набор обработчиков (конструктор, деструктор, OnInit и т.д.), но кроме этого, существует еще специальный сегмент OnBuild.

Если диаграмма требует какой-то подготовки данных в БД, то лучше ее выполнить в конструкторе, а в деструкторе освободить захваченные ресурсы.

В сегменте OnInit можно выполнить какие-то действия с окнами. Например, если разработчик хочет, чтобы диаграмма отображалась не в диалоге, а в собственном фрейме, но при этом нужно дать пользователю возможность настраивать диаграмму, то в OnInit можно выбросить прилипающую панель с настройками.

Сама же диаграмма должна строиться в сегменте OnBuild. Последовательность вызова сегментов при создании экземпляра объекта такова: конструктор, OnInit, OnBuild. Сегмент OnBuild можно, также, вызвать явным образом с помощью функции ChartRebuild.

Диаграмма строится на основе серии точек. Сколько диаграмм требуется построить, столько серий точек и должно быть задано. При построении нужно:

1. Создать серию
2. Добавить в серию точки
3. Изменить настройки, если это требуется для следующей серии
4. Повторить первые три шага требуемое количество раз

Серия создается функцией ChartCreateSeries. При создании серии указываются ее категория и тип. Можно совмещать серии различных категорий, но все категории должны быть логически совместимы. Например, можно совместить линейный график, область и гистограмму, но совмещать биржевую диаграмму с пирамидой не имеет смысла. Для большинства диаграмм используется тип CT_SIMPLE, т.е. когда все серии отображаются независимо друг от друга. Этот тип применим к любой категории. А вот тип CT_STACKED имеет смысл не для всех категорий. Например, колонки или линейные графики можно отобразить стеком, а вот для пузырьковой диаграммы это не имеет

смысла. Если при построении диаграммы используется тип, отличный от CT_SIMPLE, то все серии должны иметь одну и ту же категорию и тип.

Точки в серию добавляются с помощью семейства функций ChartAdd… Для различных категорий применяются различные функции. Связано это с тем, что разные категории диаграмм требуют разных наборов данных, а также, строятся в разных осях. В частности:

- ChartAddBubbleData – для пузырьковой
- ChartAddStockData – для биржевой
- ChartAddRangeData – для диаграмм типа CT_RANGE
- ChartAddDataXY – для двумерных с числовыми осями
- ChartAddDataXYZ – для поверхностей
- ChartAddData – для всех остальных

Для различных серий могут потребоваться различные настройки встроенных элементов: координатных осей, вида линии, вида маркеров и меток и т.д. Изначально все эти настройки задаются на этапе разработки в редакторе свойств объекта, но в период исполнения они могут быть изменены программным путем. Поэтому, если следующая серия требует других настроек, то перед ее построением настройки следует изменить. Для этого используются функции работы с настройками (см. раздел «Перечень функций по категориям»). Вернуться к исходным настройкам, заданным на этапе разработки, можно с помощью функции ChartRestoreDefault.

10.4.1 Эффект анимации

Для вызова эффекта анимации служит функция ChartAnimate. Функция может быть вызвана в любой момент после того, как диаграмма уже построена. Если анимация раздражает пользователя, то он может отключить ее через контекстное меню. В этом случае функция ChartAnimate не будет выполнять никаких действий.

Флаг блокировки анимации сохраняется вместе с остальными пользовательскими настройками объекта.

10.4.2 Автоматическое обновление диаграмм

Если требуется обновлять диаграмму автоматически, например, чтобы в реальном времени отображать динамику изменения данных в БД, то следует воспользоваться таймером. Таймеры создаются посредством функции SetTimer.

Таймер нужно создать в сегменте OnInit, а затем в сегменте OnMessage ловить его сообщения и перестраивать диаграмму (функция ChartRebuild). Не следует создавать таймер в конструкторе, т.к. конструктор отрабатывает до первого построения диаграммы.

Стоит отметить, что таймеры не имеют прямого отношения непосредственно к диаграммам. Это независимый механизм, он может использоваться и с другими объектами.

10.5 Функции объекта

- Общие настройки диаграммы
  - string ChartGetName(HOBJECT hObj = 0);
  - BOOL ChartSetName(string strName, HOBJECT hObj = 0);
  - int ChartGetTransparency(HOBJECT hObj = 0);
  - BOOL ChartSetTransparency(int nTransparency, HOBJECT hObj = 0);
  - BOOL ChartRestoreDefault(HOBJECT hObj = 0)
- Настройка осей
  - BOOL ChartAxisGetName (int nAxis, string & strName, HOBJECT hObj = 0);
  - BOOL ChartAxisSetName (int nAxis, string strName, HOBJECT hObj = 0);
  - BOOL ChartAxisGetMask (int nAxis, string & strMask, HOBJECT hObj = 0);
  - BOOL ChartAxisSetMask (int nAxis, string strMask, HOBJECT hObj = 0);
  - BOOL ChartAxisGetOptions (int nAxis, int& nOptions, double & dMin, double & dMax, HOBJECT hObj = 0);
  - BOOL ChartAxisSetOptions (int nAxis, int nOptions, double dMin = 0, double dMax = 0, HOBJECT hObj = 0);
- Настройка таблицы значений
  - int ChartTableGetOptions (HOBJECT hObj = 0);
  - BOOL ChartTableSetOptions(int nOptions, HOBJECT hObj = 0);
- Настройка легенды
  - BOOL ChartLegendGetOptions(BOOL & bFrame, int & nPos, HOBJECT hObj = 0);
  - BOOL ChartLegendSetOptions(BOOL bFrame, int nPos, HOBJECT hObj = 0);
- Настройка вида линии
  - int ChartCurveGetWidth(HOBJECT hObj = 0);
  - BOOL ChartCurveSetWidth (int nWidth, HOBJECT hObj = 0);
  - BOOL ChartCurveGetStyle (int & nCurveType, int & nDashStyle, HOBJECT hObj = 0);
  - BOOL ChartCurveSetStyle (int nCurveType, int nDashStyle, HOBJECT hObj = 0);
- Настройка маркеров
  - BOOL ChartMarkersShow (BOOL bShow, HOBJECT hObj = 0);
  - BOOL ChartMarkersGetStyle (int& nSize, int& nShape, HOBJECT hObj = 0);
  - BOOL ChartMarkersSetStyle (int nSize, int nShape, HOBJECT hObj = 0);
- Настройка меток
  - int ChartLabelsGetOptions (HOBJECT hObj = 0);
  - BOOL ChartLabelsSetOptions (int nOptions, HOBJECT hObj = 0);

    - int ChartLabelsGetPos (HOBJECT hObj = 0);
    - BOOL ChartLabelsSetPos (int nPos, HOBJECT hObj = 0);
- Команды
    - BOOL ChartRedraw (HOBJECT hObj = 0);
    - BOOL ChartReBuild (HOBJECT hObj = 0);
    - BOOL ChartClear (HOBJECT hObj = 0);
    - BOOL ChartAnimate (double dTime, int nStyle, HOBJECT hObj = 0);
- Создать серию
    - BOOL ChartCreateSeries(string strName, int nCategory, int nType, int nOptions = 0, HOBJECT hObj = 0);
- Добавить точку в числовых координатах
    - BOOL ChartAddDataXY (int nSeries, double dX, double dY, HOBJECT hObj = 0);
    - BOOL ChartAddDataXYZ (int nSeries, double dX, double dY, double dZ, HOBJECT hObj = 0);
- Добавить значение для сущности
    - BOOL ChartAddData (int nSeries, string strEntity, double dY, BOOL bPieExplosion = FALSE, HOBJECT hObj = 0);
    - BOOL ChartAddData (int nSeries, double dY, HOBJECT hObj = 0);
- Добавить точку для пузырьковой диаграммы
    - BOOL ChartAddBubbleData (int nSeries, string strEntity, double dX, double dY, double dSize, BOOL bBubble3D = TRUE, HOBJECT hObj = 0);
- Добавить пункт для биржевой диаграммы
    - BOOL ChartAddStockData(int nSeries, double dOpen, double dHigh, double dLow, double dClose, datetime time, HOBJECT hObj = 0);
- Добавить точку для серии типа CT_RANGE
    - BOOL ChartAddRangeData(int nSeries, double dX, double dY, double dSize, HOBJECT hObj = 0);
    - BOOL ChartAddRangeData(int nSeries, string strEntity, double dY, double dSize, HOBJECT hObj = 0);
- Удалить все точки серии
    - ChartRemoveData(int nSeries, BOOL bRedraw = FALSE, HOBJECT hObj = 0);

10.6 Примеры построения диаграмм

Приведенные ниже примеры можно найти в демонстрационном модуле. Путь по меню: «Прочие объекты\ Диаграммы».

10.6.1 Линейные диаграммы

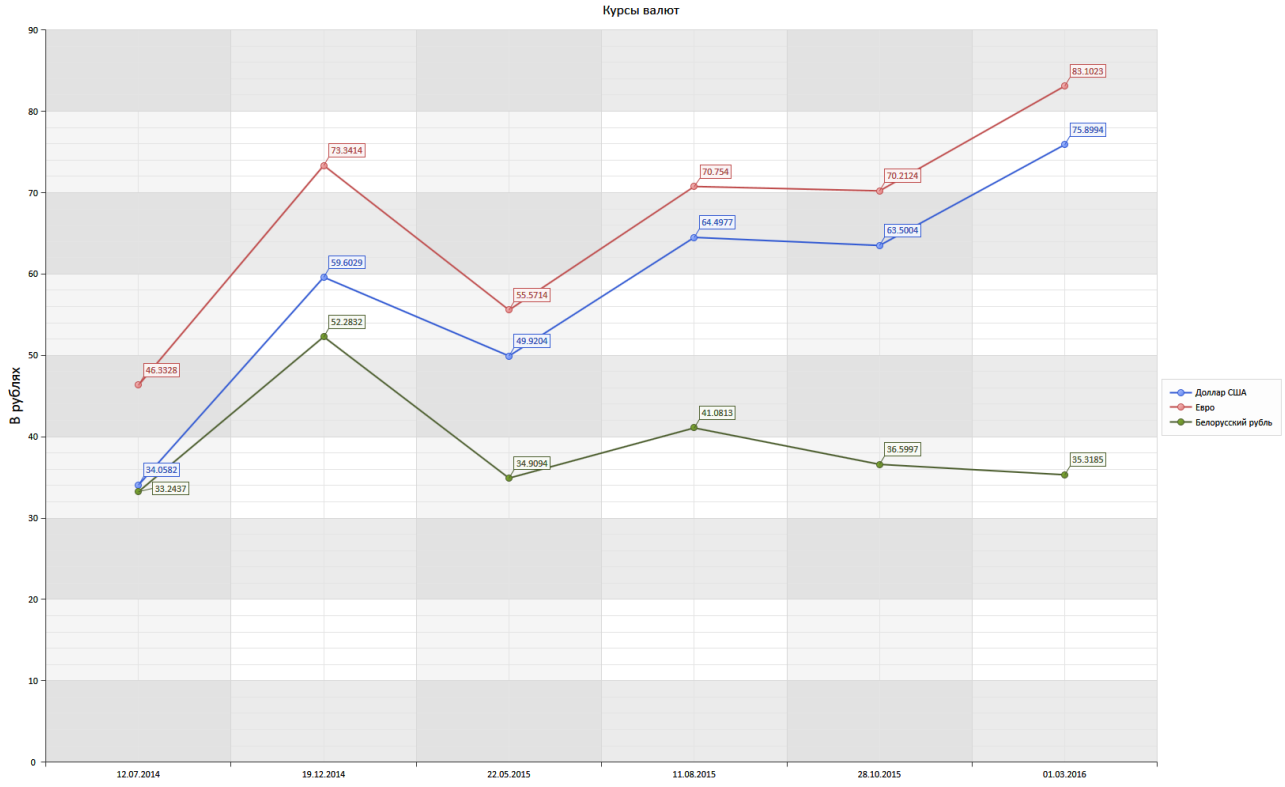
Приведенный ниже код позволяет строить линейные диаграммы всех категорий. Настройка задается двумя переменными: nCategory и nType. Меняя значения этих переменных, можно получить разные виды диаграмм.

```
int nCategory = CC_Line;
int nType = CT_SIMPLE;

ChartSetName("Курсы валют");

if (nCategory == CC_Area)
{
	ChartSetTransparency(70);
	ChartCurveSetWidth(1);
}

if (nType == CT_100STACKED)
{
	ChartAxisSetName(AXIS_Y, "В процентах");
}
else
{
	ChartAxisSetName(AXIS_Y, "В рублях");
}


ChartCreateSeries("Доллар США",		nCategory, nType);
ChartCreateSeries("Евро",		nCategory, nType);
ChartCreateSeries("Белорусский рубль",	nCategory, nType);

// USD
if (nType == CT_RANGE)
{
	ChartAddRangeData(0, "12.07.2014", 34.0582, 10.0);
	ChartAddRangeData(0, "19.12.2014", 59.6029, 20.0);
	ChartAddRangeData(0, "22.05.2015", 49.9204, 15.0);
	ChartAddRangeData(0, "11.08.2015", 64.4977, 23.0);
	ChartAddRangeData(0, "28.10.2015", 63.5004, 22.0);
	ChartAddRangeData(0, "01.03.2016", 75.8994, 30.0);
}
else
{
	ChartAddData(0, "12.07.2014", 34.0582);
	ChartAddData(0, "19.12.2014", 59.6029);
	ChartAddData(0, "22.05.2015", 49.9204);
	ChartAddData(0, "11.08.2015", 64.4977);
	ChartAddData(0, "28.10.2015", 63.5004);
	ChartAddData(0, "01.03.2016", 75.8994);
}
```

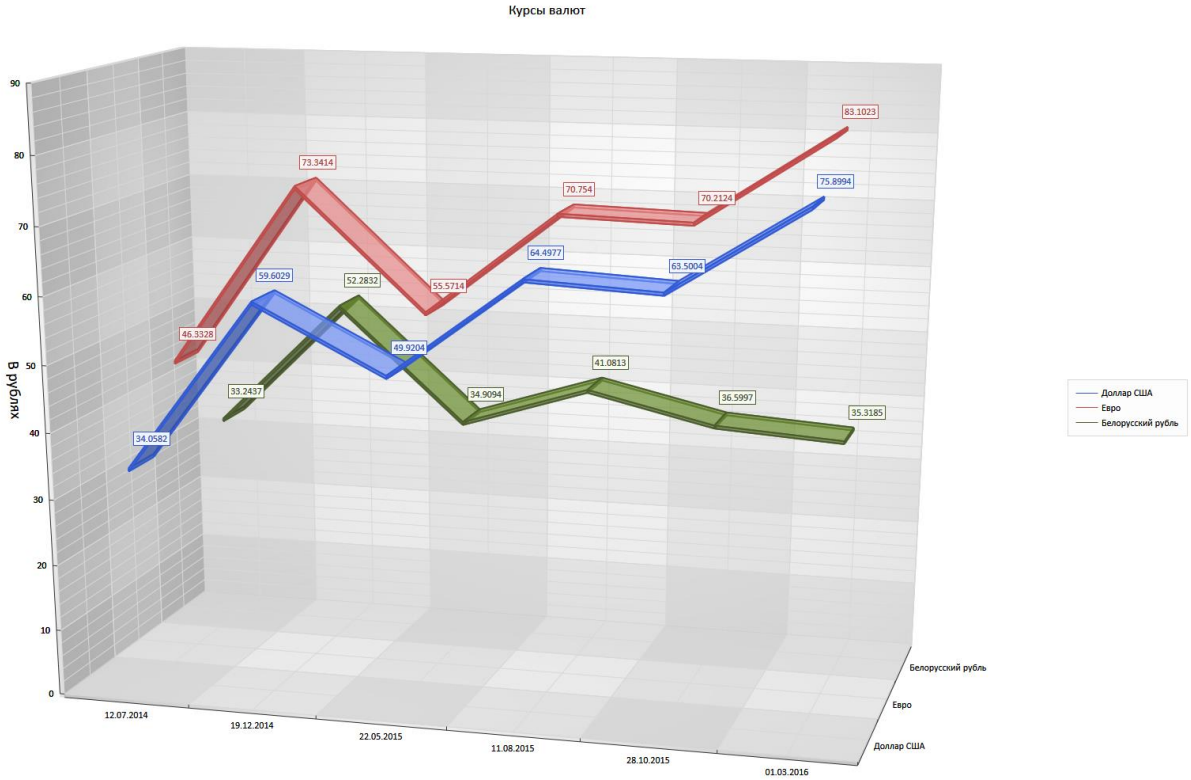
```
// Евро
ChartAddData(1, 46.3328);
ChartAddData(1, 73.3414);
ChartAddData(1, 55.5714);
ChartAddData(1, 70.754);
ChartAddData(1, 70.2124);
ChartAddData(1, 83.1023);

// Белорусский рубль (за 1000)
ChartAddData(2, 33.2437);
ChartAddData(2, 52.2832);
ChartAddData(2, 34.9094);
ChartAddData(2, 41.0813);
ChartAddData(2, 36.5997);
ChartAddData(2, 35.3185);
```

10.6.1.1 Графики

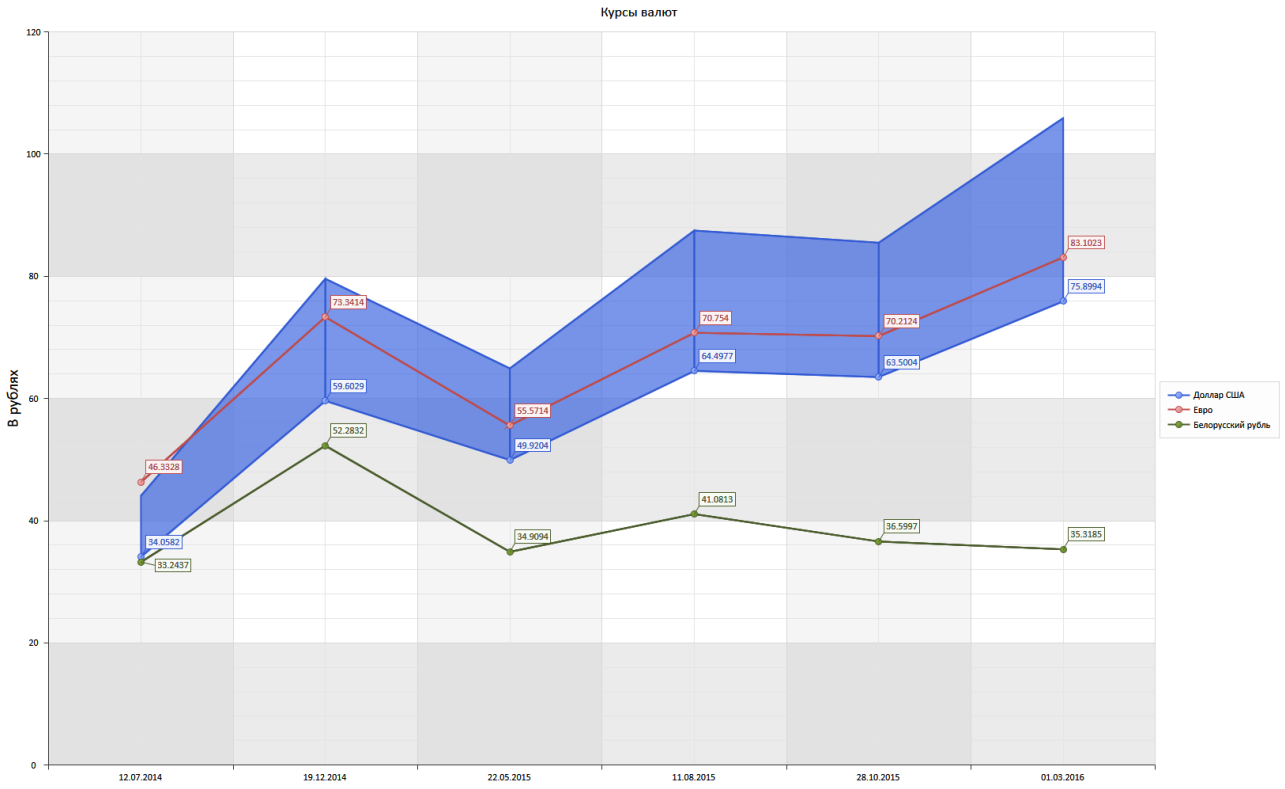
int nCategory = CC_Line;
int nType = CT_SIMPLE;



int nCategory = CC_Line3D;
int nType = CT_SIMPLE;

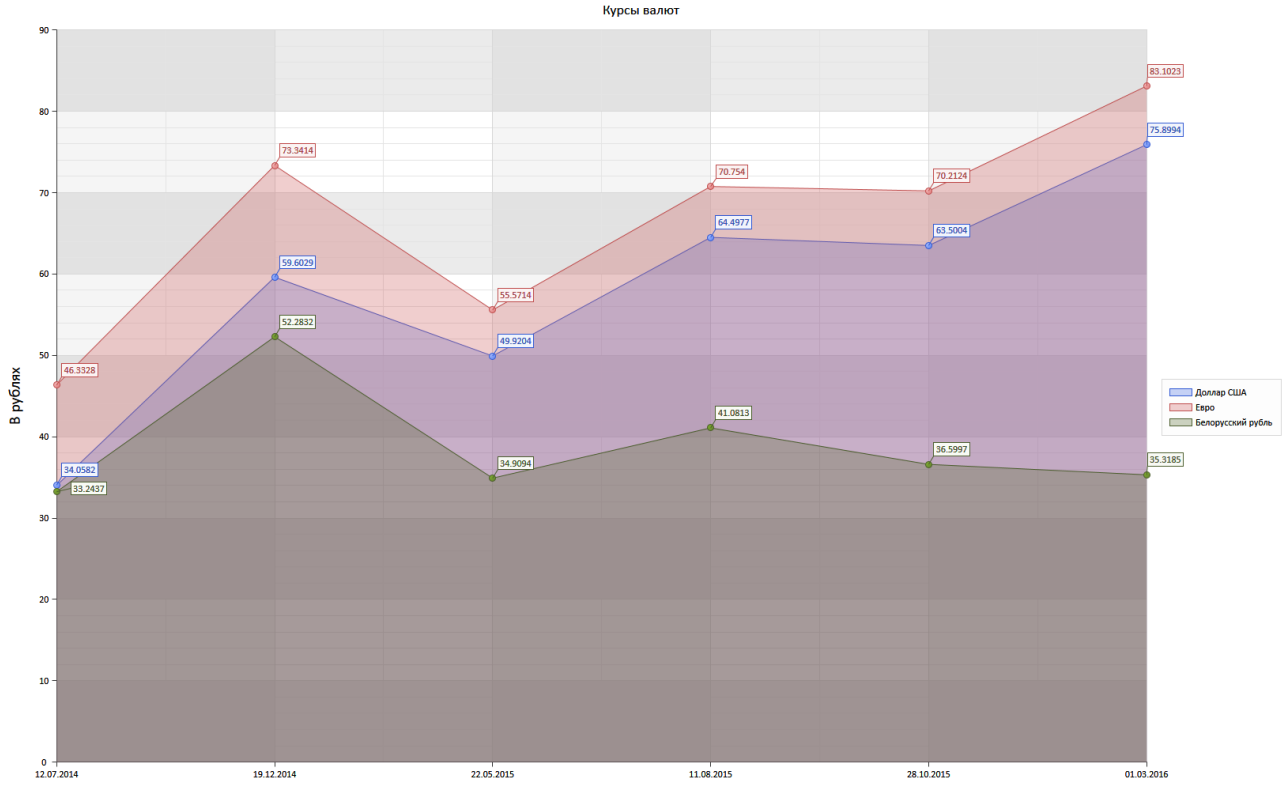


int nCategory = CC_Line;
int nType = CT_RANGE;

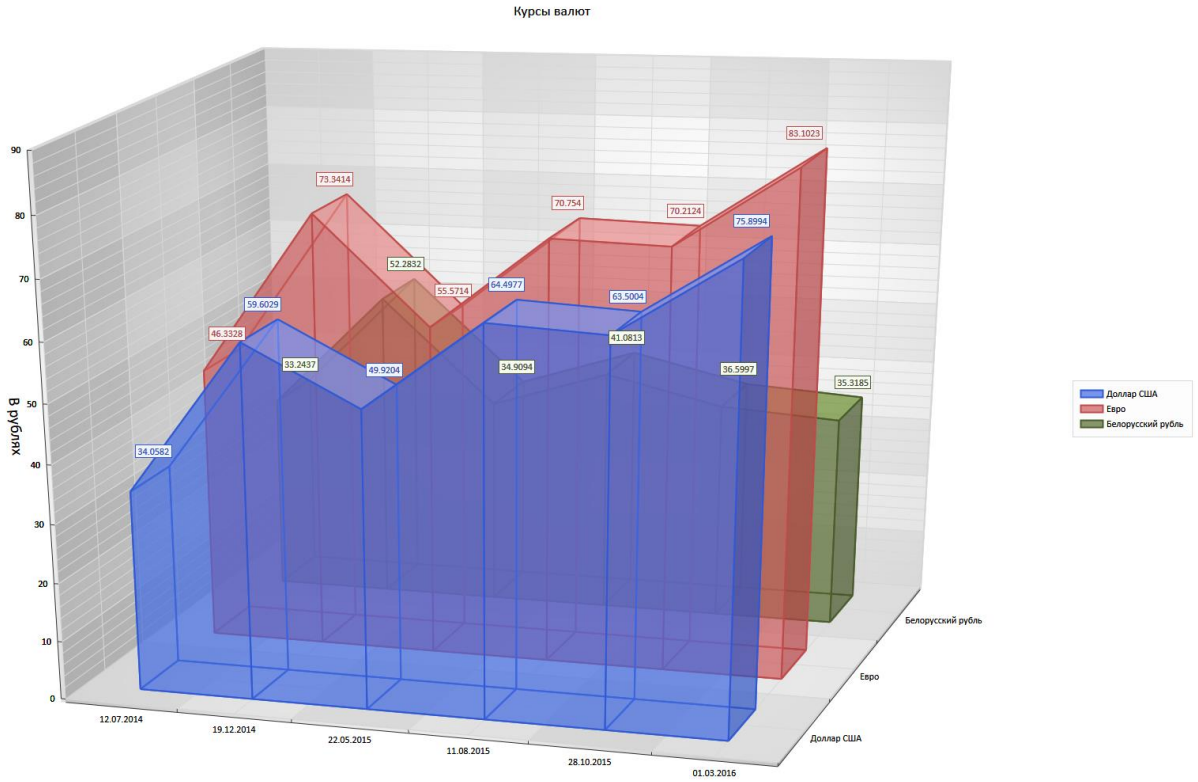


10.6.1.2 Области

int nCategory = CC_Area;
int nType = CT_SIMPLE;

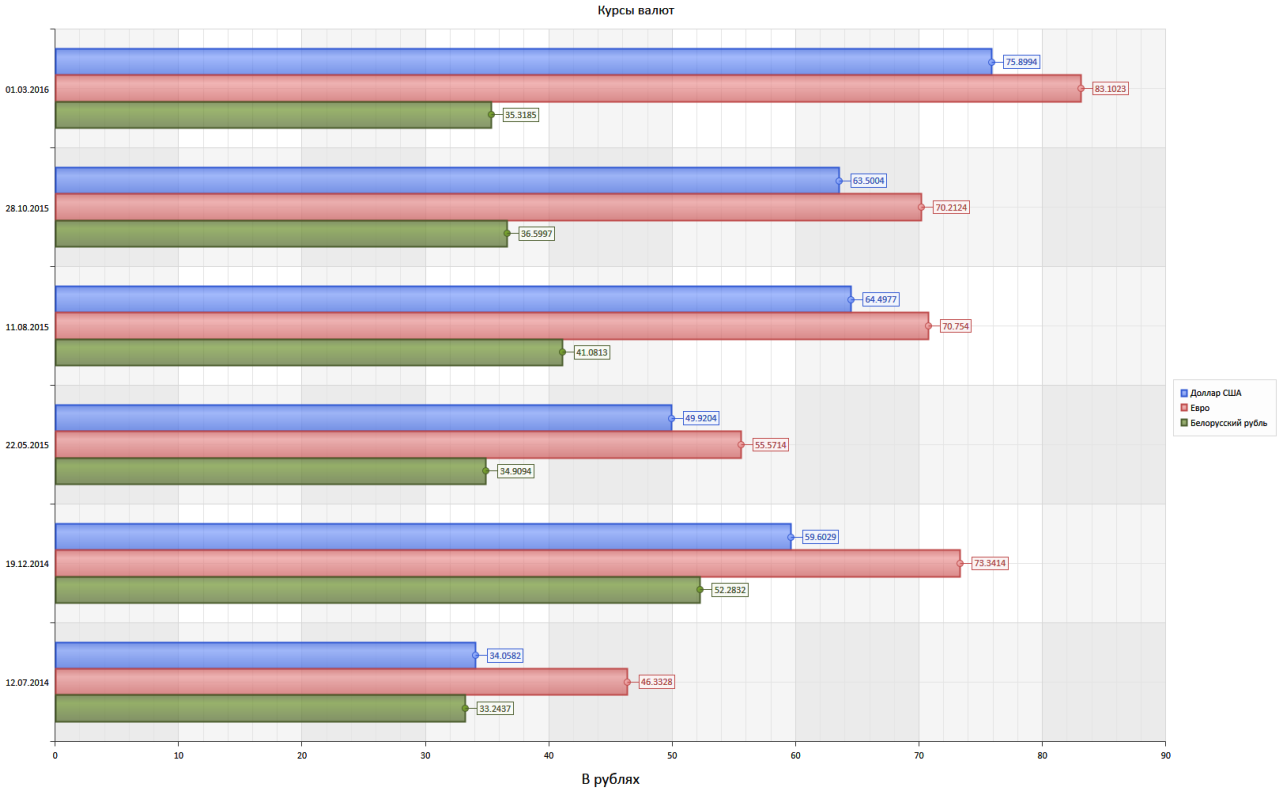


int nCategory = CC_Area3D;
int nType = CT_SIMPLE;

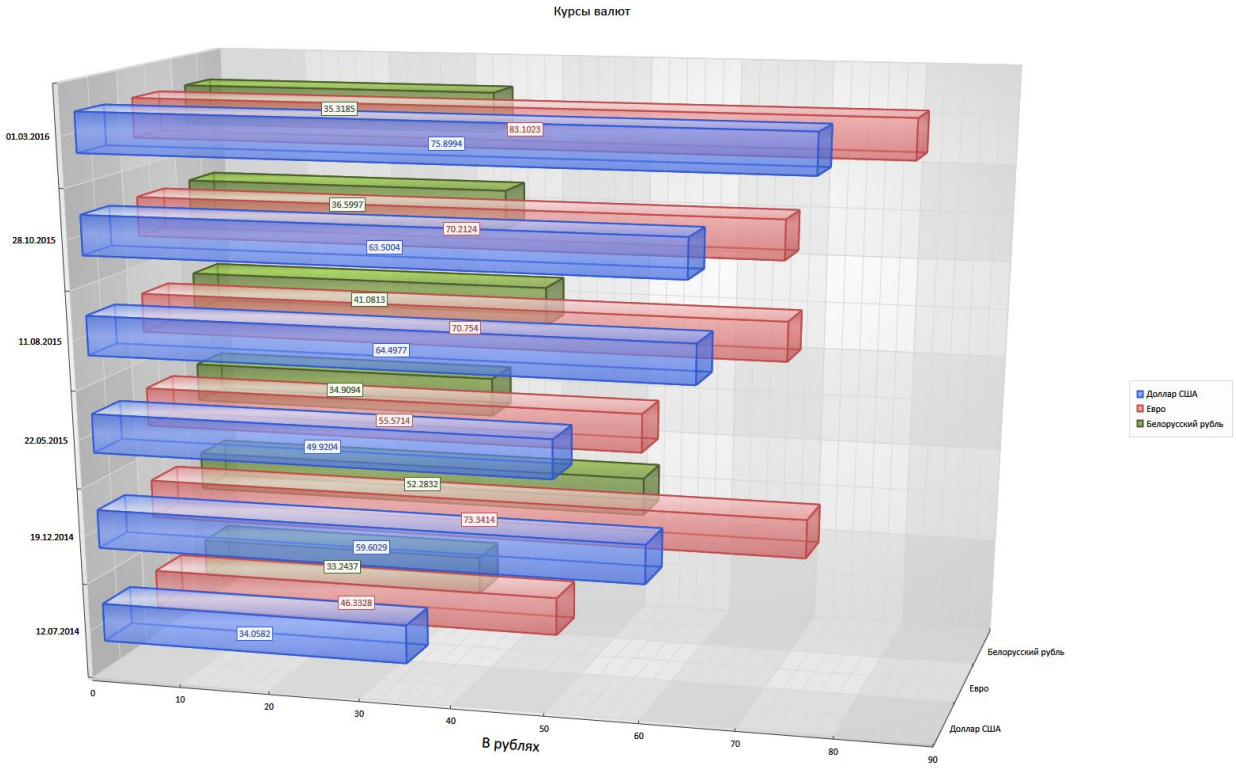


10.6.1.3 Бар

int nCategory = CC_Bar;
int nType = CT_SIMPLE;

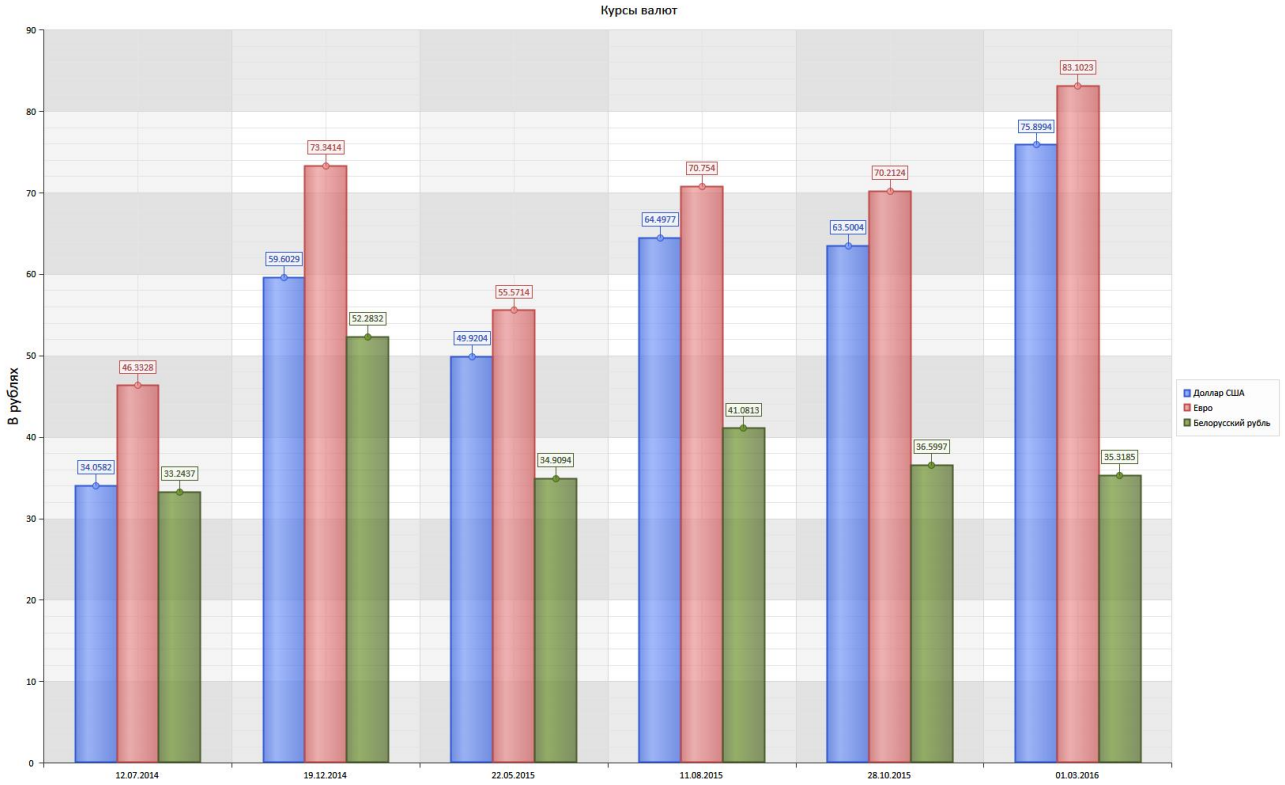


int nCategory = CC_Bar3D;
int nType = CT_SIMPLE;

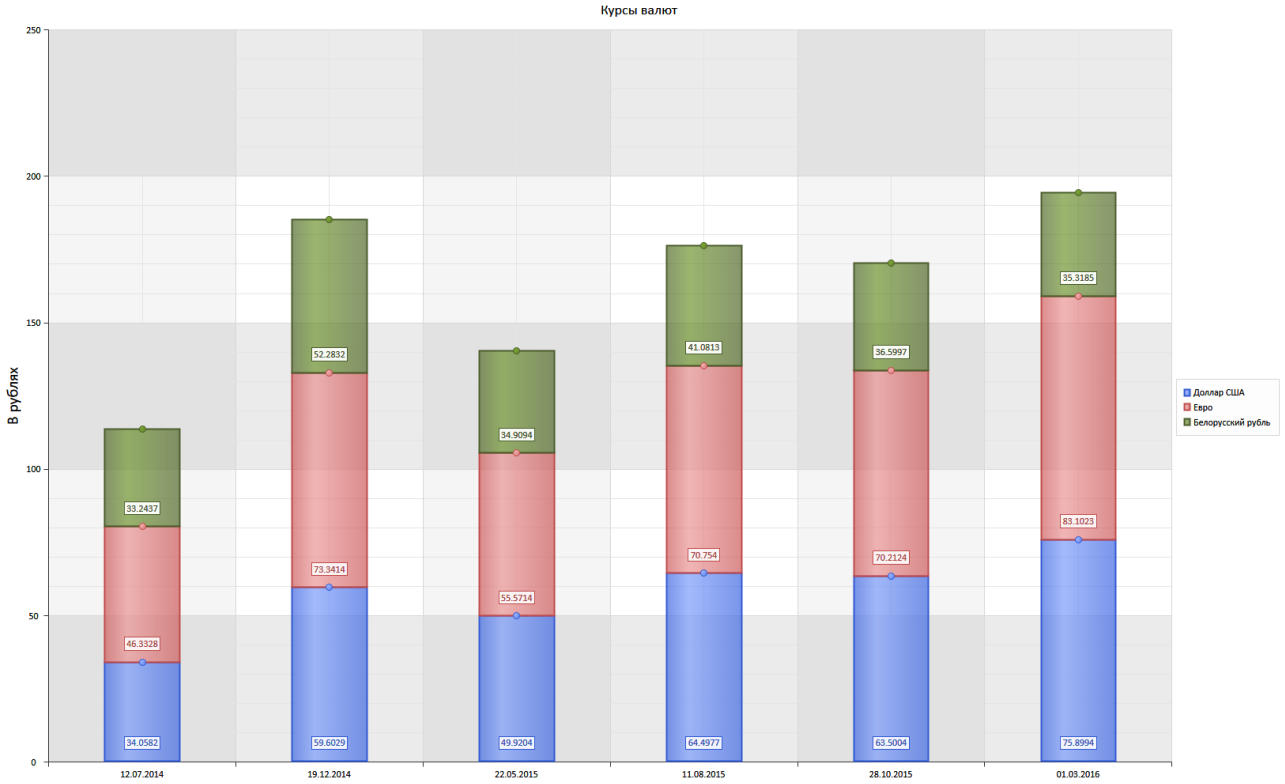


10.6.1.4 Колонки

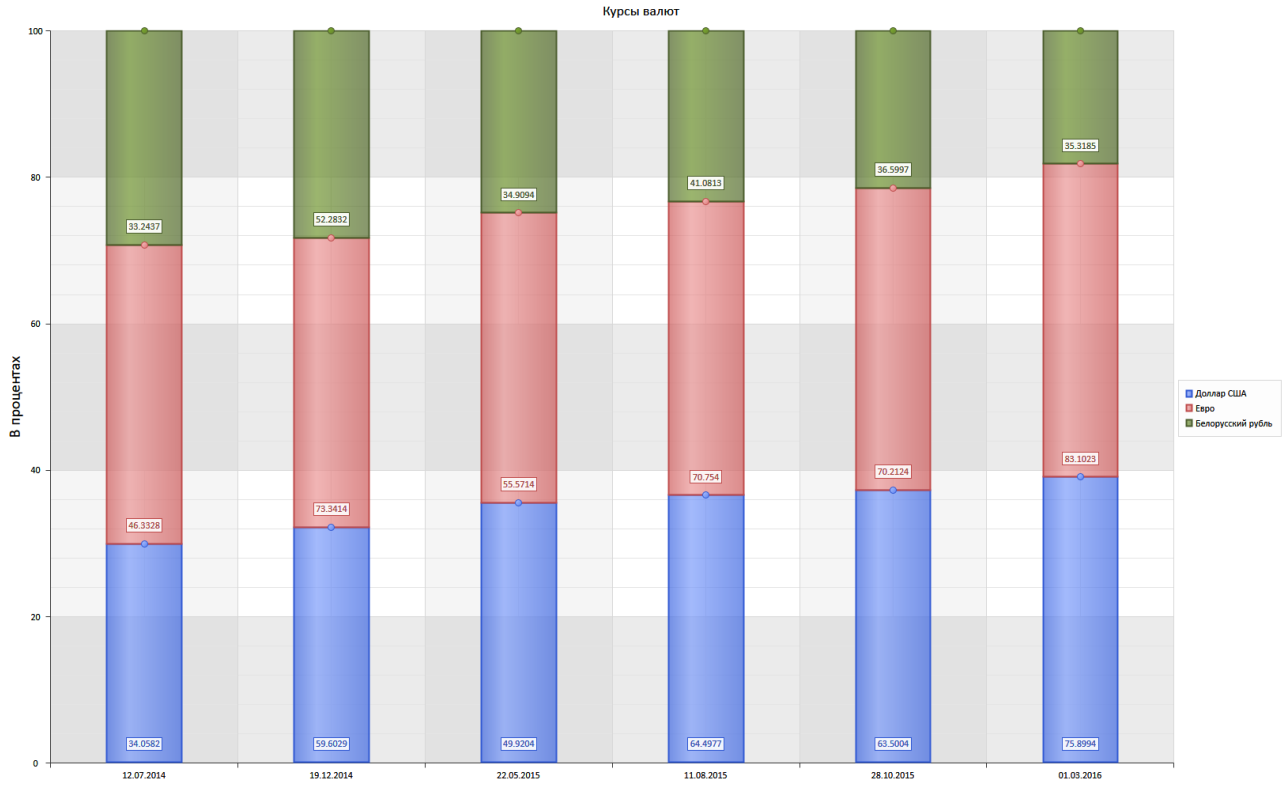
int nCategory = CC_Column;
int nType = CT_SIMPLE;



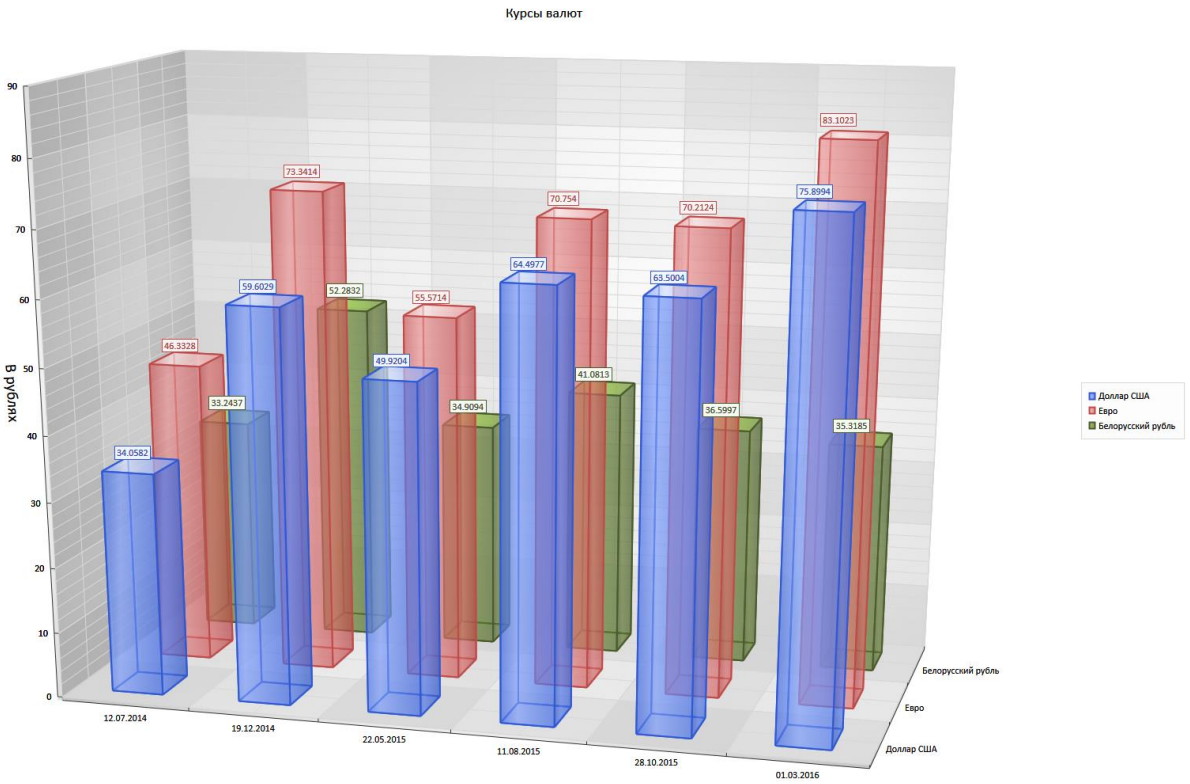
int nCategory = CC_Column;
int nType = CT_STACKED;



int nCategory = CC_Column;
int nType = CT_100STACKED;



int nCategory = CC_Column3D;
int nType = CT_SIMPLE;



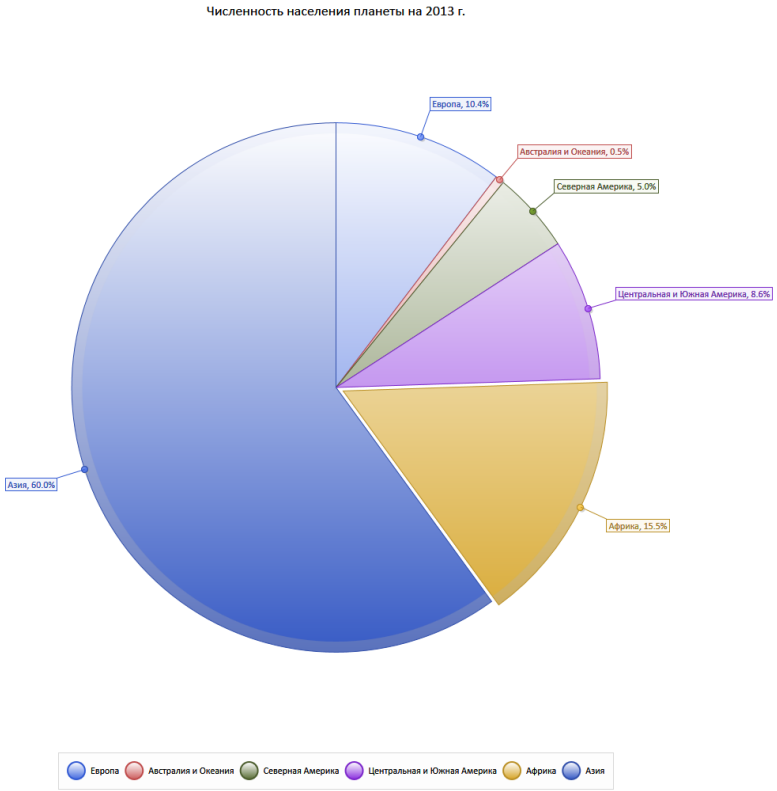
10.6.2 Фигурные диаграммы

Приведенный ниже код позволяет строить фигурные диаграммы всех категорий. Как и в предыдущем примере, настройка задается двумя переменными: nCategory и nType.

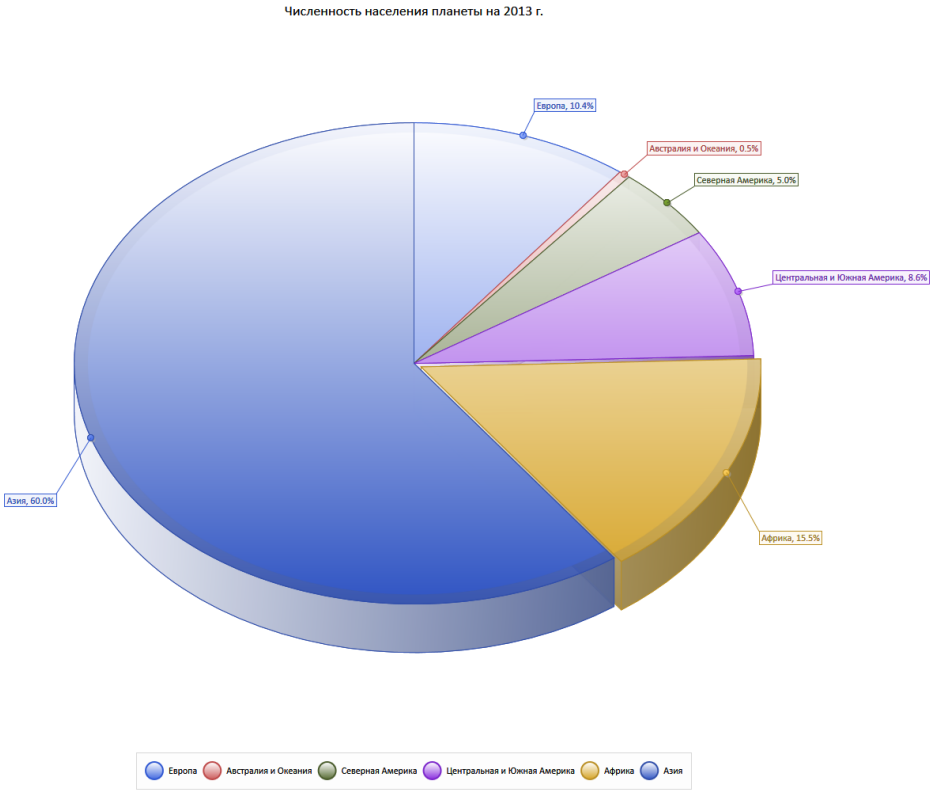
```
int nCategory = CC_Pie;
int nType = CT_SIMPLE;

ChartSetTransparency(20);

int nFlags = CSF_SHADOW;

ChartSetName("Численность населения планеты на 2013 г.");
ChartLegendSetOptions(TRUE, ELGP_BOTTOM);

ChartCreateSeries("2013", nCategory, nType, nFlags);

ChartAddData(0, "Европа",                              742452.0);
ChartAddData(0, "Австралия и Океания",                 38304.0);
ChartAddData(0, "Северная Америка",                    355361.0);
ChartAddData(0, "Центральная и Южная Америка",         616644.0);
ChartAddData(0, "Африка",                              1110635.0, TRUE);
ChartAddData(0, "Азия",                                4298723.0);

if (nCategory == CC_DoughnutNested)
{
	ChartCreateSeries("2005", nCategory, nType, nFlags);

	ChartAddData(1, "Европа",                              724722.0);
	ChartAddData(1, "Австралия и Океания",                 32998.0);
	ChartAddData(1, "Северная Америка",                    332156.0);
	ChartAddData(1, "Центральная и Южная Америка",         558281.0);
	ChartAddData(1, "Африка",                              887964.0);
	ChartAddData(1, "Азия",                                3787508.0);
}
```

10.6.2.1 Круговая

```
int nCategory = CC_Pie;
int nType = CT_SIMPLE;
```

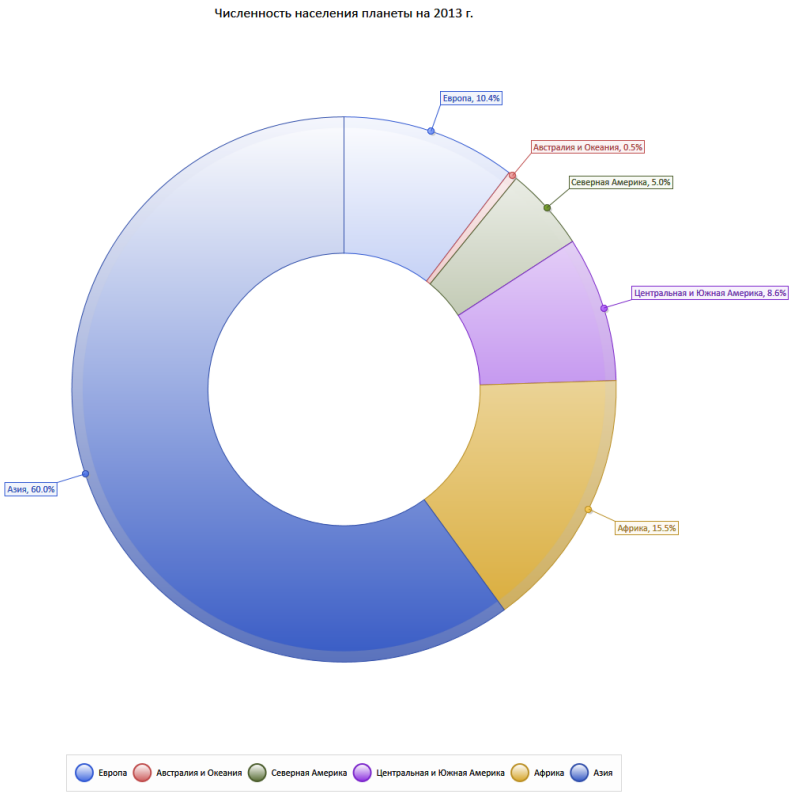



```
int nCategory = CC_Pie3D;
int nType = CT_SIMPLE;
```

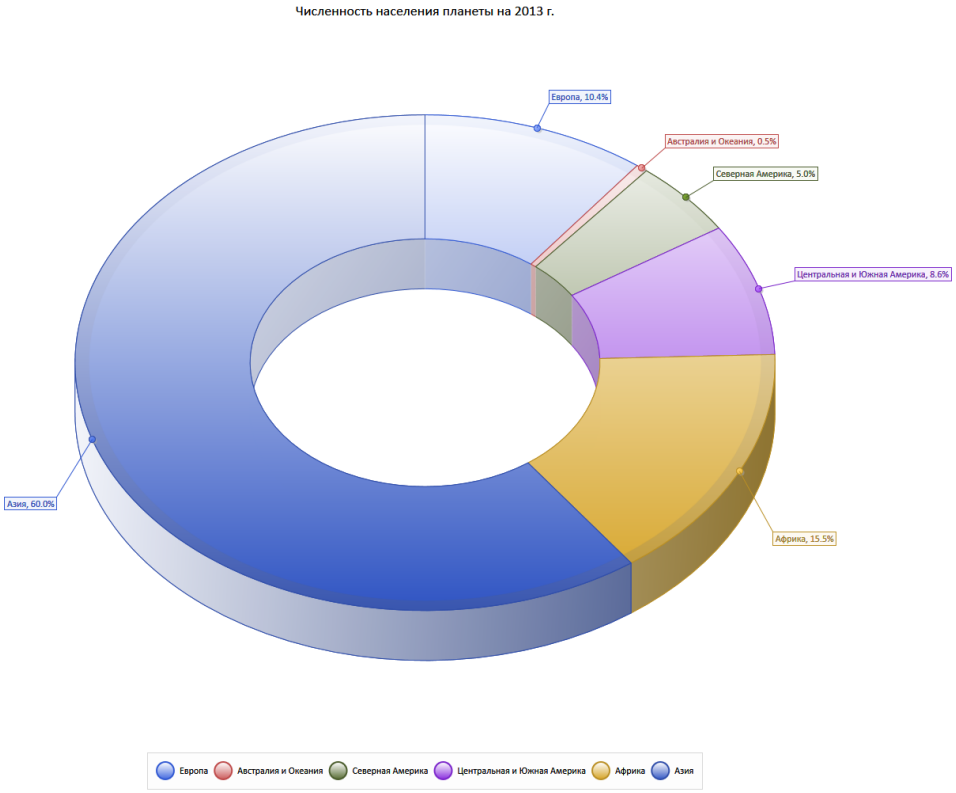


10.6.2.2 Кольцо

int nCategory = CC_Doughnut;
int nType = CT_SIMPLE;

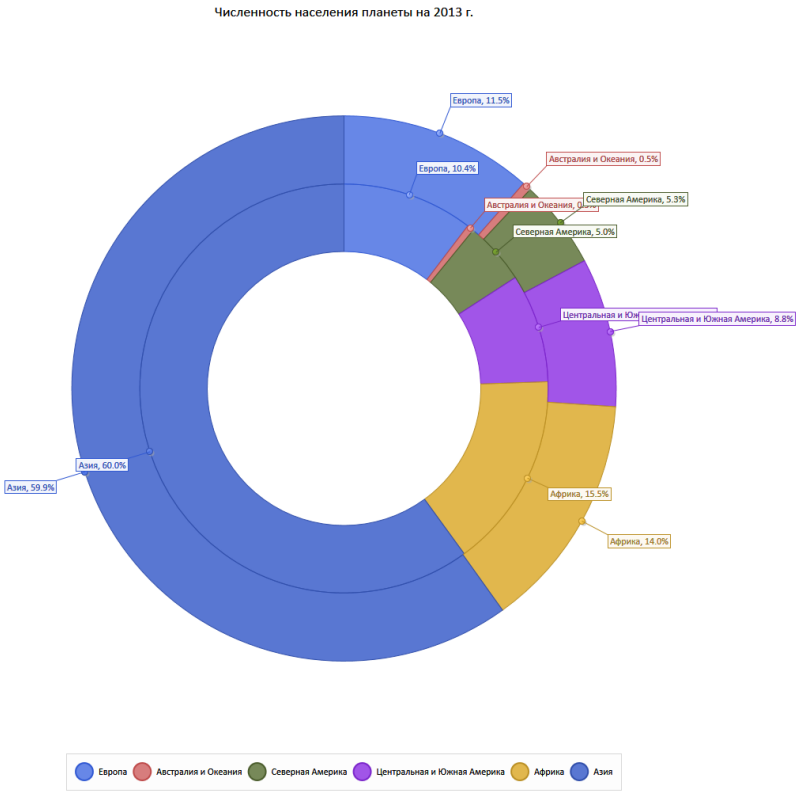


int nCategory = CC_Doughnut3D;
int nType = CT_SIMPLE;



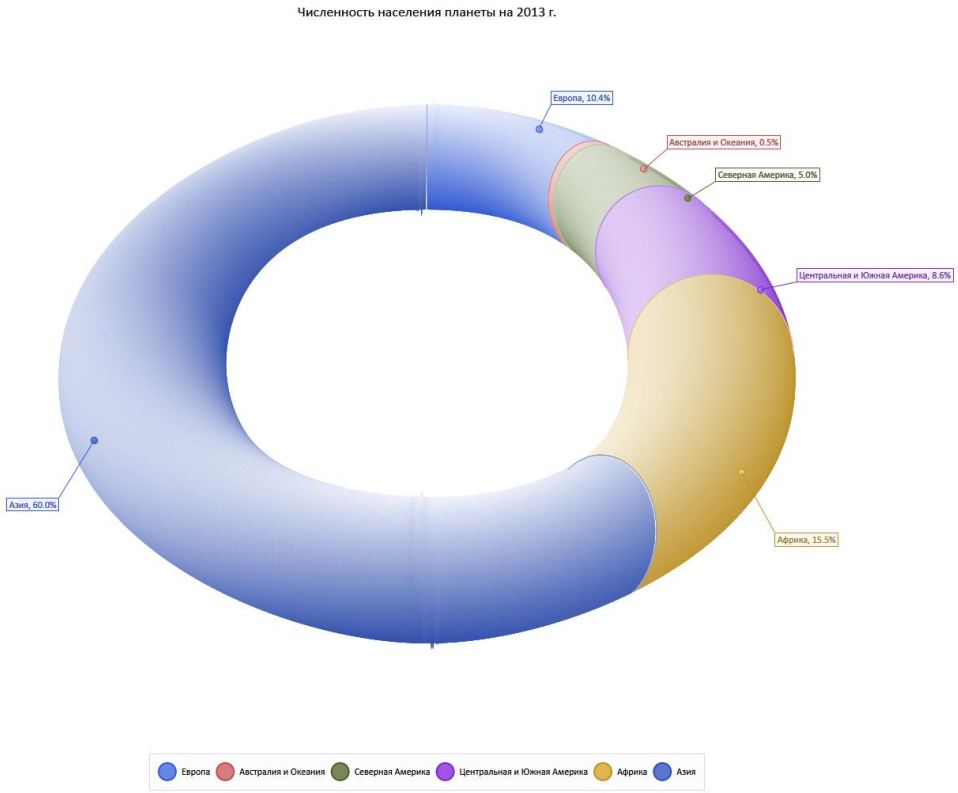
10.6.2.3 Вложенные кольца

```
int nCategory = CC_DoughnutNested;
int nType = CT_SIMPLE;
```



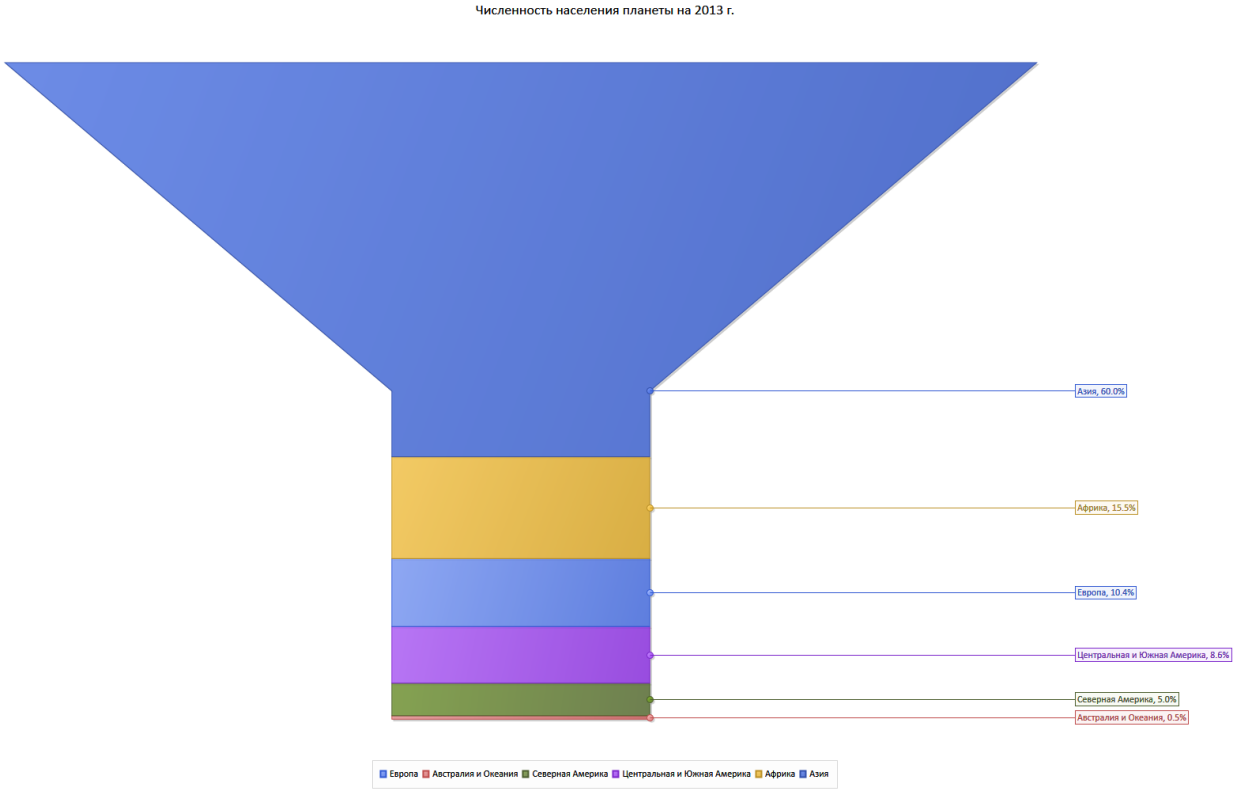
10.6.2.4 Top

```
int nCategory = CC_Torus3D;
int nType = CT_SIMPLE;
```

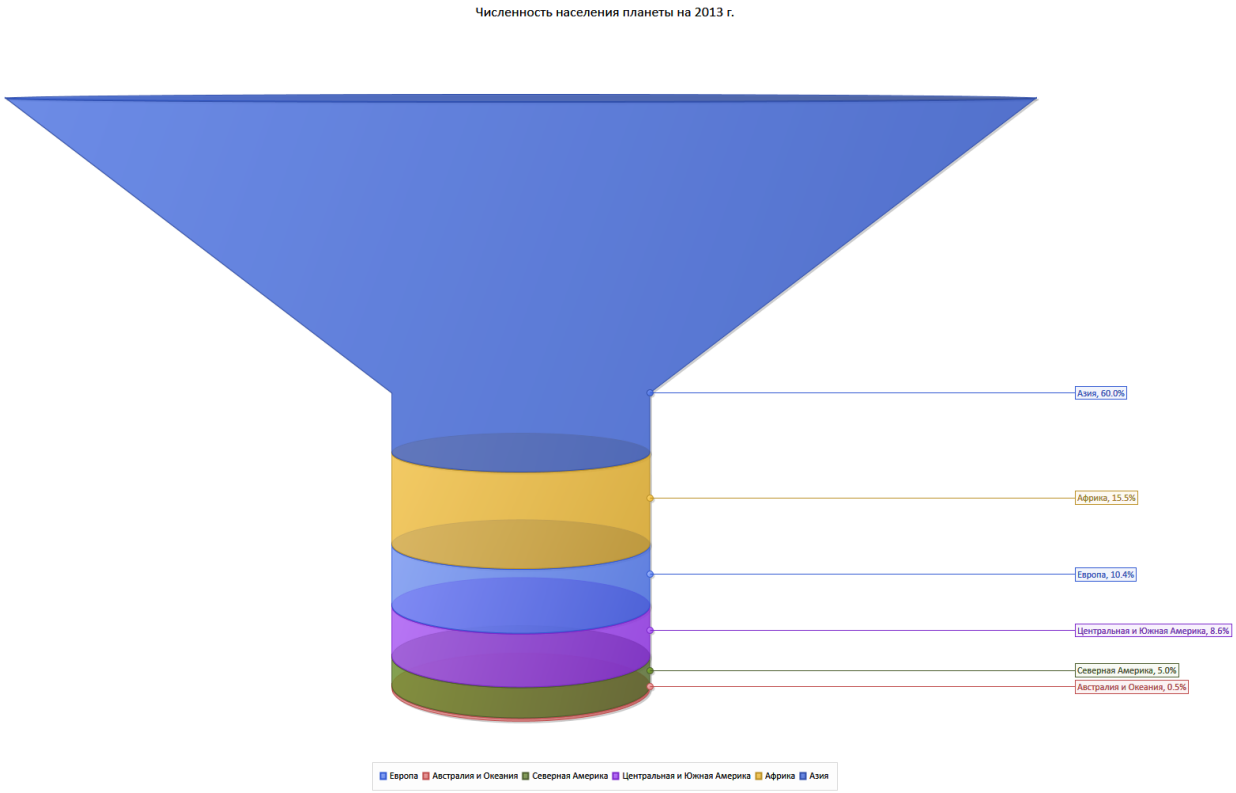


10.6.2.5 Воронка

int nCategory = CC_Funnel;
int nType = CT_SIMPLE;

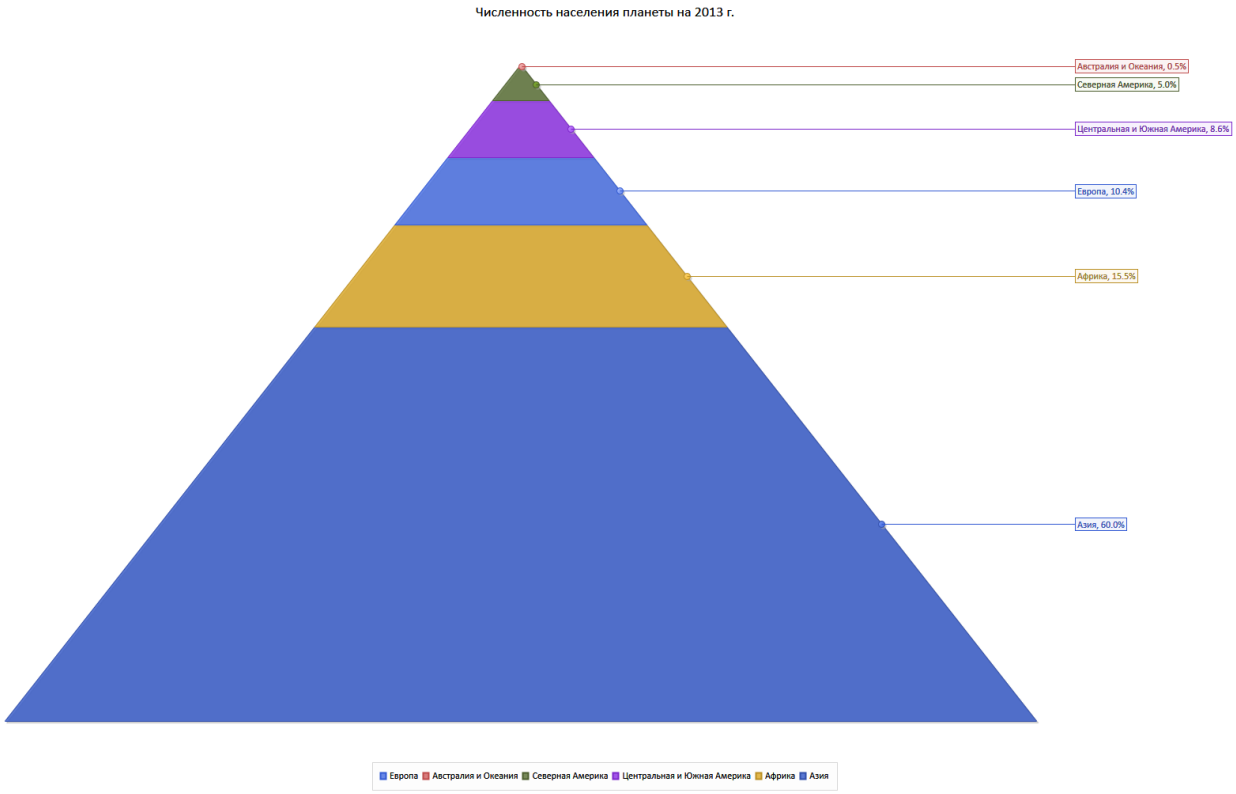


int nCategory = CC_Funnel3D;
int nType = CT_SIMPLE;

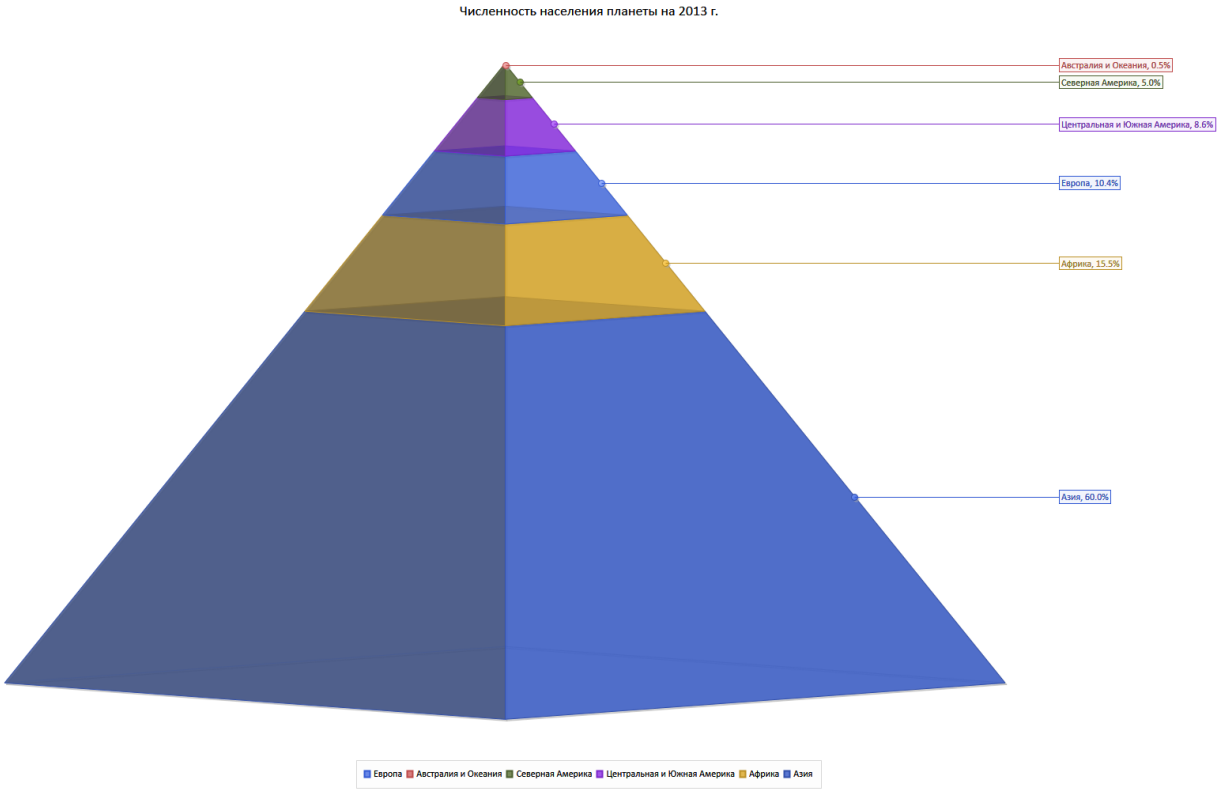


10.6.2.6 Пирамида

int nCategory = CC_Pyramid;
int nType = CT_SIMPLE;



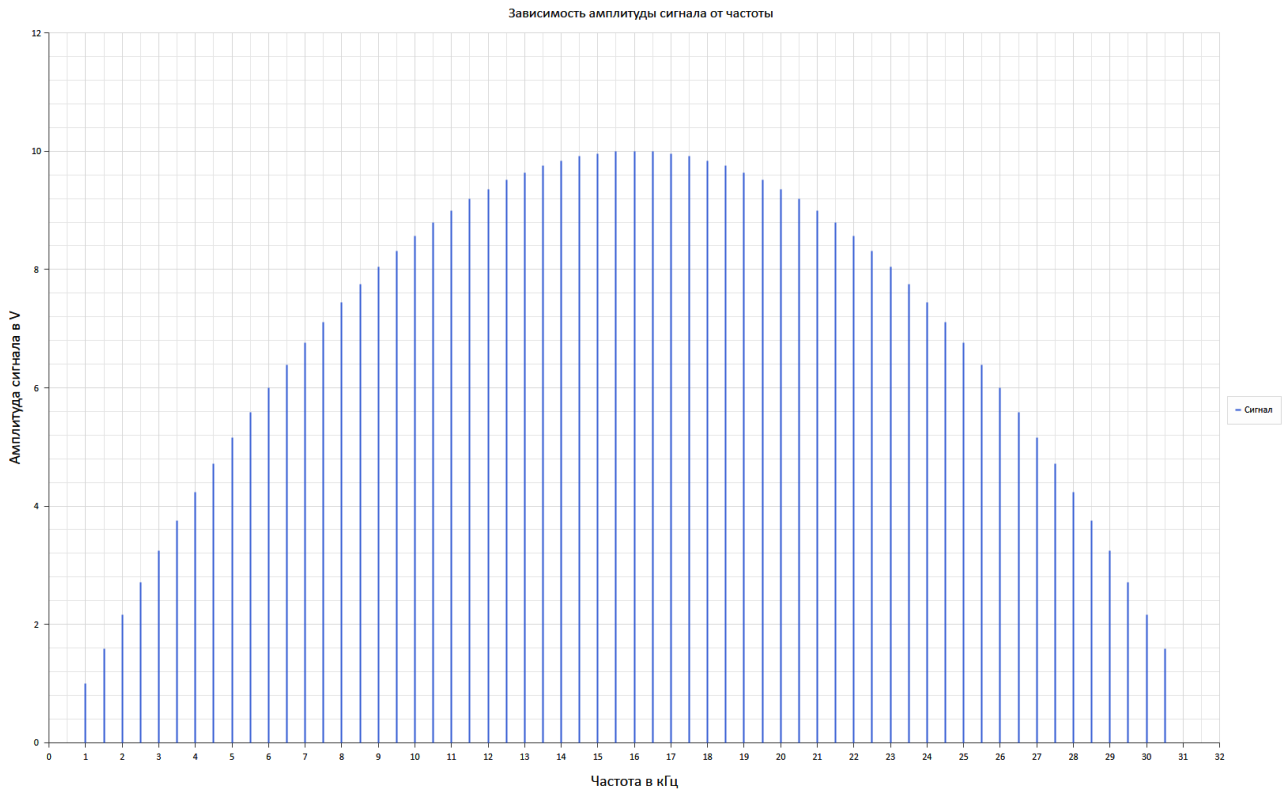
int nCategory = CC_Pyramid3D;
int nType = CT_SIMPLE;



10.6.3 Прочие диаграммы

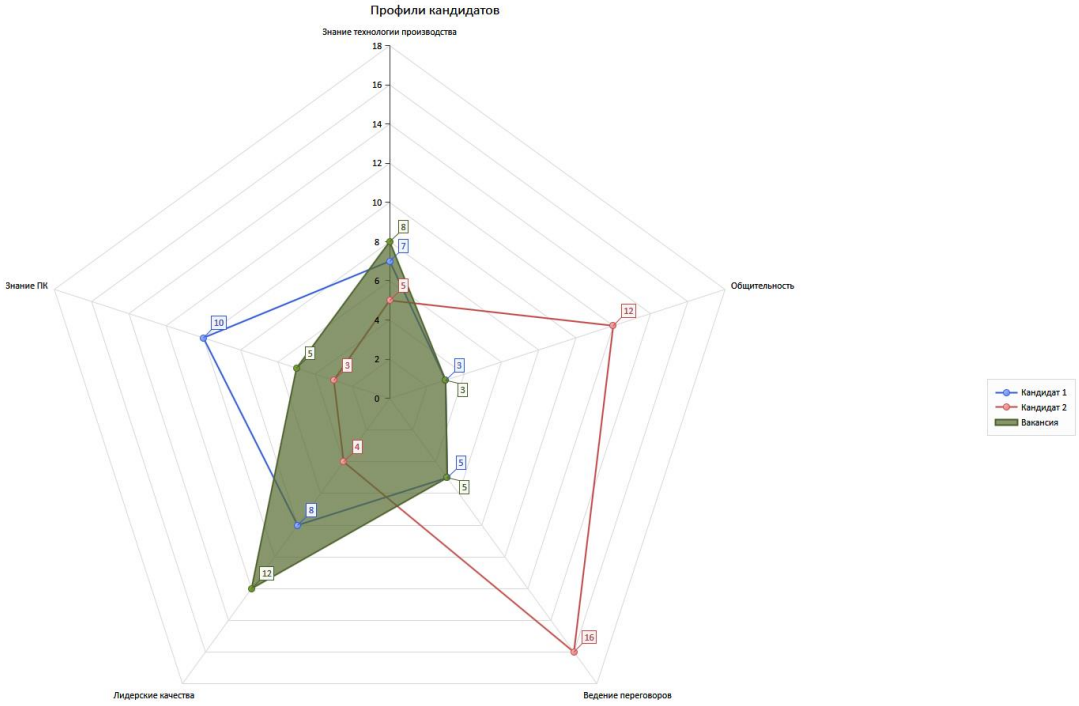
10.6.3.1 Гистограмма

```
ChartSetName("Зависимость амплитуды сигнала от частоты");
ChartAxisSetName(AXIS_X, "Частота в кГц");
ChartAxisSetName(AXIS_Y, "Амплитуда сигнала в V");

ChartMarkersShow(FALSE);
ChartLabelsSetOptions(0);

ChartCreateSeries("Сигнал", CC_Histogram, CT_SIMPLE);

double dFrequency = 1;

for (double d = -3; d <= 3; d += 0.1)
{
	double y = -d*d + 10;

	ChartAddDataXY(0, dFrequency, y);
	dFrequency += 0.5;
}
```



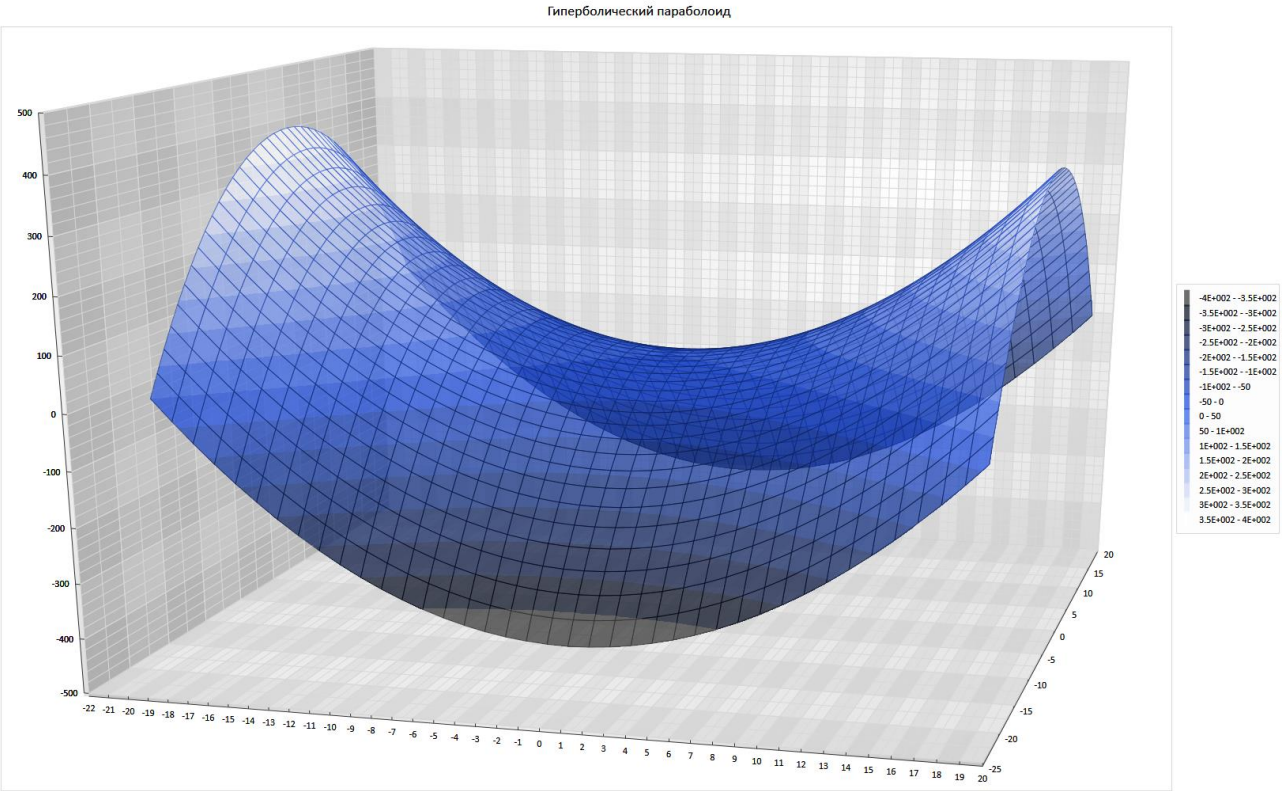
10.6.3.2 Полярные координаты

```
ChartSetName("Профили кандидатов");

ChartCreateSeries("Кандидат 1", CC_Polar, CT_SIMPLE, CSF_CLOSE_SHAPE);
ChartCreateSeries("Кандидат 2", CC_Polar, CT_SIMPLE, CSF_CLOSE_SHAPE);
ChartCreateSeries("Вакансия", CC_Polar, CT_SIMPLE, CSF_CLOSE_SHAPE | CSF_FILL_SHAPE);

ChartAddData(0, "Знание технологии производства", 7.0);
ChartAddData(0, "Общительность",                  3.0);
ChartAddData(0, "Ведение переговоров",            5.0);
ChartAddData(0, "Лидерские качества",             8.0);
ChartAddData(0, "Знание ПК",                      10.0);

ChartAddData(1, 5.0);
ChartAddData(1, 12.0);
ChartAddData(1, 16.0);
ChartAddData(1, 4.0);
ChartAddData(1, 3.0);

ChartAddData(2, 8.0);
ChartAddData(2, 3.0);
ChartAddData(2, 5.0);
ChartAddData(2, 12.0);
ChartAddData(2, 5.0);
```



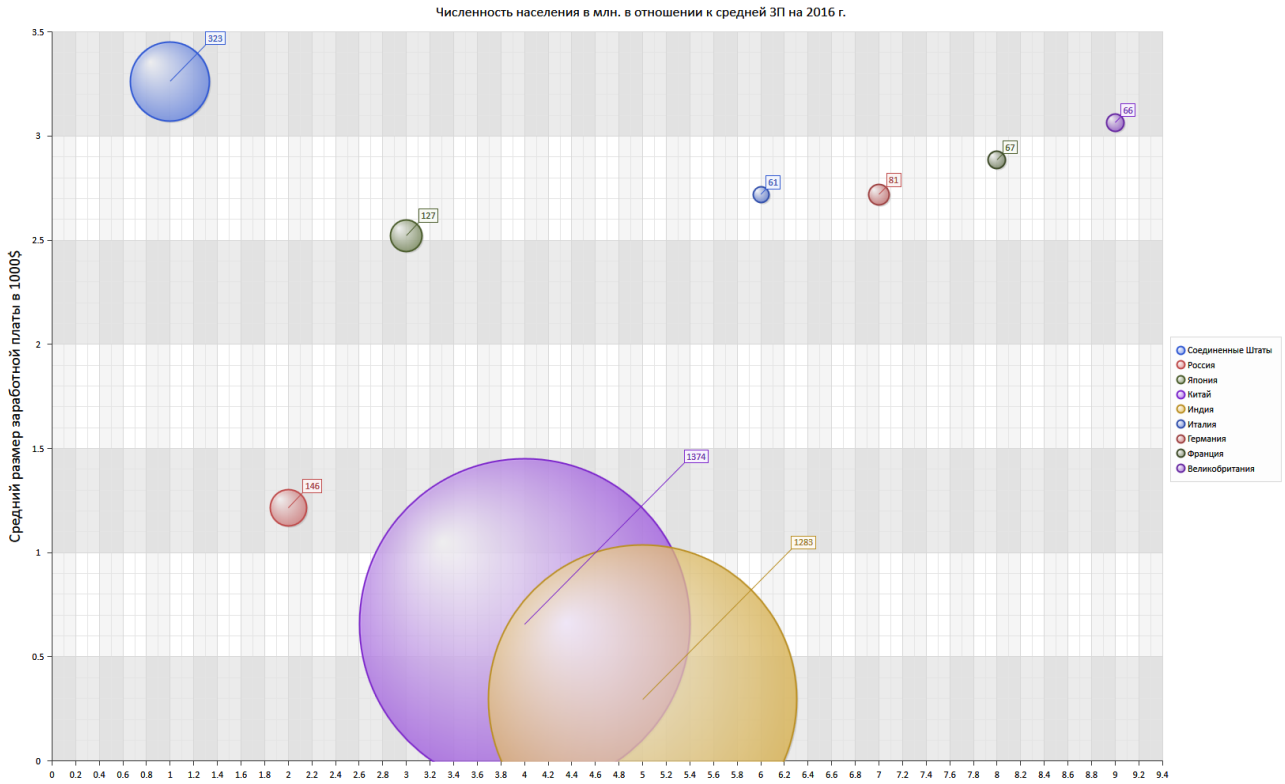
10.6.3.3 Поверхность

```
ChartSetName("Гиперболический параболоид");
ChartCreateSeries("", CC_Surface3D, CT_SIMPLE, CSF_SURFACE_MONOCHROME);

for (double y = -20; y < 20; y++)
{
	for (double x = -20; x < 20; x++)
	{
		ChartAddDataXYZ(0, x, x*x - y*y, y);
	}
}
```



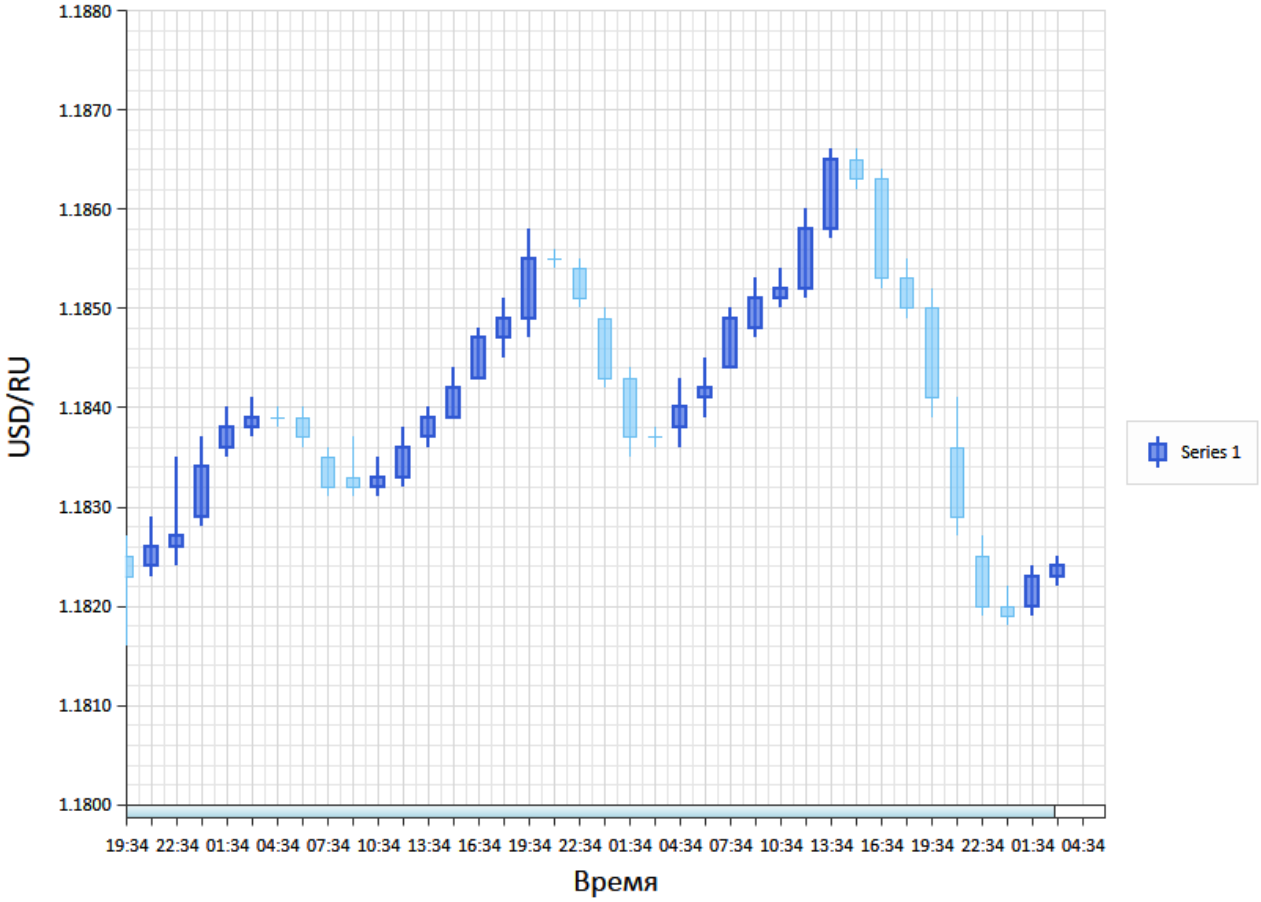
10.6.3.4 Пузырьковая

```
int nFlags = CSF_SHADOW | CSF_BUBBLE_SINGLE_SERIES | CSF_BUBBLE_SIZE3;

ChartSetName("Численность населения в млн. в отношении к средней ЗП на 2016 г.");
ChartAxisSetName(AXIS_Y, "Средний размер заработной платы в 1000$");

ChartCreateSeries("", CC_Bubble, CT_SIMPLE, nFlags);

ChartAddBubbleData(0, "Соединенные Штаты",      1.0, 3.263, 323.0, TRUE);
ChartAddBubbleData(0, "Россия",                 2.0, 1.215, 146.0, TRUE);
ChartAddBubbleData(0, "Япония",                 3.0, 2.522, 127.0, TRUE);
ChartAddBubbleData(0, "Китай",                  4.0, 0.656, 1374.0, TRUE);
ChartAddBubbleData(0, "Индия",                  5.0, 0.295, 1283.0, TRUE);
ChartAddBubbleData(0, "Италия",                 6.0, 2.720, 61.0, TRUE);
ChartAddBubbleData(0, "Германия",               7.0, 2.720, 81.0, TRUE);
ChartAddBubbleData(0, "Франция",                8.0, 2.886, 67.0, TRUE);
ChartAddBubbleData(0, "Великобритания",         9.0, 3.065, 66.0, TRUE);
```



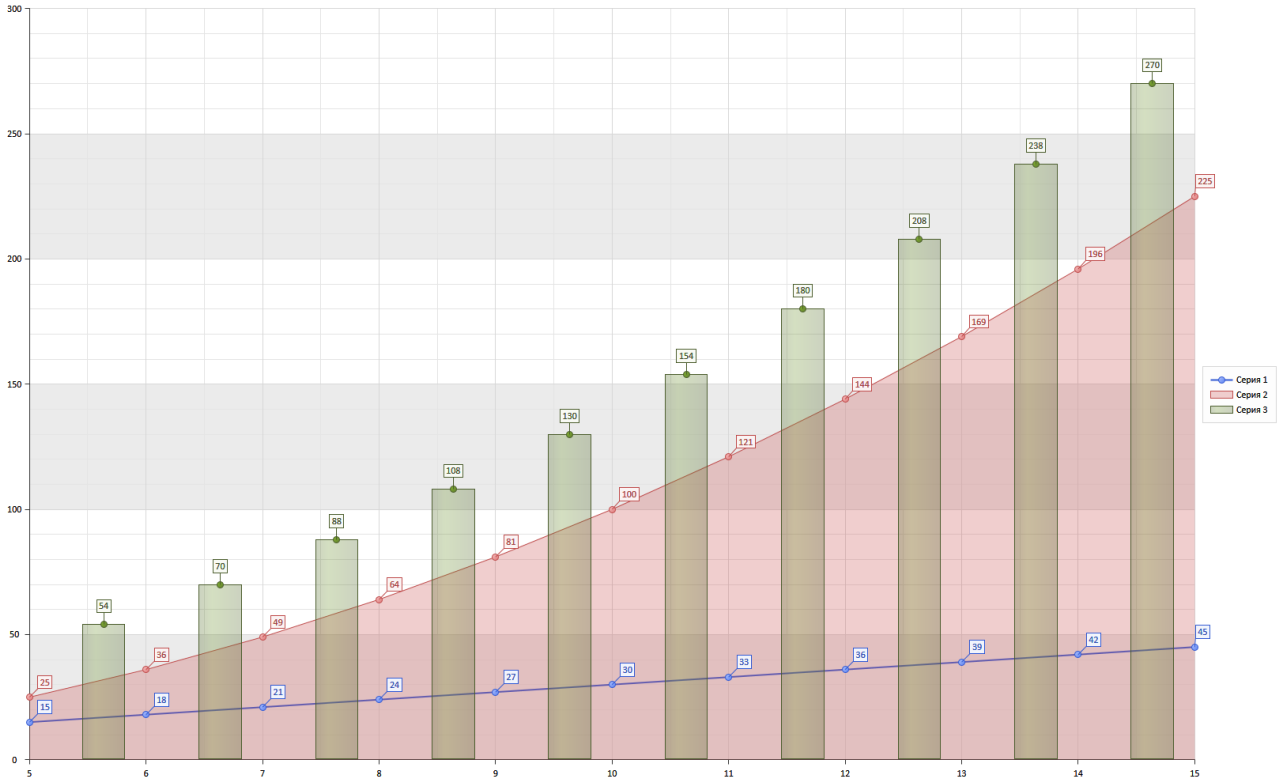
10.6.3.5 Биржевая

```
int nOptions = ChartLabelsGetOptions();
nOptions &= ~DLF_SHOW;
ChartLabelsSetOptions(nOptions);
ChartMarkersShow(FALSE);

ChartAxisSetName(AXIS_X, "Время");
ChartAxisSetOptions(AXIS_X,  DAF_SHOW  |  DAF_SHOW_NAME  |  DAF_GRID  |  DAF_AS_DATE  |
DAF_SCROLL_BAR, 0.0, 0.0);
ChartAxisSetMask(AXIS_X, "d %H:%M");

ChartAxisSetName(AXIS_Y, "USD/RU");
ChartAxisSetOptions(AXIS_Y, DAF_SHOW | DAF_SHOW_NAME | DAF_GRID, 0.0, 0.0);
ChartAxisSetMask(AXIS_Y, "n .4");

ChartCreateSeries("Series 1", CC_Stock, CT_SIMPLE, CSF_STOCK_CANDLE);

datetime dtStart = DateGetCurrent();
datedif  span = '0, 1, 30, 0';

ChartAddStockData(0, 1.1825, 1.1827, 1.1816, 1.1823, dtStart += span);
ChartAddStockData(0, 1.1824, 1.1829, 1.1823, 1.1826, dtStart += span);
ChartAddStockData(0, 1.1826, 1.1835, 1.1824, 1.1827, dtStart += span);
ChartAddStockData(0, 1.1829, 1.1837, 1.1828, 1.1834, dtStart += span);
ChartAddStockData(0, 1.1836, 1.1840, 1.1835, 1.1838, dtStart += span);
ChartAddStockData(0, 1.1838, 1.1841, 1.1837, 1.1839, dtStart += span);
ChartAddStockData(0, 1.1839, 1.1840, 1.1838, 1.1839, dtStart += span);
ChartAddStockData(0, 1.1839, 1.1840, 1.1836, 1.1837, dtStart += span);
ChartAddStockData(0, 1.1835, 1.1836, 1.1831, 1.1832, dtStart += span);
ChartAddStockData(0, 1.1833, 1.1837, 1.1831, 1.1832, dtStart += span);
ChartAddStockData(0, 1.1832, 1.1835, 1.1831, 1.1833, dtStart += span);
ChartAddStockData(0, 1.1833, 1.1838, 1.1832, 1.1836, dtStart += span);
ChartAddStockData(0, 1.1837, 1.1840, 1.1836, 1.1839, dtStart += span);
ChartAddStockData(0, 1.1839, 1.1844, 1.1839, 1.1842, dtStart += span);
ChartAddStockData(0, 1.1843, 1.1848, 1.1843, 1.1847, dtStart += span);
ChartAddStockData(0, 1.1847, 1.1851, 1.1845, 1.1849, dtStart += span);
ChartAddStockData(0, 1.1849, 1.1858, 1.1847, 1.1855, dtStart += span);
ChartAddStockData(0, 1.1855, 1.1856, 1.1854, 1.1855, dtStart += span);
ChartAddStockData(0, 1.1854, 1.1855, 1.1850, 1.1851, dtStart += span);
ChartAddStockData(0, 1.1849, 1.1850, 1.1842, 1.1843, dtStart += span);
ChartAddStockData(0, 1.1843, 1.1844, 1.1835, 1.1837, dtStart += span);
ChartAddStockData(0, 1.1837, 1.1838, 1.1836, 1.1837, dtStart += span);
ChartAddStockData(0, 1.1838, 1.1843, 1.1836, 1.1840, dtStart += span);
ChartAddStockData(0, 1.1841, 1.1845, 1.1839, 1.1842, dtStart += span);
ChartAddStockData(0, 1.1844, 1.1850, 1.1844, 1.1849, dtStart += span);
ChartAddStockData(0, 1.1848, 1.1853, 1.1847, 1.1851, dtStart += span);
ChartAddStockData(0, 1.1851, 1.1854, 1.1850, 1.1852, dtStart += span);
ChartAddStockData(0, 1.1852, 1.1860, 1.1851, 1.1858, dtStart += span);
ChartAddStockData(0, 1.1858, 1.1866, 1.1857, 1.1865, dtStart += span);
ChartAddStockData(0, 1.1865, 1.1866, 1.1862, 1.1863, dtStart += span);
ChartAddStockData(0, 1.1863, 1.1864, 1.1852, 1.1853, dtStart += span);
ChartAddStockData(0, 1.1853, 1.1855, 1.1849, 1.1850, dtStart += span);
ChartAddStockData(0, 1.1850, 1.1852, 1.1839, 1.1841, dtStart += span);
ChartAddStockData(0, 1.1836, 1.1841, 1.1827, 1.1829, dtStart += span);
ChartAddStockData(0, 1.1825, 1.1827, 1.1819, 1.1820, dtStart += span);
ChartAddStockData(0, 1.1820, 1.1822, 1.1818, 1.1819, dtStart += span);
ChartAddStockData(0, 1.1820, 1.1824, 1.1819, 1.1823, dtStart += span);
ChartAddStockData(0, 1.1823, 1.1825, 1.1822, 1.1824, dtStart += span);
ChartAddStockData(0, 1.1824, 1.1827, 1.1823, 1.1824, dtStart += span);
ChartAddStockData(0, 1.1823, 1.1824, 1.1812, 1.1814, dtStart += span);
```


1.1880
1.1870
1.1860
1.1850
1.1840
1.1830
1.1820
1.1810
1.1800
USD/RU
19:34 22:34 01:34 04:34 07:34 10:34 13:34 16:34 19:34 22:34 01:34 04:34 07:34 10:34 13:34 16:34 19:34 22:34 01:34 04:34
Время
Series 1

10.6.3.6 Комбинированная

```
double dMin = 5;
double dMax = 15;

ChartSetTransparency(70);
ChartAxisSetOptions(AXIS_X, DAF_SHOW | DAF_GRID | DAF_FIXED_RANGE, dMin, dMax);

ChartCurveSetWidth(2);
ChartCreateSeries("Серия 1", CC_Line, CT_SIMPLE);

ChartCurveSetWidth(1);
ChartCreateSeries("Серия 2", CC_Area, CT_SIMPLE);
ChartCreateSeries("Серия 3", CC_Column, CT_SIMPLE);

for (double x = dMin; x <= dMax; x++)
{
	double y1 = 3*x;
	double y2 = x*x;
	double y3 = y1+y2;

	ChartAddDataXY(0, x, y1);
	ChartAddDataXY(1, x, y2);
	ChartAddDataXY(2, x, y3);
}
```



11 Блок-схемы

11.1 Общие принципы построения блок-схем

Для построения блок-схемы служит сегмент OnBuild. Он исполняется системой при начальной инициализации объекта, но может быть, также, вызван непосредственно с помощью функции FC_Rebuild.

Элементы блок-схемы делятся на две категории: фигуры и коннекторы (соединительные линии). Элемент идентифицируется уникальным целочисленным идентификатором, который создает система при добавлении элемента в блок-схему. Этот идентификатор можно использовать в дальнейшем для обращения к данному элементу.

Для добавления фигуры служит функция FC_ShapeAdd, для добавления коннектора – FC_ConnectorAdd. Обе функции возвращают идентификатор созданного элемента. Для соединения коннектора с фигурами служит функция FC_Connect.

Можно, также, используя идентификаторы, задавать свойства фигур и коннекторов, а также, привязывать к ним пользовательские данные. Для работы с пользовательскими данными служат функции FC_GetItemData, FC_SetItemData и FC_DropItemData.

Объект может находиться в одном из двух режимов: в режиме редактирования и в режиме просмотра. Этим управляет флаг FO_ENABLE_EDIT в настройках объекта.

- В режиме редактирования элементы блок-схемы можно передвигать, изменять их размеры, вводить текст. В этом режиме, также, задействуются обработчики OnChange (изменение текста фигуры), OnMouseButton (обработчик клавиш мыши), OnDel (удаление выделенных элементов), OnSelChange (изменение списка выделенных элементов). Кроме того, рабочее поле блок-схемы может отображать сетку и выполнять выравнивание элементов по сетке. Это задается флагами FO_GRID и FO_SNAP_TO_GRID в настройках объекта.
- В режиме просмотра нельзя изменять параметры элементов, и из обработчиков событий задействован только OnMouseButton.

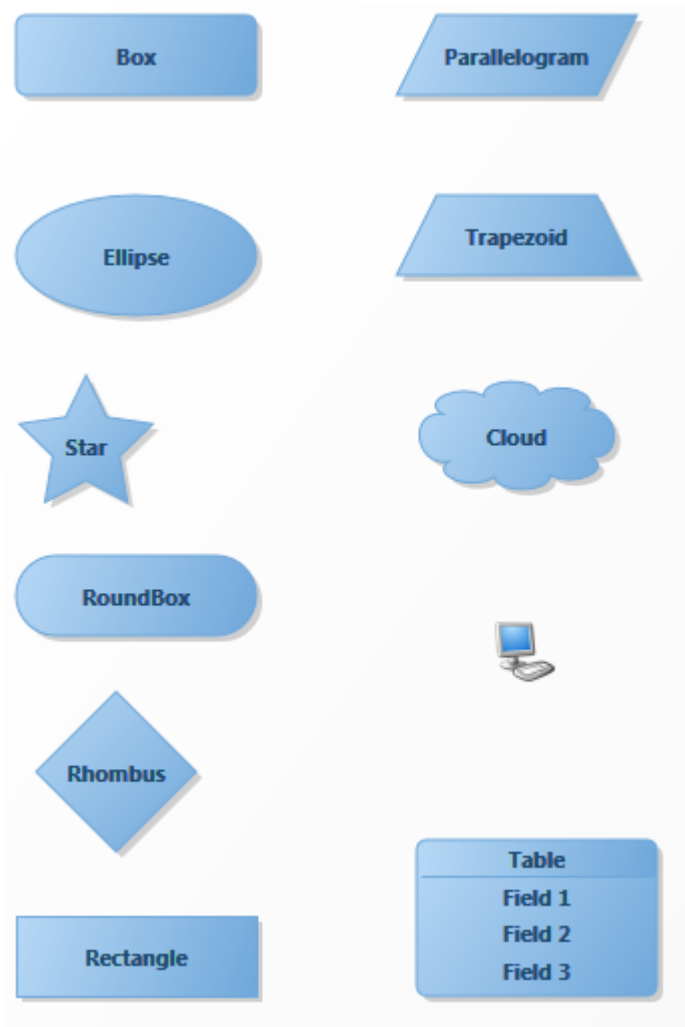
Независимо от режима отображения объект поддерживает функцию масштабирования с помощью колесика мышки с нажатой клавишей Ctrl, либо посредством команд контекстного меню. Эта возможность задается флагом FO_MOUSE_WHEEL в настройках объекта. Есть, также, возможность задавать пределы масштабирования.

Примеры построения блок-схем можно найти в демонстрационном модуле по пути: «Прочие объекты \ Блок-схемы».

11.1.1 Фигуры

Объект блок-схема поддерживает следующие виды фигур:

- SHAPE_BOX – прямоугольник со скругленными углами
- SHAPE_ELLIPSE – эллипс
- SHAPE_STAR – звезда
- SHAPE_ROUNDBOX – скругленный прямоугольник
- SHAPE_RHOMBUS – ромб
- SHAPE_RECTANGLE – прямоугольник
- SHAPE_PARALLELOGRAM – параллелограмм
- SHAPE_TRAPEZOID – трапеция
- SHAPE_CLOUD – облако
- SHAPE_PICTURE – пиктограмма
- SHAPE_TABLE – таблица



Фигура имеет следующие общие свойства:

- Координаты – функция FC_ShapeSetRect
- Текст – функция FC_ShapeSetText
- Цвет заливки рабочего поля – функция FC_ShapeSetFillColor
- Цвет контура – функция FC_ShapeSetBorderColor
- Цвет текста – функция FC_ShapeSetTxtColor

Для каждой подобной функции существует аналог «Get».

В режиме редактирования для каждой конкретной фигуры можно задавать дополнительные настройки редактирования, в частности, должен ли размер фигуры изменяться пропорционально, можно ли редактировать текст и т.д. Более подробная информация о флагах редактирования представлена в описании функции FC_ShapeSetEditFlags.

11.1.1.1 Фигура SHAPE_PICTURE

Данная фигура не имеет таких настроек, как текст и цвет. Она отображает связанную с ней пиктограмму (см. функцию FC_ShapeSetPicture).

11.1.1.2 Фигура SHAPE_TABLE

Функция FC_ShapeAdd с параметром SHAPE_TABLE создает пустую таблицу. Для управления полями таблицы служат функции FC_TableAddField и FC_TableDropField.

Данная фигура имеет две области прорисовки: рабочее поле и заголовок.

Цвет заливки и цвет текста заголовка задаются функциями FC_TableSetCaptionColor и FC_TableSetCaptionTxtColor.

Цвет заливки и цвет текста рабочего поля (где располагаются поля таблицы) задаются функциями FC_ShapeSetFillColor и FC_ShapeSetTxtColor.

11.1.2 Коннекторы

При создании коннектор имеет две точки: начальная и конечная. Функция FC_ConnectorAddPoint позволяет добавить дополнительные точки перегиба.

Коннектор имеет следующие свойства:

- Цвет – функция FC_ConnectorSetColor
- Вид стрелок (начальной и конечной) – функция FC_ConnectorSetArrow
- Вид линии: кривая или ломаная – задается при создании коннектора функцией FC_ConnectorAdd

11.2 Оператор создания экземпляра – FLOWCHART

HOBJECT FLOWCHART name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта «Блок-схема» и возвращает дескриптор экземпляра.

11.3 Настройки объекта

Помимо настроек, общих для всех визуальных объектов, данный объект имеет ряд собственных настроек. Все эти настройки определяют поведение объекта по умолчанию, но могут быть изменены в период исполнения с помощью функции SetOptions.

- Общие свойства (группа настроек)
  - Масштабирование колесиком мыши (флаг)
    - Если флаг установлен, то при нажатой клавише Ctrl вращение колесика мыши будет вызывать масштабирование
    - Имя флага для функций GetOptions и SetOptions – FO_MOUSE_WHEEL
  - Минимальный предел масштабирования (поле ввода для double)
    - Принимает значения от 0.1 до 1
  - Максимальный предел масштабирования (поле ввода)
    - Принимает значения от 1 до 10
  - Обработчик событий мыши (флаг)
    - Если флаг установлен, то при нажатии клавиш мыши будет вызываться обработчик OnMouseButton
    - Имя флага для функций GetOptions и SetOptions – FO_TRACK_MOUSE
  - Обработчик на правую клавишу мыши (флаг)
    - Если флаг установлен, то при нажатии на правую клавишу мыши будет вызываться обработчик OnMouseButton, иначе – контекстное меню
    - Имя флага для функций GetOptions и SetOptions – FO_MOUSE_RBUTTON
- Настройки редактирования (группа настроек)
  - Редактирование положения и размера элементов (флаг)
    - Если флаг установлен, то элементы можно выделять, перемещать и изменять их размер
    - Имя флага для функций GetOptions и SetOptions – FO_ENABLE_EDIT
  - Редактирование текста элемента (флаг)
    - Если флаг установлен, то доступно редактирование текста элемента «на месте»
    - Имя флага для функций GetOptions и SetOptions – FO_INPLACE_EDIT
  - Запрещено множественное выделение (флаг)
    - Если флаг установлен, то выделить можно только один элемент
    - Имя флага для функций GetOptions и SetOptions – FO_SINGLE_SEL

    - Сетка (флаг)
        - Если флаг установлен, то будет выводится сетка
        - Имя флага для функций GetOptions и SetOptions – FO_GRID
        - Доступен, если установлен флаг FO_ENABLE_EDIT
    - Выравнивать по сетке (флаг)
        - Если флаг установлен, то при перемещении элементов мышкой будет выполняться выравнивание их по сетке
        - Имя флага для функций GetOptions и SetOptions – FO_SNAP_TO_GRID
        - Доступен, если установлены флаги FO_ENABLE_EDIT и FO_GRID
    - Доступна клавиатура (флаг)
        - Если флаг установлен, то перемещение элементов возможно с помощью стрелок клавиатуры
        - Имя флага для функций GetOptions и SetOptions – FO_ENABLE_KEYBOARD
        - Доступен, если установлен флаг FO_ENABLE_EDIT
- Цвет рабочего поля (группа настроек)
    - Основной цвет (Color picker)
    - Цветовой переход (Color picker)
    - Тип цветового перехода (комбобокс)
        - Список содержит набор констант типов цветового перехода
    - Цвет сетки (Color picker)
    - Размер сетки (поле ввода для int)
- Цвет элемента
    - Основной цвет (Color picker)
    - Цветовой переход (Color picker)
    - Тип цветового перехода (комбобокс)
        - Список содержит набор констант типов цветового перехода
    - Цвет контура (Color picker)
    - Цвет текста (Color picker)

11.4 Специфические сегменты объекта

- OnBuild – построение блок-схемы
- OnChange – изменение размера или позиции элементов
- OnTool – добавление нового элемента
- OnDel – удаление элемента
- OnSelChage – изменение выделения элементов

Описание обаботчиков см. в электронной документации.

11.5 Функции работы с объектом

11.5.1 Общие сведения

Имена всех функций работы с блок-схемой имеют префикс «FC_» (FLOWCHART).

Все функции работы с объектом устанавливают код ошибки, который доступен через вызов функции GetLastError:

- В случае успешного завершения функция выставляет код ERR_OK.
- Если переданный дескриптор не достоверен, функция выставляет код ERR_BAD_HANDLE и завершается.
- Если переданный дескриптор не является дескриптором блок-схемы, или функция не может быть вызвана в данном контексте, она выставляет код ERR_NOT_SUPPORTED и завершается.
- Если объект еще не визуализирован, а функция этого требует, она выставляет код ERR_NOT_VISUALIZED и завершается.

11.5.2 Перечень функций по категориям

Общие функции

- FC_Redraw – обновить окно объекта
- FC_Rebuild – перестроить объект (вызов сегмента OnBuild)
- FC_Clear – удалить все элементы блок-схемы
- FC_DropConnectors – удалить все коннекторы
- FC_CopyToClipboard – копировать изображение в буфер обмена
- FC_SaveImage – сохранить изображение в файл
- FC_DropItem – удалить элемент

Управление элементами блок-схемы

- FC_ShapeAdd – добавить фигуру
- FC_ShapeBring – изменить позицию фигуры (Z-order)
- FC_ShapeGetType – возвращает тип фигуры
- FC_ShapeGetConnectors – возвращает все коннекторы, связанные с фигурой
- FC_ShapeGetRect – возвращает размер фигуры (прямоугольник границ)
- FC_ShapeSetRect – устанавливает размер фигуры (прямоугольник границ)
- FC_ShapeGetText – возвращает текст фигуры
- FC_ShapeSetText – устанавливает текст фигуры
- FC_ShapeGetFillColor – возвращает цвет заливки фигуры
- FC_ShapeSetFillColor – устанавливает цвет заливки фигуры
- FC_ShapeGetBorderColo – возвращает цвет контура фигуры
- FC_ShapeSetBorderColor – устанавливает цвет контура фигуры
- FC_ShapeGetTxtColor – возвращает цвет текста фигуры
- FC_ShapeSetTxtColor – устанавливает цвет текста фигуры
- FC_ShapeGetEditFlags – возвращает флаги редактирования фигуры
- FC_ShapeSetEditFlags – устанавливает флаги редактирования фигуры

Только для элемента типа ShapePicture

- FC_ShapeSetPicture – устанавливает пиктограмму фигуры типа ShapePicture

Только для элемента типа ShapeTable

- FC_TableAddField – добавляет поле в таблицу

- FC_TableDropField – удаляет поле таблицы
- FC_TableGetFieldCount – возвращает количество полей таблицы
- FC_TableGetFieldName – возвращает имя поля таблицы
- FC_TableSetFieldName – устанавливает имя поля таблицы
- FC_TableGetCaptionColor – возвращает цвет заливки заголовка таблицы
- FC_TableSetCaptionColor – устанавливает цвет заливки заголовка таблицы
- FC_TableGetCaptionTxtColor – возвращает цвет текста заголовка таблицы
- FC_TableSetCaptionTxtColor – устанавливает цвет текста заголовка таблицы

Коннекторы

- FC_ConnectorAdd – добавить коннектор
- FC_Connect – присоединить коннектор к фигуре
- FC_ConnectorAddPoint – добавить точку излома коннектора
- FC_ConnectorDropPoint – удалить точку излома коннектора
- FC_ConnectorGetPointCount – возвращает количество точек коннектора
- FC_ConnectorGetArrow – возвращает свойства стрелки коннектора
- FC_ConnectorSetArrow – устанавливает свойства стрелки коннектора
- FC_ConnectorGetShape – возвращает фигуру, связанную с коннектором
- FC_ConnectorGetColor – возвращает цвет линии коннектора
- FC_ConnectorSetColor – устанавливает цвет линии коннектора
- FC_ConnectorGetPoints – возвращает коллекцию точек коннектора
- FC_ConnectorIsSpline – возвращает тип линии (кривая/ломаная)
- FC_ConnectorSetSpline – устанавливает тип линии (кривая/ломаная)

Пользовательские данные

- FC_GetItemData – возвращает пользовательские данные, связанные с элементом
- FC_SetItemData – связывает пользовательские данные с элементом
- FC_DropItemData – удаляет пользовательские данные, связанные с элементом

Перебор элементов

- FC_GetItemCount – возвращает общее число элементов блок-схемы
- FC_GetItem – возвращает идентификатор элемента по индексу
- FC_IsConnector – является ли элемент коннектором
- FC_IsShape – является ли элемент фигурой

Работа с выделенными элементами

- FC_GetSelCount – возвращает число выделенных элементов
- FC_GetSelItem – возвращает идентификатор выделенного элемента по индексу
- FC_DropSelection – удалить все выделенные элементы
- FC_IsSelected – является ли элемент выделенным
- FC_Select – выделение элемента

Функции рабочего поля

- FC_GetColor – возвращает цвет заливки рабочего поля
- FC_SetColor – устанавливает цвет заливки рабочего поля
- FC_GetColorGrid – возвращает цвет сетки
- FC_SetColorGrid – устанавливает цвет сетки
- FC_GetGridSize – возвращает размер сетки
- FC_SetGridSize – устанавливает размер сетки
- FC_GetScaleRange – возвращает пределы масштабирования
- FC_SetScaleRange – устанавливает пределы масштабирования
- FC_GetScale – возвращает текущую величину масштаба
- FC_Scale – устанавливает текущую величину масштаба

Функции печати

- FC_Print – отправка блок-схемы на печать
- FC_PrintPreview – предварительный просмотр блок-схемы перед печатью

11.6 Объект в период исполнения

11.6.1 Контекстное меню

Структура меню:

1. Обновить (F5)
   1.1. Аналог функции Rebuild
   1.2. Комментарий: Перестроить блок-схему
2. Копировать изображение (Shift+Ctrl+C)
   2.1. Аналог функции Copy_ToClipboard. После вызова функции выдается сообщение: «Изображение скопировано в буфер обмена.»
   2.2. Комментарий: Копировать изображение в буфер обмена
3. Сохранить в файл (Ctrl+S)
   3.1. Вызывает стандартный диалог выбора файла с масками *.bmp и *.png, после чего отрабатывает функция SaveImage
   3.2. Комментарий: Сохранить изображение в файл
4. Масштабирование – группа. Все команды группы доступны только в том случае, если разрешено масштабирование (установлен флаг FO_MOUSE_WHEEL). Шаг масштабирования такой же, как при вращении колесика мыши.
   4.1. Больше
      4.1.1. Увеличивает масштаб (функция Scale)
      4.1.2. Комментарий: Увеличить масштаб
      4.1.3. Команда доступна, если не достигнута максимальная граница увеличения
   4.2. 100%
      4.2.1. Устанавливает масштаб в 100%
      4.2.2. Комментарий: Установить масштаб в 100%
      4.2.3. Команда доступна, если текущий масштаб отличается от 100%
   4.3. Меньше
      4.3.1. Уменьшает масштаб (функция Scale)
      4.3.2. Комментарий: Уменьшить масштаб
      4.3.3. Команда доступна, если не достигнута минимальная граница уменьшения
5. Печать
   5.1. Аналог функции FC_Print
   5.2. Комментарий: Печать блок-схемы
6. Предварительный просмотр
   6.1. Аналог функции FC_PrintPreview
   6.2. Комментарий: Предварительный просмотр печати

11.6.2 Горячие клавиши

- В режиме редактирования
  - Ctrl+C – копировать выделенные элементы
  - Ctrl+X – вырезать выделенные элементы
  - Ctrl+V – вставить скопированные элементы
  - Del – удалить выделенные элементы
  - Двойной клик левой клавиши мыши на коннекторе – добавить/удалить точку излома
  - Ctrl+клик левой клавиши мыши на элементе блок-схемы – выделить элемент / снять выделение
- F5 – перестроить блок-схему
- Ctrl+Shift+C – копировать изображение в буфер обмена
- Ctrl+S – сохранить изображение блок-схемы в файл
- Если доступно масштабирование (установлен флаг FO_MOUSE_WHEEL)
  - Ctrl + колёсико мыши вверх – увеличение масштаба
  - Ctrl + колёсико мыши вниз – уменьшение масштаба
  - Ctrl + нажатие по колёсику мыши – возврат масштаба к 100%

11.7 Предопределенные константы

11.7.1 Константы типов градиента

- GRAD_NO_GRADIENT
- GRAD_HORIZONTAL
- GRAD_VERTICAL
- GRAD_DIAGONAL_LEFT
- GRAD_DIAGONAL_RIGHT
- GRAD_CENTER_HORIZONTAL
- GRAD_CENTER_VERTICAL
- GRAD_RADIAL_TOP
- GRAD_RADIAL_CENTER
- GRAD_RADIAL_BOTTOM
- GRAD_RADIAL_LEFT
- GRAD_RADIAL_RIGHT
- GRAD_RADIAL_TOP_LEFT
- GRAD_RADIAL_TOP_RIGHT
- GRAD_RADIAL_BOTTOM_LEFT
- GRAD_RADIAL_BOTTOM_RIGHT
- GRAD_BEVEL
- GRAD_PIPE_VERTICAL
- GRAD_PIPE_HORIZONTAL

11.7.2 Константы портов соединения

- PORT_NONE
- PORT_CENTER
- PORT_CAPTION
- PORT_LEFT
- PORT_RIGHT
- PORT_TOP
- PORT_TOPLEFT
- PORT_TOPRIGHT
- PORT_BOTTOM
- PORT_BOTTOMLEFT
- PORT_BOTTOMRIGHT

11.7.3 Константы вида стрелок коннектора

- ARROW_NONE
- ARROW_OPEN
- ARROW_STEALTH
- ARROW_TRIANGLE
- ARROW_FILLEDTRIANGLE
- ARROW_CIRCLE
- ARROW_FILLEDCIRCLE
- ARROW_DIAMOND
- ARROW_FILLEDDIAMOND

12 Отчеты

Объект «Отчет» представляет собой контейнер для хранения шаблона отчета и программного кода, формирующего отчет из шаблона. Экземпляр объекта «Отчет» идентифицируется дескриптором (тип HOBJECT), который, как и в случае других объектов X2, возвращается оператором создания экземпляра.

При создании экземпляра объекта система выгружает шаблон в дисковый файл в специальную директорию в профиле пользователя. Имя файла формируется системой таким, чтобы оно было уникально в рамках данной директории. В том случае, если в настройках объекта указано, что отчет имеет формат Crystal Reports, система создает экземпляр просмотрщика Crystal Reports.

После этого система вызывает конструктор объекта. В конструкторе можно расположить код, формирующий отчет. Имя файла шаблона и путь можно получить с помощью функции ReportGetTemplate.

В отличие от прочих объектов X2, для отчетов существует явный признак (флаг в настройках), который указывает, должна ли система автоматически удалить экземпляр объекта сразу же после отработки его конструктора. Если этот флаг опущен, программист должен самостоятельно удалить экземпляр с помощью функции DestroyObject. Если этого не сделать, то система удалит его только при завершении сеанса.

При уничтожении экземпляра система вызовет деструктор объекта. В деструкторе можно расположить код, удаляющий временные объекты или данные из БД, которые были сформированы для данного отчета.

После этого система удалит файл шаблона и, в случае отчета Crystal Reports, уничтожит экземпляр просмотрщика.

Примеры отчетов можно найти в демонстрационном модуле. Путь по меню: «Прочие объекты \ Отчеты».

12.1 Принципы формирования отчетов

Для формирования отчета можно использовать скриптовые языки (операторы VBS и JS). В этом случае можно задействовать любое подходящее приложение, имеющее COM-интерфейс, в частности, Word или Excel.

Для отчетов в формате документов MS Office можно использовать VBA. Для этого в шаблоне нужно реализовать набор макросов, которые и будут формировать отчет.

Для отчетов в формате Crystal Reports предусмотрена специальная опция в настройках объекта и определен набор встроенных функций. При передаче параметров в отчет следуйте рекомендациям, данным в п. 3.1 «Соответствие типов X2 типам Crystal Reports».

Можно, также, использовать семейства функций с префиксами «Docx» и «Xlsx». Эти функции позволяют работать с файлом шаблона напрямую, без загрузки MS Office. Но в этом случае шаблон должен иметь формат Office Open XML.

12.2 Оператор создания экземпляра – REPORT

HOBJECT REPORT name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта «Отчет» и возвращает дескриптор экземпляра.

При создании экземпляра отчета Crystal Reports, система создает, также, экземпляр просмотрщика. Если при инициализации просмотрщика возникнет ошибка, оператор вернет FALSE. В этом случае система запросит у Crystal Reports текст сообщения об ошибке, и если текст сообщения будет не пустым, система выведет его на экран.

12.3 Общий перечень функций

1. ReportGetTemplate
2. ReportGetTitle
3. ReportSetTitle
4. Функции работы с отчетами Crystal Reports
   4.1. ReportShow
   4.2. ReportPrint
   4.3. ReportSetParam
5. Функции семейства «Docx»
   5.1. Работа с документом
      5.1.1. DocxOpen – открыть документ
      5.1.2. DocxSave – сохранить документ на диск
      5.1.3. DocxClose – завершить работу с документом и освободить ресурсы
      5.1.4. DocxGetName – получить имя документа
      5.1.5. DocxSetName – изменить имя документа
      5.1.6. DocxGetErrorString – получить строку ошибки транслятора XML или ZIP-алгоритма
   5.2. Подстановка строк в документ
      5.2.1. DocxSetVarValue – установить значение переменной документа
      5.2.2. DocxReplaceBookmark – заменить закладку на указанный текст
      5.2.3. DocxReplaceText – заменить текст на указанный текст
   5.3. Работа с таблицами

5.3.1. DocxRowAdd – добавить строку в конец таблицы

5.3.2. DocxRowInsert – вставить строку в таблицу по указанному индексу

5.3.3. DocxRowDelete – удалить строку из таблицы по указанному индексу

5.3.4. DocxRowGetCount – получить количество строк в таблице

5.3.5. DocxCellSetData – установить данные в указанной ячейке таблицы

5.3.6. DocxCellSetFont – установить шрифт в указанной ячейке таблицы

5.3.7. DocxCellSetFontSize – установить размер шрифта в указанной ячейке таблицы

5.3.8. DocxCellSetFontWeight – установить насыщенность шрифта в указанной ячейке таблицы

5.3.9. DocxCellSetFontFlags – установить флаги шрифта в указанной ячейке таблицы

5.3.10. DocxCellSetFontColor – установить цвет шрифта в указанной ячейке таблицы

5.3.11. DocxCellSetAligment – установить выравнивание шрифта в указанной ячейке таблицы

6. Функции семейства «Xlsx»

6.1. Работа с документом

6.1.1. XlsxOpen – открыть документ

6.1.2. XlsxSave – сохранить документ на диск

6.1.3. XlsxClose – завершить работу с документом и освободить ресурсы

6.1.4. XlsxGetName – получить имя документа

6.1.5. XlsxSetName – изменить имя документа

6.1.6. XlsxGetErrorString – получить строку ошибки транслятора XML или ZIP-алгоритма

6.2. Управление листами документа

6.2.1. XlsxSheetGetActive – получить номер активного листа

6.2.2. XlsxSheetSetActive – установить активным указанный лист

6.2.3. XlsxSheetGetCount – получить количество листов в документе

6.2.4. XlsxSheetGetName – получить имя листа

6.2.5. XlsxSheetSetName – изменить имя листа

6.2.6. XlsxSheetInsert – вставить новый лист в документ

6.2.7. XlsxSheetDelete – удалить указанный лист из документа

6.3. Строки и колонки

6.3.1. XlsxRowInsert – вставить строку

6.3.2. XlsxRowDelete – удалить строку

6.3.3. XlsxColInsert – вставить колонку

6.3.4. XlsxColDelete – удалить колонку

6.4. Данные в ячейках

13 Библиотеки

Объект «Библиотека» обеспечивает возможность вызова внешних модулей из кода на языке X2. Внешний модуль может представлять собою сборку .NET, либо библиотеку (dll) C++. С каждым объектом «Библиотека» можно связать только один внешний модуль.

Экземпляр объекта «Библиотека» идентифицируется дескриптором (тип HOBJECT), который, как и в случае других объектов X2, возвращается оператором создания экземпляра.

При создании экземпляра объекта система выгружает модуль в дисковый файл в специальную директорию в профиле пользователя, и позиционирует dll в память. Имя файла формируется системой таким, чтобы оно было уникально в рамках данной директории.

После этого система вызывает конструктор объекта. В конструкторе можно подготовить данные или создать временные объекты в БД, если это требуется для работы библиотеки.

С этого момента можно вызывать функции, экспортируемые внешним модулем (функция LibCall).

Когда экземпляр объекта стал не нужен, программист должен удалить его с помощью функции DestroyObject. Если этого не сделать, то система удалит его при завершении сеанса.

При уничтожении экземпляра система вызовет деструктор объекта. В деструкторе можно расположить код, удаляющий данные, подготовленные в конструкторе.

Когда система выгрузит внешний модуль из памяти и удалит файл dll, зависит от режима кэширования. Если кэширование включено (обычный режим эксплуатации), то это произойдет при завершении сеанса. Если кэширование отключено, то после того, как будет завершен последний экземпляр объекта.

13.1 Системные требования

Для использования сборок .NET в качестве библиотек необходимо, чтобы на клиентской машине была установлена платформа .NET версии 4.5.

13.2 Оператор создания экземпляра – LIBRARY

HOBJECT LIBRARY name (parameters…);

Где:

- name – имя объекта
- parameters – параметры объекта, заданные разработчиком

Оператор создает экземпляр объекта «Библиотека» и возвращает дескриптор экземпляра.

13.3 Свойства объекта

Поле «Заголовок» носит исключительно информационный характер и не используется системой. Заголовок будет помещен в поле Title таблицы x2Objects.

Поле «Библиотека» предназначено для отображения имени файла библиотеки (путь к файлу не отображается и не сохраняется в ресурсе).

Поле «Сборка .NET» (флаг LIO_NET) определяет способ загрузки, с помощью которого система будет позиционировать библиотеку в память.

13.4 Экспорт

В случае сборки .NET.

Для того, чтобы метод класса был доступен для вызова из кода на X2, он должен соответствовать одному из прототипов:

Int32 Members(OdbcConnection ^rConnect, String ^str, String ^%strOut)

Int32 Members(String ^str, String ^%strOut)

где:

- rConnect – соединение с БД
- str – строка входных параметров
- strOut – строка выходных параметров

В случае библиотеки C++.

Строки передаются в кодировке UTF16.

Прототип экспортной функции:

```
extern "C" __declspec ( dllexport ) int Func
(
        CDatabase*      pDatabase,      // соединение с БД
        LPCTSTR         szParam,        // строка входных параметров
        CString*        strOut,         // строка выходных параметров
        CWnd*           pwndParent      // окно родителя (главное окно приложения)
);
```

Функции возвращают 32-битное целое, смысл которого определяет сам программист. Значение -1 зарезервировано для системных нужд.

13.5 Пример вызова функции внешней библиотеки

Пример вызова функции внешней библиотеки можно найти в демонстрационном модуле. Путь по меню: «Прочие объекты \ Библиотеки».

Пример демонстрирует вызов функции «Func», которую экспортирует библиотека C++ «X2_Library». Работа с библиотеками .NET выполняется аналогичным образом.

Код функции на C++ приведен ниже:

```
extern "C" __declspec ( dllexport ) int Func
(
	CDatabase*	pDatabase,	// соединение с БД
	LPCTSTR		szParam,	// строка входных параметров
	CString*	strOut,		// строка выходных параметров
	CWnd*		pwndParent	// окно родителя (главное окно приложения)
)
{
	if (pwndParent && ::IsWindow(pwndParent->GetSafeHwnd()))
	{
		::MessageBox
		(
			pwndParent->GetSafeHwnd(),
			szParam,
			_T("Сообщение X2_Library.dll"),
			MB_OK | MB_ICONINFORMATION
		);
	}

	*strOut = _T("Вызов функции отработал корректно!");
	return 1;
}
```

14 Объекты замещения

При внедрении серийной версии какого-либо программного продукта, как правило, требуется его доработка под нужды конкретного клиента. Поскольку приложение X2 строится из объектов, то такая доработка предполагает внесение изменений в коды или настройки отдельных объектов. Приложение X2 имеет открытый код, поэтому внесение изменений не представляет какой-то проблемы. Но проблема возникает позже, когда клиент захочет обновить серийную версию продукта. Обновление удалит старые версии объектов и установит новые. Это может затронуть, в том числе, и те объекты, которые были доработаны на стороне клиента. Соответственно, доработки придется как-то восстанавливать. Чтобы сократить трудозатраты, возникающие в таких случаях, и вводятся объекты замещения.

14.1 Концепция

Идея состоит в том, чтобы отделить код серийной версии от доработок, выполненных на стороне клиента и хранить их отдельно друг от друга. Согласно этой концепции, при доработке серийной версии ее объекты не подвергаются модификации. Вместо этого создаются новые объекты с теми же именами, которые на этапе исполнения будут замещать серийные. Таким образом, возникает понятие базовых объектов (далее БО), из которых строится приложение, и объектов замещения (далее ОЗ), которые выступают в роли заместителей базовых. Поскольку БО и ОЗ хранятся в разных таблицах, обновление серийной версии не затронет ОЗ.

Этот подход не гарантирует, что после обновления серийной версии все ОЗ будут корректно работать. Если обновление меняет саму логику работы программы, то может случиться так, что код, написанный в ОЗ с учетом старой логики, окажется некорректным для новой логики. Возможна и такая ситуация, что БО, который был в старой версии, в новой просто исчез. Тогда и ОЗ для него вызываться не будет. Поэтому в общем случае после обновления следует протестировать доработанный функционал. Тем не менее, как показывает практика, в подавляющем большинстве случаев объекты замещения сохраняют свою работоспособность.

Платформа позволяет выполнять экспорт и импорт объектов через дисковый файл. При импорте объектов в специальных служебных таблицах сохраняется история импорта. История импорта позволяет выявить все измененные объекты, для которых существовали ОЗ. Это может существенным образом сократить объем тестирования, который необходимо провести после обновления.

Также, после обновления серийной версии рекомендуется провести перекомпиляцию всех объектов замещения.

14.1.1 Функциональные группы и механизм загрузки объектов

Объекты замещения объединяются в функциональные группы. Группы связываются с пользователями или ролями приложения (application role). Смысл группы в том, что для одних пользователей будет отрабатывать один набор ОЗ, а для других – другой. Здесь следует уточнить, что в одной группе не может быть двух одноименных объектов, а разных группах может.

Всегда существует предопределенная группа «Для всех». Данная группа не может иметь связей с ролями. Объекты этой группы замещают базовые в случае, если для роли не найдено соответствующего замещения в других группах.

В период исполнения механизм загрузки объектов работает следующим образом:

1. Запрашиваем имя текущего пользователя (роли) и ищем группу, с ним связанную
   1.1. Если есть группа, ищем в ней ОЗ
      1.1.1. Если найден, используем его
2. Если ОЗ не найден, ищем его в группе «Для всех»
   2.1. Если найден, используем его
3. Если ОЗ не найден, ищем базовый объект
   3.1. Если найден, используем его
   3.2. Если не найден, переходим в состояние ошибки

14.1.2 Замещение и наследование

Объект замещения может наследовать базовый, а может замещать его, выступая как самостоятельный объект. Это регулируется признаком, который задается явным образом на этапе разработки ОЗ. Смысл наследования состоит в том, что ОЗ в период исполнения наследует все свойства базового объекта за исключением тех, которые в нем заданы явным образом. В случае же замещения, ОЗ просто отрабатывает, как есть, взамен базового.

Сам механизм загрузки работает таким образом, что для какого-то ОЗ может и не существовать базового объекта. Это нормальная ситуация. Если на этапе внедрения потребовался какой-то объект, например, отчет или список, которого вообще нет в серийной версии то нужно создать его, как ОЗ и вызвать из другого ОЗ.

Если случай замещения не вызывает вопросов, то случай наследования требует некоторых разъяснений. Для ОЗ-наследника справедливо следующее:

1. Он должен иметь те же входные параметры, что и его базовый объект. Список его параметров не может быть изменен.
2. На этапе исполнения для ОЗ-наследника загружается и его базовый объект. Базовый объект должен существовать, иначе возникнет ошибка периода исполнения.
3. Для созданного экземпляра объекта
   - 3.1. Наследуется состав внешних деклараций, включенных в БО, т.е. в наследнике доступны все типы данных, глобальные переменные и константы, объявленные в этих декларациях.
   - 3.2. Наследуется встроенная декларация, т.е. в наследнике доступны все типы данных, локальные переменные и константы, объявленные во встроенной декларации БО.
   - 3.3. Все свойства объекта наследуются полностью. Интерфейс редактора ОЗ-наследника не позволяет менять свойства объекта, например, настройки колонок списка. Тем не менее, свойстами можно управлять программным путем помощью функции SetOptions. Аналогично можно задавать и свойства колонок.
   - 3.4. Исполняемые сегменты кода наследуются в том случае, если они не реализованы в ОЗ. Если в ОЗ реализован какой-то сегмент кода, то он будет исполняться вместо соответствующего базового. В этом случае функция CallBaseSegment позволит явным образом вызвать сегмент БО.

При разработке ОЗ функция CallBaseSegment позволяет модифицировать кодовый сегмент базового объекта так, чтобы в ОЗ исполнить собственный код до, после или вместо базового.

К примеру, для ряда объектов приложения нам требуется обеспечить протоколирование того, кто и когда вызывал их на исполнение. Допустим, мы создали для этого таблицу

```
CREATE TABLE Protocol
(
        ObjectType      int,
        ObjectName      nvarchar (255),
        UserName        nvarchar (255) DEFAULT CURRENT_USER,
        CallDate        datetime DEFAULT GetDate ()
)
```

Тогда в конструкторах этих объектов мы можем написать код, подобный этому

```
{
	INSERT INTO Protocol (ObjectType, ObjectName) VALUES
	(
		:(GetObjectType ()), :(StrQuote (GetObjectName ()))
	);

	return CallBaseSegment ();
}
```

Здесь мы вызываем вставку в таблицу протокола, а потом вызываем конструктор базового объекта. Если конструктор базового объекта по каким-то своим причинам вернет FALSE, то и наш ОЗ завршится. Если же конструктор БО не реализован, то функция CallBaseSegment вернет значение по умолчанию для конструктора.

14.1.3 Отключение поддержки ОЗ

Включение и отключение поддержки механизма замещения объектов осуществляется для пользователя БД (роли приложения). Проверка этого признака выполняется только на этапе загрузки среды исполнения и действует до ее завершения. Т.е. при переключении признака поддержки ОЗ требуется перезапуск среды исполнения.

Отключение поддержки ОЗ может быть полезно, если при исполнении возникает ошибка, и требуется быстро выяснить, чем она вызвана, кодом серийной версии или кодом доработок.

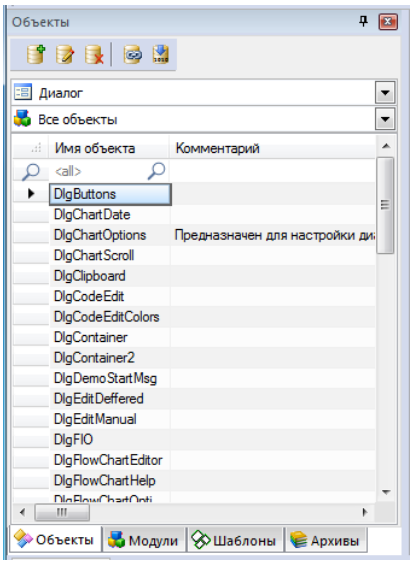
14.1.4 Ограничения по типам объектов

Механизм замещения не поддерживается для следующих типов объектов:

- Модули
- Декларации

14.2 Рекомендации по разработке ОЗ

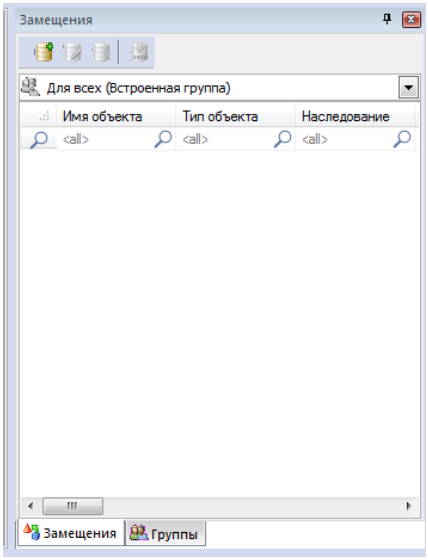
Для работы с базовыми объектами служит панель «Объекты». Путь по меню в Дизайнере: «Вид \ Панели \ Объекты».



Для экспорта и импорта базовых объектов служат пункты меню Дизайнера «Сервис \ Сопровождение \ Базовые объекты \ Экспорт» и «Сервис \ Сопровождение \ Базовые объекты \ Импорт».

К интерфейсу истории импорта базовых объектов можно получить доступ с помошью пункта меню «Сервис \ Сопровождение \ Базовые объекты \ История импорта».

Для работы с объектами замещения служат панели «Замещения» и «Группы». Путь по меню в Дизайнере: «Вид \ Панели \ Замещения» и «Вид \ Панели \ Группы».



Для экспорта и импорта объектов замещения служат пункты меню «Сервис \ Сопровождение \ Объекты замещения \ Экспорт» и «Сервис \ Сопровождение \ Объекты замещения \ Импорт».

14.2.1 Список

Изменить свойства колонки можно с помощью функций ListSetFieldOptions, ListSetColTitle. В частности, функция ListSetFieldOptions позволяет скрыть колонку. Эти функции можно вызывать из конструктора ОЗ – наследника.

Запрос списка наследуется полностью и не может быть подменен программно. Если требуется добавить поле или подменить запрос, то следует создать ОЗ без наследования.

14.2.2 Дерево

При построении ОЗ для дерева нужно следовать общим принципам разработки ОЗ. Каких-либо специальных рекомендаций на этот счет нет.

14.2.3 Меню

Каждый пункт меню имеет собственный идентификатор, уникальный в рамках объекта. По этому идентификатору система будет искать соответствующий пункт в базовом объекте.

В ОЗ-наследнике можно произвольным образом менять структуру меню, переставляя существующие пункты, либо удаляя их. Также, можно добавлять новые пункты. Новая структура меню будет сохранена в объекте замещения.

Если пункт меню имеет идентификатор, присутствующий в БО, то действует принцип наследования:

- если обработчик не пуст, будет исполнен он
- если он пуст, будет исполнен обработчик БО

Для пунктов меню, добавленных на этапе кастомизации, обработчик должен быть задан.

Управлять состоянием существующих пунктов меню можно с помощью API меню.

14.2.4 Диалог

Чтобы изменить форму диалога или его запрос, следует создавать ОЗ без наследования.

В ОЗ-наследнике можно манипулировать управляющими элементами диалога с помощью функций API этих элементов.

14.2.5 Локальные функции

Для объектов замещения с установленным флагом наследования

- Если для функции, определенной в ОЗ, нет функции БО с тем же определением, то данная функция ОЗ является самостоятельной, и может быть вызвана только из данного ОЗ
- Если функция, определенная в ОЗ-наследнике, имеет то же определение, что и функция базового объекта, то она замещает данную функцию БО. Функция имеет то же определение, означает, что:
    - Совпадает имя функции
    - Совпадает тип возврата
    - Совпадает количество параметров, их последовательность и типы (имена параметров не играют роли)
    - Значения по умолчанию должны быть заданы для всех тех же параметров, что и в функции БО, но сами значения могут быть иными. При вызове локальной функции ОЗ, замещающей функцию БО, будут применены значения по умолчанию, заданные в ОЗ.
- Из функции ОЗ, замещающей функцию БО, возможен вызов функции CallBaseSegment. Из самостоятельной функции ОЗ такой вызов не приведет к ошибке, но и не выполнит никаких действий.

```
// Пусть в базовом объекте задана функция
int Func (int i, int j = 1)

{
        MessageBox (ToString(i), + " " + ToString(j));
        return 0;
}

// В ОЗ-наследнике
int Func (int k, int m = 2)         // Замещение функции БО
int Func (string s)                 // Самостоятельная функция ОЗ
int Func (int т, string s = “”)     // Неоднозначное определение (вызовет ошибку компиляции)

// Пусть в ОЗ-наследнике функция int Func (int k, int m = 2) реализована так
{
        MessageBox (ToString(k), + " " + ToString(m));

        k = 33;
        m = 22;
        return CallBaseSegment ();
}

// Пусть БО вызывает функцию
int nResult = Func (10);
```

- В случае вызова базовой версии функции (поддержка ОЗ выключена) функция выдаст сообщение: «10 1»
- В случае вызова функции замещения (поддержка ОЗ включена) функция выдаст последовательно два сообщения: «10 2» и «33 22»