

## **ПРЕДИСЛОВИЕ**

Добро пожаловать в “Руководство программиста” по Clarion для Windows. Эта книга призвана снабдить программирующего на языке Clarion большим количеством очень полезных сведений о более глубоких слоях функционирования Clarion для Windows.

В этой книге раскрываются многие внутренние вопросы функционирования Clarion для Windows, которые не рассматриваются в остальной документации. Почему не рассматриваются? Ответ прост - большинство представленных здесь сведений становятся наиболее полезными, когда вы переходите к расширению функциональных возможностей Clarion для Windows (для собственных нужд или при создании дополняющих Clarion программных продуктов). В обычных же случаях большинство описанных здесь функций выполняются для вас средой разработки и библиотечными функциями.

Вот почему мы создали это руководство - специально для опытных программистов, для кого освоение основ (использование Генератора приложений) позади, кто хочет освоить более глубокие уровни программирования на языке Clarion (такие как написание кода вручную, или написание приложений, используя сочетание нескольких языков) и перед кем стоят новые, более сложные проблемы.

Это руководство состоит из трех частей:

### **Часть I - программирование на языке Clarion**

Здесь глубоко рассматриваются технологии кодирования на языке Clarion. Она начинается с обучения основам языка и вплоть до объектно-ориентированного программирования на Clarion'e.

### **Часть II - Ресурсы для более глубоких уровней программирования**

Подробно рассматриваются такие вопросы как: система ведения проекта среды TopSpeed (все управляющие предложения, которые можно использовать в Clarion для Windows), результаты, получаемые при использовании функции “Export Text” в Генераторе Приложений или Редакторе словаря (формат файлов .TXD//TXT), использование в одном проекте нескольких языков (Clarion и C/C++/Модула-2) и многие многие другие вопросы.

На установочном диске Clarion имеется набор полезных файлов, связанных с разделами данного руководства и представляющих собой: стандартные операторы EQUATE и прототипы функций для обращения интерфейсу прикладного программирования Windows (API), заголовочные файлы C++/Модула-2 для обращений к драйверам TopSpeed, а также прототипы на языке Clarion для многих полезных функций из стандартной библиотеки C.

### **Часть III - Язык программирования шаблонов**

Эта часть представляет собой Справочное руководство по языку программирования шаблонов, в котором полностью описан синтаксис мета-языка шаблонов. Для того, чтобы облегчить написание ваших собственных шаблонов, в нее также включены аннотированные описания нескольких стандартных шаблонов Clarion.

Все эти “низкоуровневые” средства позволяют вам насколько возможно снять ограничения, которые могут встретиться при программировании на языке Clarion. Вы увидите, что реальных ограничений нет (кроме вашего собственного воображения). Исследуйте глубины и наслаждайтесь.

## **Глава I Язык программирования CLARION**

### **Clarion-как язык программирования**

Основой Clarion для Windows является язык программирования Clarion. Это язык 4-го поколения (4GL), который соединяет в себе черты и языка общего назначения и языка баз данных. Это язык баз данных, потому что в нем есть структуры данных и команды, предельно оптимизированные для работы с файлами данных и прикладного программирования баз данных. С другой стороны - это язык общего назначения, потому что он компилируемый (а не интерпретируемый) и содержит набор операторов, функционально сопоставимый с языками 3-го поколения (такими как C, C++, Модула 2, Паскаль и т.д.). Отличия от них более детально обсуждаются в разделе Предисловие - история создания языка Clarion в документе Описание языка.

К данному моменту вы должны пройти все уроки по Генератору приложений в документе С чего начать. Назначение данных уроков по языку - дать вам представление об основных сторонах языка Clarion, особенно о прикладном программировании в рамках парадигмы событийного управления Windows. Ключевые слова языка Clarion выделяются ЗАГЛАВНЫМИ буквами и основное внимание в этих уроках обращается на конкретное использование представляемого данным ключевым словом языкового средства и его взаимодействия с другими элементами языка и только в том конкретном контексте, в котором он в данном случае используется. За более полным описанием представленного каким-либо ключевым словом элемента языка и его возможностей всегда следует обращаться к книге Описание языка.

Когда вы пройдете эти краткие уроки, ознакомьтесь:

- со структурой программы на языке Clarion;
- с наиболее общими программными структурами, используемыми для обработки событий;
- с тем как откомпилировать программу.

Если посмотреть исходный текст, сгенерированный Генератором приложений, или примеры в интерактивной системе помощи, то вы увидите, что лежащая в основе их логика и операторы обработки событий очень похожи на примеры, которые вы напишете во время этих уроков. Это должно облегчить понимание текста программы, автоматически генерируемого в соответствии с шаблонами, и осознание того, где и как вставлять свой собственный код для выполнения ваших конкретных требований.

### **Событийно-управляемые программы**

---

Программы для среды Windows являются событийно-управляемыми. Щелкнув кнопкой

мышью на каком-либо объекте в окне или нажав клавишу на клавиатуре, пользователь вызывает возникновение события. Любое такое действие приводит к тому, что Windows посылает сообщение (событие) программе, к которой относится данное окно, оповещая о том, что предпринял пользователь.

Как только Windows передала программе сообщение, сигнализирующее о событии, программа получает возможность обработать его соответствующим образом. В принципе это означает, что парадигма программирования в Windows прямо противоположна парадигме программирования для среды DOS: операционная система (Windows) сообщает программе что сделал пользователь, в противоположность ситуации, когда программа запрашивает у операционной системы выполнение каких-либо функций.

То, что действия пользователя все время находятся в центре внимания - это наиболее важная сторона программирования для Windows. Поэтому, программа в среде Windows является в большей степени реагирующей стороной, чем вызывающей какие-либо действия. Действия программы всегда больше связаны с тем, что сделал пользователь, вместо того, чтобы диктовать ему, что дальше делать. Наиболее ярким примером такого подхода служит диалоговое окно для ввода неких данных. В большинстве программ для DOS пользователь должен следовать единственным путем, вводя данные в одно поле за другим. Прежде чем перейти к следующему полю, он должен ввести данные в текущее (и обычно можно перейти только в конкретное “следующее” поле). Такой подход позволяет очень просто реализовать проверку введенных данных - она выполняется непосредственно после того, как пользователь “покинул” поле.

В программе для Windows пользователь для перехода к другому объекту в окне может использовать манипулятор “мышь” или нажатие комбинации “клавиш ускоренного доступа”. Делать он это может в любой момент времени, не придерживаясь какого-либо порядка, вообще пропуская некоторые поля. Таким образом операторы проверки правильности введенных данных, должны быть выполнены дважды, чтобы обеспечить уверенность в том, что хотя бы один раз проверка данных будет выполнена. Первый раз действия по проверке данных следует попытаться выполнить, когда пользователь после ввода данных переключается на другой объект в этом окне, и еще раз, когда он нажимает кнопку ОК, чтобы завершить ввод данных и покинуть это окно. Если в этот момент не выполнить такие проверки, то можно допустить пропуск данных, которые должны быть в этом окне обязательно введены. Такой подход и делает программы для Windows в большей мере реагирующими на действия пользователя, нежели руководящими его действиями.

## Привет Windows

---

Все книги, по которым происходит обучение какому-либо языку программирования традиционно начинаются с написания программы “Hello World” - последуем этой традиции и мы.

### **Создание файла исходного текста**

1. В окне File Manager Windows 3.1, выберите подкаталог CW20 и затем выберите в меню File пункт Create Directory. В поле Name введите Language, а затем нажмите кнопку ОК.

ИЛИ

В окне Explorer Windows95, выберите подкаталог CW20 выберите File , затем New и Folder. В поле, которое затем появится введите Language, и нажмите ENTER.

2. Вернитесь в Clarion для Windows.

3. Выберите File, затем New.

Появится диалоговое окно New. Это стандартное для Windows диалоговое окно открытия файлов, позволяющее сменить каталог и ввести имя файла.

4. Выберите карточку Source.

5. Выберите из списка каталог \CW20\LANGUAGE

6. Введите Hello в поле File Name.

7. Нажмите кнопку Create.

Таким образом создается пустой файл HELLO.CLW (.CLW - это стандартное расширение у файлов исходного текста Clarion для Windows) и вы переходите в текстовый редактор.

### **Создание проектного файла**

1. Выберите Project, затем - New.

Раскроется диалоговое окно New Project. При написании программы “вручную” все равно нужен файл описания проекта (назовем его проектным файлом - HELLO.PRJ), который управляет процессом компиляции и компоновки программы.

2. Включите радиокнопку Hand Coded Project.

3. Введите C:\CW20\LANGUAGE в поле Working Directory ( или используйте кнопку с многоточием, расположенную рядом с ним, чтобы выбрать каталог в окне Change Working Directory).

При ручном написании программы каталог, содержащий файл проекта (.PRJ), становится рабочим каталогом для проекта. При выборе проекта среда Clarion для Windows автоматически делает рабочим каталог, содержащий проектный файл.

4. Нажмите кнопку ОК.

Раскроется окно New Project File.

5. В поле Project Title введите текст Hello Windows, а затем нажмите клавишу TAB.

6. Введите в поле Main File имя файла HELLO.CLW и нажмите TAB.

Поскольку вы озаглавили исходный модуль в проекте, остальные поля в окне автоматически примут значения по умолчанию.

7. Нажмите кнопку ОК.

Раскроется диалоговое окно Project Editor. В нем указывается, какие модули исходного текста включаются компилятором в проект и библиотеки, которые компоновщик использует при создании исполняемого модуля. Когда, вместо того, чтобы использовать Генератор Приложений, все приложение пишется вручную, самостоятельно нужно изменять и файл проекта, включая в него все используемые модули исходного текста, файловые драйверы, пиктограммы и внешние библиотеки. Полную информацию о том, как описывать ваш проект можно найти в главе Использование проектной системы в Руководстве пользователя..

8. Нажмите кнопку ОК.

Теперь мы переключимся на файл HELLO.CLW в текстовом редакторе.

Отметим, что заголовок окна среды теперь выглядит следующим образом: Clarion for Windows (HELLO.PRJ) - (C:\CW20\LANGUAGE\HELLO.PRJ). В нем всегда отображается текущий рабочий каталог и текущий проект. При ручном написании программы - это очень важно, поскольку, просто открыв исходный файл в текстовом редакторе, мы не изменим рабочий каталог для компилятора.

### **Написание программы**

1. Наберите следующий текст:

```
PROGRAM
MAP
END
MyWin      WINDOW('Hello Windows'),SYSTEM
           END
CODE
OPEN(MyWin)
ACCEPT
END
```

Он начинается оператором PROGRAM. В любой программе на языке Clarion это первый отличающийся от комментария оператор. Отметим, что ключевое слово PROGRAM сдвинуто относительно слова MyWin. В исходном тексте на языке Clarion с первой позиции

строки может начинаться только оператор, имеющий метку. Метка по определению должна начинаться с первой колонки. Оператором PROGRAM в любой программе начинается секция глобальных данных.

Далее идет пустая структура MAP. Обязательным оператором, заканчивающим структуру MAP, является оператор END. В структуре MAP содержатся прототипы, которые определяют типы данных для параметров, типы возвращаемых значений и различные другие элементы, которые сообщают компилятору о том, как реализовывать обращения к процедуре или функции (все это будет рассматриваться позднее). Структура MAP требуется, когда текст вашей программы разбивается на отдельные процедуры и функции. Мы этого с вами еще не делали, но эта структура необходима, потому что мы использовали в программе оператор OPEN.

Когда компилятор обрабатывает структуру MAP, он автоматически включает в нее прототипы, находящиеся в файле \CW20\LIBSRC\BUILTINS.CLW. Этот файл содержит прототипы почти всех встроенных функций языка Clarion (включая и оператор OPEN). Если бы в тексте нашей программы не было пустой структуры MAP, компилятор выдал бы сообщение об ошибке с указанием на оператор OPEN.

MyWin - это метка структуры данных типа WINDOW (буква М должна быть в первой колонке). В языке Clarion окна объявляются как структуры данных, а не строятся динамически исполняемыми операторами как в некоторых других языках. Это один из аспектов Clarion, которые и делают его языком 4-го поколения. Хотя он тоже может динамически создавать окна во время выполнения программы, поступать так нет необходимости. Используя структуру данных компилятор для каждого диалогового окна использует ресурсы Windows, обеспечивающие во время выполнения наилучшую производительность.

Параметр ('Hello Windows') в операторе WINDOW определяет заголовок окна. Атрибутом SYSTEM в окно добавляется стандартное системное меню Windows. Оператор END является обязательным оператором, заканчивающим структуру WINDOW. В языке Clarion все составные структуры (и структуры данных, и операторные - исполняемые структуры) должны заканчиваться оператором END или точкой (.). Из этого следует, что приведенный далее текст функционально равнозначен предыдущему примеру:

```
PROGRAM
MAP.
MyWin    WINDOW('Hello Windows'),SYSTEM.
CODE
OPEN(MyWin)
ACCEPT.
```

Будучи эквивалентным функционально, этот текст стал гораздо хуже читаться, поскольку

к структурам MAP, WINDOW и ACCEPT что-то такое добавилось. По принятым соглашениям мы используем оператор END для завершения составных структур, занимающих несколько строк текста, помещая его в той же колонке, что и ключевое слово, открывающее структуру. Тогда как все содержимое структуры записывается с отступом во внутрь. И только для завершения структур, записываемых в одну строчку, таких как IF с одним только THEN, мы используем точку. Это соглашение улучшает читаемость текста и облегчает поиск пропущенного для какой-либо структуры терминатора.

Оператор CODE обязателен и используется для того, чтобы идентифицировать начало секции исполняемых операторов. Данные (переменные, файлы, окна, отчеты и т.д.) должны быть объявлены в секции данных (предшествующей оператору CODE), а исполняемые операторы могут идти только после него. Поскольку в данной программе нет разбиения на процедуры или функции (мы сделаем это в следующей главе), в ней есть только секция глобальных данных, за которой следует три строки исполняемых операторов. Переменные, объявленные в секции глобальных данных, видимы и доступны в любом месте программы.

Оператор OPEN(MyWin) открывает окно, но не выводит его на экран. Окно появляется на экране только после выполнения оператора DISPLAY или ACCEPT. Это позволяет динамически изменять свойства всего окна или любого объекта в окне прежде чем оно появится на экране.

Оператор ACCEPT - это обработчик событий. Внутри него автоматически обрабатывается большинство событий (сообщений, полученных от Windows. Это события общего характера, (такие как перерисовка окна и т.п.) обрабатываемые библиотечными процедурами. Оператор ACCEPT передает дальше в вашу программу для обработки только те события, для которых действительно требуется нестандартная реакция. Это облегчает программирование, позволяя сосредоточиться на аспектах программы более высокого уровня.

Оператор ACCEPT имеет завершающий оператор END, который говорит о том, что это есть составная операторная структура. ACCEPT неявно открывает цикл, “проходит” по всем событиям, которые программист хочет обработать (но не в данном примере - скоро мы к этому вернемся), завершая обработку одного события при достижении оператора END и переходя к обработке следующего.

Цикл ACCEPT обязателен для каждого открытого в программе окна. Открытое окно прикрепляется к циклу ACCEPT, который следует за ним в тексте программы и будет обработчиком его событий.

В этой программе цикл ACCEPT внутри себя обрабатывает все действия, связанные с системным меню, помещенным в окно атрибутом SYSTEM. Когда закрывая окно, пользователь использует системное меню, оператор ACCEPT автоматически передает управление на оператор, следующий за оператором END. Поскольку в данном случае нет



явных операторов, которые нужно выполнить, то программа завершается. Когда выполнение любой программы на языке Clarion достигает последнего исполняемого оператора, происходит выполнение неявного оператора RETURN, который в данном случае возвращает пользователя к работе с операционной системой.

2. Щелкните на кнопке Run в среде Clarion.

Происходит компиляция, компоновка и выполнение программы. В строке заголовка окна выводится текст "Hello Windows", и затем вы с помощью системного меню должны закрыть окно.

## Окно Hello Windows с объектами в нем

---

Только что созданная вами программа - это самая маленькая программа, которую только можно написать в Clarion для Windows. Теперь, чтобы продемонстрировать добавление некоторых объектов и обработку генерируемых для них событий, немного расширим программу.

### Изменение исходного текста

1. Исправьте текст, чтобы он выглядел следующим образом:

```
PROGRAM
MAP
END
MyWin      WINDOW('Hello Windows'),AT(,100,100),SYSTEM
            STRING('Hello Windows'),AT(26,23),USE(?String1)
            BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
            END
CODE
OPEN(MyWin)
ACCEPT
    IF ACCEPTED() = ?Ok THEN BREAK.

END
```

Изменения состоят в том, что в структуру WINDOW добавлены объекты типа STRING и BUTTON. Объект типа STRING помещает в окно неизменяемый текст. А BUTTON - добавляет командную кнопку.

### **Замечание**

Если вы вручную внесли изменения в структуру WINDOW, вам следует знать, что в редакторе текстов тоже имеется встроенный форматер окон, точно такой же как и в Генераторе приложений. Чтобы его вызвать просто поставьте курсор в любом месте и нажмите ctrl+f . Существует единственное ограничение - недоступны инструменты Control Template и Dictionary Field (они характерны только для Генератора приложений).

Другим добавлением является оператор IF ACCEPTED() = ?Ok THEN BREAK. . Этим оператором мы определяем, когда пользователь нажал кнопку ОК, и прерываем цикл ACCEPT, завершая программу. Функция ACCEPTED возвращает номер поля (объекта в окне), для которого было сгенерировано событие EVENT:Accepted (EVENT:Accepted - это мнемоническое имя, содержащееся в файле \CW20\LIBSRC\EQUATES.CLW, который компилятор вставляет в каждую программу).

?Ok представляет собой мнемоническое имя соответствия объекта BUTTON, определенное его атрибутом USE (см. раздел Метки соответствия полей в Описании языка). Компилятор автоматически присваивает имя ?Ok номеру, который он назначил экранному объекту (использование мнемонических меток делает текст программы более читаемым). Когда функция ACCEPTED возвращает значение равное присвоенному компилятором кнопке ?Ok номеру, то выполняется оператор BREAK и заканчивается выполнение цикла ACCEPT.

## 2. ЩЕЛКНИТЕ по кнопке Run.

Происходит компиляция, компоновка и выполнение программы. В строке заголовка окна выводится текст “Hello Windows” и теперь этот текст выводится еще и посередине окна. Можно закрыть окно или посредством системного меню или нажав кнопку ОК .

### Типовая форма исходной программы

Могут быть и другие способы обработать события внутри цикла ACCEPT, сделав фактически то же самое. Мы рассмотрим самый общий путь так как такой стиль кодирования используется при генерации исходного текста Генератором программ по стандартному набору шаблонов.

## 1. Измените текст программы, чтобы он выглядел следующим образом:

```

PROGRAM
MAP
END
MyWin      WINDOW('Hello Windows'),AT(,100,100),SYSTEM
            STRING('Hello Windows'),AT(26,23),USE(?String1)
            BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
            END
CODE
OPEN(MyWin)
ACCEPT
    CASE FIELD()
    OF ?Ok
        CASE EVENT()
        OF EVENT:Accepted
            BREAK

```

```
END  
END  
END
```

В этом примере одна структура CASE вложена в другую. Структура CASE работает следующим образом: ищется точное совпадение результата выражения, указанного следом за ключевым словом CASE и результатом другого выражения, указанного следом за ключевым словом OFF (хотя в нашем случае показан только один оператор OFF, в структуре CASE их может быть сколько угодно).

В структуре CASE FIELD() определяется, к какому экранному объекту относится текущее событие. В том случае, когда функция FIELD возвращает значение, равное номеру поля для кнопки ОК (номеру, которому присвоено имя ?ОК), начинается выполнение структуры CASE EVENT().

Структура CASE EVENT() определяет, что за событие произошло. Если функция EVENT возвращает значение, равное EVENT:Accepted (операторы присвоения мнемонических имен событиям содержатся в файле \CW20\LIBSRC\EQUATES.CLW), то выполняется оператор BREAK.

#### **Замечание**

Вложение структуры CASE EVENT() внутрь структуры CASE FIELD() позволяет поместить все операторы, связанные с обработкой событий, относящихся к одному экранному объекту, в одно место. Нетрудно было бы вложить структуру CASE FIELD() внутрь CASE EVENT(), “вывернув наизнанку” программный код, но это разбило бы обработку событий, относящихся к одному объекту, на несколько частей, что в принципе может привести к проблемам при сопровождении программы.

#### **2. Щелкните на кнопке Run.**

Мы снова можем закрыть окно или с помощью системного меню или кнопки ОК как и в предыдущем случае, однако теперь текст программы структурирован наиболее общим образом.

#### **Программа Hello Windows, содержащая обработку событий**

Оператором ACCEPT в программу два типа событий: характерные для экранного объекта и независимые от экранного объекта.

События, характерные для какого-либо экранного объекта, когда пользователь совершает что-то такое, что может потребовать от программы выполнения особых действий, относящихся к одному из объектов окна. Например, когда пользователь, введя данные в поле, нажимает клавишу TAB, для этого поля генерируется характерное для него событие - EVENT:Accepted.

Независимые от экранных объектов события не относятся к какому-либо одному объекту, но тем не менее могут требовать от программы выполнения некоторых действий (например, закрыть окно, завершить программу или переключиться на другой исполняемый процесс).

Рассматривавшееся только что вложение двух структур CASE является наиболее общим способом обработки характерных для оконного объекта событий. А наиболее общим методом обработки независимых от объекта событий служит обычная структура CASE EVENT(), помещаемая непосредственно перед структурой CASE FIELD().

### **Изменение текста программы**

1. Исправьте программу следующим образом:

```
PROGRAM
MAP
END
MyWin      WINDOW('Hello Windows'),AT(,100,100),SYSTEM
            STRING('Hello Windows'),AT(26,23),USE(?String1)
            BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
            END
CODE
OPEN(MyWin)
ACCEPT
    CASE EVENT()
    OF EVENT:OpenWindow
        MESSAGE('Opened Window')
    OF EVENT:GainFocus
        MESSAGE('Gained Focus')
    END
    CASE FIELD()
    OF ?Ok
        CASE EVENT()
        OF EVENT:Accepted
            BREAK
        END
    END
END
END
```

Новая структура CASE EVENT() отлавливает два независимых от полей события: EVENT:OpenWindow и EVENT:GainFocus. Функция MESSAGE используется здесь только для того, чтобы показать во время выполнения программы, что за событие произошло. Вместо нее здесь можно добавить любой программный код, который нужно было бы выполнить когда действия пользователя приводят к возникновению этих событий.

Этот пример демонстрирует основную логику выполнения и структуру кода для всех программ, работающих с окном, которые на основании стандартного набора шаблонов создает для вас Генератор программ, - цикл ACCEPT, содержащий структуру CASE EVENT() для обработки всех независимых от объектов событий, за которой следует структура CASE FIELD() с вложенной в нее структурой CASE EVENT() для обработки событий, присущих какому-либо конкретному объекту в окне.

## 2. ЩЕЛКНИТЕ мышью по кнопке Run.

Отметим, что как только окно выводится на экран генерируется событие EVENT:OpenWindow, а EVENT:GainFocus - нет. Событие EVENT:GainFocus происходит когда вы, переключившись ранее на другое приложение Windows, переключаетесь (например, клавишей ALT+TAB) на программу Hello Windows.

## **Добавление процедуры**

Hello Windows - это очень простая программа. Большинство современных прикладных программ гораздо сложнее. Для них приходится использовать технологию структурного программирования. Это означает, что программа разбивается на функциональные части, каждая из которых выполняет единую логическую задачу. В языке Clarion эти функциональные части называются процедурами и функциями.

Сначала добавим в программу Hello Windows процедуру.

### **Изменение текста программы**

#### 1. Изменим программу следующим образом:

```

PROGRAM
MAP
Hello PROCEDURE
END
CODE
Hello

Hello PROCEDURE
MyWin      WINDOW('Hello Windows'),AT(.,100,100),SYSTEM
            STRING('Hello Windows'),AT(26,23),USE(?String1)
            BUTTON('OK'),AT(34,60),USE(?Ok),DEFAULT
            END
CODE
OPEN(MyWin)
ACCEPT
    CASE EVENT()
    OF EVENT:OpenWindow
        MESSAGE('Opened Window')
```

```
        OF EVENT:GainFocus
          MESSAGE('Gained Focus')
      END
      CASE FIELD()
      OF ?Ok
        CASE EVENT()
        OF EVENT:Accepted
          BREAK
        END
      END
    END
  END
```

Единственное изменение находится в начале программы. В структуре MAP теперь мы видим оператор Hello PROCEDURE, который представляет собой прототип процедуры Hello. Прототип - это объявление процедуры для компилятора, сообщаящее ему, как происходит в тексте программы обращение к этой процедуре. Данный прототип говорит, что этой процедуре не передается никаких параметров и она не возвращает никакого значения (поэтому, собственно, это процедура, а не функция). Все процедуры и функции, к которым в программе есть обращения, должны быть представлены прототипом в структуре MAP. Смотрите раздел Прототипы процедур и функций в Описании языка.

Ключевое слово CODE, идущее сразу за структурой MAP, завершает секцию глобальных данных и отмечает начало секции исполняемых операторов, которая содержит только оператор Hello - обращение к процедуре Hello. Обращение к процедурам всегда выглядят как в тексте программы как отдельные операторы.

Второй оператор Hello PROCEDURE завершает глобальную секцию исполняемых операторов, и отмечает начало определения текста процедуры Hello.

В процедуре также как и в программе есть секция объявления данных. Поэтому между ней и секцией исполняемых операторов тоже требуется ключевое слово CODE. Вот почему остальная часть программы не изменилась по сравнению с предыдущим примером. В секции данных процедуры объявляются локальные данные.

Основная разница между локальными и глобальными данными состоит в области видимости объявленных переменных. К объявленному в секции локальных данных элементу можно обращаться только в пределах процедуры или функции в которой он объявлен, тогда как к глобальному элементу данных можно обращаться в любом месте программы. Смотри раздел Объявление данных и выделение памяти в Описании языка где описываются все отличия локальных и глобальных переменных.

## 2. ЩЕЛКНИТЕ по кнопке Run.

Программа выполняется точно также как и прежде. Единственным различием является

то, что к процедуре Hello можно обращаться из любого места программы, даже из другой процедуры. Это означает, что во время выполнения программы к этой процедуре можно обращаться много раз, хотя текст ее записан только один раз.

## **Добавление функции**

Единственным различием между процедурой и функцией в языке Clarion является то, что процедура не возвращает никакого значения. Обращение к ней осуществляется только в виде отдельного оператора - процедуру нельзя использовать в выражении, списке параметров и операторе присвоения. Функция же всегда должна возвращать какое-либо значение и может использоваться в выражении, списке параметров и операторе присвоения. Если возвращаемое функцией значение не нужно, то можно обращаться к ней и отдельным оператором (как к процедуре), но это вызовет предупреждающее сообщение компилятора (если в прототипе функции не указан атрибут PROC).

Структурно процедуры и функции одинаковы - и у той и у другой есть раздел объявления локальных данных, за которым идет раздел исполняемых операторов, начинающийся ключевым словом CODE.

### **Изменение текста программы**

1. Изменим структуру MAP следующим образом:

```
MAP
Hello          PROCEDURE
EventString    FUNCTION(LONG PassedEvent),STRING
END
```

Добавился прототип для функции EventString. Она получает параметр типа LONG с именем PassedEvent, который не может опускаться при обращении к ней, а возвращает строку. Типы данных, передаваемых в процедуру или функцию, задаются в круглых скобках следом за ключевым словом FUNCTION или PROCEDURE (если передаются несколько параметров, то через запятую). Тип возвращаемого функцией значения задается следом за скобкой, закрывающей список параметров. И процедура, и функция может принимать параметры. Более подробно передача параметров обсуждается в разделе Списки параметров в прототипах в Описании языка.

2. Добавим определение функции EventString в конец файла - текста программы:

```
EventString FUNCTION(LONG PassedEvent)
ReturnString STRING(255)
CODE
CASE PassedEvent
OF EVENT:OpenWindow
    ReturnString = 'Opened Window'
```

```
OF EVENT:GainFocus
    ReturnString = 'Gained Focus'
ELSE
    ReturnString = 'Unknown Event: ' & PassedEvent
END
RETURN(ReturnString)
```

Метка EventString (напомним, она должна начинаться в первой колонке) оператора FUNCTION присваивает этой функции имя, а в списке параметров, следующем за ключевым словом FUNCTION, присваивается имя PassedEvent параметру типа LONG. Число параметров в операторе FUNCTION или PROCEDURE должно всегда соответствовать числу типов данных, перечисленных в прототипе этой функции или процедуры.

ReturnString представляет собой локальную переменную, объявленную как строка длиной 255 символов в стеке. Секция локальных данных функции заканчивается оператором CODE. Структура CASE PassedEvent должна быть вам уже знакома так как это та же самая структура CASE EVENT(), но условием в этой структуре вместо функции EVENT() служит PassedEvent. В этой структуре CASE для каждого передаваемого в эту функцию события возвращаемой строке просто присваивается соответствующее значение. Здесь нас интересует оператор RETURN(ReturnString). Для процедуры явный оператор RETURN не требуется, поскольку, когда нет больше исполняемых операторов, всегда выполняется неявный RETURN. Тогда как в функции всегда имеется явный оператор RETURN, определяющий возвращаемое значение. В данном случае оператором RETURN возвращается значение, присвоенное строке ReturnString в структуре CASE.

3. Отредактируйте структуру CASE EVENT() в процедуре Hello таким образом:

```
CASE EVENT()
  OF EVENT:OpenWindow
    MESSAGE(EventString(EVENT:OpenWindow))
  OF EVENT:GainFocus
    MESSAGE(EventString(EVENT:GainFocus))
END
```

Здесь мы изменили функцию MESSAGE таким образом, чтобы выводилась возвращенная функцией EventString строка. Передача номера события посредством мнемонического имени делает текст программы более читаемым.

4. ЩЕЛКНИТЕ по кнопке Run.

Программа выполняется точно также как и прежде.



## Первый шаг к реальному приложению - добавим меню

---

Hello Windows - отличная демонстрационная программа, но в ней нет многого, что должно быть в настоящей коммерческой программе. Поэтому сейчас мы ее расширим, включив некоторые действительно необходимые элементы, в частности меню.

### Изменение исходного текста

1. Изменим начало программы следующим образом:

```
PROGRAM
  MAP
Main      PROCEDURE
Hello     PROCEDURE
EventString FUNCTION(LONG PassedEvent),STRING
  END
  CODE
  Main
```

Мы добавили в структуру MAP прототип процедуры Main и заменим вызов процедуры Hello обращением к процедуре Main.

2. В конец файла добавим определение процедуры Main:

```
Main      PROCEDURE
AppFrame  APPLICATION('Hello Windows'),AT(,,280,200),SYSTEM,RESIZE,MAX
          MENUBAR
          MENU('&File'),USE(?File)
          ITEM('&Browse Phones'),USE(?FileBrowsePhones)
          ITEM,SEPARATOR
          ITEM('E&xit'),USE(?FileExit),STD(STD:Close)
          END
          ITEM('&About!'),USE(?About)
          END
          END
  CODE
  OPEN(AppFrame)
  ACCEPT
  CASE ACCEPTED()
  OF ?About
    Hello
  END
END
```

Процедуре Main не передается никаких параметров. Она содержит структуру AppFrame APPLICATION. Это ключевой элемент при создании программ с интерфейсом нескольких документов Windows (MDI). Приложение MDI может содержать в себе несколько исполняемых процессов. В данном случае мы имеем обязательное при создании MDI-программ порождающее MDI-окно.

В структуре MENUBAR определяются предоставляемые пользователю пункты на линейке меню. Структура MENU(' &File') создает обычное спускающееся меню File, которое имеется в большинстве программ для Windows. Амперсанд (&) перед буквой "F" указывает, что она является клавишей ускоренного доступа к этому меню и во время работы программы подчеркивается операционной системой.

Оператор ITEM(' &Browse Phones') создает пункт меню, который мы будем использовать для вызова процедуры (скоро мы к этому перейдем). Оператор ITEM,SEPARATOR создает в меню после пункта Browse Phones разделительную линию.

Оператор ITEM('E&xit') создает пункт меню для выхода из процедуры (и из программы, поскольку эта процедура единственная, к которой есть обращение из глобальной секции исполняемых операторов). Атрибут STD(STD:Close) задает стандартные действия по закрытию окна, не выходящие за рамки внутреннего обработчика событий. Вот почему в цикле после обработчика АСЦЕРТ нет никаких операторов, относящихся к этому пункту меню - за вас эти действия автоматически выполняют процедуры из библиотеки исполняющей системы.

Оператор ITEM(' &About!') создает на линейке меню справа от меню File еще один элемент. Восклицательный знак в конце представляет программное соглашение, позволяющее дать конечному пользователю визуальный признак, что данный пункт на линейке меню выполняет какое-то действие, а не раскрывает спускающееся меню.

В цикле АСЦЕРТ содержится только структура CASE ACCEPTED(). Функция ACCEPTED возвращает номер поля (пункта меню), которое имело фокус ввода в момент возникновения события EVENT:Accepted . Поскольку для элементов меню может генерироваться только это событие, можно вместо сочетания функций FIELD и EVENT использовать функцию ACCEPTED. В ветви OF ?About просто вызывается процедура Hello.

## 2. Щелкните по кнопке Run.

Программа выполняется и работают только пункты About! и File д Exit. Отметим, что пункт File д Exit заканчивает работу программы, хотя мы не писали для этого ни одного оператора.

## Еще шаг в реальный мир - добавим чтение файла и форму для заполнения полей

---

Наше меню - отличная штука, но настало время сделать следующие шаги к реальной прикладной программе. Теперь добавим файл данных и процедуры для работы с ним.

### Изменение исходного текста

1. Изменим начало программы следующим образом:

```
PROGRAM
    INCLUDE('Keycodes.CLW')
    INCLUDE('Errors.CLW')
    MAP
Main                                PROCEDURE
BrowsePhonesPROCEDURE
UpdatePhones    FUNCTION(LONG Action),LONG
Hello          PROCEDURE
EventString    FUNCTION(LONG PassedEvent),STRING
    END
Phones        FILE,DRIVER('TopSpeed'),CREATE
NameKey       KEY(Name),DUP,NOCASE
Rec           RECORD
Name          STRING(20)
Number        STRING(20)
    END
    END

InsertRecord    EQUATE(1)
ChangeRecord    EQUATE(2)
DeleteRecord    EQUATE(3)
ActionComplete  EQUATE(1)
ActionAborted   EQUATE(2)
    CODE
    Main
```

Двумя новыми операторами INCLUDE добавляются операторы EQUATE - присвоение мнемонических имен кодам клавиш и кодам ошибочных ситуаций. Использование вместо кодов их символических имен делает программу более читаемой и, как следствие, легче сопровождаемой.

Структура MAP увеличилась еще на два прототипа BrowsePhones PROCEDURE и UpdatePhones FUNCTION(LONG Action),LONG. Процедура BrowsePhones выводит список записей в файле, а функция UpdatePhones производит изменение отдельной записи.

В целях упрощения кодирования (позднее вы увидите почему) процедура `BrowsePhones` просто будет выводить все записи файла в экранном поле типа `LIST`. Это не совсем то, что делает процедура `Browse` в генерируемом по шаблонам тексте, где используется “постраничная” загрузка данных, чтобы обеспечить работу с очень большими файлами. Но в данной программе мы пока сделаем так.

Также для упрощения кодирования `UpdatePhones` представляет собой функцию, которой передается параметр. Параметр типа `LONG` обозначает действие, которое следует предпринять: добавить, удалить или изменить запись. Возвращаемое значение сообщает вызвавшей процедуре выполнил или отменил действия пользователь. И снова это не одно и то же, что процедура типа `Form` в сгенерированном по шаблону тексте (в которой для обозначения действия с записью файла и результата действия используются глобальные переменные), но работает также.

Объявлением `Phones FILE` создается простой файл данных, для доступа к которому используется файловый драйвер `TopSpeed`. В записи два поля данных: `Name` и `Number`, каждое из которых длиной 20 символов. Такое объявление поля `Number` позволяет хранить в нем телефонный номер для любой страны мира (об этом чуть позднее).

Пять операторов `EQUATE` определяют пять констант и служат для улучшения читаемости программы. `InsertRecord`, `ChangeRecord`, и `DeleteRecord` определяют действия с файлом и передаются в функцию `UpdatePhones`. `ActionComplete` и `ActionAborted` определяют два возможных значения, возвращаемых этой функцией.

2. Отредактируем структуру `CASE ACCEPTED()` в процедуре `Main` следующим образом:

```
CASE ACCEPTED()
  OF ?FileBrowsePhones
    START(BrowsePhones,25000)
  OF ?About
    Hello
  END
```

Оператор `START(BrowsePhones,25000)` выполняется, когда пользователь выбирает пункт меню `Browse Phones`. Функция `START` создает для процедуры `BrowsePhones` новый исполняемый процесс, а второй параметр (25000) задает размер (в байтах) стека для нового процесса. Когда вы в окне `APPLICATION` вызываете процедуру, содержащую порожденное MDI-окно, то вы должны использовать функцию `START`, потому что каждое MDI-окно должно находиться в исполняемом процессе, отдельном от процесса, в котором раскрыто окно `APPLICATION`. Если же вы не иницилируете отдельный исполняемый процесс, то при попытке вызвать процедуру `BrowsePhones` вы получите сообщение “Unable to open MDI window on APPLICATION’s thread” и выполнение программы немедленно прекратится.

Если же новый процесс запущен, то в порожденном MDI-окне можно раскрывать следующие окна не начиная нового процесса. Это означает, что несмотря на то, процедуры BrowsePhones и UpdatePhones обе содержат MDI-окна, только в BrowsePhones должен инициироваться новый процесс, поскольку она вызывается в окне APPLICATION. Сама же она может обращаться к процедуре UpdatePhones, не создавая новый процесс.

### Добавим процедуру BrowsePhones

1. Добавим в конец файла раздел объявления данных процедуры BrowsePhones:

```

BrowsePhonesPROCEDURE
PhonesQue      QUEUE
Name           STRING(20)
Number         STRING(20)
Position       STRING(512)
               END
window        WINDOW('Browse Phones'),AT(.,185,156),SYSTEM,GRAY,RESIZE,MDI
               LIST,AT(6,8,173,100),ALRT(MouseLeft2),USE(?List)|
,FORMAT('84L|M~Name~80L~Number~'),FROM(PhonesQue),VSCROLL
               BUTTON('&Insert'),AT(20,117),KEY(InsertKey),USE(?Insert)
               BUTTON('&Change'),AT(76,117,35,14),KEY(EnterKey),USE(?Change)
               BUTTON('&Delete'),AT(131,117,35,14),KEY(DeleteKey),USE(?Delete)
               BUTTON('Close'),AT(76,137,35,14),KEY(EscKey),USE(?Close)
               END

```

Структура PhonesQue QUEUE объявляет структуру данных, которая будет содержать все записи файла Phones, для того, чтобы высвечивать их в оконном объекте типа LIST. Структура QUEUE (очередь) в Clarion похожа на файл данных, так как имеет буфер и может содержать неопределенное количество записей, однако она существует только во время выполнения в оперативной памяти. Очередь можно уподобить динамически создаваемым массивам в других языках. Более подробно об очередях смотри в главе Очереди в памяти в Описании языка.

Очередь PhonesQue содержит три поля. Name и Number повторяют поля файла Phones и будут выводиться в двух колонках, определенных атрибутом FORMAT структуры LIST. Поле Position будет содержать значение, возвращаемое функцией POSITION для этой записи в файле Phones. Запомнив в очереди положение записи в файле, мы сможем потом повторно прочитать ее перед обращением к UpdatePhones для корректировки данных.

Структура window WINDOW содержит один объект типа LIST четыре кнопки. Поле типа LIST это ключевой элемент в этой процедуре. Атрибут ALRT(MouseLeft2) отменяет автоматическую обработку двойного щелчка кнопкой мыши, поэтому обработчик событий

АССЕРТ передает событие EVENT:AlertKey для обработки в программу. Это позволит написать программный код для вызова процедуры UpdatePhones и изменения записи, по которой пользователь дважды щелкнул.

Вертикальная черта (|) в конце оператора LIST является в языке Clarion символом продолжения, означающим, что объявление структуры LIST продолжается на следующей строке атрибутом FORMAT('84L|M~Name~80L~Number~') (при желании можно разместить их на одной строке). Параметры атрибута FORMAT определяют внешний вид поля LIST во время выполнения. Поскольку строка очень быстро становится сложной, лучше всего предоставить ее редактирование Форматеру полей LIST в Форматере окон. Данная строка формата определяет две колонки. Первая шириной 84 условных единиц, выравнивание на левый край (L) с рамкой (|) изменяемых размеров (M) и заголовком колонки служит слово "Name" (~Name~). Вторая колонка шириной в 80 единиц (80) с выравниванием на левый край и заголовком "Number" (~Number~).

Атрибут FROM(PhonesQue) в структуре LIST задает очередь-источник данных, которые должны высвечиваться, а VSCROLL добавляет вертикальную линейку скроллинга. В поле LIST будут высвечиваться поля Name и Number всех элементов очереди PhonesQue, игнорируя при этом поле Position, поскольку в атрибуте FORMAT заданы только две колонки. Объект LIST автоматически управляет скроллингом записей файла без какого-либо программирования с нашей стороны. Это происходит потому, что в структуре LIST отсутствует атрибут IMM. Если бы этот атрибут был указан, то мы должны бы были написать ручную управление скроллингом (как это делается в шаблоне процедуры Browse).

Оператор BUTTON('&Insert') определяет командную кнопку, которую будет нажимать пользователь для того, чтобы добавить новую запись. Атрибут KEY(InsertKey) указывает, что данная кнопка автоматически нажимается, при нажатии пользователем клавиши INSERT на клавиатуре. Отметим, у остальных трех кнопок есть подобные атрибуты KEY. Это означает, что вам не нужно писать специальный программный код для управления программой с клавиатуры, а не мышью.

2. Добавим начало секции исполняемых операторов процедуры BrowsePhones в конец нашего исходного файла.

```
CODE
DO OpenFile
DO FillQue
OPEN(window)
ACCEPT
    CASE EVENT()
    OF EVENT:OpenWindow
        DO QueRecordsCheck
    OF EVENT:AlertKey
```

```
IF KEYCODE() = MouseLeft2
    POST(EVENT:Accepted,?Change)
END
END
```

Секция исполняемых операторов начинается оператором DO OpenFile. Оператор DO выполняет подпрограмму, в данном случае подпрограмму OpenFile и, когда операторы подпрограммы выполнены, управление возвращается на оператор, следующий за этим оператором DO. Текст самой процедуры должен находиться в конце процедуры, после операторов, реализующих основную логику работы, поскольку сам по себе оператор ROUTINE завершает секцию исполняемых операторов процедуры или функции.

Для использования подпрограмм внутри процедуры или функции есть две причины: для того, чтобы один раз написать блок операторов, который нужно выполнять в нескольких логических точках процедуры, или подстановкой одного DO вместо целого ряда операторов, выполняющих единую задачу, сделать более ясной логику работы.

В данном случае оператор DO OpenFile выполняет вторую задачу, перемещая программный код, открывающий или создающий файл данных за пределы логики главной процедуры. Далее оператор DO FillQueue выполняет операторы, заносящие в очередь PhonesQueue все записи файла данных. Благодаря этим двум операторам DO очень легко проследить логику выполнения данной процедуры.

В ветви OF EVENT:OpenWindow структуры CASE EVENT() для вызова процедуры, проверяющей наличие записей в очереди PhonesQueue, выполняется оператор DO QueueRecordsCheck.

Далее, для обнаружения двойного щелчка мыши, в ветви OF EVENT:AlertKey содержится конструкция IF KEYCODE() = MouseLeft2. Так как атрибут ALRT(MouseLeft2) имеется только у окна списка LIST, мы уверены, что оператор POST(EVENT:Accepted,?Change) выполнится только когда пользователь дважды щелкнет мышью в окне списка.

Оператор POST сообщает обработчику ACCEPT, чтобы он сгенерировал событие, указанное первым параметром (EVENT:Accepted), для объекта, указанного вторым параметром (?Change). Цель оператора POST(EVENT:Accepted,?Change) - вызвать выполнение операторов, выполняющих обработку события EVENT:Accepted для кнопки Change, как если бы пользователь нажал эту кнопку мышкой или сочетанием клавиш.

Этим иллюстрируется очень хорошая практика: написать программный код один раз и обращаться к нему из разных мест. Это концепция структурного программирования, которая породила процедуры, функции и подпрограммы. Даже не смотря на то, что программный код для обработки события EVENT:Accepted для кнопки Change не выделен в отдельную процедуру, функцию или подпрограмму, использование оператора POST таким

образом означает, что сопровождать вам нужно будет только одну часть программы - если потребуется изменить логику корректировки записи, вам нужно будет менять только в одном месте.

3. Добавим в конец файла остальную часть программной секции процедуры BrowsePhones:

```
CASE FIELD()
OF ?Close
  CASE EVENT()
  OF EVENT:Accepted
    POST(EVENT:CloseWindow)
  END
OF ?Insert
  CASE EVENT()
  OF EVENT:Accepted
    IF UpdatePhones(InsertRecord) = ActionComplete
      DO AssignToQue
      ADD(PhonesQue)
      IF ERRORCODE() THEN STOP(ERROR()).
      SORT(PhonesQue,PhonesQue.Name)
      ENABLE(?Change,?Delete)
    END
    SELECT(?List)
  END
OF ?Change
  CASE EVENT()
  OF EVENT:Accepted
    DO GetRecord
    IF UpdatePhones(ChangeRecord) = ActionComplete
      DO AssignToQue
      PUT(PhonesQue)
      IF ERRORCODE() THEN STOP(ERROR()).
      SORT(PhonesQue,PhonesQue.Name)
    END
    SELECT(?List)
  END
OF ?Delete
  CASE EVENT()
  OF EVENT:Accepted
    DO GetRecord
    IF UpdatePhones(DeleteRecord) = ActionComplete
      DELETE(PhonesQue)
```



```
        IF ERRORCODE() THEN STOP(ERROR()).
        DO QueRecordsCheck
        SORT(PhonesQue,PhonesQue.Name)
    END
    SELECT(?List)
END
END
END
FREE(PhonesQue)
CLOSE(Phones)
```

Следом за CASE EVENT() идет структура CASE FIELD(). Отметим, что каждая ветвь OF, содержит свою собственную структуру CASE EVENT() и в каждой из них проверяется только ветвь OF EVENT:Accepted. Из-за этого можно было заменить структуру CASE FIELD() структурой CASE ACCEPTED(), исключив вложение структур CASE. Реально это дало бы небольшое увеличение производительности - весьма небольшое, чтобы заметить его визуально. Причина почему мы этого не сделали - стремление к большей логичности. Часто будет возникать необходимость отслеживать не только событие EVENT:Accepted, но и другие характерные для полей события. И в этих случаях логично использовать вложенные структуры CASE, поэтому хорошим тоном будет так поступить и теперь. Кроме того, этот пример в точности повторяет структуру кода, создаваемого Генератором программ Clarion на основе стандартных шаблонов.

Когда пользователь нажимает кнопку Close, в ветви OF ?Close выполняется оператор POST(EVENT:CloseWindow). Поскольку в нем нет второго параметра, указывающего оконный объект, событие генерируется для данного окна (и должно быть независимым от поля событием). Событие EVENT:CloseWindow приводит к тому, что цикл ACCEPT заканчивается и управление передается на оператор, следующий за закрывающим цикл оператором END. В данном случае управление передается на оператор FREE(PhonesQue), который освобождает всю память, занимаемую элементами очереди (обычный способ закрыть очередь). Оператор CLOSE(Phones) закрывает файл данных). Поскольку после него нет больше операторов, выполняется неявный RETURN и происходит возврат в процедуру Main (откуда было обращение к данной процедуре).

Ветвь OF ?Insert содержит конструкцию IF UpdatePhones(InsertRecord) = ActionComplete. Здесь вызывается функция UpdatePhones, ей передается константа InsertRecord, которая определена в глобальной секции данных, и затем проверяется возвращаемое значение ActionComplete.

Оператор DO AssignToQue выполняется только когда пользователь действительно добавляет запись. AssignToQue это локальная подпрограмма, которая переносит данные из буфера файла Phones в буфер очереди PhonesQue. Затем оператор ADD(PhonesQue) добавляет новый элемент в очередь PhonesQue. Оператор IF ERRORCODE() THEN

STOP(ERROR()). - это стандартная проверка ошибочной ситуации, которая должна выполняться после любого действия с файлом или очередью, которое потенциально может закончиться ошибкой (еще одна хорошая привычка, которую следует выработать).

Оператор SORT(PhonesQue,PhonesQue.Name) сортирует записи в очереди PhonesQue по полю Name в алфавитном порядке. Поскольку в структуре PhonesQue QUEUE нет атрибута PRE, можно обращаться к ее полям, используя синтаксис уточнения имен Clarion, присоединяя спереди к имени поля (Name) имя содержащей его структуры ((PhonesQue) через точку (PhonesQue.Name). Более подробно смотри раздел Синтаксис уточнения имен в Описании языка.

Оператор ENABLE(?Change,?Delete) делает активными кнопки Change и Delete (если это первая запись в очереди, то эти кнопки были “затемнены” подпрограммой QueRecordsCheck). Оператор SELECT(?List) возвращает пользователя в окно списка LIST.

Программный код в ветви OF ?Change почти идентичен коду в ветви OF ?Insert. Здесь есть дополнительный оператор DO GetRecord, который выполняет подпрограмму, которая считывает из файла данных запись, соответствующую выделенному в очереди элементу. Единственным отличием является оператор PUT(PhonesQue), который помещает сделанные пользователем изменения в очередь PhonesQue.

В ветви OF ?Delete код почти идентичен коду в OF ?Change. Разница в операторе DELETE(PhonesQue), который удаляет запись из очереди PhonesQue и в обращении к подпрограмме QueRecordsCheck, чтобы проверить не удалил ли пользователь последнюю запись.

4. Добавим в конец файла текст подпрограммы вызываемой в процедуре BrowsePhones:

```
AssignToQue ROUTINE
    PhonesQue.Name = Phones.Rec.Name
    PhonesQue.Number = Phones.Rec.Number
    PhonesQue.Position = POSITION(Phones)
```

```
QueRecordsCheck ROUTINE
    IF NOT RECORDS(PhonesQue)
        DISABLE(?Change,?Delete)
        SELECT(?Insert)
    ELSE
        SELECT(?List,1)
    END
```

```
GetRecord ROUTINE
```

```
GET(PhonesQue,CHOICE(?List))
IF ERRORCODE() THEN STOP(ERROR()).
REGET(Phones,PhonesQue.Position)
IF ERRORCODE() THEN STOP(ERROR()).
```

```
OpenFile  ROUTINE
  OPEN(Phones,42h)
  CASE ERRORCODE()
  OF NoError
  OROF IsOpenErr
    EXIT
  OF NoFileErr
    CREATE(Phones)
    IF ERRORCODE() THEN STOP(ERROR()).
    OPEN(Phones,42h)
    IF ERRORCODE() THEN STOP(ERROR()).
  ELSE
    STOP(ERROR())
  RETURN
END
```

```
FillQue   ROUTINE
  SET(Phones.NameKey)
  LOOP
    NEXT(Phones)
    IF ERRORCODE() THEN BREAK.
    DO AssignToQue
    ADD(PhonesQue)
    IF ERRORCODE() THEN STOP(ERROR()).
  END
```

Как видно, в этой процедуре пять подпрограмм. Отметим, что хотя подпрограмма похожа на процедуру, в ней нет оператора CODE. Это потому, что подпрограмма использует локальные данные процедуры и в ней не может быть своей собственной секции данных. Подпрограмма состоит только из исполняемых операторов.

Подпрограмма AssignToQue выполняет три оператора присвоения. Оператор `PhonesQue.Name = Phones.Rec.Name` копирует данные из поля Name буфера файла Phones в поле Name буфера очереди PhonesQue. Поскольку в структуре Phones FILE нет атрибута PRE, можно обращаться к ее полям, используя синтаксис уточнения имен Clarion, присоединяя спереди к имени поля (Name) имя содержащего его буфера (Rec) и имя структуры FILE (Phones) через точку (Phones.Rec.Name). Более подробно смотри раздел Синтаксис уточнения имен в Описании языка.

Оператор `PhonesQue.Number = Phones.Rec.Number` копирует данные из поля `Number` буфера файла `Phones` в поле `Number` буфера очереди `PhonesQue`. Оператор `PhonesQue.Position = POSITION(Phones)` заносит в поле `PhonesQue.Position` возвращаемое функцией `POSITION` значение. Это значение позволяет прочитать с диска одну конкретную запись, которая в момент выполнения функции `POSITION` находилась в буфере записи. Функция `POSITION` работает для каждого файлового драйвера Clarion, поэтому это рекомендуемый метод для указания конкретной записи в любой файловой системе.

Подпрограмма `QueRecordsCheck` проверяет наличие записей в очереди `PhonesQue`. В конструкции `IF NOT RECORDS(PhonesQue)` к возвращаемому функцией `RECORDS` применяется оператор логического отрицания `NOT`. Если функция `RECORDS(PhonesQue)` возвращает нуль, то `NOT` делает условие истинным и выполнится следующий за `IF` программный код (нуль всегда есть ЛОЖЬ и `NOT` делает его ИСТИНОЙ). Если функция `RECORDS(PhonesQue)` возвращает отличное от нуля значение, то выполнится код, следующий за ключевым словом `ELSE` (любое отличное от нуля значение всегда есть ИСТИНА и `NOT` делает его ЛОЖЬЮ). Поэтому, если очередь `PhonesQue` пуста, то для того, чтобы “затенить” кнопки `Change` и `Delete` выполняется оператор `DISABLE(?Change,?Delete)`, затем оператором `SELECT(?Insert)` внимание пользователя перемещается на кнопку `Insert` (следующее логичное действие). Если же в очереди есть записи, то оператором `SELECT(?List)` внимание пользователя переключается на окно списка `LIST`.

Подпрограмма `GetRecord` приводит в соответствие содержимое буфера записи файла `Phones` и буфера очереди `PhonesQue` для выбранной пользователем записи в окне списка. Оператор `GET(PhonesQue,CHOICE(?List))` использует функцию `CHOICE` для того чтобы “указать” на выделенный в данный момент элемент в структуре `LIST` и прочитать соответствующий элемент в буфер очереди (конечно, проверяя возможные ошибки). Затем, для повторного чтения записи из файла в буфер оператором `REGET(Phones,PhonesQue.Position)` используется запомненная информация о положении ее в файле.

Подпрограммой `OpenFile` или открывается или создается файл `Phones`. Оператор `OPEN(Phones,42h)` пытается открыть файл `Phones` для совместного использования. Второй параметр (`42h`) представляет собой шестнадцатеричное число (на что указывает буква `h` в конце). Язык Clarion поддерживает десятичную, шестнадцатеричную, двоичную и восьмеричную системы счисления. Это число представляет режим доступа к файлу “Чтение/Запись, Нет Запрета” (подробнее о режимах доступа к файлу см. описание оператора `OPEN` в Описании языка). Мы запрашиваем совместный доступ к файлу потому, что это MDI-программа и пользователь может запустить несколько копий этой процедуры в одной программе. Тем не менее в данной программе не производится проверка совместного использования обязательная для реального многопользовательского приложения. О проблемах при разработке многопользовательского приложения смотрите раздел Вопросы

совместного использования в Описании языка.

В структуре CASE ERRORCODE() проверяется наличие каких-либо ошибок при выполнении оператора OPEN. В ветви OF NoError OROF IsOpenErr (теперь видно зачем мы включили в программу файл ERRORS.CLW) для немедленного выхода из подпрограммы выполняется оператор EXIT. Очень важно не путать оператор EXIT с оператором RETURN, поскольку RETURN прекращает выполнение процедуры или функции, тогда как EXIT осуществляет выход только из локальной подпрограммы. Можно использовать в подпрограмме и ROUTINE, но если вы уверены, что хотите завершить процедуру или функцию, а не просто выйти из подпрограммы.

В ветви OF NoFileErr определяется, что открываемого файла нет. Затем оператор CREATE(Phones) создает новый пустой файл. Если вы собираетесь использовать оператор CREATE, то убедитесь, что в объявлении файла имеется атрибут CREATE, в противном случае оператор CREATE не сможет создать файл. Оператор CREATE не открывает созданный файл, этим и объясняется второй оператор OPEN(Phones,42h). Если при открытии файла возникла какая-то другая ошибка, то выполняется программный код в ветви ELSE. Оператор STOP(ERROR()) высвечивает пользователю в системном модальном окне сообщение, полученное от функции ERROR, предоставляя ему возможность или прервать программу (возврат в Windows), или проигнорировать ошибку. Затем оператор RETURN завершает выполнение процедуры, если пользователь выбрал игнорировать ошибку.

Локальная подпрограмма FillQue заносит в очередь PhonesQue все записи файла Phones. Оператор SET(Phones.NameKey) устанавливает последовательность и начальную точку обработки файла Phones. Параметр Phone.NameKey устанавливает последовательность обработки на основе значений поля Name. Пропуск второго параметра означает, что обработка начинается с начала (или конца) файла. Цикл LOOP не содержит условий, а это означает, что где-то в теле цикла мы должны поставить оператор BREAK, или, в противном случае, получим бесконечный цикл. Оператор NEXT(Phones) считывает следующую запись из файла, затем оператор IF ERRORCODE() THEN BREAK. обеспечивает выход из цикла когда все записи будут прочитаны. DO AssignToQue передает управление подпрограмме AssignToQue, которую мы уже рассматривали, а ADD(PhonesQue) добавляет новую запись в очередь.

### Добавим функцию UpdatePhones

1. Добавим в конец файла секцию данных функции UpdatePhones:

```
UpdatePhones    FUNCTION(LONG Action)
ReturnValue     LONG,AUTO
window WINDOW('Update Phone'),AT(.,185,92),SYSTEM,GRAY,RESIZE,MDI,MASK
                PROMPT('N&ame:'),AT(14,14),USE(?Prompt1)
                ENTRY(@s20),AT(68,13),USE(Phones.Rec.Name),REQ
```

```
PROMPT('N&umber:'),AT(14,43),USE(?Prompt2)
ENTRY(@s20),AT(68,42),USE(Phones.Rec.Number)
BUTTON('OK'),AT(45,74),USE(?Ok),REQ,DEFAULT
BUTTON('Cancel'),AT(109,74,32,14),USE(?Cancel)

END
```

Оператор UpdatePhones FUNCTION(LONG Action) объявляет функцию, которая принимает единственный параметр типа LONG, который будет называться в процедуре “Action” (не имеет значения, какая переменная или константа передается в функцию).

Оператор ReturnValue LONG,AUTO объявляет переменную, которая из-за атрибута AUTO остается неинициализированной компилятором. По умолчанию, в языке Clarion переменные инициализируются пробелами или нулем ( в зависимости от типа данных). Указание атрибута AUTO экономит немного памяти, однако следует иметь ввиду, что нужно быть уверенным в том, что до какой-либо проверки содержимого такой переменной, ей присвоено какой-нибудь значение, в противном случае это может привести к перемежающейся ошибки, которую действительно трудно отловить.

Структура WINDOW имеет атрибут MASK, который означает, что в отличие от стандартного для Windows “свободного” ввода данных, введенные данные проверяются на соответствие заданному шаблону.

Сочетание двух объектов PROMPT и ENTRY служат для ввода данных пользователем. Две командные кнопки позволяют ему завершить или прервать действие с файлом.

Поле типа PROMPT определяет текст подсказки на экране и клавишу ускоренного перехода к следующему за ним полю типа ENTRY. Клавиша ускоренного перехода формируется с помощью клавиши ALT плюс буква в подсказке, перед которой стоит амперсанд. Например, ('N&ame:') означает, что сочетание ALT+A переключит фокус ввода на поле ENTRY(@s20),AT(68,13),USE(Phones.Rec.Name).

Атрибут USE поля ENTRY указывает переменную, в которую во время выполнения автоматически заносятся введенные пользователем данные. Библиотечная процедура следит за тем, чтобы при переключении фокуса в поле выводилось текущее значение переменной. Когда пользователь ввел данные и переключился на другое поле, библиотечная процедура переносит высвечиваемое в поле значение в переменную, указанную атрибутом USE.

Атрибут REQ у первого поля типа ENTRY означает, что пользователь не может переключиться из этого поля на другое, оставив его пустым, тогда как атрибут REQ у кнопки ОК требует, чтобы была выполнена проверка заполнения всех полей, у которых имеется атрибут REQ. Необходимая проверка заполнения полей выполняется только тогда, когда пользователь нажимает кнопку, имеющую атрибут REQ.

2. Добавим в конец файла операторы, реализующие основную логику функции UpdatePhones:

```
CODE
OPEN(window)
DO SetupScreen
ACCEPT
    CASE FIELD()
    OF ?Phones:Rec:Number
        CASE EVENT()
        OF EVENT:Selected
            DO SetupInsert
        END
    OF ?Ok
        CASE EVENT()
        OF EVENT:Accepted
            EXECUTE Action
                ADD(Phones)
                PUT(Phones)
                DELETE(Phones)
            END
            IF ERRORCODE() THEN STOP(ERROR()).
            ReturnValue = ActionComplete
            POST(EVENT:CloseWindow)
        END
    OF ?Cancel
        CASE EVENT()
        OF EVENT:Accepted
            ReturnValue = ActionAborted
            POST(EVENT:CloseWindow)
        END
    END
END
RETURN(ReturnValue)
```

Оператор DO SetupScreen вызывает для выполнения некоторых действий по инициализации окна подпрограмму SetupScreen. Заметим, он следует за оператором OPEN(Window). Когда вы собираетесь в процедуре динамически изменить окно, его нужно сначала открыть.

Ветвь OF ?Phones:Rec:Number в структуре CASE FIELD() демонстрирует две важные вещи. Во-первых, саму по себе мнемоническую метку соответствия. Атрибут

USE(Phones.Rec.Number) содержит точки в имени поля, а они в языке Clarion в метках не допускаются. Поэтому при создании мнемонической метки для Phones.Rec.Number компилятор вместо точек подставляет двоеточия (так как двоеточия в именах допустимы).

Во-вторых, еще одним важным моментом является ветвь OF EVENT:Selected в структуре CASE EVENT(). Событие EVENT:Selected генерируется когда на экранный объект переключается фокус, но перед тем как пользователь ввел в него данные. Чтобы предложить пользователю возможность установить формат вывода и ввода данных в это поле, выполняется оператор DO SetupInsert.

Программный код в ветви OF EVENT:Accepted внутри ветви OF ?Ok это тот код, который реально записывает данные на диск. Структура EXECUTE Action выполняет один из операторов ADD(Phones), PUT(Phones), или DELETE(Phones).

Структура EXECUTE похожа на IF и CASE в том, что она выполняет операторы основываясь на результате вычисления условного выражения. Условие в структуре EXECUTE должно давать в результате целое число в интервале от 1 до n (где n это число операторов в структуре). После вычисления условия выполняется оператор, порядковый номер которого совпадает с результатом выражения.

В данном примере в структуре EXECUTE проверяется параметр Action и затем выполняется оператор ADD(Phones), если значение Action равно единице (1), PUT(Phones), если Action равно двум (2), или DELETE(Phones), если Action равно трем (3).

В общем, когда вы выбираете какую из структур языка Clarion использовать для организации условного выполнения (IF/ELSIF, CASE, или EXECUTE), учитывайте, что IF/ELSIF наиболее гибкая и наименее эффективная из трех, а EXECUTE жесткая и очень эффективная. Поэтому, когда выражения дает целочисленный результат от 1 до n, старайтесь использовать EXECUTE, в противном случае используйте CASE. Если CASE недостаточно гибка, то остается IF/ELSIF.

Независимо от того, какой оператор выполняется в EXECUTE оператор, IF ERRORCODE() THEN STOP(ERROR()). проверит наличие ошибки. Оператор ReturnValue = ActionComplete устанавливает возвращаемое в вызвавшую процедуру значение, сигнализирующее о выполнении пользователем действия с файлом, затем POST(EVENT:CloseWindow) закрывает цикл ACCEPT, передавая управление на оператор RETURN(ReturnValue).

Программный код в ветви OF ?Cancel делает почти тоже самое, но не записывая ничего на диск. Оператор присвоения ReturnValue = ActionAborted устанавливает возвращаемое в вызвавшую процедуру значение, сигнализирующее о том, что пользователь прервал



операцию с файлом. затем POST(EVENT:CloseWindow) закрывает цикл ACCEPT, передавая управление на оператор RETURN(ReturnValue).

3. Добавим в конец файла определение подпрограммы, вызываемой в функции UpdatePhones.

```

SetupScreen ROUTINE
  CASE Action
    OF InsertRecord
      CLEAR(Phones.Rec)
      TARGET{PROP:Text} = 'Adding New Number'
    OF ChangeRecord
      TARGET{PROP:Text} = 'Updating ' & CLIP(Phones.Rec.Name) |
        "s Phone Number" &
      IF Phones.Rec.Number[1] <> '+'
        ?Phones.Rec.Number{PROP:Text} = '@P###-###-####P'
      END
    OF DeleteRecord
      TARGET{PROP:Text} = 'Deleting ' & CLIP(Phones.Rec.Name) |
        Phone Number' & "s
      DISABLE(FIRSTFIELD(),LASTFIELD())
      ENABLE(?Ok,?Cancel)
    END

SetupInsert ROUTINE
  IF Action = InsertRecord
    CASE MESSAGE('International?', 'Format', ICON:Question, |
      BUTTON:Yes+BUTTON:No,BUTTON:No,1)
    OF BUTTON:Yes
      TARGET{PROP:Text} = 'Adding New International Number'
      ?Phones.Rec.Number{PROP:Text} = '@S20'
      Phones.Rec.Number[1] = '+'
      DISPLAY
      SELECT(?,2)
    OF BUTTON:No
      TARGET{PROP:Text} = 'Adding New Domestic Number'
      ?Phones.Rec.Number{PROP:Text} = '@P###-###-####P'
    END
  END
END

```

Подпрограмма SetupScreen начинается с анализа в структуре CASE Action необходимого действия. Когда пользователь добавляет запись (OF InsertRecord) оператор CLEAR(Phones.Rec) очищает буфер записи, присваивая всем полям пробелы или нули. В операторе TARGET{PROP:Text} = 'Adding New Number' с помощью синтаксиса присвоения значений свойствам, принятого в языке Clarion, динамически изменяется строка заголовка окна на "Adding New Number". Синтаксис присвоения значений свойствам в языке Clarion позволяет динамически изменить значение любого свойства (атрибута) структуры APPLICATION, WINDOW или REPORT. Подробнее о свойствах смотри Приложение В - Присвоение значений свойствам в Описании языка.

TARGET - это встроенная переменная, которая всегда "указывает" на текущую открытую структуру WINDOW. Фигурные скобки ({} ) ограничивают само свойство, а PROP:Text это мнемоническое имя (определение которого содержится в файле PROPERTY.CLW, автоматически включаемом компилятором как и файл EQUATES.CLW), которое соотносит параметр и элемент данных (в этом случае структуру WINDOW).

В ветви OF ChangeRecord оператор TARGET{PROP:Text} = 'Updating ' & CLIP(Phones.Rec.Name) & "'s Phone Number' делает то же самое, но изменяет текст заголовка на "Updating некий\_номер Phone Number." Амперсанд (&) это оператор конкатенации в языке Clarion, а функция CLIP(Phones.Rec.Name) удаляет пробелы в (кроме одного) в конце строки. Структура IF Phones.Rec:Number[1] <> '+' проверяет первый символ на "плюс" строковое поле Number. Символ плюс используется как признак того, что номер включает международные телефонные коды.

Заметим, что Phones.Rec:Number[1] адресует первый байт поля, как если бы оно было массивом. Но вы же помните, что в его объявлении не было атрибута DIM (атрибутом DIM объявляется массив). Строки типа STRING, CSTRING и PSTRING в языке Clarion неявно объявляются как массивы STRING(1), DIM(SIZE(Строка)). Это значит, что независимо от того, объявлено поле массивом или нет, можно обращаться к любому байту непосредственно.

Если номер телефона не международный, оператор ?Phones.Rec:Number{PROP:Text} = '@P###-###-####P', то для того, чтобы изменить шаблон ввода данных в поле. Отметим, что для этого как и в предыдущем случае для изменения строки заголовка окна используется свойство PROP:Text. Причина в том, что свойство PROP:Text адресует разные атрибуты, в зависимости от того, параметром какого объекта оно выступает. Поэтому, для WINDOW('текст заголовка') оно адресует строку заголовка, а для ENTRY(@S20) - шаблон ввода @S20.

Программный код в ветви OF DeleteRecord похож на код в ChangeRecord. В операторе DISABLE(FIRSTFIELD(),LASTFIELD()) используются функции FIRSTFIELD() и LASTFIELD(), чтобы "затемнить" (сделать недоступными) все поля в окне, а затем оператором ENABLE(?Ok,?Cancel) включаются кнопки OK и Cancel.

Подпрограмма SetupInsert выполняется перед тем как перейдет к полю Number. В IF Action = InsertRecord проверяется значение Action и, только когда пользователь добавляет запись, выполняется структура CASE MESSAGE. Для организации альтернативы (да/нет) можно использовать функцию MESSAGE. В данном случае мы спрашиваем пользователя, является ли номер международным.

Код в ветви OF BUTTON:Yes выполняется когда пользователь нажал кнопку Yes в окне MESSAGE. Оператором TARGET{PROP:Text} = 'Adding New International Number' изменяется строка заголовка окна, затем оператор ?Phones:Rec:Number{PROP:Text} = '@S20' изменяет шаблон для ввода в поле номера. Оператор Phones:Rec:Number[1] = '+' помещает символ плюс в первую позицию номера, после чего он выводится оператором DISPLAY, а SELECT(? ,2) помещает курсор на место второго символа текущего поля.

В ветви OF BUTTON:No также меняется заголовок окна и шаблон ввода в поле.

#### 4. ЩЕЛКНИТЕ по кнопке Run.

Программа выполнится.

### Что делать дальше

---

Этот урок служит только кратким введением в программирование на языке Clarion. Возможности языка гораздо шире, чем рассмотренные здесь, поэтому еще много чего можно изучать. Так что же читать дальше?

- Несколько следующих глав в данном Руководстве программиста. В них освещаются различные аспекты программирования на языке Clarion. Хотя они построены не в виде уроков, в них приведена подробная информация о конкретных вопросах, которым они посвящены.
- Описание языка - это "Библия" языка Clarion. Прочитать все руководство это всегда очень неплохо.
- Исследовать и разобрать исходный текст, создаваемый Генератором программ. После данного урока основная структура кода должна выглядеть знакомой, даже если конкретная логика программы и незнакома.
- Принять участие в одном из учебных семинаров, предлагаемых фирмой Арсис. Звоните (095)-530-22-42
- Присоединитесь (или создайте) местную Группу Пользователей и участвуйте в их заседаниях.
- Примите участие в форуме TopSpeed в CompuServe (GO TOPSPEED) или телеконференции Internet (comp.lang.clarion), чтобы общаться по всему миру с программистами, пишущими на Clarion.



## **Глава 2 Структура программы**

### **Структурное программирование**

“Правильная” структура компьютерной программы - это вопрос, который может спровоцировать крайне жаркую дискуссию. У каждого программиста есть свои определенные и неопровержимые идеи о том, что есть “правильная” структура программы, идеи, не всегда разделяемые другими программистами. Поэтому в этой главе просто рассматриваются средства языка Clarion, обеспечивающие структурирование программы в соответствии с вашими собственными представлениями о “правильной” структуре.

### **ПРОЦЕДУРЫ и ФУНКЦИИ**

---

Основой структурного программирования является возможность разбить текст программы на отдельные функциональные куски, выполняемые по мере необходимости. В языке Clarion для такого разбиения есть два оператора: PROCEDURE и FUNCTION. Структурно процедуры и функции одинаковы. Различие между ними состоит в том, что функция может возвращать некоторое значение и, таким образом, использоваться в выражениях и списках параметров. Процедура же не может возвращать значения, и поэтому может вызываться в программе только отдельным оператором - ее нельзя использовать как составляющую в выражении или в списке параметров.

В языке Clarion элементарные блоки для структурирования программы создаются операторами PROCEDURE и FUNCTION. В целях упрощения последующего изложения везде, где подходила бы фраза “процедура и/или функция” будет употребляться более общий термин процедура. Все дальнейшие упоминания процедуры одинаково относятся и к процедурам, и к функциям.

Внутри процедуры часто повторяющиеся последовательности операторов можно оформить в подпрограмму (оператором ROUTINE). Подпрограммы полезны с точки зрения оптимизации размеров программного кода, и как средство вынесения мелких функциональных частей за рамки логики главной процедуры, делая более понятной общую логику программы. Однако подпрограммы локальны по отношению к процедуре и могут использоваться только в той процедуре, в которой они расположены. И поэтому подпрограммы не рассматриваются в этой главе.

### **Объявление локальных данных, размещаемых в стеке**

---

В каждой процедуре есть раздел объявления данных и раздел исполняемых операторов. Секция объявления данных следует за ключевым словом PROCEDURE (или FUNCTION)

и заканчивается ключевым словом **CODE**. После оператора **CODE** идет раздел исполняемых операторов. Переменные и структуры данных, объявленные в секции данных процедуры являются для этой процедуры локальными. Это значит, что они доступны только в данной процедуре, если только не передаются в другую процедуру как параметры.

При выполнении процедуры объявленным локально (без атрибута **STATIC**) в ней переменным память выделяется динамически. Если размер локальной переменной не превышает установленного предела, то память ей выделяется в программном стэке, в противном случае - из кучи (но и тут ее свойства точно такие же, как если бы она располагалась в стэке). Когда выполнение процедуры заканчивается и управление передается в ту точку, откуда был вызов процедуры, отведенная локальным переменным память освобождается и используется программой для других целей. В стэке располагаются только переменные, объявленные локальными по отношению к процедуре.

Динамически распределяемые локальные переменные делают возможным написание рекурсивных и реентерабельных (повторно входимых) процедур. Процедура является рекурсивной, если внутри ее программного кода есть обращения к ней самой. При каждом рекурсивном обращении к процедуре для нее распределяются новые копии локальных переменных, не являющихся статическими. Рекурсия является передовой технологией программирования процедур, которые должны выполняться в последовательных итерациях.

## Структура MAP программы

---

Также как в процедуре в программе есть секция объявления данных (между ключевыми словами **PROGRAM** и **CODE**) и секция исполняемых операторов (следом за ключевым словом **CODE**). Все переменные и структуры данных, объявленные в разделе объявления данных программы являются глобальными (доступными в любом месте приложения) и память им выделяется статически.

В разделе объявления данных программы располагается структура MAP всей программы. В этой структуре описывается, какие процедуры являются глобальными и доступны в любом месте программы.

В программе на языке Clarion все процедуры должны быть представлены прототипом в структуре MAP. В прототипе компилятору сообщается имя процедуры и как к ней обращаться. Полное описание прототипов смотри в разделе Прототипы процедур и функций в Описании языка.

## МОДУЛЬ

В структуре MAP вы можете встретить структуру MODULE, которая объявляет отдельный файл исходного текста, содержащий процедуры, представленные прототипами в этой структуре MODULE. Структура MODULE реализует механизм, с помощью которого можно осуществить следующий уровень в организации программ: группировку процедур в отдельных исходных файлах.

Для группировки процедур в отдельных исходных файлах может быть много причин. Это момент, относительно которого дискуссии между программистами, придерживающимися различных точек зрения достигают особого накала. Один подход это группировка процедур, выполняющих связанные меж собой задачи. Другой причиной группировки может быть оптимизация времени компиляции. Еще одной причиной группировки процедур может служить намерение создать библиотеку с динамическими связями (DLL). Безотносительно причины структура MODULE определяет метод, с помощью которого в вашей программе найдут отражение ваши идеи структурирования.

## MEMBER-модуль

Прототипы процедур которые определяются в отдельном от программного файле должны быть объявлены в структуре MODULE внутри структуры MAP. Исходный файл, указанный структурой MODULE должен начинаться оператором MEMBER. Часто файл начинающийся оператором MEMBER называют MEMBER-модулем.

В операторе MEMBER задается имя исходного файла, содержащего программу, к которой принадлежит этот MEMBER-модуль. Структура MODULE в структуре MAP указывает на исходный файл MEMBER-модуля, а оператор MEMBER в MEMBER-модуле указывает обратно на исходный файл, содержащий оператор PROGRAMM.

Например, исходный файл MYPROG.CIW содержит:

PROGRAM	!начало секции глобальных данных
MAP	!глобальная структура MAP
Proc1 PROCEDURE	!прототип процедуры,
	! текст которой находится в MYPROG.CIW
MODULE('MYPROG2')	!отдельный исходный файл MYPROG2.CIW,
Proc2 PROCEDURE	! содержащий другие процедуры
END	!конец структуры MODULE
END	!конец структуры MAP
CODE	!Начало исполняемых операторов программы
! исполняемые операторы	

```
Proc1PROCEDURE
CODE
! исполняемые операторы
```

!Начало секции объявления локальных данных  
!Начало исполняемых операторов процедуры

Второй файл исходного текста, MYPROG2.CLW, объявленный в структуре MODULE программной структуры MAP в приведенном выше примере содержит следующий текст.

```
MEMBER('MYPROG')
!MEMBER-модуль принадлежащий программе в MYPROG.CLW
```

```
Proc2PROCEDURE
CODE
! исполняемые операторы
```

!Начало секции объявления локальных данных  
!Начало исполняемых операторов процедуры

В этом примере программа разбита на две процедуры Proc1 и Proc2. Исходный текст процедуры Proc2 находится в отдельном файле MYPROG2.CLW. Поэтому в структуре MAP программы имеется структура MODULE, в которой объявляется, что Proc2 содержится в файле MYPROG2.CLW, а первым оператором в этом файле стоит оператор MEMBER, определяющий к какой программе принадлежит этот файл.

## **MAP в MEMBER-модуле**

---

В MEMBER-модуле тоже есть секция объявления данных. Она начинается следом за ключевым словом MEMBER и заканчивается первым оператором PROCEDURE или FUNCTION.

Любая переменная или структура данных, объявленная в секции объявления данных MEMBER-модуля является локальной по отношению к модулю и им выделяется статическая память. Это означает, что доступ к ним может осуществляться только из процедур данного модуля (за исключением передачи в качестве параметра в процедуру, располагающуюся в другом исходном модуле), что делает их “личными” (private) переменными.

Дополнительно к объявлению данных в этом разделе MEMBER-модуля может содержаться своя собственная структура MAP. Она структурно похожа на MAP в программном модуле, она содержит прототипы процедур которые доступны только внутри данного MEMBER-модуля (делая их личными процедурами).

Как в предыдущем примере файл MYPROG.CLW содержит:



```

PROGRAM          ! начало секции глобальных данных
MAP              ! глобальная структура MAP
  MODULE('MYPROG2') ! отдельный исходный файл MYPROG2.CLW,
Proc1  PROCEDURE ! содержащий процедуру
      END        !конец структуры MODULE
END            !конец структуры MAP
CODE          !начало секции исполняемых операторов программы
!исполняемые операторы

```

Второй файл, MYPROG2.CLW теперь содержит следующий текст:

```

MEMBER('MYPROG') !начало секции данных MEMBER-модуля
                  ! declaration section
MAP              ! структура MAP MEMBER модуля
Proc2  PROCEDURE !прототип процедуры, которая локальна для
                  ! MEMBER-модуля MYPROG2.CLW
      END        !конец структуры MAP
Var1 BYTE        !статическая переменная локальная для MEMBER-модуля
Proc1  PROCEDURE !начало секции объявления локальных данных
      CODE      !начало исполняемых операторов процедуры
      !исполняемые операторы

Proc2  PROCEDURE ! начало секции объявления локальных данных
      CODE      ! начало исполняемых операторов процедуры
      ! исполняемые операторы

```

Процедура Proc2 объявлялась в структуре MAP не программного модуля, а MEMBER-модуля. Поэтому она может вызываться только процедурами, находящимися внутри MEMBER-модуля MYPROG2.CLW (Proc1). Переменная Var1 объявлена в секции данных MEMBER-модуля, ее могут использовать только процедуры, расположенные в этом модуле - Proc1 и Proc2.

## Структуры MODULE в структуре MAP MEMBER-модуля

Также как структура MAP программного модуля структура MAP MEMBER-модуля может содержать структуру MODULE, если процедуры, прототипы которых она содержит, находятся в отдельном файле. Любая процедура, не представленная прототипом в структуре MAP программного модуля, должна быть представлена прототипом в MEMBER-модуле, в котором она располагается. Это означает, что требуются идентичные прототипы:

- в структуре MODULE структуры MAP MEMBER-модуля, в тексте которого имеются обращения к этой процедуре.
- в структуре MAP второго MEMBER-модуля, который в действительности содержит писание этой процедуры.

Опять используем предыдущий пример. Файл MYPROG.CLW содержит:

```

PROGRAM          !начало секции объявления глобальных данных
MAP              !глобальная структура MAP
  MODULE('MYPROG2') !отдельный файл, MYPROG2.CLW
Proc1    PROCEDURE      ! содержит процедуру
          END            !конец структуры MODULE
END        !конец структуры MAP
CODE      !начало исполняемых операторов программы
!исполняемые операторы

```

Второй файл, MYPROG2.CLW, теперь содержит:

```

MEMBER('MYPROG') !начало объявления данных MEMBER-модуля
MAP              !структура MAP MEMBER-модуля
  MODULE('MYPROG3') !отдельный файл, MYPROG3.CLW
Proc2    PROCEDURE      !содержит другую процедуру
          END            !конец структуры MODULE
END        !конец структуры MAP
Proc1    PROCEDURE      !начало секции объявления локальных данных
CODE      !начало исполняемых операторов процедуры
!исполняемые операторы

```

Третий файл, MYPROG3.CLW содержит:

```

MEMBER('MYPROG') !начало данных MEMBER-модуля
MAP              !структура MAP MEMBER-модуля
Proc2    PROCEDURE      !прототип процедуры, идентичный с
                        !прототипом объявленным в структуре MAP
                        ! MEMBER-модуля MYPROG2.CLW
END        !конец структуры MAP
Proc2    PROCEDURE      !начало секции объявления локальных данных
CODE      !начало секции исполняемых операторов
           !процедуры
!исполняемые операторы

```

Прототип процедуры Proc2 в структуре MAP в модуле MYPROG2.CLW повторяет прототип в структуре MAP в модуле MYPROG3.CLW. Это дублирование обязательно для всех процедур объявленных в структуре MODULE структуры MAP MEMBER-модуля.

## Резюме

---

- В языке Clarion можно разбивать задачи на процедуры.
- Процедура может содержать подпрограммы, которые оптимизируют использование повторяющихся кусков кода.
- В процедуре могут быть локальные переменные, размещаемые в стеке, применение которых оптимизирует требования к памяти и делают возможной рекурсию.
- В программной секции объявления данных объявляются данные, доступные в любом месте программы.
- В структуре MAP программы объявляются прототипы для всех процедур, доступных в любом месте программы.
- Структура MAP может содержать структуру MODULE, которая указывает на то, что исходный текст процедур, располагается в отдельном файле.
- Оператор MEMBER указывает обратно на программный исходный модуль.
- В секции объявления данных MEMBER-модуля могут объявляться переменные и структуры данных, которые доступны только в пределах данного модуля.
- В MEMBER-модуле может быть своя структура MAP, в которой объявляются процедуры доступные только в пределах данного модуля.
- Структура MAP MEMBER-модуля может содержать структуру MODULE, которая указывает на процедуры, расположенные в другом MEMBER-модуле. Тот модуль в свою очередь тоже должен содержать идентичные прототипы этих процедур.



## **Объектно-ориентированное программирование (ООП)**

### **Введение в объекты**

#### **Что такое объекты?**

---

Объект это единая “вещь”.

Мы, люди, склонны группировать (классифицировать) все, что нас окружает, в классы вещей и явлений. Мы классифицируем на основе общих для них свойств и поведения. Поэтому, предметы, принадлежащие к одному классу, имеют свойства и поведение, которые характеризуют данный тип предметов, который они представляют.

Мы также склонны подразделять классы окружающих нас вещей на иерархию классов, которая напоминает перевернутое дерево. Иерархическое дерево начинается с наиболее общего вверху и спускается вниз к наиболее частному. Каждая ветвь дерева представляет собой отдельный класс который имеет общие свойства с классом наверху, к которому он присоединен (его родителем), потому что он произведен от этого родителя. Каждый производный класс должен также иметь уникальные для него свойства, которые обособливают его от родителя и от других классов, порожденных от этого же родителя.

Объект это экземпляр класса предметов, обычно с самого низшего уровня иерархического дерева. Он имеет свойства и поведение, которые определяют, что он собой представляет. Например, мы видим два основных класса предметов в окружающем мире: живые и неживые. Можно подразделить класс живых вещей на два подкласса: растения и животные. Внутри класса животных можно выделить много подклассов: млекопитающие, птицы, рыбы и т.д. Далее эти подклассы еще подразделяются на под- подклассы... Это дробление продолжается до тех пор, пока мы не придем к отдельному предмету (единой “вещи”).

#### **Почему объекты?**

---

Технология объектно-ориентированного программирования (ООП) была разработана для того, чтобы более точно смоделировать программу, которую нужно написать, таким же образом, каким мы смотрим на окружающий мир. Проектирование иерархии классов для решения реальной проблемы это повод посмотреть на проблему с той же точки зрения с какой мы смотрим на природу. Мы начинаем с наиболее абстрактного, общего класса, свойства и поведение которого характерны для всех членов класса, затем производим из этого абстрактного класса особые классы, которые полностью описывают набор отдельных объектов, составляющих проблему.

Одним из главных преимуществ объектно-ориентированного программирования является широкое повторное использование кода. Как только поведение (метод) запрограммировано для более общего класса, его не нужно повторно переписывать для производных классов, в которых его не нужно изменять.

Следовательно, код пишется единожды и используется для всех объектов, производных от данного класса.

## **Что делает объект объектом?**

---

В ООП есть три основных понятия: инкапсуляция, наследование и полиморфизм. Объектно-ориентированное расширение языка Clarion охватывают все три эти понятия.

### **Инкапсуляция**

Инкапсуляция означает связывание свойств класса (его членов-данных) с его методами (процедурами, которые оперируют этими данными) в одну связную единицу. Наибольшая польза от инкапсуляции состоит в том, что возможность рассматривать объект как единое целое. Это позволяет использовать его, ничего не зная о его устройстве. Кроме того, это позволяет изменить один объект, не затрагивая другие, не связанные с ним объекты.

Свойства объекта - это то, что объект знает о себе самом, а методы это его поведение - операции, которые он может выполнять. У каждого объекта есть собственные, уникальные для него свойства, и он делит одни и те же методы со всеми другими объектами этого же типа (другими экземплярами того же самого класса).

Возьмем, объекты, которые принадлежат к некоторому классу (такому как класс “нормальных здоровых людей”), каждый из них имеет свой уникальный набор свойств (как, например, голубые или карие глаза), но все они имеют одинаковую базовую способность, общую для класса объектов, к которому они принадлежат (способность видеть).

### **Наследование**

Наследование это механизм, который позволяет нам построить иерархию классов. Производный класс наследует все свойства и методы класса, от которого он произошел (базового класса). Этот механизм снабжает производный класс начальным набором всех свойств и методов, более общего (абстрактного) класса. Затем производному классу можно добавить свойства и методы, отличающие его от родителя.

Также наследование это один из центральных элементов объектно-ориентированного подхода, которые обеспечивают повторную используемость кода. Код унаследованных методов, которые не переопределены для производного класса, существует в единственном экземпляре - в классе, в котором они определены.

## **Полиморфизм**

В общем случае полиморфизм означает способность обращаться к методам, которые работают по-разному в зависимости от того, как к нему обращаются. Например, оператор OPEN языка Clarion полиморфная, потому что она выполняет разные действия в зависимости от того, использовали ли его для открытия файла или окна.

Этот тип полиморфизма обычно называют “перегрузкой функций”, потому что вы “перегружаете” то, что выглядело обращением к одной функции, несколькими операциями. В действительности перегрузка функций осуществляется просто определением нескольких функций с одинаковым именем и различными списками параметров. Подробное обсуждение этой темы, смотри в разделе Перегрузка функций в Описании языка.

Полиморфизм в объектно-ориентированном смысле чаще всего принято понимать как возможность методов базового класса обращаться к методам порожденного класса - не зная точно во время компиляции к какому методу следует обратиться. Это называется “использование виртуальных методов”. Если рассматривать наследование как возможность методов порожденных классов обращаться к методам базовых классов, то можно рассматривать виртуальные методы, как способность методов базового класса обращаться к методам порожденного класса. Это выглядит немного невероятно, поскольку базовый класс никогда не знает о том, какой класс произведен от него.

Для работы с виртуальными методами во время создания исполняемого модуля компилятор должен реализовать “позднее связывание”, вместо “раннего связывания. При раннем связывании компилятор может разрешать ссылки на не виртуальные методы во время компиляции конкретными адресами в исполняемом модуле. Это означает непосредственные обращения к процедуре, что является очень эффективным.

Тогда как, при позднем связывании компилятор должен создать таблицу виртуальных методов (Virtual Method Table - VMT), которая содержит конкретные адреса кодов для всех виртуальных методов. Компилятор должен включить программный код для реализации во время выполнения всех обращений к виртуальным методам, просматривая сначала таблицу виртуальных методов, а затем обращаясь к соответствующему методу. Может показаться, что это влечет за собой явные издержки производительности, но при использовании компилятора Clarion эти обращения почти также быстры, как обращения к не виртуальным методам.

Этот вид полиморфизма с помощью виртуальных методов представляет собой механизм, позволяющий нам управлять индивидуальными отличиями между классами, порожденными от одного базового класса, позволяющий в то же время базовым классам игнорировать эти отличия. Это делает возможным общим методам базового класса обращаться к методам одного из своих порожденных классов для того, чтобы особым образом выполнять некоторые действия, способом характерным для конкретного порожденного класса.

Например, у нас есть класс “повозки”, то в нем есть виртуальный метод “управление повозкой”, поскольку для любой повозки есть некий способ ею управлять. Когда вы порождаете от класса “повозки” класс “велосипеды”, этот порожденный класс имеет свой собственный особый виртуальный метод “управление”, характерный для велосипедов. Потом, когда вы произведете от класса “повозки” класс “автомобили”, у него тоже будет свой собственный особый виртуальный метод “управление автомобилем”. Конечно, и класс “велосипеды” и класс “автомобили” унаследуют метод “движение” который использует метод “управление”. Наследуемый метод “движение” не знает точно обращается ли он во время выполнения программы к методу “управление автомобилем” или к методу “управление велосипедом”, он просто сообщает объекту (будь то автомобиль или велосипед) управлять собой (и он это делает).



## Объектно-ориентированное расширение языка Clarion

Конкретный синтаксис языка Clarion, который позволяет объявить классы и произвести классы от ранее объявленных классов, начинается со структуры CLASS.

### Структура CLASS - инкапсуляция

Структура CLASS языка Clarion объявляет объект класса, содержащий свойства и объявляющий методы, которые оперируют этими свойствами. В языке Clarion свойства - это члены-данные, объявленные в классе, а методы - это процедуры и функции, представленные прототипами в структуре CLASS.

В этом примере объявляется очень простая структура CLASS:

```
SomeClass          CLASS,MODULE('SomeClas.CLW')
PropertyA          LONG
PropertyB          LONG
ManipulateAandB    PROCEDURE
                                END
```

Этот класс называется SomeClass и содержит два свойства (данные-члены): PropertyA и PropertyB. Он также содержит один метод: процедуру ManipulateAandB. Атрибут MODULE в операторе CLASS указывает, что программный код определения процедуры ManipulateAandB содержится в файле SomeClas.CLW.

### Свойства классов

Свойства (члены-данные) которые могут содержаться в структуре CLASS, ограничиваются типами данных, которые допустимы в структуре GROUP. Это значит, что ими могут быть все простые типы данных (LONG, SHORT, REAL и т.д.), а сложные типы (FILE, QUEUE, WINDOW и т.д.) не допускаются. Использование в структуре CLASS только простых классов может показаться ограничением, но в действительности это не так, потому что переменные-указатели тоже допустимы в классах (но об этом позднее).

### Методы в классах

Методы (процедуры и функции) объявленные в структуре CLASS определяются в файле исходного текста, имя которого указывается в атрибуте MODULE. Таким образом все методы инкапсулируются в один исходный файл, облегчая сопровождение и, кроме того, такой подход имеет дополнительные достоинства, которые мы увидим позже.

Для приведенного ранее объявления класса в файле SomeClas.CLW будет содержаться нечто подобное:

```

MEMBER()
SomeClass.ManipulateAandB    PROCEDURE
CODE
MESSAGE('A = ' & SELF.PropertyA)
MESSAGE('B = ' & SELF.PropertyB)

```

Метка оператора PROCEDURE начинается с имени класса, к которому этот метод принадлежит, затем, после точки идет само имя метода. Можно также поименовать метод и класс, к которому он принадлежит, явно задав его в первом параметре (и отметив именем класса SELF) как здесь:

```

MEMBER()
ManipulateAandB PROCEDURE(SomeClass SELF)
CODE
MESSAGE('A = ' & SELF.PropertyA)
MESSAGE('B = ' & SELF.PropertyB)

```

В определении метода для обращения к свойствам или методам текущей реализации класса используется ключевое слово SELF, а не имя класса.

### **Создание объектов**

Объект это одна конкретная реализация класса, содержащая свой собственный набор свойств, характерных для этого объекта. Все реализации объектов совместно используют методы данного класса, поэтому эти методы существуют в единственном экземпляре.

Объявление класса, имеющее атрибут TYPE, только объявляет его - пока вы не создадите экземпляр данного класса, нет никаких объектов данного класса. Объявление же без атрибута TYPE и объявляет класс и создает первый экземпляр объекта данного класса.

Существует два способа создать объект: объявить его в секции данных или создать динамически в разделе исполняемых операторов, используя оператор NEW. При объявлении в секции данных можно использовать LIKE или просто поименовать класс как тип данных.

PROGRAM		
SomeClass	CLASS	!создать класс и экземпляр
класса		
PropertyA	LONG	
PropertyB	LONG	
ManipulateAandB	PROCEDURE	
	END	
MyClass	CLASS,TYPE	!создать только класс
PropertyA	LONG	

```

PropertyB                                LONG
ManipulateAandB                          PROCEDURE
                                          END

SomeClassObject1 LIKE(SomeClass)         !создать первый объект класса
SomeClassObject2 SomeClass               !еще один объект класса SomeClass
MyClassObject1    LIKE(MyClass)          !создать первый объект класса
MyClass
MyClassObject2    MyClass                !еще один объект класса MyClass

CODE

```

Для того, чтобы создать объект динамически в разделе исполняемых операторов, используя оператор NEW, объявите переменную указатель на класс. Затем выполните оператор присвоения указателя, указав в качестве параметра оператора NEW имя нужного класса:

```

PROGRAM
SomeClass      CLASS

PropertyA      LONG
PropertyB      LONG
ManipulateAandB      PROCEDURE
                  END

SomeClassObjectRef    &SomeClass    !указатель на объект класса SomeClass
CODE
SomeClassObjectRef &= NEW(SomeClass)    !создать объект

```

Преимущество второго способа в том, что можно поместить переменную указатель на объект внутри другого объекта в качестве свойства.

### **Использование указателей в качестве свойств**

Поскольку указатели можно использовать повсюду где можно использовать метку типа данных, на который они указывают, из класса можно обращаться ко всем сложным структурам, которые не могут непосредственно входить в класс. Если прибавить сюда использование операторы NEW и DISPOSE для динамического создания и уничтожения объектов, то получится, что классы способны содержать почти все эти сложные структуры.

Например, используя в классе переменную указатель, можно объявить ссылку на очередь (&QUEUE), которая в свою очередь передается методам класса. Все это справедливо и для всех составных типов данных (&QUEUE, &FILE, &BLOB, &VIEW и &WINDOW). Хотя класс не может непосредственно содержать объявления этих сложных структур, методы класса могут действовать с ними как если бы это была их собственная структура. Например, хотя объявление файла должно быть внешним по отношению к структуре CLASS, объявив

в ней переменную указатель на файл (&FILE), а затем присвоив ей (как свойству одного конкретного объекта данного класса) адрес конкретного типа структуры FILE, мы получим, объект “владеет” файлом, и в его методах можно выполнять любые операторы, в которых требуется указание метки файла, эффективно работая прямо со структурой FILE.

Также можно объявить указатель на любой конкретный тип QUEUE, GROUP или CLASS (в виде &QueueName, &GroupName или &ClassName), которую в методе можно динамически создать. Это позволяет иметь в классе конкретные виды очередей и других классов, не объявляя их в данной структуре CLASS.

В этом примере объявляется простая структура CLASS, содержащая переменную указатель, которая “указывает” на конкретный тип структуры QUEUE.

```

MyQueue          QUEUE,TYPE      !Объявим конкретный тип очереди
Field1           LONG
Field2           STRING(20)
                END

SomeClass        CLASS
QueRef           &MyQueue        !очередь из переменных LONG и STRING(20)
CreateQue        PROCEDURE       !создать очередь для каждого объекта
                END

SomeClass.CreateQue  PROCEDURE
CODE
SELF.QueRef &= NEW(MyQueue)      !создать очередь для текущего объекта
SELF.QueRef.Field1 = 1
SELF.QueRef.Field2 = 'First Entry'
ADD(SELF.QueRef)
                ! и добавить первую запись в очередь
IF ERRORCODE() THEN STOP(ERROR()).

```

Использование таких поименованных ссылок и присвоение указателя на конкретную структуру CLASS делает возможным одному объекту действительно содержать экземпляры другого объекта. Это технология ООП известная как “композиция” является альтернативой множественному наследованию (которое мы коротко затронем, когда будем обсуждать наследование).

### **Конструкторы и деструкторы**

Конструкторы и деструкторы являются особенностями некоторых объектно-ориентированных языков. Они представляют собой методы (которые вы должны написать), которые выполняются автоматически, без явного обращения к ним. Конструктор выполняется при создании объекта, а деструктор при уничтожении. Наиболее общее назначение конструктора получить для объекта память и присвоить начальные значения членам-данным, чтобы избежать появления ошибок, к которым приводит использование

неинициализированных переменных. Деструктор же обычно освобождает память, занимаемую объектом.

Недостатком этих автоматически выполняемых методов является то, что вы не можете увидеть их в тексте - они скрыты в определении класса. Вы не можете сказать, глядя на строку кода, в которой создается объект, выполненлся или нет автоматически конструктор.

В языке Clarion память выделяется и освобождается автоматически, также автоматически и инициализируются переменные (нулем, пробелами или указанным в объявлении переменной значением). А это означает что наибольшая причина существования конструкторов и деструкторов отсутствует. Поэтому язык Clarion и не поддерживает их. Это не означает что вы не можете их сделать. Просто это значит, что если у вас есть какие-либо особые требования по инициализации или действия, которые нужно выполнить непосредственно перед уничтожением объекта, то вы явно обращаетесь к методу для выполнения этих действий после создания объекта или перед его уничтожением.

```
SomeClass      CLASS,TYPE
PropertyA      LONG
InitClass      PROCEDURE      !мой метод - конструктор
                END
ClassRef      &SomeClass
                CODE
ClassRef &= NEW(SomeClass)      !создать объект класса SomeClass
ClassRef.InitClass      !обратиться к методу - конструктору
```

### **Общее или личное**

Если свойства и методы класса не объявлены с атрибутом PRIVATE, то они являются общими (public). В этом контексте “общие” означает, что они доступны для использования в любом месте где доступен сам объект.

Атрибут PRIVATE указывает, что свойство или метод, к которому он относится, доступен только в процедурах и функциях, определенных в исходных модулях, содержащих данный метод структуры CLASS. Это полностью инкапсулирует данные и методы от других классов.

```
SomeClass      CLASS,MODULE('SomeClas.CLW'),TYPE
PublicVar      LONG      !Объявить общее свойство
PrivateVar      LONG,PRIVATE      ! Объявить личное свойство
BaseProc      PROCEDURE(REAL Parm)      ! Объявить общий метод
Proc      PROCEDURE(REAL Parm),PRIVATE      ! Объявить личный метод
                END

TwoClass      SomeClass      !объявить объект
                CODE
```

```

TwoClass.PublicVar = 1           !допустимое присвоение
TwoClass.PrivateVar = 1         !недопустимое присвоение
TwoClass.Proc(2)                 ! недопустимое обращение к
Proc

!Файл SomeClass.CLW содержит:
    MEMBER()
    MAP                           !локальную MAP, в которой объ-
SomeLocalProc    PROCEDURE(SomeClass)!является “ друг” класса SomeClass
                                   !и ему передается текущий объект
    END

SomeClass.BaseProc    PROCEDURE(REAL Parm)
    CODE
    SELF.PrivateVar = Parm        ! допустимое присвоение
    SELF.Proc(Parm)              ! допустимое обращение к Proc
    SomeLocalProc(SELF)          !обращение к другу с передачей ему текущего объекта

SomeClass.Proc          PROCEDURE(REAL Parm)
    CODE
    RETURN(Parm)

SomeLocalProc    PROCEDURE(PassedObject) !видима только в данном модуле
    CODE
    PassedObject.PrivateVar = 1      ! допустимое присвоение

```

Побочным преимуществом кларионовской реализации личных свойств и методов является появление друзей. В C++ друг это функция, которая не являясь методом класса, может совместно с ними использовать свойства данного класса. В приведенном выше примере SomeLocalProc является другом класса OneClass поскольку определена в том же исходном модуле, что и методы, принадлежащие этому классу, что дает ей доступ к личным членам-данным. Передача текущего объекта в SomeLocalProc позволяет ему непосредственно обращаться к личным свойствам текущего объекта (текущего экземпляра класса)

## **Порожденные классы - наследование**

Язык Clarion поддерживает наследование - одно объявление класса может строиться на основе свойств и методов другого класса. Класс, от которого производится данный класс обычно называется базовым классом.

В этом примере объявляется структура CLASS, производная от другой:

```

SomeClass          CLASS
!объявить базовый класс
PropertyA          LONG
PropertyB          LONG
ManipulateAandB    PROCEDURE
                  END

AnotherClass       CLASS(SomeClass)    !производный от SomeClass класс
PropertyC          LONG
ManipulateAndC     PROCEDURE
                  END

```

В этом простом примере AnotherClass производится от класса SomeClass (его базового класса), наследуя все его свойства и методы. В классе AnotherClass дополнительно объявляются одно новое свойство и один новый метод, которых нет в базовом классе. Следовательно, объекты, относящиеся к классу SomeClass, имеют по одному свойству и по одному методу, тогда как объекты класса AnotherClass имеют три свойства и два метода.

Для приведенного выше объявления двух классов допустимы будут следующие операторы:

```

CODE
SomeClass.PropertyA = 10    !присвоим значения свойствам
SomeClass.PropertyB = 10
SomeClass.ManipulateAandB   !обратимся к методу объекта

AnotherClass.PropertyA = 10 ! присвоим значения свойствам
AnotherClass.PropertyB = 10
AnotherClass.PropertyC = 10
AnotherClass.ManipulateAandB ! обратимся к методу объекта
AnotherClass.ManipulateAndC

```

А такие операторы будут неверны:

```

SomeClass.PropertyC = 10    !неправильно, у объекта нет этого свойства
SomeClass.ManipulateAndC    ! неправильно, у объекта нет этого метода

```

Эти операторы неправильны вследствие того, что хотя производный объект имеет все свойства и методы базового класса, обратное утверждение неверно. Наследование - односторонне. Существует механизм, с помощью которого можно обойти это ограничение, позволяющий методам базового класса обращаться к методам производного класса (виртуальные методы), но мы рассмотрим его позже.

## Переопределение унаследованных методов

Бывают обстоятельства когда нужно, чтобы в производном классе унаследованный метод был переопределен для того, чтобы обеспечить необходимые для этого объекта функциональные возможности. В языке Clarion это делается просто. Переобъявите метод в производном классе (точно с таким же списком параметров), затем переопределите его, реализовав новые функциональные возможности.

Пример переопределения унаследованного метода:

```
SomeClass                CLASS
PropertyA                LONG
PropertyB                LONG
ManipulateAandB          PROCEDURE
                        END

AnotherClass             CLASS(SomeClass)
PropertyC                LONG
ManipulateAandB          PROCEDURE      !Переобъявить унаследованный метод
ManipulateAndC           PROCEDURE
                        END

SomeClass.ManipulateAandB  PROCEDURE
Product                  LONG
CODE
    Product = SELF.PropertyA * SELF.PropertyB
    MESSAGE('Product of A*B is ' & Product)

AnotherClass.ManipulateAandB  PROCEDURE
                        !Переопределить унаследованный метод
DoubleProduct            LONG
CODE
    DoubleProduct = (SELF.PropertyA * SELF.PropertyB) * 2
    MESSAGE('Double the Product of A*B is ' & DoubleProduct)

AnotherClass.ManipulateAndC  PROCEDURE
DoubleProduct            LONG
CODE
    DoubleProduct = (SELF.PropertyA * SELF.PropertyC) * 2
    MESSAGE('Double the Product of A*C is ' & DoubleProduct)
```

В этом примере в классе AnotherClass переопределяется метод ManipulateAandB. В нем реализуется совершенно другой алгоритм. Заметим, что список параметров не изменился (метод не получает никаких параметров). Это ключевой момент в переопределении унаследованных методов: список параметров должен быть одинаков.



Если бы списки параметров этих двух методов ManipulateAandB различались, то это была бы перегрузка функций в классе AnotherClass. В этом случае классу AnotherClass реально принадлежали бы два метода с именем ManipulateAandB и компилятор, основываясь на различии в списках параметров, должен был бы в каждом случае решать к какому методу происходит обращение. Более детальное рассмотрение этой технологии смотрите в разделе Перегрузка функций в Описании языка .

### **Множественное наследование или композиция**

Одинарное наследование означает, что у производного класса есть только один базовый класс, от которого он наследует свойства и методы. В языке Clarion одинарное наследование поддерживается напрямую. Некоторые объектно-ориентированные языки (в частности C++) допускают множественное наследование, когда производный класс наследует свойства и методы нескольких классов. Преимущество состоит в том, что легко сочетая существующие классы, мы создаем производные классы. Есть в этом и недостаток, состоящий в том, что компилятор должен разобраться с потенциальной двусмысленностью когда два или более классов, от которых производится новый класс, имеют методы с одинаковыми именами - нужно написать дополнительный программный код, устраняющий двусмысленность перегружаемых методов.

Хотя Clarion как и многие другие объектно-ориентированные языки программирования поддерживает только одинарное наследование, легко можно обойти это ограничение, используя стандартную технологию ООП, называемую “композиция”. Композиция означает, что объект одного класса помещается внутрь объекта другого класса. Она обеспечивает достоинства множественного наследования, не создавая при этом возможной двусмысленности. Для тех экземпляров, для которых требуется множественное наследование, сначала решите, какой класс будет базовым, и поместите объект другого класса (содержимый объект) в объект нового, создаваемого, класса (содержащий объект). Затем нужно реализовать конструктор, чтобы подтвердить наличие содержимого объекта. Вызовите конструктор в начале области видимости объекта, следом за оператором CODE, завершающим секцию данных, в которой объект объявлен, или когда содержащий объект (объект-контейнер) создается динамически.

В языке Clarion композиция реализуется помещением указателя на объект внутрь объявления класса:

```
PROGRAM
SomeClass          CLASS,TYPE
PropertyA          LONG
PropertyB          LONG
ManipulateAandB    PROCEDURE
                  END
AnotherClass       CLASS,TYPE
```

```

PropertyC                LONG
ManipulateAndC           PROCEDURE

                                END

MultiClass                CLASS(SomeClass),TYPE  !наследует от SomeClass и
AnotherClassRef           &AnotherClass          !содержит объект класса
AnotherClass
InitClass                PROCEDURE              ! и метод - конструктор
                                END

MClass                    MultiClass            !объявить объект
MClassRef                 &MultiClass          ! объявить указатель на объект
CODE
MClass.InitClass          !создать экземпляр содержимого объекта
                                ! когда содержащий объект становится доступным
MClassRef &= NEW(MultiClass) !создать новый содержащий объект
MClassRef.InitClass       ! затем вызвать его конструктор

MultiClass.InitClass      PROCEDURE
CODE
SELF.AnotherClassRef &= NEW(AnotherClass) !создать содержимый объект

```

## **Виртуальные методы - полиморфизм**

Наследование позволяет производному классу обращаться к методам, которые он унаследовал от своего базового класса. С другой стороны виртуальные методы позволяют методам базового класса обращаться к методам производного класса, не смотря на то, что они не знают точно к чему они обращаются. Чтобы это осуществить, поместите прототип виртуального метода и в базовом классе, и в производном классе.

Атрибут **VIRTUAL** в прототипе метода, объявляет виртуальный метод. Этот атрибут должен быть указан и в прототипе в базовом классе, и в прототипе в производном классе. Обычно, определение виртуального метода в базовом классе представляет собой определение пустой процедуры (которая ничего не делает) или процедуру, которая обеспечивает выполнение некоторых простых действий по умолчанию для производных классов, которым не нужно свое собственное определение метода.

Пример определения двух простых виртуальных методов в двух производных классах:

```

SomeClass                CLASS,TYPE
PropertyA                LONG
PropertyB                LONG
InitAandB                PROCEDURE(LONG PassedA, LONG PassedB)
ManipulateAandB          PROCEDURE,VIRTUAL      !объявить виртуальный метод
                                END

AnotherClass             CLASS(SomeClass),TYPE
ManipulateAandB          PROCEDURE,VIRTUAL      !переопределить виртуальный метод

```

END

DifferentClass CLASS(SomeClass),TYPE

ManipulateAandB PROCEDURE,VIRTUAL

!переоопределить виртуальный метод

END

Object1

AnotherClass !объявить объект

Object2

DifferentClass !объявить другой объект

CODE

Object1.InitAandB(10,20)

!обращение к методу InitAandB класса AnotherClass

Object2.InitAandB(30,40)

!обращение к методу InitAandB класса DifferentClass

SomeClass.InitAandB PROCEDURE(LONG PassedA, LONG PassedB)

CODE

SELF.PropertyA = PassedA

SELF.PropertyB = PassedB

SELF.ManipulateAandB !обращение к методу, который соответствует  
! текущему объектуSomeClass.ManipulateAandB PROCEDURE !пустой метод базового класса,  
CODE ! который ничего не делаетAnotherClass.ManipulateAandB PROCEDURE !виртуальный метод  
CODE

MESSAGE('The Product of A\*B is ' &amp; (SELF.PropertyA \* SELF.PropertyB))

DifferentClass.ManipulateAandB PROCEDURE ! виртуальный метод  
CODE

MESSAGE('The Sum of A+B is ' &amp; (SELF.PropertyA + SELF.PropertyB))

В этом примере метод ManipulateAandB виртуальный. Метод InitAandB обращается к SELF.ManipulateAandB, не зная к какому методу он в действительности обращается. При выполнении оператора Object1.InitAandB(10,20) метод SELF.ManipulateAandB обращается к AnotherClass.ManipulateAandB, при выполнении Object2.InitAandB(30,40) он обращается к методу DifferentClass.ManipulateAandB.

## Резюме

---

- В объектно-ориентированном программировании есть три основные концепции: инкапсуляция, наследование и полиморфизм.
- Структура CLASS реализует инкапсуляцию.
- Исходный текст всех методов, принадлежащих к одному классу, размещаются в одном исходном модуле, облегчая его сопровождение.
- Объект представляет собой экземпляр класса со своим собственным набором данных - членов (свойств), который совместно с другими объектами данного класса и производных классов использует методы принадлежащие к данному классу.
- Класс может содержать данные-члены (свойства), объявляемые как простые типы данных или как указатели на составные типы (включая другие классы).
- К конструктору и деструктору нужно обращаться явно, что облегчает нахождение их в исходном тексте программы.
- Все методы и свойства являются общими, если только в их объявлении не указан атрибут PRIVATE.
- Наследование достигается порождением одного класса другим.
- Можно заместить унаследованный метод, заново объявив и заново определив его в производном классе с таким же точно списком параметров.
- Композиция обеспечивает жизнеспособную альтернативу множественному наследованию.
- Виртуальные методы реализуют стандартный в ООП полиморфизм, тогда как перегрузка функций в языке Clarion дает не объектно-ориентированную форму полиморфизма.
- Виртуальные методы представляются прототипом и в базовом и в производном классах.

## **РАЗРАБОТКА БАЗЫ ДАННЫХ**

### **Разработка базы данных**

На сегодняшний день существует несколько способов организации базы данных. Ранее широкое распространение получили три способа организации: Модель Инвертированного Списка, Иерархическая Модель, Сетевая Модель. Все они применялись главным образом на больших ЭВМ - мэйнфреймах, а на персональных компьютерах не имели широкого применения. В языке Clarion разработаны средства, которые позволяют использовать любую из перечисленных моделей.

Гораздо более распространенным методом организации базы данных на персональных компьютерах является определенная Коддом реляционная модель. Но ни в одной работающей базе данных не реализованы все правила Кодда для реляционной модели, поскольку это очень сложная математическая модель. Однако, большинство реализаций баз данных поддерживают достаточный набор правил, чтобы позволить практическое применение принципов реляционной модели. В данной главе представлен краткий обзор основных аспектов разработки реляционной базы данных насколько это пересекается с прикладным программированием.

### **Разработка Реляционной базы данных**

---

В основу разработки реляционной базы данных положен следующий принцип: элемент данных, будучи сохранен в одном месте базы, не должен повторяться в других ее местах. Это дает сразу два преимущества: сокращение объема требуемого дискового пространства и упрощение сопровождения данных. При разработке реляционной базы данных этот принцип реализуется путем разбиения данных на части - связанные друг с другом файлы. Рассмотрим, например, упрощенную модель системы приема заказов, в которой хранится следующая информация:

Customer Name	(Имя заказчика)
Customer Address	(Адрес заказчика)
ShipTo Address	(Адрес доставки)
Order Date	(Дата заказа)
Product Ordered	(Заказанный товар)
Quantity Ordered	(Заказанное количество)
Unit Price	(Цена за единицу товара)

Эту информацию можно было бы хранить в каждой записи в одном файле, что весьма неэффективно. В этом случае для каждого заказа и для каждого товара в заказе пришлось бы хранить одну и ту же информацию: имя заказчика, адрес заказчика, адрес доставки и дату заказа. Чтобы избежать повторения разобьем данные на отдельные файлы.

**Файл заказчиков:**

имя заказчика  
адрес заказчика

**Файл заказов:**

адрес доставки  
дата заказа

**Файл компоненты заказа:**

заказанный товар  
заказанное количество товара  
цена за единицу товара

При таком разбиении файл “Заказчики” содержит всю информацию о заказчике, файл “Заказы” - всю информацию о заказе, файл “Компоненты” - всю информацию по каждому товару в заказе. Очевидно, что данные не дублируются. Однако, совершенно неясно, какие записи в других файлах соответствуют конкретной записи данного файла. Для решения этого вопроса были введены реляционные понятия “Первичный Ключ” (Primary Key) и “Внешний Ключ” (Foreign Key).

Первичный ключ - это указатель на запись файла, представляющий собой поле (или поля), значение которого не может повторяться в файле или быть пустым. В языке Clarion первичный ключ задается объявлением (оператором KEY) без атрибута DUP, причем для ввода данных поля-компоненты ключа должны иметь атрибут REQ. При строгом соблюдении канонов разработки реляционной базы данных для каждого файла базы должен быть первичный ключ.

Внешний ключ - это это указатель на запись файла, представляющий собой поле (или поля), значение которого повторяет значение поля первичного ключа другого файла, связанного с данным. Иными словами, внешний ключ содержит “ссылку” на первичный ключ другого файла.

Первичный и внешний ключи составляют основу взаимосвязей файлов реляционной базы данных. Совпадающие значения первичного и внешнего ключей указывают на наличие связи между записями (являются “указателями” на эти записи). Записи внешнего ключа в “Файле А” отсылают к записи первичного ключа в “Файле Б”, а первичный ключ “Файла Б” на записи внешнего ключа в “Файле А”.

Чтобы файлы предыдущего примера удовлетворяли требованиям “реляционности”, в них необходимо добавить некоторые поля для определения первичных и внешних ключей.

**Файл заказчиков:**

номер заказчика	- первичный ключ
имя заказчика	
адрес заказчика	

**Файл заказов:**

номер заказа	- первичный ключ
номер заказчика	- внешний ключ
адрес доставки	
дата заказа	

**Файл компоненты заказа:**

номер заказа	- первый компонент первичного ключа и внешний ключ
заказанный товар	- второй компонент первичного ключа
заказанное количество товара	
цена за единицу товара	

Поскольку нельзя гарантировать, что в файле “заказчики” не повторится имя заказчика, то в качестве первичного ключа добавлено поле “номер заказчика”. Номер заказа добавлен в файл “заказы” в качестве первичного ключа, поскольку в файле нет поля, которое можно считать уникальным. в файл заказа введен внешний ключ - поле номер заказчика - для связи с файлом заказчика. Внешний ключ “номер заказа” введен в файл “компоненты” для связи его с файлом “заказы”. Это же поле стало первой компонентой составного первичного ключа (поля номер заказа и заказанный товар).

Понятие ключа (KEY-объявление) языка Clarion никак не связано с реляционными понятиями первичного и внешнего ключей. Это значит, что, несмотря на существование этих ключей в теории, в языке ключ используется только тогда, когда в приложении необходим специфический доступ к файлу. Вообще говоря, для большинства первичных ключей будут созданы ключи в смысле языка Clarion, и весьма редко то же самое будет сделано для внешнего ключа.

**Взаимосвязь файлов**

Между любыми двумя файлами реляционной базы данных можно установить три типа отношений: Одна-к-Одной; Одна-ко-Многим (или родитель-дети) и обратные Многие-к-Одной; Многие-ко-Многим. Эти отношения характеризуют какое число записей одного файла связано с каким числом записей другого файла.

Между файлом Заказчики и файлом Заказы из предыдущего примера установлено отношение Одна-ко-Многим. Одна запись файла Заказчики связана со многими записями в файле Заказы. Такое же отношение установлено между файлом Заказы и файлом

Компоненты: один заказ может состоять из нескольких товаров. Отношение Одна-ко-Многим (“родитель-дети”) очень распространено в коммерческих приложениях баз данных.

Отношение Одна-к-Одной означает, что ровно одна запись в одном файле связана с ровно одной записью другого файла. Это отношение полезно тогда, когда поля некоторого файла не всегда требуют наличия в них информации. При сосредоточении таких полей в одном файле происходит бесполезная трата дискового пространства под пустые поля записей, которые не нуждаются в дополнительной информации. Поэтому выгодно создать второй файл с отношением Одна-к-Одной с первым файлом, в котором хранились бы необязательные поля с информацией в них.

Вернемся к примеру. В файле Заказы наличие отдельного поля Адрес Доставки не является необходимым. Поэтому в базу данных можно ввести еще один файл - файл Доставки.

#### **Файл заказов:**

номер заказа            - первичный ключ  
номер заказчика        - внешний ключ  
дата заказа

#### **Файл доставки:**

номер заказа            - первичный и внешний ключ  
адрес доставки

Теперь запись будет добавлена в файл Доставки только тогда, когда заказ должен быть доставлен по адресу отличному от адреса в файле Заказчики. Между файлом Доставки и файлом Заказы установлено отношение Одна-к-Одной.

Труднее всего иметь дело с отношением Многие-ко-Многим. Оно означает, что некоторое множество записей одного файла связано с некоторым множеством записей другого файла. Распространим наш пример на промышленный концерн, который покупает детали и производит изделия. Одна деталь может входить в разные изделия, а одно изделие может состоять из многих деталей.

#### **Файл детали:**

Номер детали - первичный ключ  
Описание детали

#### **Файл изделий:**

Номер изделия - первичный ключ  
Описание изделия



Не вдаваясь в теоретические подробности укажем, что в этом случае необходимо создать третий файл, который принято называть Связующим файлом. Как видно из примера Связующий файл порождает два отношения Одна-ко-Многим:

#### **Файл детали:**

Номер детали - первичный ключ

Описание детали

#### **Файл детали-изделия:**

Номер детали - 1-й компонент первичного ключа и внешний ключ

Номер изделия 2-й компонент первичного ключа и внешний ключ

Используемое количество

#### **Файл изделий:**

Номер изделия - первичный ключ

Описание изделия

В файле Деталь-Изделие есть составной первичный ключ и два внешних ключа. Между файлом Детали и файлом Деталь-Изделие установлено отношение Одна-ко-Многим. Этот же тип отношения установлен между файлом Изделие и файлом Деталь-Изделие. Связующий файл становится как бы “посредником” между двумя файлами, связанными отношением Многие-ко-Многим.

Ценно то, что обычно существует информация, которая, по логике вещей, должна находится в Связующем файле. В нашем случае Количество (деталей одного вида в изделии) логически попадает в файл Деталь-Изделие.

### **Переложение теории на язык Clarion**

---

В практической разработке реляционной базы данных при определении некоторых файлов нет необходимости объявлять оператором KEY ключ в смысле Clarion для первичных ключей. Если вообще нет необходимости в доступе к отдельной записи файла, то нет необходимости и в ключе в смысле языка Clarion для поддержания первичного ключа. Обычно такое характерно для дочернего файла (из отношения родитель-дети), записи которого нужны только в сочетании с записью файла-родителя.

Для внешнего ключа также необязательно использовать ключ в смысле языка Clarion. Решение об использовании объявления ключа оператором KEY зависит от способа, каким будет осуществляться доступ к файлу, содержащему внешний ключ. Если доступ к записям с внешним ключом осуществляется через первичный ключ, то необходимо создавать ключ в смысле языка Clarion. Если же внешний ключ используется только для того, чтобы гарантировать правильность значения в поле значения внешнего ключа, то языковый ключ

создавать не нужно. Возьмем предыдущие теоретические примеры и переложим их на язык Clarion:

Customer	FILE,DRIVER('Clarion'),PRE(Cus)	
CustKey	KEY(Cus:CustNo)	! Первичный ключ
Record	RECORD	
CustNo	LONG	! номер заказчика - первичный ключ
Name	STRING(30)	! Имя заказчика
Address	STRING(30)	! Адрес заказчика

..

Order	FILE,DRIVER('Clarion'),PRE(Ord)	
OrderKey	KEY(Ord:OrderNo)	! Первичный ключ
CustKey	KEY(Ord:CustNo),DUP	! Внешний ключ
Record	RECORD	
OrderNo	LONG	! Номер заказа - первичный ключ
CustNo	LONG	! Номер заказчика - внешний ключ
Date	LONG	! Дата заказа

..

ShipTo	FILE,DRIVER('Clarion'),PRE(Shp)	
OrderKey	KEY(Shp:OrderNo)	! Первичный ключ
Record	RECORD	
OrderNo	LONG	!номер заказа - первичный ключ и внешний ключ
Address	STRING(30)	! Адрес доставки
Item	FILE,DRIVER('Clarion'),PRE(Itm)	
OrderKey	KEY(Itm:OrderNo,Itm:ProdNo)	! Первичный ключ
Record	RECORD	
OrderNo	LONG	! Номер Заказа-поле первич.ключа;внеш. ключ
ProdNo	LONG	!Номер Изделия-поле первич.ключа;Внеш. ключ
Quantity	SHORT	! Заказанное количество
Price	DECIMAL(7,2)	! Цена за единицу

..

Product	FILE,DRIVER('Clarion'),PRE(Pro)	
ProdKey	KEY(Pro:ProdNo)	! Первичный ключ
Record	RECORD	
ProdNo	LONG	!номер изделия - первичный ключ
Description	STRING(30)	!описание изделия

..

Parts2Prod	FILE,DRIVER('Clarion'),PRE(P2P)	
ProdPartKey	KEY(P2P:ProdNo,P2P:PartNo)	! первичный ключ

PartProdKey	KEY(P2P:PartNo,P2P:ProdNo)	!альтернативный ключ
Record	RECORD	
PartNo	LONG	!Номер Детали-поле Первич. ключа; Внеш. ключ
ProdNo	LONG	!Номер изделия-поле Первич. ключа; Внеш. ключ
Quantity	SHORT	
..		
Parts	FILE,DRIVER('Clarion'),PRE(Par)	
PartKey	KEY(Par:PartNo)	! Первичный ключ
Record	RECORD	
PartNo	LONG	!Номер детали - первичный ключ
Description	STRING(30)	!Описание детали
..		

Отметим, что только для одного внешнего ключа (в файле Order) был объявлен ключ в терминах языка. По счастливой случайности остальные внешние ключи - как компоненты первичных ключей - попали в KEY-объявления для поддержки первичных ключей.

Первичный ключ (Itm:OrderKey) объявлен в качестве ключа файла Item для того, чтобы исключить возможность дублирования наименований изделий в заказе. Если такая возможность допускается, то ключ Itm:OrderKey будет состоять только из Itm:OrderNo и будет объявлен с атрибутом DUP, что разрешает дублирование значений ключа. Тем самым мы перешли от первичного ключа к внешнему ключу и его объявлению в файле.

Файл Item связан с файлом Product отношением Многие-к-Одной, которое с противоположной точки зрения выглядит как Одна-ко-Многим. Такой взгляд часто используется при контроле правильности ввода данных. Например, процедура ввода данных в файл Item может проверять правильность ввода Номера Изделия по значению Номера Изделия в записях файла Product.

## Целостность ссылок

“Целостность ссылок” - один из важнейших вопросов Реляционной модели. Проблема целостности должна решаться на уровне программного обеспечения, поскольку она касается взаимосвязи данных во время ее работы с базой данных.

Целостность ссылок означает, что ни один внешний ключ не может иметь значение, которое не совпадало бы со значением первичного ключа. Поддержание целостности базы данных влечет за собой решение следующих двух вопросов:

- Что нужно делать, если пользователь хочет удалить запись с первичным ключом?
- Что нужно делать, если пользователь хочет изменить значение первичного ключа?

Есть три наиболее общих ответа на каждый из этих вопросов: Ограничить действие,

Каскадировать действие и (реже всего) Обнулить значения внешнего ключа. Разумеется, в каждом конкретном приложении могут быть свои ответы. Можно, например, как это требуется в отдельных программах скопировать всю информацию в файлы “предыстории” перед выполнением определенного действия.

### **Ограничение действия**

Ограничить действие означает, что когда пользователь пытается удалить запись с первичным ключом или изменить значение первичного ключа, то действие разрешено, если только нет внешних ключей, которые ссылаются на данный первичный ключ. Действие запрещено, если такие внешние ключи есть.

Покажем, как может выглядеть фрагмент программы, в котором ограничиваются удаления записей или изменение значения первичного ключа. Определения файлов приведены выше.

```
ChangeRec EQUATE(2)      !символическое имя признака для изменения данных
DeleteRec EQUATE(3)      ! символическое имя для признака удаления данных
SaveKey   LONG           !переменная для сохранения значения первич.ключа
```

```
CODE
```

```
SaveKey = Cus:CustNo      ! сохранение значения первич.ключа
```

```
OPEN(window)
```

```
ACCEPT
```

```
CASE ACCEPTED()          !обработать элемент
                           !обработка отдельного оконного объекта
```

```
OF ?OKButton              !кнопка завершения ввода в окне
```

```
IF Action = ChangeRec AND Cus:CustNo <> SaveKey
                           !проверить не изменился ли первичный ключ?
```

```
Cus:CustNo = SaveKey      ! восстановить значение
```

```
MESSAGE('Key Field changes not allowed!')!сообщим пользов-лю
```

```
SELECT(1)                 ! начать сначала
```

```
CYCLE
```

```
ELSIF Action = DeleteRec  !проверить не удаление ли это
```

```
Ord:CustNo = Cus:CustNo   !начальное значение полю ключа
```

```
GET(Order,Ord:CustKey)    !и попробуем получить связанную запись
```

```
IF NOT ERRORCODE()        !если GET успешный
```

```
MESSAGE('Delete not allowed!') ! сообщим пользователю
```

```
SELECT(1)                 ! начнем сначала
```

```

        CYCLE
    ELSE                                     !если GET неудачный
        DELETE(Customer)                  ! удалим запись о покупателе
        BREAK                             ! и выйдем
    END
                                         !другие исполняемые операторы
END
END
END
END

```

### **Каскадирование действия**

Каскадировать действие - когда пользователь пытается удалить запись с первичным ключом или изменить значение первичного ключа, то действие распространяется и на (каскадируется) любые внешние ключи, ссылающиеся на первичный ключ. Если такие внешние ключи существуют, то при удалении удаляются и записи с внешними ключами, а при замене значения заменяются значения у всех внешних ключей, ссылающихся на данный первичный ключ.

При Каскадировании действия следует учитывать одно важное обстоятельство: как быть, если файл, подвергающийся Каскадированию (дочерний файл), в свою очередь является родительским по отношению к другому файлу? Такую ситуацию необходимо выявлять и обрабатывать, поскольку Каскадирование распространяется на записи всех зависимых файлов. При создании программы обработки данной ситуации необходимо знать взаимосвязи файлов и программировать Каскадирование на всю “глубину” связей, чтобы исключить возможность “повисания” связи.

И вновь продемонстрируем, как может выглядеть фрагмент программы, в котором Каскадируются удаления записей или изменение значения первичного ключа. Определения файлов приведены выше.

```

ChangeRec EQUATE(2)    !символическое имя признака для изменения данных
DeleteRec EQUATE(3)    ! символическое имя признака для удаления данных
SaveKey   LONG          ! переменная для сохранения значения первич.ключа
CODE
SaveKey = Cus:CustNo    ! сохраним первичный ключ
OPEN(window)
ACCEPT
CASE ACCEPTED()         ! обработка отдельного оконного объекта
OF ?OKButton            ! кнопка завершения ввода в окне
    IF Action = ChangeRec AND Cus:CustNo <> SaveKey

```

! проверить не изменился ли первичный ключ?

```

    DO ChangeCascade          ! и каскадировать изменение
    ELSIF Action = DeleteRec   ! проверить не удаление ли это
    DO DeleteCascade          ! и каскадировать удаление
    END
    ! другие исполняемые операторы
    END
END

```

ChangeCascade ROUTINE

```

    Ord:CustNo = SaveKey      ! начальное значение полю ключа
    SET(Ord:CustKey,Ord:CustKey) ! и установим обработку всех

```

! заказов одного покупателя

LOOP

```

    NEXT(Order)              ! по одной записи
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
                                ! проверка окончания данных по покупателю и выход
    Ord:CustNo = Cus:CustNo !изменить на новое значение
    PUT(Order)                ! и записать обратно
    IF ERRORCODE() THEN STOP(ERROR()).
END

```

DeleteCascade ROUTINE

```

    Ord:CustNo = SaveKey      ! начальное значение полю ключа
    SET(Ord:CustKey,Ord:CustKey) ! и установим обработку всех
    LOOP                      ! заказов одного покупателя
    NEXT(Order)               ! по одной записи
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
                                ! проверка окончания данных по покупателю и выход
    CLEAR(Itm:Record)         !очистить буфер
    Itm:OrderNo = Ord:OrderNo ! начальное значение полю ключа
    SET(Itm:OrderKey,Itm:OrderKey) ! и установим обработку всех
    LOOP UNTIL EOF(Item)      ! позиций одного заказа
    NEXT(Item)                ! по одной записи
    IF Itm:OrderNo <> Ord:OrderNo OR ERRORCODE() THEN BREAK.
                                ! проверка окончания данных по заказу и выход
    DELETE(Item)              ! и удаление записи по одному товару
    IF ERRORCODE() THEN STOP(ERROR()).
END                          !конец цикла про файлу товаров
    Shp:OrderNo = Ord:OrderNo !проверить наличие запись о доставке
    GET(ShipTo,Shp:OrderKey)

```

```

IF NOT ERRORCODE()      ! если GET успешный
  DELETE(ShipTo)        ! удалить запись о доставке
  IF ERRORCODE() THEN STOP(ERROR()).
END
DELETE(Ord)  ! и удалить запись о заказе
  IF ERRORCODE() THEN STOP(ERROR()).
END          !конец цикла по файлу заказов

```

### Обнулить внешний ключ

Обнуление внешнего ключа - это когда пользователь пытается удалить запись с первичным ключом или изменить значение первичного ключа, то обнуляются значения внешних ключей, ссылающихся на первичный ключ (если поля внешних ключей допускают нулевое значение).

Используя те же определения файлов, покажем, как может выглядеть фрагмент программы, в котором обнуляются внешние ключи при удалении записей или изменении значения первичного ключа.

```

ChangeRec EQUATE(2)  ! символическое имя признака для изменения данных
DeleteRec EQUATE(3)  ! символическое имя признака для удаления данных
SaveKey    LONG      ! переменная для сохранения значения первич.ключа
CODE
SaveKey = Cus:CustNo    ! сохраним первичный ключ
OPEN(window)
ACCEPT
CASE ACCEPTED()        ! обработка отдельного оконного объекта
OF ?OKButton           ! кнопка завершения ввода в окне
  IF Action = ChangeRec AND Cus:CustNo <> SaveKey
                        ! проверить не изменился ли первичный ключ?
    DO ChangeNullify    ! и обнулить порожденные записи
  ELSIF Action = DeleteRec ! проверить не удаление ли это
    DO DeleteNullify! и обнулить порожденные записи
  END
  ! другие исполняемые операторы
END
END

ChangeNullify ROUTINE
Ord:CustNo = SaveKey    ! начальное значение полю ключа
SET(Ord:CustKey,Ord:CustKey) ! и установим обработку всех
LOOP                   ! заказов одного покупателя
NEXT(Ord)              ! по одной записи
IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.

```

```
                                ! проверка окончания данных по покупателю и выход
Ord:CustNo = 0                !обнулить значение
PUT(Order)                    ! и записать обратно
    IF ERRORCODE() THEN STOP(ERROR()).
END

DeleteNullify ROUTINE
Ord:CustNo = SaveKey          ! начальное значение полю ключа
SET(Ord:CustKey,Ord:CustKey)  ! и установим обработку всех
LOOP                          ! заказов одного покупателя
    NEXT(Order)                ! по одной записи
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
                                ! проверка окончания данных по покупателю и выход
Ord:CustNo = 0                !обнулить поле
PUT(Order)                    ! и записать обратно
    IF ERRORCODE() THEN STOP(ERROR()).
END
```

Вариант Обнуления менее “трудоемок”, чем вариант Каскадирования, поскольку при Каскадировании приходится удалять все связанные записи по всей глубине связи файлов. При Обнулении обнуляются только отдельные внешние ключи - ссылающиеся на первичный ключ, который удаляют или изменяют его значение.



## Резюме

---

- Каждый элемент данных должен храниться в одном экземпляре.
- Для предотвращения дублирования данных используют отдельные файлы.
- Связь между файлами осуществляется посредством первичных и внешних ключей.
- Первичный ключ - уникальный (и ненулевой) индекс файла, по которому осуществляется доступ к отдельной записи в файле.
- Внешний ключ содержит ссылку на первичный ключ какого-либо другого файла.
- “Одна-ко-Многим” - самое распространенное отношение файлов. Его также называют “Родитель-Дети” и “Многие-к-Одной” (то же самое отношение, только с противоположной точки зрения).
- Отношение “Одна-к-Одной” создается главным образом для хранения данных, которые не всегда нужны в каждой записи.
- Отношение “Многие-ко-Многим” нуждается в “Связующем” файле, который выступает в роли посредника между двумя файлами. Связующий файл расчленяет одно отношение “Многие-ко-Многим” на два отношения “Одна-ко-Многим”.
- На практике объявления ключей в смысле языка Clarion необходимо только для тех первичных и внешних ключей, с помощью которых в приложении осуществляется особый доступ к файлам.
- Целостность на уровне ссылок означает, что все внешние ключи содержат значимые ссылки на первичные ключи.
- Поддержание Целостности обеспечивается путем выявления - на программном уровне - операций удаления или изменения значений первичного ключа.
- Наиболее распространены три метода поддержания Целостности на уровне ссылок: Ограничение (изменение/удаление запрещено), Каскадирование (изменение/удаление распространяется и на внешний ключ) и Обнуление внешнего ключа (присвоение нулевого значения внешнему ключу).



## Работа с файлами

Прикладные системы баз данных по определению хранят данные в файлах. Цель любой прикладной базы данных - извлечение их из файлов и преобразование в осмысленную выходную информацию. В данном разделе речь пойдет о возможностях языка Clarion, предоставляющих программисту средства доступа и обработки файлов данных.

### Методы доступа

---

Записи в файлах данных располагаются, вообще говоря, в том порядке, в каком они туда добавляются (это обычно так, но не всегда). Порядок, в котором записи расположены в файле, называется “физическим” или “по номеру”. Совсем необязательно, чтобы такой порядок нес в себе какую-либо смысловую нагрузку.

Существуют два способа доступа к записи в файле: последовательный доступ и произвольный доступ. Последовательный доступ означает извлечение нескольких записей для обработки их в определенной последовательности. Произвольный доступ подразумевает извлечение и обработку одной конкретной записи. Почти в любой программе работающей с базой данных применяется как тот, так и другой способ.

Если требуется извлекать записи из файла в их физической последовательности - по номеру, - то для этого достаточно только файла данных. Если требуется произвольно извлечь запись и точно известно ее местоположение в файле (ее номер), то и для этого достаточно только файла данных. Однако, для большинства приложений такая однобокость была бы слишком большим ограничением.

### Ключи и Индексы

---

Применение в языке Clarion объявлений ключей и индексов предоставляет возможность использовать для последовательного и произвольного доступа другой порядок выбора записей, отличный от физического. Такой порядок определяется полями - компонентами, - из которых составляется ключ (KEY) или индекс (INDEX). Каждая компонента ключа или индекса может быть упорядочена по возрастанию или по убыванию.

Основное различие между ключом и индексом состоит в том, что ключ поддерживается динамически. Всякий раз, когда добавляется запись, изменяется или удаляется, изменяется и ключ. Поскольку ключ всегда отражает текущее состояние файла, его следует использовать для наиболее часто используемых последовательностей сортировки записей.

Индекс динамически не поддерживается. Поэтому, чтобы быть уверенным, что индекс правильно отражает текущее состояние файла, непосредственно перед использованием его нужно каждый раз перестраивать. Перестройка индекса осуществляется оператором BUILD. Оператор BUILD требует много времени и исключительного права доступа к файлу. В силу этого, индекс следует использовать для редко используемых последовательностей

выбора записей.

Специальной формой индекса является “динамический” индекс. Это такой индекс, для которого в определении файла не указаны составляющие его поля (оператор INDEX без параметров). Поля “динамического” индекса объявляются в операторе BUILD во время исполнения программы.

В отличие от “статического” индекса, оператор BUILD для построения “динамического” индекса не требует исключительного права доступа к файлу. Отсюда следует очевидное преимущество: пользователь сам может определять порядок выбора записей.

Sample	FILE,DRIVER('TopSpeed'),PRE(Sam)	
Field1Key	KEY(Sam:Field1)	! ключ по полю Field1
Field2Ndx	INDEX(Sam:Field2)	!статический индекс по полю Field2
DynNdx	INDEX()	!динамический индекс
Record	RECORD	
Field1	LONG	
Field2	STRING(10)	
Field3	DECIMAL(7,2)	
	..	

#### CODE

OPEN(Sample,42h)	! Открыть; доступ:Чт./Зп. + Нет запрета
LOCK(Sample)	! Заблокировать для исключительного
BUILD(Sam:Field2Ndx)	! доступа при построении индекса
UNLOCK(Sample)	! и разблокировать
BUILD(Sam:DynNdx,'-Sam:Field1,+Sam:Field2')	! Создать динамический индекс

В этом примере ключ по полю Sam:Field1 отражает текущее состояние, индекс по полю Sam:Field2 строится, когда файл открыт и заблокирован (требуется исключительное право доступа). “Динамический” индекс строится во время выполнения программы. Порядок выбора записей по полю Sam:Field1 - по убыванию, по полю Sam:Field2 - по возрастанию. Конечно, можно создать индекс с любым возможным для файла порядком выбора.

За исключением различия в их поддержке, ключи и индексы функционально эквивалентны. Они используют один и тот же формат файла. Они равноправно могут выступать в качестве параметров (если того требует синтаксис) в любом операторе языка, осуществляющем доступ к файлу. Поэтому в дальнейшем, в целях упрощения, вместо фразы “ключ и/или индекс” будем применять общий термин индекс. Если не оговорено особо, то любое употребление слова индекс равно применимо как к ключу, так и к индексу.

## Последовательный доступ к файлу

Последовательный доступ к файлу в языке Clarion осуществляют три оператора: SET, NEXT и PREVIOUS. Оператор SET инициирует последовательную обработку; он не производит считывания записи файла. Операторы NEXT и PREVIOUS считывают записи, порядок доступа к которым задан оператором SET, в возрастающем (NEXT) или убывающем (PREVIOUS) порядке.

Оператор SET является “отправным пунктом” в последовательной обработке файла. Чтобы установить начало и порядок доступа для считываемых записей операторам NEXT или PREVIOUS должен предшествовать оператор SET. Как правило, оператор SET является последним исполняемым оператором перед структурой LOOP, в которой осуществляется последовательная обработка записей файла. Оператор NEXT или PREVIOUS будет тогда первым оператором в цикле LOOP:

```
SET(Sam:Field1Key)      ! Начало файла; доступ по ключу
LOOP                   ! Обработать весь файл
    NEXT(Sample)        ! читать каждую запись
    IF ERRORCODE() THEN BREAK.    ! По концу файла выйти из цикла
                                ! операторы обработки записи
END                     ! Конец цикла
```

В Описании языка приведены семь форм оператора SET. По существу, они распадаются на две группы, в зависимости от порядка доступа к записи: по физическому номеру - три формы; по индексу - четыре формы.

### Физический порядок

Начало/Конец Файла  
Физический номер записи  
Значение индекса

### По индексу

Начало/Конец Файла  
номер записи в индексе  
Значение индекса  
Значение индекса и Физический номер записи

Оператор SET инициализирует - посредством механизма “нечеткой логики” - указатель записи для последовательной обработки. При установке на Начало/Конец файла указатель записи реально ни на что не указывает. Если за оператором SET следует оператор NEXT, то считывание записей производится с начала файла к его концу. Если за оператором SET следует оператор PREVIOUS, то считывание записей производится с конца файла к его началу. Считывая записи в одном направлении операторами NEXT или PREVIOUS, невозможно вновь попасть в Начало/Конец файла. Для этого нужно еще раз обратиться к оператору SET.

Тот же механизм “нечеткой логики” задействован при выборе начальной записи по

значению индекса. Если запись, удовлетворяющая значению индекса, найдена, то оператор SET выставляет на нее указатель записи для последовательной обработки. В этом случае как оператор NEXT, так и оператор PREVIOUS прочтут ее в качестве первой записи.

Если, однако, запись, удовлетворяющая значению индекса, не найдена, то оператор SET выставляет указатель записи для последовательной обработки “между” последней записью, у которой индекс меньше (больше, для убывающей последовательности значений индекса) заданного значения, и первой, у которой индекс больше (меньше, для убывающей последовательности значений индекса) заданного значения. В этом случае операторы NEXT и PREVIOUS прочтут - в качестве первой записи - разные записи. Оператор NEXT считает следующую запись в индексной последовательности, тогда как PREVIOUS - предыдущую.

Преимущество механизма “нечеткой логики” проявляется при использовании составного ключа.

Рассмотрим пример:

Sample	FILE, DRIVER('TopSpeed'), PRE(Sam)	
FieldsKey	KEY(Sam:Field1, Sam:Field2), DUP	! индекс по Field1 и Field2
Record	RECORD	
Field1	LONG	
Field2	STRING(10)	
Field3	DECIMAL(7,2)	

CODE

OPEN(Sample, 42h)	! доступ: Чт./Зп. + Нет запрета
CLEAR(Sam:Record)	! Очистить буфер записи
Sam:Field1 = 10	! Первая компонента индекса
SET(Sam:FieldsKey, Sam:FieldsKey)	! В индексной послед-ти; “10-пусто”-начало
LOOP	! Обработать каждую запись
NEXT(Sample)	! по одной
IF ERRORCODE() THEN BREAK.	! выйти по концу файла
IF Sam:Field1 <> 10	! Конец группы записей?
BREAK	! “да” - выход из цикла
END	
	! Операторы обработки записи
END	! Конец цикла обработки

Сначала очищается буфер записи: обнуляются поля Sam:Field1 и Sam:Field3, заполняется пробелами поле Sam:Field2. Первой компоненте индекса Sam:FieldsKey присваивается значение, которое совпадает с соответствующим значением в записях, какие нужно обработать. Оператор SET устанавливает порядок доступа к записи: по индексу, а начало последовательной обработки - по значению индекса. В примере - число 10 для поля

Sam:Field1 и строка пробелов для поля Sam:Field2.

<u>Записи Файла Sample:</u>	<u>Номер индекса</u>	<u>Field1</u>	<u>Field2</u>	<u>Field3</u>
	1	5	ABC	14.52
	2	5	DEF	14.52
<u>указатель после SET &gt;&gt;</u>				
	3	10	ABC	14.52
	4	10	ABC	29.04
	5	10	DEF	14.52
	6	15	ABC	14.52
	7	15	DEF	14.52

Поскольку нет ни одной записи, которая точно соответствовала бы индексу, то оператор SET устанавливает указатель записи так, как показано выше. Значение второй записи - 5'DEF' меньше, чем 10'пробелы', а значение третьей записи - 10'ABC' больше, чем 10-пробелы, поэтому указатель записи установлен "между" ними. На первом шаге цикла LOOP оператор NEXT считывает третью запись. Оператор IF завершает цикл обработки при считывании оператором NEXT шестой записи.

Существует явное отличие между номером записи и номером индекса. Номер записи определяет относительное положение записи в файле данных. Функция POINTER(Метка оператора FILE) возвращает номер записи. Номер индекса определяет относительное положение записи в индексной последовательности. Функция POINTER(Метка оператора KEY) возвращает номер индекса.

Файл данных может выглядеть, например, так (не следует думать, конечно, что номера записей и индексов хранятся в файле):

<u>Файл Sample:</u>	<u>Физ. номер</u>	<u>номер индекса</u>	<u>Field1</u>	<u>Field2</u>	<u>Field3</u>
	1	3	10	ABC	14.52
	2	6	15	ABC	14.52
	3	5	10	DEF	14.52
	4	2	5	DEF	14.52
	5	4	10	ABC	29.04
	6	7	15	DEF	14.52
	7	1	5	ABC	14.52

Формы оператора SET, в которых для выбора начальной записи последовательной обработки используется номер, весьма похожи, поэтому нужно ясно осознавать какого типа номер используется (номер записи или номер индекса).

SET(Sample,1)	! По номеру записи, номер записи=1, номер индекса=3
SET(Sam:FieldsKey,1)	!По номеру индекса, номер индекса=1, номер записи=7
Sam:Field1 = 10	
Sam:Field2 = 'ABC'	
SET(Sam:FieldsKey,Sam:FieldsKey,5)	
! По номеру индекса, номер индекса=4, номер записи=5	

Последняя форма оператора SET устанавливает указатель на конкретную запись в последовательности записей с одинаковыми ключевыми значениями. В последовательности таких записей оператор SET выбирает ту, номер которой совпадает со значением его третьего параметра. Данная форма оператора SET становится незаменимой при обработке файлов, в которых присутствуют записи с одинаковым значением ключа, когда нужно начать обработку с одной из них.

## **Произвольный доступ к файлу**

Оператор GET - единственный в языке оператор, с помощью которого осуществляется произвольный доступ к файлу. В отличие от оператора SET, оператор GET либо считывает запись из файла, либо выдает сообщение об ошибке. Для оператора GET не используется механизм “нечеткой логики”.

Существует три формы оператора GET. С их помощью можно считывать записи по значению индекса, по номеру записи и по номеру индекса.

Sam:Field1 = 15	
Sam:Field2 = 'ABC'	
GET(Sample,Sam:FieldsKey)	!Считать; номер индекса=6, номер записи=2
GET(Sample,1)	!Считать; номер записи=1, номер ключа=3
GET(Sam:FieldsKey,1)	!Считать; номер ключа=1, номер записи=7

В первом случае оператор GET считывает первую (по значению индекса) запись файла, индексное значение которой совпадает со значением индекса, заданного к моменту исполнения оператора GET. Во втором случае считывается первая (по номеру записи) запись файла. В третьем - считывается первая (по номеру индекса) запись файла.

Когда запись с заданным значением ключа не найдена в файле, оператор GET выдает сообщение об ошибке. Поэтому для сложного индекса до исполнения оператора GET должны быть заданы все компоненты.

Оператор GET не влияет на последовательную - SET/NEXT или SET/PREVIOUS - обработку файла. Это означает, что исполнение оператора GET в процессе последовательной обработки не приводит к изменению указателя записи для последовательной обработки.



```
SET(Sam:FieldsKey)           ! На начало файла
LOOP                         ! Обработать каждую запись по индексу
  NEXT(Sample)               ! Считать следующую (по значению индекса)
  IF ERRORCODE() THEN BREAK. ! Выйти по концу файла
    ! операторы последовательной обработки записи
  GET(Sam:FieldsKey,1)       ! Считать первую (по значению индекса)
    ! операторы обработки записи для произвольного доступа
END
```

В этом примере записи файла обрабатываются в индексной последовательности (по значению индекса). После обработки текущей записи считывается и обрабатывается первая - по значению индекса - запись файла. Это ни коим образом не влияет на процесс последовательной обработки, так что на каждом шаге цикла оператор NEXT будет считывать следующую (по значению индекса) запись, несмотря на “вмешательство” оператора GET.

## Резюме

---

- Существуют два способа, какими можно считывать записи из файла: последовательный доступ к записи и произвольный доступ к записи.
- Объявления KEY и INDEX определяют особый порядок следования записей в файле, к которому относятся эти объявления.
- Ключ поддерживается динамически и его можно использовать в любой момент. Индекс динамически не поддерживается и его нужно создавать перед использованием.
- “Динамический” индекс позволяет во время исполнения программы определять порядок следования записей при обработке файла.
- Оператор SET устанавливает порядок доступа и начало последовательной обработки. Оператор SET должен исполняться перед первым исполнением оператора NEXT или PREVIOUS.
- Оператор SET использует механизм “нечеткой логики” для определения начала последовательной обработки. Указатель записи устанавливается либо на конкретную запись, либо “между” записями, из которых одна еще не удовлетворяет, а другая уже не удовлетворяет параметру начальной установки оператора SET.
- Номер записи и номер ключа записи - два существенно различных понятия. Необходимо избегать путаницы при их использовании.
- Оператор GET осуществляет произвольный доступ к записи в файле.
- Оператор GET не влияет на последовательную SET/NEXT или SET/PREVIOUS обработку записей.







## Совместное использование файлов

Программирование приложений для баз данных стремительно смещается в сторону их сетевой реализации. Поскольку все большее число фирм подключают свои персональные компьютеры (ПК) к Локальным Вычислительным Сетям (ЛВС), то автономные приложения дорабатываются для их применения в многопользовательских средах. В крупных фирмах приложения для универсальных ЭВМ разукрупняются и переписываются для их функционирования в ЛВС. С появлением распределенных и многозадачных операционных систем для ПК даже для автономных компьютеров необходимо создавать приложения с учетом требований коллективного доступа. В этой главе речь пойдет о средствах языка Clarion, которые позволяют создавать приложения, предназначенные для применения в многопользовательской среде.

### Открытие файлов

---

Чтобы получить доступ к файлу, его сначала нужно открыть. Функцию открытия выполняют операторы OPEN и SHARE. Функционально эти операторы эквивалентны, единственное отличие - значение по умолчанию второго параметра (режим доступа) каждого оператора.

Режим доступа определяет предоставляемый пользователю тип доступа к открытому им файлу и тип доступа, разрешенный другим пользователям файла. Код доступа к файлу на уровне ДОС формируется путем сложения значений этих типов. Значения режима доступа:

	<u>Доступ</u>	<u>Десятич.</u>	<u>Шестнадц.</u>
Доступ Пользователя	Только Чтение	0	0h
	Только Запись	1	1h
	Чтение/Запись	2	2h
Доступ Со-пользователей	Полный Запрет	16	10h
	Запрет Записи	32	20h
	Запрет Чтения	48	30h
	Нет Запрета	64	40h

По умолчанию режим доступа в операторе OPEN устанавливается Чтение/Запись + Запрет Записи (22h), при котором исключительное право записи в файл предоставляется пользователю, открывающему этот файл (режим одного пользователя). По умолчанию режим доступа в операторе SHARE устанавливается Чтение/Запись + Нет Запрета (42h), при котором пользователю, открывающему этот файл, исключительных прав не предоставляется (многопользовательский режим). Возможные методы доступа в операторах

OPEN и SHARE приведены ниже:

OPEN(file)	! Чтение/Запись + Запрет Записи
OPEN(file,22h)	! Чтение/Запись + Запрет Записи
SHARE(file,22h)	! Чтение/Запись + Запрет Записи
SHARE(file)	! Чтение/Запись + Нет Запрета
SHARE(file,42h)	! Чтение/Запись + Нет Запрета
OPEN(file,42h)	! Чтение/Запись + Нет Запрета
OPEN(file,40h)	! Только Чтение + Нет Запрета
SHARE(file,40h)	! Только Чтение + Нет Запрета

Здесь представлены три часто встречающихся метода доступа. В многопользовательских приложениях обычно применяется метод Чтение/Запись + Нет Запрета (42h), при котором пользователю разрешен полный доступ к файлу. Метод Только Чтение + Нет Запрета (40h) используется в тех случаях, когда данному пользователю нужно только просмотреть файл, а у другого пользователя возникла потребность произвести запись в файл.

## Проверка на вмешательство

---

При совместном доступе к файлу главное внимание приходится уделять возможности одновременного изменения записей в файле несколькими пользователями. Процедура “проверка на вмешательство” (“concurrency checking”) предохраняет файл данных от “порчи” при одновременном изменении записи. Смысл проверки на вмешательство состоит в выяснении: изменилась или нет запись на диске за время пока пользователь редактировал в памяти ее копию.

Ясно, что для добавления записи к файлу проверка на вмешательство не требуется. Если два пользователя попытаются добавить одну и ту же запись в файл с уникальным ключом (оператор KEY без атрибута DUP), то вторичное исполнение оператора ADD приведет к выдаче ошибки “две записи с одним ключом” (“Creates Duplicate Key”) без добавления записи. Если же для файла разрешено дублирование ключа (KEY с атрибутом DUP), то нет общего метода, по которому программа может отличить обдуманное (корректное) дублирование записи от случайного (некорректного). Проверка на вмешательство не требуется и при удалении записи. Если первый пользователь удалил запись, то другой попытается удалить уже удаленную запись.

Проверка на вмешательство необходима тогда, когда пользователь вносит изменения в запись. Внесение изменений - редактирование - происходит в три этапа: получение записи, ее редактирование, занесение в файл. Может случиться, что за время выполнения пользователем второго этапа, другой пользователь успеет отредактировать эту же запись и записать ее в файл. Теперь, когда первый пользователь готов занести свою редакцию записи на диск, там уже находится “чужая” запись. Должен ли он проигнорировать чужую запись и занести свою? Если оба пользователя вносили изменения в разные элементы данных

(поля) записи, то первый пользователь не имеет права на “затирание” чужой записи. Даже если и тот и другой пользователь вносили одни и те же изменения, то и в этом случае первый пользователь должен знать, что кто-то уже проделал его работу.

Простейший способ проверки на вмешательство - это выполнить оператор WATCH перед считыванием записи с диска. Это сигнал файловому драйверу автоматически выполнять проверку на вмешательство и при возникновении конфликта выдавать код ошибки. К сожалению, не все файловые драйверы поддерживают такую возможность.

Самый простой способ проверки на вмешательство без оператора WATCH состоит в том, что в программе нужно:

- 1 Сохранить копию записи до внесения в нее изменений.
- 2 Непосредственно перед записью правок на диск еще раз прочесть запись с диска и сравнить с сохраненным оригиналом.
- 3 В случае совпадения записать пользовательские правки на диск. В противном случае, оповестить пользователя и вывести на экран запись с правками другого пользователя.

Пусть имеют место следующие объявления данных:

```

Sample      FILE,DRIVER('TopSpeed'),PRE(Sam)  ! Объявление файла данных
Field1Key   KEY(Sam:Field1)
Record      RECORD
Field1      LONG
Field2      STRING(10)

Action      LONG          !тип производимых изменений
AddRec      EQUATE(1)
ChangeRec   EQUATE(2)

```

Предположим, что есть некая процедура, которая читает запись из файла, задает тип операции над выбранной записью (Добавить, Изменить или Удалить запись) и вызывает процедуру корректировки. Процедура корректировки работает только с выбранной записью и завершает заданную операцию. Программа, реализующая процедуру корректировки, могла бы выглядеть так:

```

Update      PROCEDURE      ! Процедура корректировки
Screen      WINDOW        ! объявления полей в окне

                                END

SaveQue     QUEUE,PRE(Sav)  ! Очередь хранения записи - копии
SaveRecord  LIKE(Sam:Record),PRE(Sav) ! буфера записи файла

                                END

```

```

SavRecPos  STRING(512)      ! Переменная для хранения положения записи
CODE
OPEN(Screen)
Sav:SaveRecord = Sam:Record      ! Сохраним копию записи
ADD(SaveQue,1)                   ! в первой записи очереди
SavRecPos = POSITION(Sample)      ! Сохраним положение записи
DISPLAY                          ! Отображаем запись на экране
ACCEPT                          ! Цикл обработки полей окна
CASE ACCEPTED() ! Процедуры редактирования отдельных полей окна
OF ?OKButton      ! Поле завершения обработки всего окна
  IF Action = ChangeRec      ! Если операция - Изменить
    Sav:SaveRecord = Sam:Record      ! Храним запись с изменениями
    ADD(SaveQue,2)              ! во второй записи очереди
    GET(SaveQue,1)              ! Получить исходную запись из очереди
    REGET(Sample,SavRecPos) ! Вновь прочитать запись из файла
    IF ERRORCODE()
      IF ERROR() = 'RECORD NOT FOUND' !Кто-то ее удалил?
        Action = AddRec ! тип операции Добавить
        GET(SaveQue,2) !Взять вариант записи из очереди
        Sam:Record = Sav:SaveRecord ! и поместить в буфер
      ELSE
        STOP(ERROR())      ! “Непредвиденная” ошибка
      END
    ELSEIF Sav:SaveRecord <> Sam:Record
      ! Если запись уже изменена другим
      Sav:SaveRecord = Sam:Record
      !пользователем, сохранить ее
      ADD(SaveQue,1)      ! в первом элементе очереди
      DISPLAY      ! и отобразить на экране
      BEEP      ! Оповестить пользователя
      MESSAGE('Changed by another station')
      SELECT(1)      ! и начать снова
      CYCLE      ! с первого поля окна
    ELSE      !Если запись не менялась
      GET(SaveQue,2)      ! Взять редакцию пользователя
      Sam:Record = Sav:SaveRecord      ! и поместить в буфер
    ..
  EXECUTE Action      ! Выполнить запись на диске для:
    ADD(Sample)      ! операции Добавить (AddRec)
    PUT(Sample)      ! операции Изменить (ChangeRec)
    DELETE(Sample)      ! операции Удалить (DeleteRec)
END

```



ErrorCheck	! Проверка на ошибки общего характера
FREE(SaveQue)	! Освободить память “из-под” очереди
BREAK	! и выйти из цикла обработки
..	! Конец LOOP и CASE

Выше представлен простейший вариант проверки на вмешательство без использования оператора WATCH. Исходная запись сохраняется в первом элементе очереди, а указатель записи в переменной типа LONG. После этого пользователь может заняться редактированием записи. Фрагмент программы, в котором реализована проверка на вмешательство, помещен в структуру CASE FIELD() OF ?OKButton. OKButton должно быть полем, по завершению которого пользователь готов к занесению отредактированной записи на диск.

Чтобы определить была ли запись в файле изменена другим пользователем, сначала нужно сохранить во втором элементе очереди QUEUE изменения, сделанные данным пользователем, затем взять копию исходной записи из первого элемента очереди. Для чтения записи из файла воспользуемся сохраненным указателем этой записи. Если запись не обнаружена в файле, - значит, кто-то ее уже удалил. Поэтому отредактированная запись просто добавляется в файл. Если запись найдена, то она сравнивается с сохраненной исходной записью. При несовпадении: найденная запись помещается в первый элемент очереди QUEUE; оповещается пользователь; ему вновь предоставляется возможность редактировать запись. При совпадении (вмешательства не было) - отредактированная пользователем запись извлекается из очереди (второй QUEUE-элемент) и заносится в буфер файла для последующей записи на диск. Это вполне ясный и логичный вариант. Однако, он дополнительно использует три “куска” памяти, каждый объемом буфера записи файла: буфер очереди и два элемента этой очереди (плюс 28 байт заголовка для каждого элемента очереди). Наличие большого числа полей в записи файла потребует значительного объема памяти при реализации этого варианта.

Существует другой вариант проверки на вмешательство. В нем вместо копирования и сохранения исходной записи вычисляется ее контрольная сумма по методу циклического избыточного кода (Cyclical Redundancy Check - CRC). В этом варианте сначала вычисляется контрольная сумма записи, затем пользователь редактирует эту запись, и перед занесением отредактированной записи в файл снова вычисляется контрольная сумма исходной записи. Если значения контрольных сумм не совпадают, то запись в файле была кем-то изменена - было вмешательство. И в данном варианте необходим дополнительный объем памяти для хранения отредактированной записи, поскольку требуется повторное чтение записи с диска в буфер файла.

Ниже представлен пример функции, вычисляющей 16-битовое CRC-значение (контрольную сумму), с ее прототипом в структуре MAP. Подобные функции используются в некоторых стандартных коммуникационных протоколах. Получив на вход байтовый массив, функция вычисляет 16-битовое CRC-значение для этого массива и возвращает его

в переменной типа USHORT (16-битовое беззнаковое).

```

MAP                                ! Прототип функции в структуре MAP
    CRC16(*BYTE[]),USHORT          ! CRC16 ожидает на входе массив байт
END                                ! и возвращает USHORT-значение
!~~~~~
CRC16    FUNCTION(Array)           ! Вычисление 16 битового CRC-значения
CRC      ULONG                     ! Рабочая переменная
CODE
    LOOP X# = 1 TO MAXIMUM(Array,1) ! Цикл по всему массиву
        CRC = BOR(CRC,Array[X#])    ! Обрабатываем текущий байт
    LOOP 8 TIMES                     ! Цикл по каждому биту
        CRC = BSHIFT(CRC,1)         ! Сдвиг значения на 1 бит влево
    IF BAND(CRC,1000000h) ! 24 бит до сдвига был установлен?
        CRC = BXOR(CRC,102100h) ! Если Да, то накладываем "маску"
    .                               ! Конец обоих циклов
RETURN(BAND(BSHIFT(CRC,-8),0000FFFFh)) !Сдвиг; выделим возвращ. значения

```

Проверка на вмешательство с применением CRC функции примет следующий вид:

```

Update      PROCEDURE              ! Процедура корректировки
Screen      WINDOW                  ! объявления полей окна
END
Sav:SaveRecord LIKE(Sam:Record),PRE(Sav),STATIC
            ! Переменная для хранения записи - копии буфера записи файла
PassArray    BYTE,DIM(SIZE(Sam:Record),OVER(Sam:Record)
            !объявим массив поверх Sam:Record
SavRecPos    STRING(512)           !Переменная для хранения положения записи
SavCRC       USHORT                ! Переменная для CRC-кода
CODE
    OPEN(Screen)
    SavCRC = CRC16(PassArray)       ! Храним CRC-значение исходной записи
    SavRecPos = POSITION(Sample)     ! Храним положение записи
    DISPLAY                                       ! Отображаем запись на экране
    ACCEPT                                       ! Цикл обработки полей окна
        CASE ACCEPTED()                  ! Процедуры редактирования полей окна
        OF ?OKButton                     ! завершение обработки всего экрана
            IF Action = ChangeRec        ! Если операция - Изменить
                Sav:SaveRecord = Sam:Record ! Храним запись с изменениями
                REGET(Sample,SavRecPos) ! Вновь читать запись из файла
                IF ERRORCODE()
                IF ERROR() = 'RECORD NOT FOUND' ! Кто-то ее удалил?

```

```

        Action = AddRec      ! тип операции Добавить
        Sam:Record = Sav:SaveRecord
                           ! и поместить в буфер файла
    ELSE
        STOP(ERROR())      ! “Непредвиденная” ошибка
    END
    ELSIF SavCRC <> CRC16(PassArray)
        ! Если запись уже изменена другим
        SavCRC = CRC16(PassArray)! сохраним CRC код для нее
        DISPLAY              ! и отобразим на экране
        BEEP                 ! Оповестить пользователя
        IF MESSAGE(‘Changed by another station’).
        SELECT(1)             ! и начать снова
        CYCLE                 ! с первого поля окна
    ELSE
        ! Если запись не менялась
        Sam:Record = Sav:SaveRecord ! и поместить в буфер файла
    ..
    EXECUTE Action           ! Выполнить запись на диске для:
        ADD(Sample)          ! операции Добавить (AddRec)
        PUT(Sample)          ! операции Изменить (ChangeRec)
        DELETE(Sample)       ! операции Удалить (DeleteRec)
    END
    ErrorCheck              ! типовая проверка ошибочных ситуаций
    BREAK                   ! и выйти из цикла обработки
    ..                      ! Конец LOOP и CASE

```

Заметим, что процедура корректировки в этом примере стала короче, а ее логика - яснее. Появились новые данные: переменная SavCRC - для хранения CRC-значений, и массив PassArray, совмещенный (OVER) в памяти с буфером записи в структуре файла, - для передачи записи - в виде входного массива - в функцию CRC16.

Здесь может возникнуть вопрос: почему переменная Sav:SaveRecord в процедуре объявлена с атрибутом STATIC? Тому имеются четыре причины:

При повторном вычислении контрольной суммы требуется место для хранения отредактированной пользователем записи.

При вызове процедуры или функции память для размещения локальных данных отводится в стеке.

В стеке может не хватить места для хранения буфера записи большого размера. Поэтому память должна быть выделена в статической области.

Статическая память в процедурах и функциях отводится только для структур данных (SCREEN, PULLDOWN, REPORT, FILE и QUEUE).

Конечно, если область хранения объявлена в секции глобальных данных (между ключевыми словами PROGRAM и CODE) или в секции объявления данных MEMBER модуля (между ключевыми словами MEMBER и PROCEDURE или FUNCTION), то она автоматическим становится статической и необязательно объявлять ее с атрибутом STATIC.

## **Блокировка и освобождение записей**

---

Оператор HOLD предотвращает изменение записи другими пользователями во время ее редактирования. Последующие за HOLD операторы GET, NEXT или PREVIOUS при выборе записи устанавливают флаг “запись заблокирована”. Если другой пользователь попытается получить доступ к такой записи, то он получит уведомление о том, что с записью уже работают - “запись заблокирована”. Запись блокируется до тех пор, пока она не будет освобождена: явным образом - оператором RELEASE; неявным образом - операторами PUT, DELETE или обращением за другой записью файла.

Благодаря реализации понятия сменяемого драйвера файла, язык Clarion может взаимодействовать с разными файловыми системами. Каждая файловая система по-своему организует блокирование записи. Поэтому действительный результат выполнения оператора HOLD зависит от драйвера файла, который знает какое действие соответствует данной файловой системе. В некоторых файловых системах блокирование записи допускает чтение ее другими пользователями. В других - блокирует любой доступ к записи. Одни файловые системы освобождают заблокированную запись при сбое, другие - нет, оставляя запись помеченной как заблокированная. Конкретные действия, связанные с оператором HOLD, описаны в документации на каждый драйвер файла.

Может возникнуть желание отказаться от выполнения процедуры проверки на вмешательство, блокируя запись с момента доступа к ней и освобождая ее после записи в файл. Но хорошо ли это? В зависимости от реализации оператора HOLD в используемой файловой системе, ответ может быть и Да и Нет.

“Нет” - если файловая система запрещает любой доступ к записи другим пользователям или возможный сбой системы оставляет запись в состоянии блокировки. Однако, это не значит, что нужно вообще отказаться от использования оператора HOLD. Оператор HOLD не рекомендуется применять там, где запись блокируется на период общения пользователя с клавиатурой (длительность которого непредсказуема). Его с успехом можно применить в процедуре проверки на вмешательство в момент самой проверки, чтобы быть уверенным, что никто не изменит запись в период между вторым GET и тем PUT, который записывает правки пользователя на диск.

С использованием оператора HOLD, приведенный выше пример изменится только в

части CASE FIELD() OF ?OKButton.

```

OF ?OKButton                                ! Поле завершения обработки всего окна
  IF Action = ChangeRec                      ! Если операция - Изменить
    Sav:SaveRecord = Sam:Record             ! Храним запись с изменениями
    HOLD(Sample,1)                          ! Блокировать на время проверки
    REGET(Sample,SavRecPos)                 ! Вновь прочитать запись из файла
    IF ERRORCODE()
      IF ERROR() = 'RECORD ALREADY HELD'
        ! Кто-нибудь уже заблокировал?
        BEEP                               ! Оповестить пользователя
        SHOW(25,1,'Held by another station')
        SELECT(1) ! и начать снова
        CYCLE                               ! с первого поля
      ELSIF ERROR() = 'RECORD NOT FOUND'
        ! Кто-нибудь ее удалил?
        Action = AddRec                    ! тип операции Добавить
        Sam:Record = Sav:SaveRecord
        ! и поместить в буфер файла
      ELSE
        STOP(ERROR()) ! "Незапланированная" ошибка
      END
    ELSIF SavCRC <> CRC16(PassArray) !Если запись уже изменена другим
      RELEASE(Sample) ! Освободить запись
      SavCRC = CRC16(PassArray) ! сохранить новый CRC код
      DISPLAY        ! отобразить на экране изменения
      BEEP           ! Оповестить пользователя
      IF MESSAGE('Changed by another station').
      SELECT(1)      ! и начать снова
      CYCLE          ! с первого поля экрана
    ELSE
      ! Если запись не менялась
      Sam:Record = Sav:SaveRecord ! и поместить изменения в буфер
    ..
  EXECUTE Action                               Совместное использование файлов
  ! Выполнить запись на диске для:
  ADD(Sample) ! операции Добавить (AddRec)
  PUT(Sample) ! операции Изменить (ChangeRec)
  DELETE(Sample) ! операции Удалить (DeleteRec)
END
ErrorCheck ! типовая процедура обработки ошибок
BREAK      ! и выйти из цикла

```

В этом примере запись блокируется ровно на столько, сколько времени требуется для выяснения - изменилась ли исходная запись, и для записи правок пользователя на диск. Если кто-либо уже заблокировал запись, то пользователь уведомляется об этом и ему предоставляется возможность еще одной попытки. Если при многократном повторе выясняется, что запись все еще заблокирована, то, вероятно, она осталась в состоянии блокировки из-за сбоя в работе файловой системы другого пользователя. В этом случае запись может быть освобождена только средствами файловой системы, в которой произошел сбой. Можно написать программу, которая обработает такую ситуацию, но она будет зависеть от конкретной файловой системы, а здесь рассматривается типовой пример.

Если возникновение конфликтов при обновлении записи угрожает дальнейшей работе, то оператор HOLD необходим для удержания контроля над записью во время ввода данных пользователем. При этом, однако, может возникнуть неприятная ситуация, когда записи остаются заблокированными при сбое системы. Разблокировать записи приходится либо вручную, с помощью утилит файловой системы, либо при помощи программы, специально созданной для обработки таких ситуаций в данной файловой системе. Другая неприятность, обусловленная таким применением оператора HOLD, возникает когда файловая система запрещает другим пользователям читать заблокированные записи. Записи будут время от времени “исчезать” и “вновь появляться” по мере того, как пользователи блокируют или освобождают их. Поэтому, если нет необходимости, следует избегать использовать оператор HOLD таким образом.

Для того, чтобы использовать этот метод, необходимо внести оператор HOLD в процедуру, которая реально получает записи из файла. В большинстве случаев такая процедура будет выводить на экран разного рода таблицы просмотра записей. Таблицы просмотра можно отображать на экране с помощью окна списка - LIST. Следующий пример демонстрирует это.

```

OF ?List                                !окно списка
CASE EVENT()
OF EVENT:Accepted                       !выбираем запись
  GET(TableQue,CHOICE())                 !берем из очереди номер записи
  HOLD(Sample,1)                         !выдаем HOLD
  REGET(Sample,Que:RecPosition)          ! и получаем запись из файла
  IF ERROR() = 'RECORD ALREADY HELD' !кто-то уже блокировал?
    BEEP                                ! оповестить пользователя
    IF MESSAGE('Held by another station').
    SELECT(?List)                        ! и попробовать снова
    CYCLE
  ELSE                                  !никто не блокировал
    Action = ChangeRec !установить признак изменения
    Update              ! и вызвать процедуру корректировки
  END

```

! Фрагмент программы обработки других клавиш  
END

Такая технология сильно упрощает программу корректировки:

```
Update      PROCEDURE                      !процедура корректировки
Screen      WINDOW                        !объявления полей окна
            END
CODE
OPEN(Screen)
DISPLAY
ACCEPT      !отобразить запись
            !цикл обработки полей окна
CASE FIELD()
    !операторы обработки отдельных полей окна
OF ?OKButton      !кнопка завершения ввода данных?
CASE EVENT()
OF EVENT:Accepted
    EXECUTE Action      !выполнить запись на диск
        ADD(Sample) ! если Action = 1 (AddRec)
        PUT(Sample) ! если Action = 2 (ChangeRec)
        DELETE(Sample) ! если Action = 3 (DeleteRec)
    END
    ErrorCheck      !типовая процедура обработки ошибок
    BREAK           ! и выйти из цикла
..      .           !конец цикла и CASE
```

Оператор HOLD позволяет пользователю блокировать только одну запись в файле. Если требуется обновить несколько записей в файле, запретив при этом другим пользователям вносить изменения в обновляемые записи, то необходимо заблокировать (оператором LOCK) файл.

## Блокирование и разблокирование файлов

Оператор LOCK исключает возможность доступа другим пользователям к записям файла до тех пор, пока не будет выполнен оператор UNLOCK. Как и для оператора HOLD, результат исполнения оператора LOCK зависит от драйвера файла, который предпринимает действия, соответствующие данной файловой системе. В некоторых файловых системах системный сбой автоматически разблокирует файл, в других - файл остается заблокированным. В документации на соответствующий драйвер описано конкретное исполнение оператора LOCK.

Полный запрет доступа другим пользователям к записям заблокированного файла обуславливает сравнительно редкое использование оператора LOCK. Чаще всего оператор

LOCK можно встретить в сочетании с оператором BUILD, чтобы заново создать индекс перед тем, как им воспользоваться (в чем нет необходимости, если индекс - “динамический”). При такой “пакетной” обработке файл приходится блокировать в течение значительного отрезка времени, и ее лучше всего отложить до тех пор, пока все пользователи не закончат всю работу. Значительно чаще оператор LOCK используется при обработке транзакций, но это - предмет отдельного обсуждения.

Если в приложении действительно требуется заблокировать файл, то нужно стараться максимально сузить период времени, в течение которого запрещается доступ к файлу другим пользователям. Желательно, чтобы в промежутке исполнения программы между операторами LOCK и UNLOCK не требовался ввод данных пользователем. Иначе, когда пользователь пойдет “слегка подкрепиться”, файл останется заблокированным. В частности, оператор LOCK должен непосредственно предшествовать оператору BUILD, и файл должен быть разблокирован сразу после завершения выполнения оператора BUILD.

ReportProc	PROCEDURE	
Sample	FILE,DRIVER('TopSpeed'),PRE(Sam)	!объявление файла данных
Field1Key	KEY(Sam:Field1)	
Field2Ndx	INDEX(Sam:Field2)	!индекс
Record	RECORD	
Field1	LONG	
Field2	STRING(10)	
	..	
Report	REPORT	!операторы объявления структуры отчета
	END	
	CODE	
	OPEN(Sample,42h)	! Чтение/Запись + Нет Запрета
	LOCK(Sample,1)	!заблокировать файл
	IF ERROR() = 'FILE IS ALREADY LOCKED'	!проверить не заблокирован ли
	BEEP	!оповестить пользователя
	IF MESSAGE('Locked by another station').	
	RETURN	! и выйти
	END	
	BUILD(Sam:Field2Ndx)	!построить индекс
	UNLOCK(Sample)	!разблокировать файл
	OPEN(Report)	
	SET(Sam:Field2Ndx)	!использовать индекс
	LOOP	
	NEXT(Sample)	
	IF ERRORCODE() THEN BREAK.	
	!операторы формирования отчета	
	END	



Из примера видно, что при открытии файла значение режима доступа устанавливается в 42h (Чтение/Запись + Нет Запрета) - полное совместное использование. Далее производится попытка заблокировать файл на время равное одной секунде. Если попытка удалась, то немедленно выполняется оператор BUILD. При неудачной попытке происходит оповещение пользователя и возврат из процедуры. По завершению исполнения оператора BUILD оператор UNLOCK вновь предоставляет другим пользователям доступ к файлу, а процедура формирования отчета использует индексный порядок доступа к записям.

## Взаимная блокировка

Есть две формы взаимной блокировки. Первая - когда два пользователя пытаются в разном порядке заблокировать одну и ту же группу файлов. Вот примерный сценарий:

Пользователь А	блокирует файл А
Пользователь Б	в это же время блокирует файл Б
Пользователь А	пытается заблокировать файл Б и не может, поскольку Б его уже заблокировал
Пользователь Б	пытается заблокировать файл А и не может, поскольку А его уже заблокировал

Пытаясь получить контроль над файлами, оба пользователя “повисают”. Выход из данной ситуации состоит в принятии на программном уровне простого соглашения: блокировку файлов осуществлять всегда в одном и том же порядке (например, по алфавиту), при этом отслеживать блокировки со стороны других пользователей. Проиллюстрируем этот принцип примером:

```
LOOP
  LOCK(FileA,1)      ! Попытка заблокировать на 1 секунду
  IF ERROR() = 'FILE IS ALREADY LOCKED'
    CYCLE            ! и попытаться снова
  END
  LOCK(FileB,1)      ! Попытка заблокировать на 1 секунду
  IF ERROR() = 'FILE IS ALREADY LOCKED'
    UNLOCK(FileA)    !разблокировать уже заблокированный файл
    CYCLE            ! и попытаться снова
  END
  BREAK              !выйти из цикла когда заблокированы оба файла
END
```

Легко понять, что будут заблокированы оба файла. Если FileA уже заблокирован другим пользователем, то сначала в выполнении цикла возникает односекундная пауза, затем - следующий шаг цикла. Пауза дает другому пользователю шанс завершить свои действия.

Если первый оператор LOCK был исполнен, то производится попытка заблокировать FileB. Если FileB уже заблокирован другим пользователем, то сначала немедленно разблокируется FileA - для его использования другим пользователем, - затем - пауза и последующий шаг цикла. В приведенном примере выход из цикла (оператор BREAK) произойдет только тогда, когда оба файла будут успешно заблокированы.

Вторая форма взаимной блокировки - когда “некорректно” применяются операторы HOLD и LOCK. В некоторых файловых системах операторы HOLD и LOCK совершенно независимы, поэтому возможны ситуации когда один пользователь блокирует запись в файле, а другой - блокирует тот же файл. Первый не может ни вернуть запись в файл, ни даже освободить ее; при этом второй пользователь не может отредактировать заблокированную запись.

Существуют два способа предотвращения таких ситуаций:

Можно договориться о корректном применении операторов HOLD и LOCK. Это вынуждает использовать либо оператор HOLD (что более распространено), либо оператор LOCK. Такая договоренность должна соблюдаться во всех приложениях, в которых осуществляется запись в общую группу файлов.

Можно условиться отслеживать блокируемые записи файла когда он заблокирован. При этом подразумевается, что заранее известно как нужно поступать с взаимно заблокированной записью при ее обнаружении.

Первый способ применяется значительно чаще. Второй - заставляет погружаться в область программирования, которая успешнее применяется при обработке транзакций, где осуществляется отслеживание блокируемой записи.

## Резюме

---

- Только после открытия файла разрешается доступ к его записям.
- Режим доступа определяет тип доступа, который ДОС предоставляет как пользователю, открывшему файл, так и другим пользователям.
- Возможность нескольким пользователям одновременно иметь доступ к одной и той же записи должна всегда быть в центре внимания при программировании для монгопользовательской среды.
- Проверка на вмешательство осуществляется для того, чтобы пользователь своими правками не мог “затереть” записи, откорректированные другими пользователями.
- Очень часто оператор HOLD используется совместно с проверкой на вмешательство. Он запрещает другим пользователям вносить изменения в запись в момент, когда осуществляется проверка на неизменность ее исходного значения.
- Оператор LOCK чаще всего используется для обеспечения исключительного права доступа к файлу в процессе создания индекса файла.
- Взаимная блокировка - вопрос программирования, который проще всего разрешается посредством неукоснительного соблюдения соглашений по написании программ.







## ***ЧАСТЬ II***

### ***РЕСУРСЫ ДЛЯ УГЛУБЛЕННОГО ПРОГРАММИРОВАНИЯ***

## **НАСТРОЙКА И РАСШИРЕНИЕ СРЕДЫ**

### **Введение**

Среда Разработки Clarion настроена оптимальным образом для разработки приложений. Однако, для того чтобы удовлетворить вашим конкретным требованиям, ее можно расширить. Можно добавлять пункты меню для вызова апплетов (небольших программ, выполняющих одну задачу) или внешних инструментальных средств, которые вы хотите использовать. Среда Разработки Clarion разбита на отдельные апплеты, к которым можно обращаться для выполнения ими определенной функции. Например, чтобы открыть и редактировать файл шаблонов, можно вызвать редактор исходных текстов (CWedt).

Управляет расширением файл установок конфигурации -CW20.INI . Это стандартный для Windows .INI-файл, который в Clarion можно модифицировать с помощью операторов GETINI и PUTINI.

Как и большинство приложений для Windows Среда Разработки Clarion при запуске считывает .INI файл и по завершении работы записывает в него текущие установки. При редактировании этого файла имейте это в виду. Если вы внесете какие-либо изменения в него в то время, когда Clarion запущен, то при завершении его работы эти изменения будут перекрыты текущими установками. По этой причине пользуйтесь для изменения .INI файла редактором Notepad или любым другим текстовым редактором.

**Замечание:** Настоятельно рекомендуем прежде чем вносить какие-либо изменения сделайте резервную копию .INI файла.



## **Неизменяемые разделы файла CW20.INI**

### **User Information (данные о пользователе)**

---

Среда Разработки Clarion хранит информацию о пользователе (фамилию, компанию, серийный номер и ключевой код) в разделе [user] файла .INI .

Вам не следует изменять данные в этом разделе.

### **Paths (пути к каталогам)**

---

В разделе [paths] Среда Разработки Clarion хранит данные о последнем открывавшемся проекте. Эта информация обновляется при каждом завершении работы Среды .

Этот раздел файла можно изменять, но при каждом завершении работы Среды он будет ею снова изменен.

Пример:

```
[paths]  
prjfile=C:\CW20\APPS\MYAPP.APP
```

### **Опции Среды**

---

В разделе [Environment] файла .INI задаются два параметра: увеличивается или нет до максимального размер окна приложения и используется по умолчанию или нет режим “быстрого старта”.

Этот раздел файла можно изменять, но при каждом завершении работы Среды он будет ею снова изменен.

Пример:

```
[Environment]  
maximized=off  
quickstart=off
```

### **Апплеты Clarion**

---

В Среде Разработки Clarion есть несколько встроенных апплетов, к которым можно обращаться из определяемого пользователем меню (User Defined Menu) или в определяемой пользователем карточке - закладке (User Defined Tab) в одном из диалоговых окон работы с файлами (New, Open, or Pick). Эти апплеты “регистрируются” в разделе [Clarion Applets] файла настроек (.INI).

Этот раздел файла не следует изменять. Апплеты описываются далее в этой главе, поэтому

вам будет известно к каким из них можно обращаться из меню или на закладке.

Апплеты, помеченные звездочкой (\*), предназначены для внутренних целей и отдельно к ним обращаться нельзя. Они приведены в списке только для сведения.

Cwedt	Редактор исходных текстов
Cwcif16	Интерфейсный модуль компилятора
Cwscr16	Форматер Окон*
Cwprj16	Редактор проекта
Cwrpt16	Форматер печатных документов*
Cwadm16	Редактор Словаря данных
Cwgen16	Генератор Приложений
Cwasv16	Служебные функции Генератора Приложений *
Cwfrm16	Редактор Формул*
Cwbrw16	Менеджер баз данных
Cwqck16	Мастер “быстрого старта”*

Эти апплеты перечислены в разделе [Clarion Applets] файла .INI.

Например:

[Clarion Applets]

\_version=8

CWedt16=on

CWcif16=on

CWscr16=on

CWprj16=on

CWrpt16=on

CWadm16=on

CWgen16=on

CWasv16=on

CWfrm16=on

CWbrw16=on

CWqck16=on

## Изменение Среды Разработки Clarion

### Указание определенных пользователем приложений

---

Для того, чтобы вызвать приложение или апплет из Среды Разработки Clarion, сначала нужно определить его в разделе [User Applications] файла настроек CW20.INI. Тем самым приложение “регистрируется” в Среде. Чтобы добавить свое приложение, измените раздел [User Applications] следующим образом:

AppReference=CommandLine

AppReference имя под которым будет фигурировать ваше приложение.

CommandLine Команда, которая должна быть выполнена. Она может содержать макрорасширение %f или %a.

Пример:

[User Applications]

notepad=notepad %f

wordpad=wordpad %f

notepad.2=notepad c:\windows\win.ini

notepad.3=notepad %a.txt

**Замечание:** Если ваше приложение располагается в каталоге, который не указан в PATH или использует длинные имена (Windows 95), нужно использовать досовский эквивалент этих каталогов или имен. Например, если Wordpad находится в каталоге C:\Program Files\Accessories, то нужно указать C:\Progra~1\ACCESS~1\wordpad.

Отметим, что последние две строки указывают notepad.2 и notepad.3. Они определяют, что notepad.2 должен открыть для редактирования файла WIN.INI. А Notepad.3 должен открыть (или создать) файл, используя имя текущего приложения и расширение .TXT. Для того, чтобы допустить возможность одновременного запуска нескольких одинаковых пользовательских приложений и чтобы при этом они вели себя по-разному, можно добавить расширение.

Макрокоманда расширения %f, показанная в примере обозначает текущее имя файла. Оно передается как параметр командной строки прикладной программе. Допустимые макрокоманды расширения, которые можно использовать как параметры командной строки:

%f Имя файла и путь

%-f Только имя файла

%a Текущее приложение (\*.APP) или проект (\*.PRJ)

Дополнительно можно добавить к макрокоманде расширение, чтобы заменить расширение передаваемого имени файла. Например, вызов Notepad с параметром %a.txt открыл бы файл, используя имя текущей прикладной программы и расширение TXT. Это дает возможность определить пользовательское приложение, которое можно вызывать для того, чтобы открыть (или создать) текстовый файл, соответствующий текущему имени прикладной программы.

## Добавление пунктов в меню Clarion

---

Чтобы вызывать внешнюю прикладную программу из Среды Разработки Clarion, нужно сначала определить как описано выше прикладную программу в разделе [User Applications] файла CW20.INI. Тем самым приложение “регистрируется” в среде.

Затем, добавляете Меню и определяете пункты в меню в разделе [User Menus] следующим образом:

Пример:

[User Menus]

\_version=1

1=&Editors/&Wordpad|wordpad

2=&Editors/&Notepad|notepad

3=&Utilities/&Calculator|Calculator

4=&Utilities/&Paint|Paint

**Замечание:** Первая строка (\_version=1) обозначает номер версии, используемый для внутренних нужд. Не следует изменять эту строку.

## Добавление пунктов в меню Clarion Setup

---

Можно также добавлять пункты в Setup Menu Среды Разработки Clarion. Чтобы вызывать внешнюю прикладную программу из Среды Разработки Clarion, нужно сначала определить как описано выше прикладную программу в разделе [User Applications] файла CW20.INI. Тем самым приложение “регистрируется” в среде.

Затем, добавляете пункты меню и определяете их в разделе [Setup Menu] как показано ниже в строках 10 и 11. В строке 11 используется определенное пользователем приложение (См. Указание определенных пользователем приложений). Строки 4, 8, и 10 создают разделители, используя синтаксис n=-.

n=Text|Application

nНомер элемента

TextТекст, выводимый в меню.

ApplicationАплет Clarion или приложение, которое следует открыть. Все внешние приложения должны быть указаны в разделе [User Applications]

Пример:

```
[Setup Menu]
_version=10
1=&Editor Options|CWedt
2=&Dictionary Options|CWadm
3=&Application Options|CWgen.1
4=-
5=&Template Registry|CWgen.103
6=Data&base Driver Registry|CWasl.38
7=&VBX Custom Control Registry|CWasl.37
8=-
9=Edit Redirection &File|CWasl.36
10=-
11=Edit WIN.INI|notepad.2
```

**Замечание:** Первая строка (`_version=10`) обозначает номер версии, используемый для внутренних нужд. Не следует ее изменять.

## **Добавление масок файлов выпадающий список типов файлов**

---

Раздел [File Types] определяет маски, используемые в раскрывающемся списке в любом из диалоговых окон для выбора файла (New, Open, or Pick) в Clarion для Windows и апплет или приложение, которое открывает или создает этот файл. Должен быть один дополнительный элемент, в формате ALL=n. Он определяет, что должно использоваться для ВСЕХ файлов, позволяя нескольким типам использовать \*.\* в качестве маски файлов.

**Замечание:** Первая строка (`_version=8`) обозначает номер версии, используемый для внутренних нужд. Не следует ее изменять.

По умолчанию:

```
[File Types]
_version=8
1=Application (*.app)=*.app|CWgen
2=Clarion source (*.clw)=*.clw|CWedt
3=Dictionary (*.dct)=*.dct|CWadm
4=Project (*.prj)=*.prj|CWprj
5=Database files=*.tps;*.dbf;*.dat;*.db|CWbrw
6=All Database files (*.*)=*.|CWbrw
7=All files (*.*)=*.|CWedt
all=7
```

Чтобы добавить собственные маски, измените раздел [File Types] в CW20.INI следующим способом:

**n=DisplayText=Filemask|Application**

**n** номер элемента

**DisplayText** текст, выводимый в раскрывающемся списке.

**Filemask** маска, используемая для списка файлов

**Application** Апплет Clarion или приложение, которое следует открыть. Все внешние приложения должны быть указаны в разделе [User Applications]

Пример:

```
[File Types]
1=Application (*.app)=*.app|CWgen
2=Clarion source (*.clw)=*.clw|CWedt
3=Dictionary (*.dct)=*.dct|CWadm
4=Project (*.prj)=*.prj|CWprj
5=Text (*.txt)=*.txt|notepad
6=Doc (*.doc)=*.doc|wordpad
7=Database files (*.tps;*.dbf;*.dat;*.db)|CWbrw
8=All Database files (*.*)=*.|CWbrw
9=All files (*.*)=*.|CWedt
all=9
```

## **Добавление карточек-закладок в диалоговые окна New, Open и Pick File**

Можно изменить Clarion для Windows любое из диалоговых окон File, для того чтобы пользовательскую карточку закладку.

Чтобы добавить свою карточку закладку измените раздел [Pick Dialog], [New Dialog], или [Open Dialog] в файле CW20.INI следующим образом:

**n=TabText=FileMask|Application**

**n** Номер элемента

**TabText** Текст, выводимый на карточке.

**FileMask** Расширение выводимых имен файлов

**Application** Апплет Clarion или приложение, которое следует открыть. Все внешние приложения должны быть указаны в разделе [User Applications]

**Замечание:** Первая строка (\_version=8) обозначает номер версии, используемый для внутренних нужд. Не следует ее изменять.

Пример:

[Pick Dialog]

```
_version=8  
1=&Application=app|CWgen  
2=&Dictionary=dct|CWadm  
3=&Project=prj|CWprj  
4=Data&base=*&|CWbrw  
5=&Source=clw|CWedt  
6=&Text=txt|notepad  
7=&Doc=doc|wordpad  
8=A&ll=*&|CWasl  
lines=2  
default=-8
```

[Open Dialog]

```
_version=8  
1=&Application=app|CWgen  
2=&Dictionary=dct|CWadm  
3=&Project=prj|CWprj  
4=Data&base=*&|CWbrw  
5=&Source=clw|CWedt  
6=A&ll=*&|CWasl  
7=&Text=txt|notepad  
8=&Doc=doc|wordpad  
lines=2  
default=-2
```

[New Dialog]

```
_version=8  
1=&Application=app|CWgen  
2=&Dictionary=dct|CWadm  
3=&Project=prj|CWprj  
4=&Source=clw|CWedt  
5=&Text=txt|notepad  
6=&Doc=doc|wordpad  
lines=2  
default=-1
```

Последние две строки каждого из этих разделов влияют на карточки-закладки в каждом из диалоговых окон File. Строка lines=2 определяет, что пространство выделяется для двух рядов карточек.





## Спецификации печати

---

По умолчанию, все файлы, печатаемые из Среды имеют заголовок, в котором приводятся имя файла, дата и номер страницы. Печать заголовка управляется установками в разделе [printing] файла настроек INI. По желанию можно изменить этот раздел и печатать другой заголовок. В этом разделе используются следующие макрорасширения:

%F Имя файла  
%D Текущая системная дата  
%T Текущее системное время  
%P Текущий номер страницы

**Замечание:** Первая строка (`_version=1`) обозначает номер версии, используемый для внутренних нужд. Не следует ее изменять.

```
[printing]  
_version=1  
header=File: %F Date: %D Time: %T Page: %P
```

## Установки опций Среды

### Опции автоматического размещения полей

---

В разделе [Auto Populate] задается то, как оконные объекты размещаются при их автоматическом добавлении (двойным щелчком) на основе Fields Box в форматере.

- startX=5**                      **Задаёт начальное положение на оси X (горизонтальной оси) для первого размещаемого объекта.**
- startY=5**                      **Задаёт начальное положение на оси Y (вертикальной оси) для первого размещаемого объекта.**
- incrementX=50**              **Задаёт отступ по горизонтали в условных единицах от объекта до соответствующей ему подсказки (поля типа Prompt).**
- incrementY=16**              **Задаёт расстояние по вертикали в условных единицах между размещаемыми автоматически объектами.**
- alignment=top**              **Задаёт выравниваются ли объекты по вертикали относительно соответствующих им подсказок по верхнему, нижнему краю или центру.**

Пример:

```
[Auto Populate]
startX=5
startY=5
incrementX=50
incrementY=16
alignment=top
```

### Опции словаря

---

Опции словаря данных устанавливаются в разделе [Administrator] файла INI. Эти опции устанавливаются в самой Среде, но можно поменять их и прямо в INI файле.

Default driver=DriverName

Файловый драйвер, используемый по умолчанию для новых файлов.

Set threaded=on|off

Указывает, добавляется или нет по умолчанию в определения новых файлов атрибут THREAD.

Assign message=on|off

Указывает, используется ли описание, задаваемое вами при объявлении поля, в качестве параметра текст в атрибуте MSG.

Order files=on|off

Указывает, сортируется ли список файлов в алфавитном порядке или в порядке создани описаний файлов.

Display field type=on|off

Задает, высвечивается ли в списке полей тип поля.

Display field prefix=on|off

Задает, высвечивается ли в списке полей префикс.

Display field picture=on|off

Задает, высвечиваются ли в списке полей шаблоны полей.

Display field description=on|off

Задает, высвечиваются ли в списке полей описания полей.

Display key type=on|off

Задает, высвечивается ли в списке ключей тип ключа.

Display key prefix=on|off

Задает, высвечивается ли в списке ключей префикс.

Display key description=on|off

Задает, высвечивается ли в списке ключей описание ключа.

Display key primary=on|off

Задает, высвечивается ли в списке ключей “Primary” для первичного ключа.

Display key unique=on|off

Задает, высвечивается ли в списке ключей “Unique” для ключа, не допускающего дублирование.

Display key attributes=on|off

Задает, высвечиваются ли в списке ключей остальные атрибуты ключа.

Display file driver=on|off

Задает, высвечивается ли в списке файлов название файлового драйвера.

Display file prefix=on|off

Задает, высвечивается ли в списке файлов префикс.

Display file description=on|off

Задает, высвечивается ли в списке файлов описание.

Пример:

[Administrator]

Default driver=TOPSPEED

Set threaded=on

Assign message=on

Order files=off

Display field type=on

Display field prefix=off

Display field picture=on

Display field description=on

Display key type=off

Display key prefix=off

Display key description=on

Display key primary=off

Display key unique=off  
Display key attributes=off  
Display file driver=off  
Display file prefix=off  
Display file description=on

## Параметры системы управления проектом

---

Параметры системы управления проектом задаются в разделе [Project System] файла настроек. Эти параметры устанавливаются в самой Среде, но можно поменять их и прямо в INI файле.

automake=on|off      Задает, должно ли перед выполнением приложение автоматически компилироваться и компоноваться (если это необходимо).  
autosave=on|off      Задает, должно ли приложение автоматически сохраняться перед компиляцией и компоновкой.  
runmin=on|off      Задает, должно ли окно среды быть автоматически минимизировано на время выполнения приложения.  
runwait=on|off      Задает, должна ли Система управления проектом на время выполнения приложения командой RUN приостанавливать Clarion для Windows до завершения работы приложения.  
newtype=<extension>      Задает, определен ли в разделе [Make File Types] .INI файла какой-нибудь новый тип для файла описания проекта. Эта строка автоматически добавляется Средой при добавлении нового типа файла описания проекта.

Пример:

```
[Project System]
automake=on
autosave=on
runmin=off
runwait=off
newtype=MAK
```

## Опции Генератора программ

---

Опции Генератора программ задаются в разделе [Application] файла настроек. Эти параметры устанавливаются в самой Среде, но можно поменять их и прямо в INI файле.

CondGeneration=on|off      Чтобы указать, что должны компилироваться только те модули, которые изменились с момента последней компиляции, задайте ON.

DebugGeneration=on off	Указывает, записывается или нет в текстовый файл протокол событий для Генератора и включает или выключает регистрацию. В случае возникновения критической ошибки при работе Генератора этот протокол облегчает обнаружение места возникновения проблемы. Имя файла протокола задается в пункте Debug Filename. Это особенно полезно при разработке шаблонов.
Repeat Procedures=on off	Указывает, высвечивает или нет Генератор в поле-индикаторе степени выполнения задания (progress box) имена встречаемых в процессе генерации исходного текста функций.
PopulateMain=on off	Указывает, записывает или нет Генератор процедуры в главный модуль исходного текста. Когда эта опция установлена в off, главный модуль исходного текста содержит только главную процедуру, глобальный программный код, стандартные для каждого приложения процедуры и функции. Все остальные процедуры располагаются в других файлах.
RequireDictionary=on off	Указывает, должно ли каждое новое приложение иметь словарь данных.
MultiUser=on off	Указывает, используются или нет опции управления файлами для проектов, которые выполняют несколько разработчиков. Смотрите приложение Разработка проекта несколькими программистами в Руководстве пользователя.
ApplicationWizard=on off	Указывает, используется или нет по умолчанию Мастер создания Приложений. Эту опцию можно при создании приложения проигнорировать, отметив или убрав отметку в поле Application Wizard в диалоговом окне установки свойств приложения.
ProcedureWizard=on off	Указывает, используется или нет по умолчанию при создании процедуры соответствующий Мастер создания Процедур. Эту опцию можно при создании процедуры проигнорировать, отметив или убрав отметку в поле Procedure Wizard в диалоговом окне выбора типа процедуры (Select Procedure Type).
NameClash=n	Указывает, как Генератор обходится с именами процедур, импортированных из другого приложения, которые конфликтуют именами уже имеющихся процедур.

Возможные варианты:

0 - Запрашивать при первом конфликте

- 1 - Запрашивать альтернативное имя
- 2 - Автоматически переименовывать
- 3 - Замещать предыдущую

`ModuleProcs=n`     Задает число процедур, помещаемых Генератором в один модуль исходного текста. Когда включена условная компиляция (Conditional Generation), эта опция может влиять на время компиляции. Например, указание одной процедуры на модуль означает, что при каждой успешной компиляции пересоздаются только те процедуры, которые изменились с момента последней компиляции, и никакие более. Недостатком является большая потребность в дисковой памяти. В общем, меньшие по размеру модули обрабатываются быстрее.

`Debug File=filename`

Указывает имя файла, в который записывается отладочная информация.

`DisableField=on|off`

Указывает, будут или нет высвечиваться зависящие от шаблона поля-подсказки. Этой опцией нельзя выключить поля-подсказки созданные Управляющими шаблонами.

Пример:

[Application]

`CondGeneration=on`

`DebugGeneration=off`

`Repeat Procedures=on`

`PopulateMain=off`

`RequireDictionary=off`

`MultiUser=off`

`ApplicationWizard=on`

`ProcedureWizard=on`

`NameClash=0`

`ModuleProcs=1`

`Debug File=c:\cw20\apps\mydebug.txt`

`DisableField=off`

## Опции синхронизации словаря

---

Опции синхронизации словаря задаются в разделе [Control Synchronization] файла настроек. Эти параметры устанавливаются в самой Среде, но можно поменять их и прямо в INI файле.

`ApplicationLoad=on|off`     ON устанавливает повторную синхронизацию приложения со словарем данных при каждом открытии файла приложения.

`Windows=on|off`     ON устанавливает повторную синхронизацию окон приложения со словарем данных при каждой

	синхронизации приложения.
Reports=on off	ON устанавливает повторную синхронизацию отчетов со словарем данных при каждой синхронизации приложения.
SyncVariables=on off	ON устанавливает повторную синхронизацию полей для переменных в памяти со словарем данных при каждой синхронизации приложения.
PrimaryAttrOnly=on off	ON устанавливает повторную синхронизацию только для “первичных” атрибутов.
ChangeControlTypes=on off	ON устанавливает возможность изменения типов объектов управления в окне.
SyncDropNonDrop=on off	ON устанавливает возможность изменения поля LIST с атрибутом DROP, на поле LIST без атрибута DROP.
ClearHelpEtc=on off	ON отменяет атрибуты закладки Help для оконного объекта при отсутствии в словаре.
OverrideSize=on off	ON устанавливает возможность изменения размера поля в соответствии со словарем.
Refreeze=on off	ON устанавливает “разморозку” “замороженного” объекта после синхронизации.
IgnoreFreeze=on off	ON устанавливает синхронизацию всех оконных объектов, несмотря на то, что некоторые из них “заморожены”.
SyncListbox=on off	ON устанавливает синхронизацию строк форматирования окон списков (полей типа LIST box).
SyncHeading=0 1 2 3	Задает синхронизацию заголовков: 0 - всегда, 1 для заголовков в словаре, 2 для заголовков в словаре и заголовков окон, 3 никогда не синхронизировать.
WarnDialog=on off	Установка ON задает вывод предупреждающего окна при возникновении проблем во время синхронизации.
WarnOnResize=on off	Установка ON задает вывод в файл журнал предупреждающего сообщения при изменении размеров объекта.
WarnFile=textfile.txt	Задает имя текстового файла в который выводится отчет о синхронизации.
ClearAlrtEtc=on off	Установка ON задает сброс всевозможных атрибутов при их отсутствии в словаре.
ClearFont=on off	Установка ON задает сброс атрибута FONT, если он отсутствует в словаре.
ClearCursor=on off	Установка ON задает сброс атрибута CURSOR, если он отсутствует в словаре.
ClearIcon=on off	Установка ON задает сброс атрибута ICON, если он отсутствует в словаре.
ClearAlert=on off	Установка ON задает сброс атрибута ALRT, если он отсутствует в словаре.
ClearKey=on off	Установка ON задает сброс атрибута KEY, если он отсутствует

в словаре.

ClearColor=on|off

Установка ON задает сброс атрибута COLOR, если он отсутствует в словаре.

ClearTally=on|off Установка

ON задает сброс атрибута TALLY, если он отсутствует в словаре.

Пример:

[Control Synchronization]

ApplicationLoad=on

Windows=on

Reports=on

SyncVariables=off

MajorAttrOnly=off

ChangeControlTypes=on

SyncDropNonDrop=off

ClearHelpEtc=off

ClearAlrtEtc=off

OverrideSize=on

Refreeze=on

IgnoreFreeze=off

SyncListbox=on

SyncHeading=2

WarnDialog=on

WarnOnResize=off

WarnFile=sync.txt

ClearFont=off

ClearCursor=off

ClearIcon=off

ClearAlert=off

ClearKey=off

ClearColor=off

ClearTally=off

ClearKeep=off

ClearBreak=off



## Опции реестра шаблонов

---

Текст на языке шаблонов может быть логически разбит на множество файлов. Clarion для Windows использует эти файлы для создания логически единого набора шаблонов, используемого при генерации приложения. Опции реестра шаблонов предназначены в основном для тех программистов, кто создает свои собственные файлы шаблонов или вносит изменения в стандартные шаблоны. Эти опции устанавливаются в самой Среде, но можно поменять их и прямо в INI файле. Опции реестра шаблонов задаются в разделе [Registry] файла настроек.

Reregister if changed=on|off

Установка ON задает автоматическую повторную регистрацию ваших шаблонов при обнаружении Генератором изменений в них.

Update Template Chain=on|off

Установка ON задает автоматическое изменение фалов шаблонов, когда вносятся изменения в Реестр шаблонов.

Recreate deleted templates=on|off

Установка ON указывает, что Генератор приложений должен повторно сгенерировать файлы .TPL и .TPW из файла REGISTRY.TRF, когда из реестра удаляются файлы.

MessageLines=3

Указывает, что выводится при генерации.

Возможные варианты:

0-Нет сообщений

1-Имена модулей

2- Имена модулей и процедур

3-Все сообщения.

Пример:

Reregister if changed=on

Update Template Chain=off

Recreate deleted templates=off

MessageLines=3

## Опции формatera окон

---

Опции формatera окон задаются в разделе [Window Formatter] файла настроек. Эти опции устанавливаются в самой Среде, но можно поменять их и прямо в INI файле.

grid=on|off Задаёт включена или нет в окне образцов функция Snap to Grid (привязка к сетке).

showtoolbox=on|off Переключает вывод инструментальной Панели Оконных Объектов (Controls toolbox).

showalignbox=on|off Переключает вывод панели выравнивания (Alignment Box).

showpropertybox=on|off Переключает вывод панели свойств (Property Box).

showfieldbox=on|off Переключает вывод панели полей (Fields Box).

tbposx=n Горизонтальная координата инструментальной Панели Оконных Объектов (Controls toolbox)

tbposy=n Вертикальная координата инструментальной Панели Оконных Объектов

tbposw=n Ширина инструментальной Панели Оконных Объектов.

tbposh=n Высота инструментальной Панели Оконных Объектов.

pbposx=n Горизонтальная координата панели свойств (Property Box).

pbposy=n Вертикальная координата панели свойств

pbposw=n Ширина панели свойств.

pbposh=n Высота панели свойств.

abposx=n Горизонтальная координата панели выравнивания (Alignment Box).

abposy=n Вертикальная координата панели выравнивания.

abposw=n Ширина панели выравнивания.

abposh=n Высота панели выравнивания.

Пример:

```
[window formatter]
```

```
grid=off
```

```
showtoolbox=on
```

```
showalignbox=on
```

```
showpropertybox=on
```

```
showfieldbox=on
```

```
tbposx=720
```

```
tbposy=286
```

```
tbposw=116
```

```
tbposh=199
```

```
pbposx=279
```

```
pbposy=122
```

```
pbposw=338
```

```
pbposh=65
```

```
abposx=835
```

```
abposy=126
```

abposw=72  
abposh=187

## Опции формatera отчетов

---

Опции формatera отчетов задаются в разделе [Report Formatter] файла настроек. Эти опции устанавливаются в самой Среде, но можно поменять их и прямо в INI файле.

grid=on|off Задаёт включена или нет в окне образцов функция Snap to Grid (привязка к сетке).

showtoolbox=on|off Переключает вывод инструментальной Панели Объектов (Controls toolbox).

showalignbox=on|off Переключает вывод панели выравнивания (Alignment Box).

showpropertybox=on|off Переключает вывод панели свойств (Property Box).

showfieldbox=on|off Переключает вывод панели полей (Fields Box).

tbposx=n Горизонтальная координата инструментальной Панели Объектов (Controls toolbox)

tbposy=n Вертикальная координата инструментальной Панели Объектов

tbposw=n Ширина инструментальной Панели Объектов.

tbposh=n Высота инструментальной Панели Объектов.

pbposx=n Горизонтальная координата панели свойств (Property Box).

pbposy=n Вертикальная координата панели свойств

pbposw=n Ширина панели свойств.

pbposh=n Высота панели свойств.

abposx=n Горизонтальная координата панели выравнивания (Alignment Box).

abposy=n Вертикальная координата панели выравнивания.

abposw=n Ширина панели выравнивания.

abposh=n Высота панели выравнивания.

Пример:

[report formatter]

grid=off

showtoolbox=on

showalignbox=off

showpropertybox=off

showfieldbox=off

tbposx=540

tbposy=26

```
tbposw=62
tbposh=259
pbposx=113
pbposy=145
pbposw=338
pbposh=65
abposx=540
abposy=76
abposw=88
abposh=229
```

## Опции текстового редактора

---

Опции текстового редактора задаются в разделе [Report Formatter] файла настроек. Эти опции устанавливаются в самой Среде, но можно поменять их и прямо в INI файле.

**Замечание:** Первая строка (`_version=3`) обозначает номер версии, используемый для внутренних нужд. Не следует ее изменять.

`font=ibm` Задаёт шрифт, используемый редактором. По умолчанию - `ibm` - специальный набор символов, поставляемый с Clarion для Windows, в котором имеются символы линий. По желанию можно задать системный моноширинный шрифт, используемый в вашей системе.

`maximized=off` Указывает устанавливается или нет максимальный размер окна редактора

`insert=on` Указывает устанавливается или нет режим вставки для редактора.

Пример:

```
[editor]
```

```
_version=3
```

```
font=system
```

```
maximized=off
```

```
insert=on
```

## Табуляция в редакторе

---

Метки табуляции в редакторе устанавливаются в самой Среде, но можно поменять их и прямо в INI файле, используя следующий синтаксис:





## **CLARION КАК DDE СЕРВЕР**

### **Введение**

Динамический обмен данными (DDE) это очень мощное средство среды Windows, позволяющее пользователю обмениваться данными с другим, отдельно выполняющимся приложением Windows. Среда разработки Clarion представляет собой DDE сервер, к которому можно обращаться из клиентского приложения. Это позволяет создавать программы, которые обращаются к Среде разработки Clarion для выполнения какой-либо задачи или набора задач.

Например, можно написать программу для регистрации новых классов шаблонов в процессе инсталляции набора шаблонов. Другим примером может служить программа, которая управляет всеми вашими проектами и будет выполнять генерацию текста, компиляцию и компоновку нескольких приложений.

Динамический обмен данными основывается на установлении “канала связи” между двумя одновременно выполняющимися приложениями. Одно из этих приложений действует как DDE сервер, передавая данные, а другое как клиент, принимающий эти данные. Одно приложение может быть одновременно и DDE сервером, и клиентом, получая некие данные от одних приложений и передавая какие-то данные другим приложениям. Несколько “каналов связи” может устанавливаться между одним конкретным DDE сервером и несколькими DDE клиентами.

**Когда запускается Clarion для Windows, он устанавливает себя в качестве DDE сервера. Для этого он должен:**

- Используя функцию DDESERVER зарегистрироваться в Windows как DDE сервер.
- Обеспечить приложениям-клиентам возможное обслуживание. Возможное обслуживание и обсуждается в этой главе.
- Когда динамический обмен данными больше не требуется, то с помощью оператора DDECLOSE канал связи закрывается. Таким образом, когда пользователь прекращает работу Clarion для Windows, автоматически прекращается DDE обмен.

**Для того, чтобы выступать в качестве DDE клиента, приложение на языке Clarion должно:**

- Поскольку DDE обслуживание должно быть связано с окном, нужно открыть по крайней мере одно окно. Кроме того, приложение должно установить системному свойству DDETimeout (SYSTEM{Prop:DDETimeOut} = nn)

некое значение `nn` - величину интервала времени (зависящее от запрашиваемого приложением обслуживания).

- Открыть канал с DDE сервером к его клиент, используя функцию `DDECLIENT`. В процессе DDE обмена генерируются особые, неотносящиеся к конкретному полю события и передаются в цикл `ACCEPT` окна, для которого открыт канал обмена.
- Используя оператор `DDEREAD` или оператор `DDEEXECUTE`, запросить от сервера данные.
- Когда DDE обмен больше не нужен, закрыть канал с помощью оператора `DDECLOSE`.

Прототипы DDE функций находятся в файле `DDE.CLW`, который оператором `INCLUDE` нужно включать в структуру `MAP` для всей программы.

Для того, чтобы включить этот файл в Генераторе приложений:

1. Нажмите кнопку `Global`.
2. Нажмите кнопку `Embeds`.
3. Выделите `Inside the Global Map`, затем нажмите кнопку `Insert`.
4. Выделите `Source`, затем нажмите кнопку `Select`.
5. В Текстовом Редакторе, введите:  
`INCLUDE('dde.clw')`
6. Сохраните текст и выйдите из редактора
7. Нажмите кнопку `Close`, затем нажмите кнопку `Ok`.

## Соединение с Clarion для Windows как с DDE сервером

```
Сервер =DDECLIENT('Clarion Win')
```

**Сервер** Переменная типа `LONG`, предназначенная для хранения номера DDE канала данного клиента.

**DDECLIENT** Эта функция возвращает новый номер DDE канала для приложения и темы. Если приложение в данный момент не выполняется, то `DDECLIENT` возвращает ноль (0).

**ClarionWin** Идентификатор для Clarion для Windows.

Таким образом клиентское приложение присоединяется к Среде Clarion как к серверу.



Прежде чем выполнять какие-либо DDE команды нужно установить соединение.

В примере показывается, как можно определить, запущена Среда Clarion для Windows или нет.

Пример:

!Вставляемый в обработку событий окна код - после сгенерированного кода

```
Server = DDECLIENT('ClarionWin')
```

```
IF Server      ! Если сервер работает
```

```
DO ProcedureReturn      ! Return to caller
```

```
ELSIF NOT Flag#      ! признак того, что выполнялась команда RUN
```

```
  RUN('cw15')      ! запустить Среду
```

```
  Flag# =1      ! установить признак того, что RUN выполнялась
```

```
END
```

## Отсоединение от DDE сервера Clarion для Windows

**DDECLOSE**(*Сервер*)

**DDECLOSE** Процедура, прекращающая связь с DDE сервером.

*Сервер* Переменная типа LONG, которая содержит номер канала с DDE сервером.

Ликвидируется DDE канал между клиентским приложением и Средой Clarion как сервером. Прежде чем выполнять какие-либо DDE команды нужно установить соединение.

Пример:

```
Server = DDECLIENT('ClarionWin')      !...исполняемые операторы
```

```
DDECLOSE(Server)
```

## Экспорт словаря в текстовый (TXD) формат

**DDEEXECUTE**(*Сервер*, '[ExportDct(ИмяDct,ТекстФайл,ВыводНаЭкран)]')

**DDEEXECUTE** Посылает командную строку на DDE сервер.

*Сервер* Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.

**ExportDct** Сервисная функция экспорта словаря в текстовый файл.

*ИмяDct* Имя словаря, из которого осуществляется экспорт.

*ТекстФайл*                   Имя текстового файла, в который осуществляется экспорт.  
*ВыводНаЭкран*           Если этот параметр равен (1) или ИСТИНА, то в процессе экспорта на экран выводятся сообщения, если же он равен (0) или ЛОЖЬ, то вывод на экран сообщений запрещается. Если этот параметр опущен, то вывод сообщений подавляется.

Эта сервисная функция позволяет вывести данные из словаря в текстовый файл. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение.

Пример:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT
```

```
CODE
```

```
SYSTEM{PROP:DDETimeOut} = 12000    !перерыв после двух минут
```

```
DDEEXECUTE(Server, '[ExportDct(c:\CW15\APPS\Myapp1.DCT,MYTXD.TXD,0)]')
DO CheckDDEError
```

```
CheckDDEError  ROUTINE
```

```
DDEErrorMsg = ''
```

```
err# = ERRORCODE()
```

```
! проверить нет ли ошибок
```

```
IF err# > 600    ! если ошибка связана с DDE
```

```
    IF err# = 603    ! не выполнена DDEExecute
```

```
        DDEREAD(Server, DDE>manual, 'GetErrorNum', DDEErrorNum)
```

```
        DDEREAD(Server, DDE>manual, 'GetErrorMsg', DDEErrorMsg)
```

```
        MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
```

```
        ELSIF err# = 605    ! ошибка по времени
```

```
            MESSAGE('DDE timeout')
```

```
    ELSE
```

```
        MESSAGE(err#)
```

```
    END
```

```
END
```

## Импорт словаря из текстового (TXD ) формата

**DDEEXECUTE**(Серве, '[ImportDct(ТекстФайл, ИмяDct, ВыводНаЭкран)]')

### DDEEXECUTE

Посылает командную строку на DDE сервер.

*Сервер*

Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.

**ImportDct**

Сервисная функция импорта словаря из текстового файла.

*ТекстФайл*

Имя текстового файла, из которого осуществляется импорт.

*ИмяDct*

Имя словаря, в который осуществляется импорт.

*ВыводНаЭкран*

Если этот параметр равен (1) или ИСТИНА, то в процессе импорта на экран выводятся сообщения, если же он равен (0) или ЛОЖЬ, то вывод на экран сообщений запрещается. Если этот параметр опущен, то вывод сообщений подавляется.

Эта сервисная функция позволяет занести данные в словарь из текстового файла. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение. Кроме того, нельзя импортировать данные в открытый словарь.

Пример:

```
Server          LONG
DDEErrorMsg    CSTRING(300)
DDEErrorNum    USHORT
```

CODE

SYSTEM{PROP:DDETimeout} = 12000 ! перерыв после 2-х минут

```
DDEEXECUTE(Server, '[ImportDct(MYTXD.TXD,c:\CW15\APPS\Myapp1.DCT,0)]')
DO CheckDDEError
```

CheckDDEError ROUTINE

DDEErrorMsg = ''

err# = ERRORCODE()

! проверка ошибок

IF err# > 600 ! ошибка связана с DDE ?

IF err# = 603 !ошибка при выполнении DDEExecute

DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)

DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)

MESSAGE('Error ' & DDEErrorNum & ': ' & CLIP(DDEErrorMsg))

ELSIF err# = 605 ! таймаут

MESSAGE('DDE timeout')

ELSE

```

        MESSAGE(err#)
    END
END

```

## Экспорт данных из файла .APP в текстовый формат (TXA)

**DDEEXECUTE**(*Сервер*, '[ExportApp(*ИмяApp*, *ТекстФайл*, *ВыводНаЭкран*)]')

<b>DDEEXECUTE</b>	Посылает командную строку на DDE сервер.
<i>Сервер</i>	Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.
<b>ExportApp</b>	Сервисная функция экспорта данных из файла приложения в текстовый файл.
<i>ИмяApp</i>	Имя файла описания приложения Clarion из которого производится экспорт в формат TXA.
<i>ТекстФайл</i>	Имя текстового файла в который осуществляется экспорт.
<i>ВыводНаЭкран</i>	Если этот параметр равен (1) или ИСТИНА, то в процессе экспорта на экран выводятся сообщения, если же он равен (0) или ЛОЖЬ, то вывод на экран сообщений запрещается. Если этот параметр опущен, то вывод сообщений подавляется

Эта сервисная функция позволяет вывести данные из файла описания приложения в текстовый файл. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение. Эта сервисная функция работает с любым файлом .APP, независимо от того открыт ли он в данный момент Средой Clarion.

Пример:

```

Server          LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

```

CODE

SYSTEM{PROP:DDETimeOut} = 12000    ! перерыв после двух минут

```

DDEEXECUTE(Server, '[ExportApp(c:\CW15\APPS\Myapp1.APP, MyTXA.TXA, 0)]')
DO CheckDDEError

```

CheckDDEError ROUTINE

```

DDEErrorMsg = ''
err# = ERRORCODE()
! проверка ошибок

```

```

IF err# > 600    ! ошибка связана с DDE
  IF err# = 603    !ошибка выполнения DDEExecute
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
  ELSIF err# = 605    ! ошибка по времени
    MESSAGE('DDE timeout')
  ELSE
    MESSAGE(err#)
  END
END
END

```

## Импорт данных в файл .APP из текстового формата (ТХА)

**DDEEXECUTE**(Сервер, '/ImportApp(ТекстФайл,ИмяApp,ВыводНаЭкран,КонфлИмен)')

<b>DDEEXECUTE</b>	Посылает командную строку на DDE сервер.
<i>Сервер</i>	Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.
<b>ImportApp</b>	Сервисная функция импорта данных из текстового файла в файл приложения.
<i>ТекстФайл</i>	Имя текстового файла из которого осуществляется импорт.
<i>ИмяПриложения</i>	Имя файла описания приложения Clarion в который производится импорт из формата ТХА.
<i>ВыводНаЭкран</i>	Если этот параметр равен (1) или ИСТИНА, то в процессе импорта на экран выводятся сообщения, если же он равен (0) или ЛОЖЬ, то вывод на экран сообщений запрещается. Если этот параметр опущен, то вывод сообщений подавляется
<i>КонфлИмен</i>	Указывает, как Генератор обходится с именами процедур, импортированных из текстового файла, которые конфликтуют именами уже имеющихся процедур. Если указать "Rename", то такие процедуры автоматически переименовываются. Если указать "Replase", то такие процедуры автоматически замещают уже существующие. Если этот параметр опущен, то пользователю выдается запрос о том, что делать с такими именами.

Эта сервисная функция позволяет ввести данные в файла описания приложения из текстового файла. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение. Эта сервисная функция работает с любым файлом .APP, независимо от того открыт ли он в данный момент Средой Clarion.

Пример:

```
Server          LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT
CODE
SYSTEM{PROP:DDETimeout} = 12000    ! перерыв после двух минут
```

```
DDEEXECUTE(Server,['ImportApp(MYTXA.TXA,c:\CW15\APPS\Myapp1.APP,Replace)'])
DO CheckDDEError
CheckDDEError ROUTINE
DDEErrorMsg = ''
err# = ERRORCODE()
! проверка ошибок
IF err# > 600    ! ошибка связана с DDE ?
    IF err# = 603    ! не выполнялась DDEExecute
        DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
        DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
        MESSAGE('Error ' & DDEErrorNum & ': ' & CLIP(DDEErrorMsg))
        ELSIF err# = 605    ! Timeout Error
            MESSAGE('DDE timeout')
    ELSE
        MESSAGE(err#)
END
END
```

## Загрузка файла описания приложения

```
DDEEXECUTE(Сервер,['LoadApplication(ИмяApp)'])
```

<b>DDEEXECUTE</b>	Посылает командную строку на DDE сервер.
<i>Сервер</i>	Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.
<b>LoadApplication</b>	Сервисная функция загрузки файла приложения.
<i>ИмяПриложения</i>	Имя файла приложения, который следует загрузить.

Эта сервисная функция позволяет загрузить файл приложения в Генератор Приложений. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение.

Пример:

```
Server          LONG
DDEErrorMsg CSTRING(300)
```

DDEErrorNum USHORT  
CODE

```
SYSTEM{PROP:DDETimeOut} = 12000    ! перерыв после двух минут
DDEEXECUTE(Server,'[LoadApplication(c:\CW15\APPS\Myapp.app,0)]')
DO CheckDDEError
```

CheckDDEError ROUTINE

```
DDEErrorMsg = ''
err# = ERRORCODE()
! проверить ошибки
IF err# > 600    ! ошибка связана с DDE ?
    IF err# = 603    !ошибка выполнения DDEExecute
        DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
        DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
        MESSAGE('Error ' & DDEErrorNum & ': ' & CLIP(DDEErrorMsg))
    ELSIF err# = 605    ! ошибка по времени
        MESSAGE('DDE timeout')
    ELSE
        MESSAGE(err#)
    END
END
```

## Генерировать текст приложения

***DDEEXECUTE(Сервер, '[GenerateApp(ИмяApp, ВыводНаЭкран)]')***

DDEEXECUTE	Посылает командную строку на DDE сервер.
Сервер	Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.
GenerateApp	Сервисная функция генерации исходного текста приложения.
ИмяApp	Имя приложения, текст которого следует сгенерировать.
ВыводНаЭкран	Если этот параметр равен (1) или ИСТИНА, то в процессе генерации на экран выводятся сообщения, если же он равен (0) или ЛОЖЬ, то вывод на экран сообщений запрещается. Если этот параметр опущен, то вывод сообщений подавляется

Эта сервисная функция позволяет сгенерировать исходный текст для некоторого приложения. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение. Эта сервисная функция работает с любым файлом .APP, независимо от того открыт ли он в данный момент Средой Clarion.

Пример:

```
Server      LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT
```

CODE

```
SYSTEM{PROP:DDETimeOut} = 12000    ! перерыв после двух минут
DDEEXECUTE(Server, '[GenerateAPP(c:\CW15\APPS\Myapp.app,0)]')
DO CheckDDEError
```

CheckDDEError ROUTINE

```
DDEErrorMsg = ''
err# = ERRORCODE()
! проверить ошибки
IF err# > 600    ! ошибка связана с DDE ?
    IF err# = 603    !ошибка исполнения DDEExecute?
        DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
        DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
        MESSAGE('Error ' & DDEErrorNum & ': ' & CLIP(DDEErrorMsg))
        ELSIF err# = 605    ! ошибка по истечении времени?
            MESSAGE('DDE timeout')
    ELSE
        MESSAGE(err#)
    END
END
```

## Выполнить проект или приложение.

**DDEEXECUTE**(Сервер, '[ExecuteProject(ИмяПроекта)]')

<b>DDEEXECUTE</b>	Посылает командную строку на DDE сервер.
<i>Сервер</i>	Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.
<b>ExecuteProject</b>	Сервисная функция компиляции и компоновки проекта (.PRJ) или приложения (.APP).
<i>ИмяПроекта</i>	Проект или приложение Clarion, для которого нужно выполнить процесс создания.

Эта сервисная функция позволяет выполнить компиляцию и компоновку проекта или приложения. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение. Эта сервисная функция работает с любым файлом .APP, независимо



от того открыт ли он в данный момент Средой Clarion. Вывод сообщений на экран при этом производится, но по завершении процесса окно закрывается без какого-либо дальнейшего взаимодействия.

Пример:

```
Server          LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT
```

CODE

```
SYSTEM{PROP:DDETimeOut} = 12000    ! перерыв после двух минут
    DDEEXECUTE(Server, '[ExecuteProject(c:\CW15\APPS\Myapp.app)]')
    DO CheckDDEError
```

CheckDDEError ROUTINE

```
DDEErrorMsg = ''
err# = ERRORCODE()
! проверить ошибки
IF err# > 600          ! ошибка связана с DDE ?
    IF err# = 603      !ошибка выполнения DDEExecute ?
        DDEREAD(Server, DDE>manual, 'GetErrorNum', DDEErrorNum)
        DDEREAD(Server, DDE>manual, 'GetErrorMsg', DDEErrorMsg)
        MESSAGE('Error ' & DDEErrorNum & ':' & CLIP(DDEErrorMsg))
        ELSIF err# = 605 ! таймаут
            MESSAGE('DDE timeout')
    ELSE
        MESSAGE(err#)
    END
END
```

## Выполнение шаблонов-утилит

**DDEEXECUTE**(Сервер,'GenerateUtilityTemplate(ИмяШаблонаУтилиты,ИмяПриложения,ИмяПриложения)')

**DDEEXECUTE**           Посылает командную строку на DDE сервер.  
**Сервер**                Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.  
**GenerateUtilityTemplate**   Сервисная функция генерации Шаблона-утилиты для заданного приложения.  
**ИмяШаблонаУтилиты**       Имя шаблона-утилиты который следует сгенерировать.  
**ИмяПриложения**       Имя приложения, для которого генерируется утилита-шаблон.  
**ВыводНаЭкран**       Если этот параметр равен (1) или ИСТИНА, то в процессе генерации на экран выводятся сообщения, если же он равен (0) или ЛОЖЬ, то вывод на экран сообщений запрещается. Если этот параметр опущен, то вывод сообщений подавляется

Эта сервисная функция позволяет выполнить шаблон-утилиту для некоторого приложения. Прежде чем выполнить эту (или какие-либо другие) команду нужно установить соединение. Эта сервисная функция работает с любым файлом .APP, независимо от того открыт ли он в данный момент Средой Clarion.

Пример:

```
Server       LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT
```

```
CODE
SYSTEM{PROP:DDETimeOut} = 12000   ! таймаут на две минуты
```

```
DDEEXECUTE(Server,'[GenerateUtilityTemplate(MyUtil,c:\CW15\APPS\Myapp.APP,0)]')
DO CheckDDEError
```

CheckDDEError ROUTINE

```
DDEErrorMsg = ''
err# = ERRORCODE()
! Check for DDE error
IF err# > 600                   ! ошибка связана с DDE
  IF err# = 603               ! ошибка при выполнении DDEExecute
    DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
    DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
    MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
```

```

        ELSIF err# = 605 ! ошибка таймаута
            MESSAGE('DDE timeout')
        ELSE
            MESSAGE(err#)
        END
    END
END

```

## Регистрация класса шаблона

**DDEEXECUTE(*Сервер*, '[RegisterTemplateChain(*ИмяШаблона*, *ИмяПриложения*)]')**

**DDEEXECUTE**           Посылает командную строку на DDE сервер.  
**Сервер**                Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.  
**RegisterTemplateChain**   Сервисная функция регистрации конкретного набора шаблонов.  
**ИмяШаблона**           Имя набора шаблонов, который следует зарегистрировать.  
**ВыводНаЭкран**        Если этот параметр равен (1) или ИСТИНА, то в процессе регистрации на экран выводятся сообщения, если же он равен (0) или ЛОЖЬ, то вывод на экран сообщений запрещается. Если этот параметр опущен, то вывод сообщений подавляется

Эта сервисная функция в первую очередь касается продуктов сторонних фирм для Clarion. Зарегистрировав из клиентского приложения набор набор шаблонов, разработчик шаблонов может добавить свои функциональные возможности в процедуру инсталляции. Тем самым отпадает необходимость регистрации стороннего набора шаблонов конечным пользователем.

Пример:

```

Server           LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT

```

### CODE

```

SYSTEM{PROP:DDETimeOut} = 12000 ! перерыв на две минуты
DDEEXECUTE(Server, '[RegisterTemplateChain(MYTPL.TPL,0)]')
DO CheckDDEError

```

CheckDDEError ROUTINE

```

DDEErrorMsg = ''
err# = ERRORCODE()

```

! проверить ошибки

IF err# > 600 ! ошибка связана с DDE

IF err# = 603 !! ошибка при выполнении DDEExecute

DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)

DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)

MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))

ELSIF err# = 605 ! ошибка по таймауту

MESSAGE('DDE timeout')

ELSE

MESSAGE(err#)

END

END

## Получение сообщений об ошибках DDE

**DDEREAD**(Сервер,DDE:Manual,'GetErrorMsg',ТекстСообщения)  
**DDEREAD**(Сервер,DDE:Manual,'GetErrorNum',НомерОшибкиDDE )

**DDEREAD** Получить данные от DDE сервера.  
*Сервер* Переменная типа LONG, которая содержит номер канала соединения с DDE сервером.  
*DDE:Manual* Мнемоническая метка соответствия, определяющая тип связи как “ручная” (определяется в файле EQUATES.CLW).  
**'GetErrorMsg'** Запрос текста сообщения об ошибке от сервера. Должно быть написано точно так как показано.  
**'GetErrorNum'** Запрос номера сообщения об ошибке от сервера. Должно быть написано точно так как показано.  
*ТекстСообщения* Переменная типа STRING(255), в которую заносится текст сообщения.  
*НомерОшибкиDDE* Переменная типа USHORT, в которую заносится номер сообщения.

Эта сервисная функция предназначена для контроля клиентском приложении ошибочных ситуаций при динамическом обмене данными. Для того, чтобы выявить ошибку, связанную с DDE, можно проверить результат, возвращаемый функцией Clarion ERRORCODE, а затем запросить DDE сервер проанализировать полученное значение.

Пример:

```
Server          LONG
DDEErrorMsg CSTRING(300)
DDEErrorNum USHORT
```

CODE

```
DDEEXECUTE(Server,['GenerateAPP(c:\CW15\APPS\Myapp.app,0)'])
DO CheckDDEError
```

CheckDDEError ROUTINE

```
DDEErrorMsg = '
err# = ERRORCODE()
! проверка ошибок
IF err# > 600                ! ошибка связана с DDE
    IF err# = 603            ! ошибка выполнения DDEExecute
        DDEREAD(Server, DDE:manual, 'GetErrorNum', DDEErrorNum)
        DDEREAD(Server, DDE:manual, 'GetErrorMsg', DDEErrorMsg)
        MESSAGE('Error ' & DDEErrorNum & ' : ' & CLIP(DDEErrorMsg))
```

```
        ELSIF err# = 605          ! ошибка по таймауту
            MESSAGE('DDE timeout')
        ELSE
            MESSAGE(err#)
        END
    END
```

## Ошибки при динамическом обмене данными с Clarion

---

601 Invalid DDE Channel	(неправильный номер канала DDE)
602 DDE Channel Not Open	(канал DDE не открыт)
603 DDEEXECUTE Failed	(ошибка выполнения DDEEXECUTE)
604 DDEPOKE Failed	(ошибка выполнения DDEPOKE)
605 Time Out	(ошибка по истечению времени)

Ошибки при выполнении сервисных функций DDE и сообщения о них

### 1 “Screen output must be either ‘TRUE’ or ‘FALSE’”

(Переменная “ВыводНаЭкран” должна иметь значение или “ИСТИНА”, или “ЛОЖЬ”)

### 2 “Cannot open specified dictionary”

(Нельзя открыть заданный словарь данных)

### 3 “Cannot create specified text file”

(Нельзя создать заданный текстовый файл)

### 4 “Cannot open specified text file”

(Нельзя открыть заданный текстовый файл)

### 5 “Cannot create specified dictionary”

(Нельзя создать заданный словарь данных)

### 6 “Cannot open specified application”

(Нельзя открыть заданное приложение)

### 7 “Cannot create specified application”

(Нельзя создать заданное приложение)

### 8 “Cannot open specified template file”

(Нельзя открыть заданный файл шаблонов)

### 9 “Cannot open specified project file”

(Нельзя открыть заданный файл описания проекта)

### 10 “Wrong number of parameters”

(Неверное число параметров)

### 11 “Requested DDE service is not supported”

(Запрошенная сервисная функция DDE не поддерживается)

Для следующих ошибок возвращаются сгенерированные запрошенными сервисными функциями сообщения:

**12 Export dictionary failed**

(Ошибка при экспорте словаря)

**13 Import dictionary failed**

(Ошибка при импорте словаря)

**14 Export application failed**

(Ошибка при экспорте приложения)

**15 Import application failed**

(Ошибка при импорте приложения)

**16 Register templated chained failed**

(Ошибка при регистрации цепочки шаблонов)

**17 Generate utility template failed**

(Ошибка при генерации утилиты-шаблона)

**18 Generate application failed**

(Ошибка при генерации текста приложения)

**19 Error in Make**

(Ошибка при создании приложения)





## **Формат файла .TXD**

### **Файлы .TXD: словари Clarion в формате ASCII**

Файлы .TXD представляют собой просто текстовый вариант файлов-словарей (.DCT) Clarion для Windows. В файлах .DCT Среда Clarion для Windows хранит всю используемую информацию о базе данных. Файл .TXD содержит всю хранящуюся в словаре информацию, но в текстовом, доступном для работы в любом текстовом редакторе формате.

Загрузив словарь в Среду Clarion для Windows, а затем выбрав в меню File д Export Text легко можно создать текстовый файл.

Зачем может потребоваться текстовый файл? По крайней мере по трем причинам:

**Копирование**                      Поскольку файл .TXD также легко можно импортировать, он может служить резервной копией текущей версии словаря (.DCT).  
Предыдущие версии в формате .TXD не сохраняются.

**Массовые изменения**

В связи с тем, что с файлом .TXD можно работать привычным для вас текстовым редакторе, всю мощь которого можно использовать для проведения массовых изменений в словаре, или подобным же образом любых изменений, которые легче сделать в текстовом редакторе.

**Первая помощь**                      Иногда со словарем .DCT может быть связано несколько странное поведение в Среде разработки. Экспорт в .TXD файл, а затем импорт обратно может выявить и исправить эти проблемы в словаре.

## Структура файла .TXD

Файл .TXD состоит из пяти основных разделов: [DICTIONARY], [FILES], [ALIASES], [RELATIONS] и [VIEWS]. Эти разделы соответствуют пяти основным структурам, относящимся к базе данных, в языке Clarion и содержат подразделы и ключевые слова, которые полностью описывают эти структуры. Например, раздел [FILES] помимо всего остального содержит определения полей и ключей файла данных.

### Схема файла .TXD

---

На следующей странице приведен упорядоченный список разделов файла .TXD, подразделов и ключевых слов. Он приведен для того, чтобы дать вам общее представление о структуре и организации файла .TXD в целом.

Каждый раздел начинается с заголовка, заключенного в квадратные скобки, и заканчивается с началом следующего раздела. В каждом разделе могут содержаться подразделы и ключевые слова.

Начало подраздела похоже на начало основного раздела - заголовок, заключенный в квадратные скобки.

Ключевые слова записываются большими буквами, а следом - его значение. Значение для ключевых слов приводятся в различных форматах, которые описываются ниже.

Пробелы и отступы в схеме сделаны для улучшения читаемости и в реальном .TXD файле отсутствуют.

Следом за схемой подробно обсуждается каждый раздел, подраздел и ключевое слово.

```
[DICTIONARY]
  VERSION
  CREATED
  MODIFIED
  PASSWORD
  [DESCRIPTION]
```

```
[FILES]
  [LONGDESC]
  [QUICKCODE]
  [USEROPTION]
  Определение файла
  !!> Список ключевых слов
```

Повторяющаяся группа описания файла

[LONGDESC]  
[QUICKCODE]  
[USEROPTION]           Необязательная повторяющаяся группа описания ключа  
Определение ключа  
!!> Список ключевых слов

[LONGDESC]  
[QUICKCODE]  
[USEROPTION]  
[SCREENCONTROLS]   Необязательная повторяющаяся группа описания Мемо  
[REPORTCONTROLS]  
Определение Мемо  
!!> Список ключевых слов

[LONGDESC]  
[QUICKCODE]  
[USEROPTION]  
[SCREENCONTROLS]   Необязательная повторяющаяся группа описания Blob  
[REPORTCONTROLS]  
Определение Blob  
!!> Список ключевых слов

Определение записи  
[LONGDESC]  
[QUICKCODE]  
[USEROPTION]  
[SCREENCONTROLS]   Необязательная повторяющаяся группа описания поля  
[REPORTCONTROLS]  
Определение поля  
!!> Список ключевых слов  
END  
END

[ALIASES]           Необязательный  
[LONGDESC]  
[QUICKCODE]  
[USEROPTION]           Необязательная повторяющаяся группа описания алиаса  
Определение алиаса  
!!> Список ключевых слов  
[RELATIONS]       Необязательный  
[LONGDESC]  
[USEROPTION]       Необязательная повторяющаяся группа описания связи

Определение связи  
[VIEWS]

Определение структуры View  
!!> Список ключевых слов

Необязательный

Необязательная повторяющаяся группа

## Разделы файла .TXD

### [DICTIONARY]

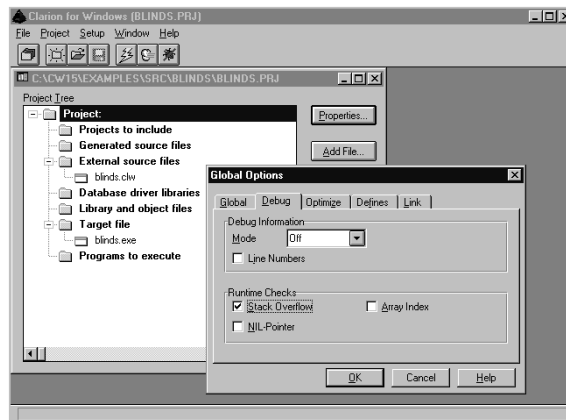
Раздел [DICTIONARY] является обязательным, хотя все ключевые слова в ней в отдельности необязательны. Этот раздел содержит следующие подразделы и ключевые слова:

#### [DICTIONARY]

VERSION  
CREATED  
MODIFIED  
PASSWORD

#### [DESCRIPTION]

VERSION	Приводится версия словаря (необязательное ключевое слово). При импорте версия игнорируется. Например: VERSION '1.0'
CREATED	Приводится дата и время изначального создания словаря (необязательное ключевое слово). Например: CREATED '19 AUG 95' '11:02:33AM'
MODIFIED	Приводится дата и время последнего сохранения словаря (необязательное ключевое слово). Например: MODIFIED ' 7 NOV 95' ' 9:44:18AM'
PASSWORD	Приводится пароль, необходимый для доступа к словарю (необязательное ключевое слово). Для установки словарю пароля используйте кнопку Password в диалоговом окне Dictionary Properties.



Пароль нечувствителен к регистру букв. В файле .DCT он зашифрован, но при экспорте в текстовый файл он не зашифровывается. Например:

PASSWORD cablecar

[DESCRIPTION]      Необязательный описательный текст длиной до 1000 символов на нескольких строках. Каждая строка текста начинается с восклицательного знака ( ! ) и содержит до 75 символов. Это текст с вкладки Comments диалогового окна Dictionary Properties. Раздел [DESCRIPTION] в словаре будет разделяться на строки длиной 75 символов. Если текст разделяется символами “возврат каретки”, то в текстовый файл будет выведена дополнительная пустая строка. Это справедливо для любого описания ([DESCRIPTION] и [LONGDESC]) в файле .TXD и для переменных в файле .TXA .

Например:

Первоначальный текст в словаре:

Это описание : <CR>

и оно продолжается на следующей строке в следствие чего текст здесь записывается без символа “возврат каретки”.

Текст в файле .TXD:

! Это описание :

!

! и оно продолжается на следующей строке в следствие чего текст здесь записывается без символа “возврат каретки”.

### Пример-[DICTIONARY]

[DICTIONARY]

VERSION '1.0'

CREATED '19 AUG 95' '11:02:33AM'

MODIFIED ' 7 NOV 95' '9:44:18AM'

PASSWORD cablecar

[DESCRIPTION]

!До 1000 символов текста, описывающего данный словарь.

!Каждая строка начинается с восклицательного знака и длиной до 75 знаков

### [FILES]

Раздел [FILES] имеется в файле .TXD только в одном экземпляре. Все файлы в этом словаре определяются в этом разделе. В нем также определяются поля и ключи как часть

определения каждого файла.

### Группа определения файла

Группа определения файла представляет собой последовательность подразделов и ключевых слов, которые полностью описывают одну структуру FILE в словаре. Эта группа повторяется для каждого файла, так что все они полностью описываются.

[FILES]

[LONGDESC]
[QUICKCODE]
[USEROPTION]
Определение файла
!!> Список ключей

Повторяющаяся группа описания файла

[LONGDESC]  
[QUICKCODE]

См. ниже Общие подразделы.

Информация, используемая Мастером Clarion для Windows для конфигурирования генерируемых Мастером приложений и процедур (необязательный подраздел). Для ввода данных в подраздел [QUICKCODE] используйте вкладку Options в окне File Properties. Единственной опцией Wizard/QUICKCODE для файлов является NOPOPULATE. Она означает, что Мастер не будет генерировать для этого файла процедуры форм, таблиц и отчетов.

[USEROPTION]  
Определение файла

См. ниже Общие подразделы.

Начинается с первой строки объявления файла на языке Clarion. Возможен необязательный комментарий длиной до 40 символов. В определение файла включается также список ключевых слов и группы определения ключей, мемо-полей, полей типа Blob (больших двоичных объектов), записи и полей. Вся структура завершается оператором END. Например:

```
Customers FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE
Группа определения ключей
Группа определения Мемо-полей
Группа определения группы
Группа определения записи
Группа определения поля
END
```

Список ключевых слов -

список ключевых слов, которые описывают опции, не заданные в предложении FILE (необязательные). Список начинается с !!>.

Единственным допустимым ключевым словом для файла является IDENT. Оно определяет внутренний номер-ссылку, по которому Среда идентифицирует предложение FILE.

Например:    !!> IDENT(1)

### **Пример - Группа определения файла**

[FILES]

[LONGDESC]

! Это главный файл покупателей. Содержит фамилии, адреса и  
! телефоны. Одна запись на покупателя. Каждый покупатель имеет  
! уникальный ключ, который присваивается автоматически.

[QUICKCODE]

!NOPOPULATE

[USEROPTION]

!ThirdPartyTemplateSwitch(on)

!ThirdPartyPreProcessLevel(release)

Customers FILE,DRIVER('TOPSPEED'),PRE(CUS),CREATE,THREAD

!!> IDENT(1)

.  
. .  
. .  
END

### **Группа определения ключа**

Группа определения ключа необязательна. Она представляет собой последовательность подразделов и ключевых слов в файле .TXD, которые целиком описывают один ключ или индекс файла. Для каждого ключа или индекса эта группа повторяется, так что все они полностью описываются. Группа определения ключа содержит следующие ключевые слова.

[LONGDESC]

[QUICKCODE]

[USEROPTION]    Группа определения ключа - необязательная, повторяющаяся

Определение ключа

!!> список ключевых слов

[LONGDESC]  
[QUICKCODE]

См. ниже Общие подразделы..

Информация, используемая Мастерами Clarion для Windows для того, чтобы конфигурировать генерируемые ими приложения и

процедуры. Для ввода данных в этот раздел используется вкладка Options в диалоговом окне Key Properties.

Данные в разделе [QUICKCODE] могут располагаться на нескольких строках, каждая из которых начинается восклицательным знаком. Однако Среда Clarion для Windows соединяет эти строки в одну. Поэтому ключевые слова несмотря на то, что они располагаются в .TXD файле на отдельных строках, должны отделяться запятыми. Для ключей в этом разделе возможны две опции: NOPOPULATE и ORDER.

ORDER()

NOPOPULATE Мастер не генерирует отчетов и таблиц, упорядоченных по этому ключу (необязательная опция).

Порядок, в котором отсортированные по этому ключу отчеты и таблицы просмотра, выводятся внутри сгенерированных Мастером меню и окон просмотра (необязательная опция).

Здесь можно указать Normal, First или Last. Normal означает, что отсортированные по этому ключу отчеты и таблицы просмотра выводятся в том порядке, в котором этот ключ располагается в словаре. First приводит к тому, что отсортированные по этому ключу отчеты и таблицы просмотра выводятся прежде таблиц, отсортированных по ключам с опцией Normal и Last. И наоборот, Last приводит к тому, что отсортированные по этому ключу отчеты и таблицы просмотра выводятся после таблиц, отсортированных по ключам с опцией First и Normal.

Например: [QUICKCODE]  
!ORDER(First)

[USEROPTION]

См. ниже Общие подразделы..(необязательная опция)

Определение ключей Перечисляются объявления ключей и индексов на языке Clarion (обязательно). Включает в себя необязательный комментарий длиной до 40 символов. Например:

KeyCust KEY(CUS:CustNumber),NOCASE !Автонумерация

IDENT()

Список ключевых слов - список ключевых слов, которые описывают опции, не заданные в предложении KEY (необязательные). Список начинается с !!>. Допустимы два ключевых слова: IDENT и AUTO. Внутренний указатель используемый Средой для идентификации ключа (необязательный).

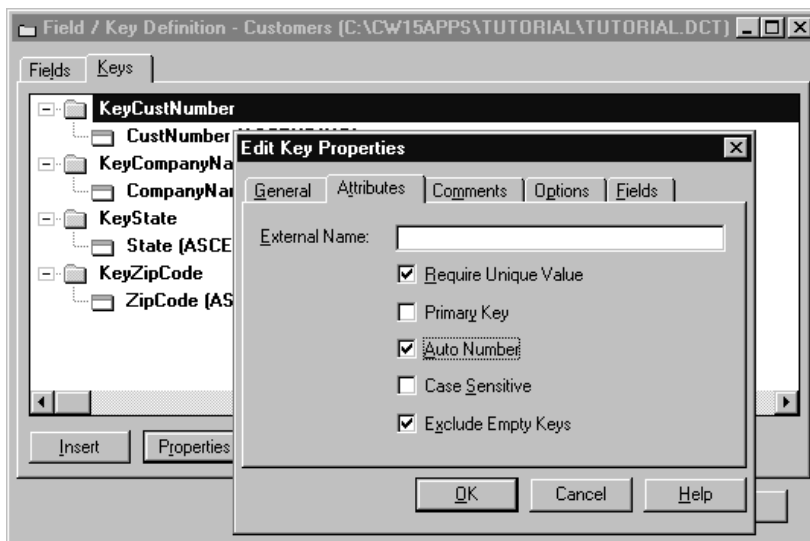
AUTO

Автонумеруемый ключ (необязательный). Это означает, что генератор приложений будет вставлять программный код, для автоматического увеличения значения ключевого поля при добавлении записи. Для установки автонумерации ключа используется вкладка Attributes в диалоговом окне Key Properties. Например:



!!> IDENT(1),AUTO

### Пример группы определения ключа



KeyCustNumber KEY(CUS:CustNumber),NOCASE,OPT !автонумеруемый ключ Cust  
!!> IDENT(1),AUTO  
[QUICKCODE]  
!ORDER(First)

### Группа определения поля Мемо

Группа определения поля Мемо не является обязательной. Она представляет собой последовательность подразделов и ключевых слов, которые исчерпывающе описывают одно Мемо-поле файла. Для каждого Мемо-поля эта группа повторяется, так что все они полностью описываются. Группа определения Мемо-поля содержит следующие ключевые слова и подразделы:

[LONGDESC]

[QUICKCODE]

[USEROPTION]

[SCREENCONTROLS]

[REPORTCONTROLS]

MEMO Definition

!!> список ключевых слов

Группа Мемо-поля, необязательная, повторяемая

[LONGDESC]  
[QUICKCODE]

См. ниже Общие подразделы.. (необязательные)  
Информация, используемая Мастерами Clarion для Windows для

того, чтобы конфигурировать генерируемые ими приложения и процедуры (необязательная). Для ввода данных в этот раздел используется вкладка Options в диалоговом окне Key Properties.

Данные в разделе [QUICKCODE] могут располагаться на нескольких строках, каждая из которых начинается восклицательным знаком. Однако Среда Clarion для Windows соединяет эти строки в одну. Поэтому ключевые слова несмотря на то, что они располагаются в .TXD файле на отдельных строках, должны отделяться запятыми.

Для MEMO-полей в этом разделе возможны опции: NOPOPULATE, ORDER, VERTICALSPACE, и TAB().

NOPOPULATE	Это поле не включается в создаваемые Мастером отчеты, формы и таблицы просмотра (необязательный параметр) .
ORDER()	Для MEMO-полей не имеет смысла, они всегда выводятся на отдельной вкладке, в порядке, в котором они идут в словаре данных.
VERTICALSPACE	Для MEMO-полей не имеет смысла, они всегда выводятся на отдельной вкладке.
TAB()	Для MEMO-полей не имеет смысла, они всегда выводятся на отдельной вкладке
[QUICKCODE]	
!NOPOPULATE	
[USEROPTION]	См. ниже Общие подразделы..(необязательная опция)
[SCREENCONTROLS]	MEMO-поля всегда выводятся в поле типа TEXT. См. ниже Общие подразделы.. (необязательная опция)
[REPORTCONTROLS]	MEMO-поля всегда выводятся в поле типа TEXT. См. ниже Общие подразделы..(необязательная опция)
<p>Определение MEMOОбъявление поля MEMO на языке Clarion (обязательное). Может включать необязательный комментарий длиной до 40 символов. Например:</p>	
<p>CustomerMemo MEMO(500) !Текстовое поле в 500 символов</p>	

### **Список ключевых слов для полей MEMO**

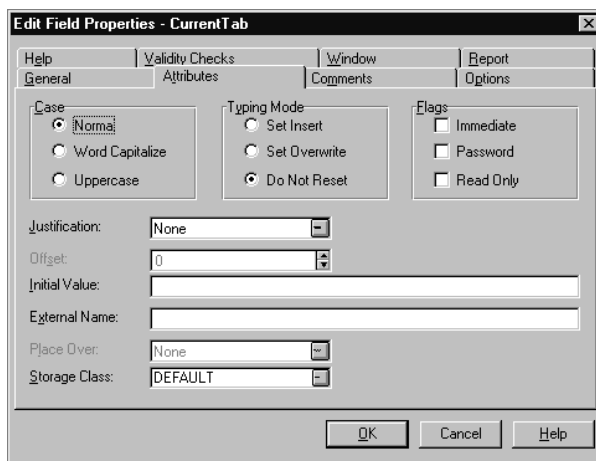
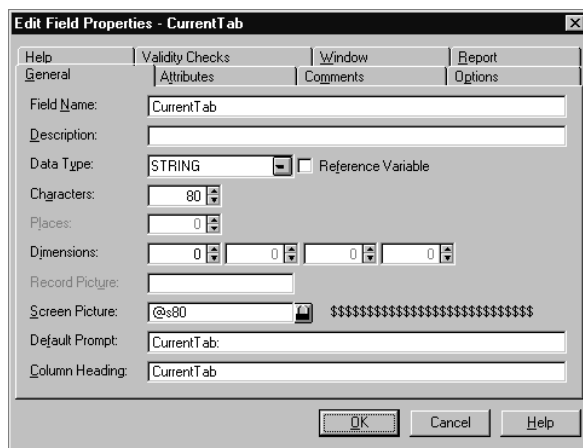
Список ключевых слов, который описывает опции, не указываемые в операторе MEMO

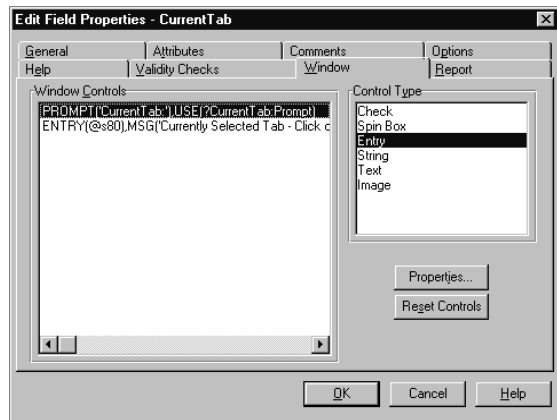
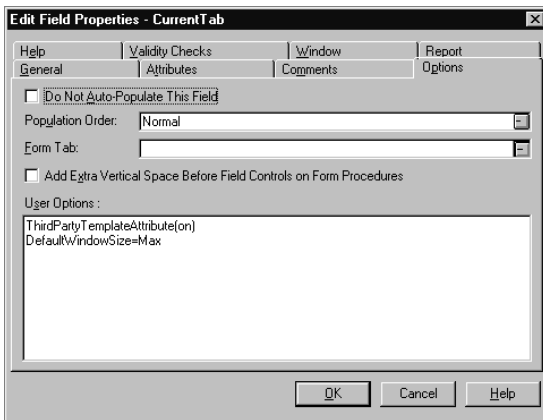
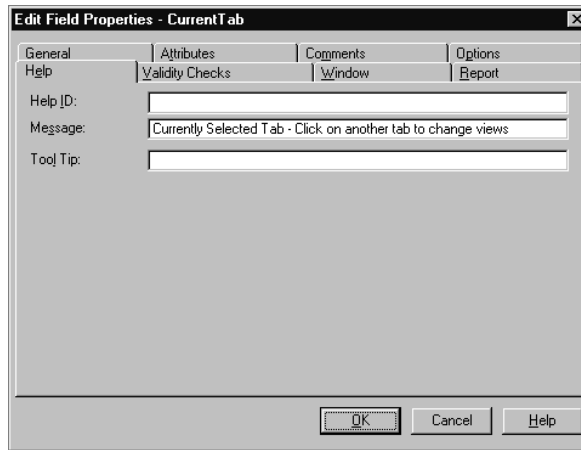
языка Clarion. Он начинается с !!> и не является обязательным. В файле .TXD этот список приводится на одной строке через запятую.

Значения этих ключевых слов в этом списке задаются в диалоговом окне Field Properties, многие из них непосредственно соответствуют ключевым словам языка Clarion. Разъяснение по этим ключевым словам смотрите в Описании языка.

Вот полный список ключевых слов для полей MEMO

```
IDENT()  
VALID()  
INITIAL()  
PROMPT()  
HEADER()  
HELP()  
MESSAGE()  
TOOLTIP()  
PICTURE()  
CASE()  
TYPEMODE()  
PASSWORD  
READONLY
```





Отметим ключевые слова в списке соответствуют вкладкам и приглашениям и вкладкам в диалоговом окне Field Properties. Это потому что там устанавливаются значения этих ключевых слов (подробнее об этих полях смотрите Руководство пользователя, Описание языка и систему интерактивной помощи).

IDENT()

Внутренний идентификатор, используемый Средой для того, чтобы различать этот элемент (необязательный). Например:

IDENT(3)

VALID()

Задаёт программный код для проверки допустимости значения, который генерируется для этого поля (необязательно присутствует). У ключевого слова VALID имеется список параметров, которые представлены ниже в диаграмме, где альтернативные параметры расположены столбцом, а необязательные представлены в фигурных скобках:

VALID(NONZERO)

INRANGE({minimum}{,maximum})  
BOOLEAN  
INFILE(filename,parentfile:childfile)  
INLIST('item1|item2|...|itemn')

Например, для обязательно присутствующего поля:  
VALID(NONZERO)

Для положительного числового поля:  
VALID(INRANGE(0))

Для логического поля (да/нет, истина/ложь):  
VALID(BOOLEAN)

Для поля проверяемого по другому полю:  
VALID(INFILE(States,Customers:States))

Для поля проверяемого по ограниченному списку:  
VALID(INLIST('Left|Center|Right'))

Эти значения устанавливаются в диалоговом окне Field Properties на вкладке Validity Checks. О проверке допустимости значений смотрите главу Использование Редактора Словаря в Руководстве пользователя.

INITIAL()

Необязательное начальное значение для поля. Это значение устанавливается в диалоговом окне Field Properties на вкладке Attributes. Например:

INITIAL(0)

PROMPT()

Текстовая строка, используемая в качестве пояснения к данному полю в окнах (необязательная). Это значение устанавливается в диалоговом окне Field Properties на вкладке General. Например:

PROMPT('&Product Number')

HEADER()

Текстовая строка, используемая в качестве заголовка колонки для данного поля в окнах и отчетах (необязательная). Это значение устанавливается в диалоговом окне Field Properties на вкладке General. Например:

HEADER('PRODUCT NUMBER')

HELP()

Имя раздела интерактивной помощи для данного поля. Это значение устанавливается в диалоговом окне Field Properties на вкладке Help.

Например:

HELP('PRODNUMBER')

#### MESSAGE()

Необязательный текст сообщения длиной до 75 символов, используемого для этого поля по умолчанию. Это значение устанавливается в диалоговом окне Field Properties на вкладке Help. Например:

MESSAGE('Enter the 6 digit product number')

#### TOOLTIP()

Необязательный текст всплывающей подсказки длиной до 75 символов, используемой для этого поля по умолчанию. Это значение устанавливается в диалоговом окне Field Properties на вкладке Help. Например:

TOOLTIP('Enter the 6 digit product number')

#### PICTURE()

Необязательный шаблон для отображения этого поля на экране, используемый по умолчанию. Полный список шаблонов и их использование приведены в Описании языка. Это значение устанавливается в диалоговом окне Field Properties на вкладке General. Например:

PICTURE('@n6')

#### CASE()

Перевод регистра букв, используемый по умолчанию для этого поля (необязательный). Варианты: UPPER и CAPS. UPPER преобразует введенные символы к верхнему регистру. CAPS преобразует к верхнему регистру только первую букву каждого слова. Это значение устанавливается в диалоговом окне Field Properties на вкладке Attributes. Например:

CASE('UPPER')

#### TYPEMODE()

Используемый по умолчанию режим ввода в это поле (необязательно задается). Возможные значения: INS и OVR. В режиме INS (режим вставки) существующий текст сохраняется, а вводимые символы "вставляются". OVR устанавливает режим замены: в этом режиме находящиеся в поле символы "заменяются" вводимыми. Это значение устанавливается в диалоговом окне Field Properties на вкладке At

tributes. Например:

TYPEMODE('INS')

#### PASSWORD

Вводимый в поле текст высвечивается как последовательность звездочек (необязательное свойство). Это значение устанавливается в диалоговом окне Field Properties на вкладке Attributes. Например: PASSWORD

#### READONLY

Значение в поле предназначено “только для чтения” и вводить данные в такое поле нельзя. Это значение устанавливается в диалоговом окне Field Properties на вкладке Attributes. Например:

READONLY

Например:

!!> IDENT(47),PROMPT('Customer Memo:'),PICTURE(@s255)

#### Пример - группа определения Мемо-поля

[LONGDESC]

!Long description of Memo field

[QUICKCODE]

!TAB('TabName'),ORDER(Last),VERTICALSPACE

[SCREENCONTROLS]

! PROMPT('Customer Memo:'),USE(?CUS:CustomerMemo:Prompt)

! TEXT,USE(CUS:CustomerMemo)

[REPORTCONTROLS]

! TEXT,USE(CUS:CustomerMemo)

CustomerMemo MEMO(500)

!!> IDENT(47),PROMPT('Customer Memo:'),PICTURE(@s255)

#### Группа определения поля Blob

Группа определения поля Blob не является обязательной. Это последовательность подразделов .TXD файла и ключевых слов, которая полностью описывает одно поле типа Blob в файле. Для каждого поля Blob в файле эта группа повторяется, так что все они полностью документируются. Группа определения поля Blob содержит следующие ключевые слова и подразделы.

[LONGDESC]

[QUICKCODE]

[USEROPTION]

[SCREENCONTROLS]

[REPORTCONTROLS]

Группа определения поля Blob, необязательная повторяющаяся.

## BLOB Definition

!!> Список ключевых слов.

[LONGDESC]	См. ниже Общие подразделы..(необязательная опция)
[QUICKCODE]	Для полей BLOB не имеет значения.
[USEROPTION]	См. ниже Общие подразделы..(необязательная опция)
[SCREENCONTROLS]	См. ниже Общие подразделы..(необязательная опция)
[REPORTCONTROLS]	См. ниже Общие подразделы..(необязательная опция)
Определение BLOB	Список объявлений полей BLOB на языке Clarion (обязательный элемент). Может включать в себя необязательный комментарий длиной до 40 символов. Например:

CustomerPhoto BLOB,BINARY !Customer Image/Logo

Список ключевых слов      Список ключевых слов, которые описывают опции, не задаваемые в объявлении BLOB. Этот список начинается с символов !!>.

Единственное допустимое для BLOB ключевое слово - это IDENT. Это внутренний идентификатор, используемый Средой для того, чтобы идентифицировать файл (необязательный элемент).

Например:

!!> IDENT(48)

### **Пример группы определения поля Blob**

```
[LONGDESC]
!это оцифрованное изображение может содержать логотип или фото покупателя
[QUICKCODE]
!TAB('Personal'),ORDER(First)
[SCREENCONTROLS]
! IMAGE,USE(?CUS:CustomerPhoto)
[REPORTCONTROLS]
! IMAGE,USE(?CUS:CustomerPhoto)
CustomerPhoto      BLOB,BINARY !Фото/лого покупателя
!!> IDENT(48)
```

### **Определение записи**

Определение записи является обязательным. Оно начинается с кларионовского оператора RECORD для этого файла. В объявление RECORD включаются описания всех полей файла. Вся структура заканчивается оператором END. Например:

CustomerRecord      RECORD



группы определения полей  
END

### **Группа определения поля**

Группа определения поля не является обязательной. Она представляет собой последовательность подразделов .TXD файла и ключевых слов, которая полностью описывает одно поле файла. Для каждого поля в файле эта группа повторяется, так что все они полностью документируются. Группа определения поля содержит следующие ключевые слова и подразделы.

[LONGDESC]	
[QUICKCODE]	
[USEROPTION]	
[SCREENCONTROLS]	Группа определения поля.
[REPORTCONTROLS]	Повторяемая, необязательная
Определение поля	
!!> Список ключевых слов	

[LONGDESC]	См. ниже Общие подразделы..(необязательная опция)
[QUICKCODE]	Информация, используемая Мастерами Clarion для Windows для того, чтобы конфигурировать генерируемые ими приложения и процедуры (необязательная). Для ввода данных в этот раздел используется вкладка Options в диалоговом окне Field Properties. Данные в разделе [QUICKCODE] могут располагаться на нескольких строках, каждая из которых начинается восклицательным знаком. Однако Среда Clarion для Windows соединяет эти строки в одну. Поэтому ключевые слова несмотря на то, что они располагаются в .TXD файле на отдельных строках, должны отделяться запятыми. Для полей файла в этом разделе возможны опции: NOPOPULATE, ORDER, VERTICALSPACE, и TAB.
NOPOPULATE	Это поле не включается в создаваемые Мастером отчеты, формы и таблицы просмотра (необязательный параметр) .
ORDER()	Порядок в котором поля включаются в отчеты, формы и таблицы просмотра (необязательный параметр). Здесь можно указать Normal, First или Last. Normal означает, поля отображаются в том порядке, в котором они находятся в слове. First приводит к тому, что данный элемент выводится перед элементами с признаками Normal и Last. И наоборот, Last означает, что данный элемент выводится после элементов с признаками Normal и First
VERTICALSPACE	В сгенерированных Мастером формах между этим полем и предыдущим вставляются дополнительные пробелы.
TAB()	Название вкладки в сгенерированной Мастером форме, на которой выводится данное поле.

Например:

[QUICKCODE]

!ORDER(Last),VERTICALSPACE,TAB('Personal')

[USEROPTION]      См. ниже Общие подразделы..(необязательная опция)

[SCREENCONTROLS]

См. ниже Общие подразделы..(необязательная опция)

[REPORTCONTROLS]

См. ниже Общие подразделы..(необязательная опция)

Определение поля      Оператор объявления поля на языке Clarion (необязательный).

Может включать необязательный комментарий длиной до 40 символов. Например:

CustNumber DECIMAL(7,2) !уникальный ключ

### **Ключевые слова - для полей**

Список ключевых слов, который описывает опции, не указываемые в операторе объявления поля языка Clarion. Он начинается с !!> и не является обязательным. В файле .TXD этот список приводится на одной строке через запятую.

Значения этих ключевых слов в этом списке задаются в диалоговом окне Field Properties, многие из них непосредственно соответствуют ключевым словам языка Clarion. Разъяснение по этим ключевым словам смотрите в Описании языка.

Полный список ключевых слов для полей в словаре данных:

IDENT()

VALID()

INITIAL()

PROMPT()

HEADER()

HELP()

MESSAGE()

TOOLTIP()

PICTURE()

CASE()

TYPEMODE()

PASSWORD

READONLY

Заметим, что ключевые слова в этом списке тесно связаны с вкладками и приглашениями в диалоговом окне Field Properties. Это потому, что именно там и устанавливаются их значения (Подробнее об этих полях смотрите Руководство пользователя, Описание языка

и диалоговую помощь.)

**IDENT()**                    Необязательный внутренний идентификатор, используемый Средой для того, чтобы различать этот элемент. Например:  
**IDENT(3)**

**VALID()**                    Задает программный код, генерируемый для проверки правильности заполнения этого поля. Ключевое слово **VALID** имеет список параметров, которые представлены ниже в диаграмме, где альтернативные параметры расположены столбцом, а необязательные представлены в фигурных скобках:

**VALID(NONZERO)**  
**INRANGE**({minimum}{,maximum])  
**BOOLEAN**  
**INFILE**(filename,parentfile:childfile)  
**INLIST**('item1|item2|...|itemn')

Например, для обязательного поля:  
**VALID(NONZERO)**

Для положительного числового поля:  
**VALID(INRANGE(0))**

Для двоичного логического поля (да/нет, истина/ложь):  
**VALID(BOOLEAN)**

Для поля, проверяемого на соответствие полю в другом файле:  
**VALID(INFILE(States,Customers:States))**

Для поля, проверяемого на наличие в список

**VALID(INLIST('Left|Center|Right'))**

Эти значения задаются в диалоговом окне **Field Properties** на вкладке **Validity Checks**. Информацию по проверке правильности значения поля смотрите в главе **Использование словаря данных в Руководстве пользователя**.

**INITIAL()**                    Необязательное начальное значение поля. Это значение устанавливается на вкладке **Attributes** диалогового окна **Field Properties**. Например:  
**INITIAL(0)**

**PROMPT()**                    Текстовая строка, используемая по умолчанию как поясняющая

надпись к данному полю на экране. Этот текст устанавливается на вкладке General диалогового окна Field Properties. Например:  
PROMPT('&Product Number')

**HEADER()** Текстовая строка, используемая по умолчанию как заголовок колонки для данного поля в окнах списка (необязательный атрибут). Этот текст устанавливается на вкладке General диалогового окна Field Properties. Например:  
HEADER('PRODUCT NUMBER')

**HELP()** Раздел диалоговой подсказки для этого поля. Это значение устанавливается на вкладке Help диалогового окна Field Properties. Например:  
HELP('PRODNUMBER')

**MESSAGE()** Необязательный текст используемого по умолчанию сообщения длиной до 75 символов. Это значение устанавливается на вкладке Help диалогового окна Field Properties. Например:  
MESSAGE('Enter the 6 digit product number')

**TOOLTIP()** Необязательный текст длиной до 75 символов, используемый по умолчанию в качестве всплывающей подсказки. Это значение устанавливается на вкладке Help диалогового окна Field Properties. Например:  
TOOLTIP('Enter the 6 digit product number')

**PICTURE()** Используемый по умолчанию шаблон (необязательный) для отображения этого поля на экране. Полное описание шаблонов и их использования смотри в Описании языка. Этот шаблон задается на вкладке General в диалоговом окне Field Properties. Например:

PICTURE('@n6')

**CASE()** Устанавливаемое по умолчанию преобразование регистра букв в этом поле (необязательно). Варианты: UPPER и CAPS. UPPER устанавливает преобразование букв в прописные. А CAPS устанавливает преобразование в прописные только первой буквы каждого слова. Это значение устанавливается на вкладке Attributes диалогового окна Field Properties. Например:

CASE('UPPER')

**TYPEMODE()** Устанавливаемый по умолчанию режим ввода в это поле

(необязательно). Варианты: INS и OVR. В режиме INS (режим вставки) существующий текст сохраняется, а вводимые символы “вставляются”. OVR устанавливает режим замены: в этом режиме находящиеся в поле символы “заменяются” вводимыми. Это значение устанавливается в диалоговом окне Field Properties на вкладке Attributes. Например:  
 TYPEMODE(‘INS’)

**PASSWORD** Вводимый в поле текст высвечивается как последовательность звездочек (необязательное свойство). Это значение устанавливается в диалоговом окне Field Properties на вкладке Attributes. Например:  
 PASSWORD

**READONLY** Значение в поле предназначено “только для чтения” и вводить данные в такое поле нельзя. Это значение устанавливается в диалоговом окне Field Properties на вкладке Attributes. Например:  
 READONLY  
 Пример:  
 !!> IDENT(5),PROMPT(‘&Cust Number:’),PICTURE(@n4)

### **Пример группы определения поля**

```
[QUICKCODE]
!ORDER(First)
[SCREENCONTROLS]
! PROMPT(‘&Cust Number:’),USE(?CUS:CustNumber:Prompt)
! ENTRY(@n4),USE(CUS:CustNumber)
[REPORTCONTROLS]
! STRING(@n4),USE(CUS:CustNumber)
CustNumber          DECIMAL(7,2)
!!> IDENT(5),PROMPT(‘&Cust Number:’),HEADER(‘Cust Number’),PICTURE(@n4)
```

## **[ALIASES] (алиасы)**

Раздел “алиасы” существует в файле .TXD только в одном экземпляре, или может отсутствовать вовсе. В нем определяются все алиасы из словаря данных.

### **Группа определения алиасов**

Группа определения алиасов представляет собой последовательность подразделов и ключевых слов, полностью описывающих один алиас из словаря данных. Для каждого алиаса в словаре эта группа повторяется. Группа определения алиасов содержит следующие ключевые слова и подразделы.

```
[ALIASES]
  [LONGDESC]
```

[QUICKCODE]

[USEROPTION]

определение алиаса

!!&gt; список ключевых слов

повторяющаяся группа определения алиасов

[LONGDESC]

[QUICKCODE]

См. ниже Общие подразделы..(необязательная опция)

Информация, используемая Мастерами Clarion для Windows для того, чтобы конфигурировать генерируемые ими приложения и процедуры (необязательная). Для ввода данных в этот раздел используется вкладка Options в диалоговом окне Alias Properties. Для полей файла в этом разделе возможна только опция NOPOPULATE. Она означает, что это Мастер не будет генерировать отчеты, формы и таблицы просмотра для этого алиаса (необязательный параметр) .

[USEROPTION]

Определение алиаса

См. ниже Общие подразделы..(необязательная опция).

Задаёт файл, для которого переопределяется алиас и обязательный новый префикс. Может включать необязательный комментарий длиной до 40 символов. Например:

Sales

ALIAS(Orders),PRE(SAL) !Алиас для Order

Список кл. слов

Список ключевых слов, который описывает опции, не указываемые при определении алиаса. Он начинается с !!> и не является обязательным.

Единственное ключевое слово для алиаса - IDENT. Оно определяет необязательный внутренний идентификатор, используемый Средой для того, чтобы различать этот элемент. Например:

!!&gt; IDENT(6)

### Пример группы определения алиаса

[ALIASES]

[QUICKCODE]

!NOPOPULATE

Sales ALIAS(Orders),PRE(SAL) !Алиас для Order

!!&gt; IDENT(6)

### [RELATIONS]

В файле .TXD имеется только один раздел [RELATIONS]. Этот раздел необязателен, и в нем определяются связи между файлами. Здесь также определяются методы обеспечения целостности ссылок, как неотъемлемая часть каждой связи.

Группа определения связей

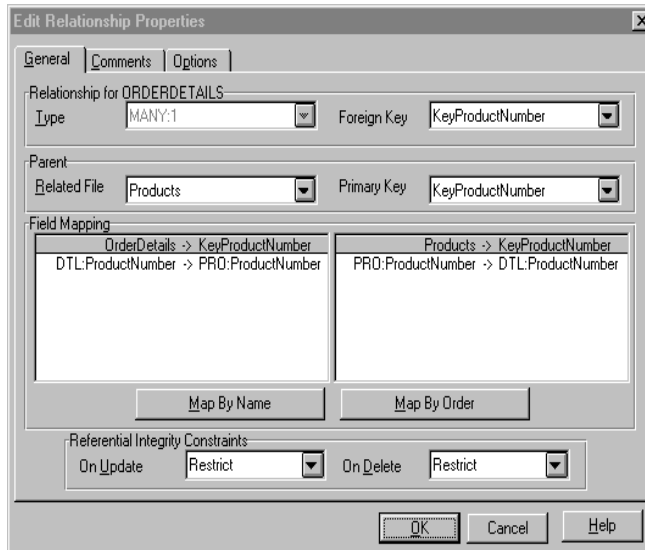
Группа определения связей представляет собой последовательность подразделов файла .TXD, ключевых слов и специальных описаний связей. Который полностью определяют взаимосвязи между двумя файлами в словаре. Для каждой связи эта группа повторяется. Группа определения связей содержит следующие ключевые слова и подразделы:

[RELATIONS]	
[LONGDESC]	
[USEROPTION]	Группа определения связей
Определение связи	
[LONGDESC]	См. ниже Общие подразделы..(необязательная опция)
[USEROPTION]	См. ниже Общие подразделы..(необязательная опция)
	Определение связи Связь определенная в окне Edit Relationship Properties (обязательный элемент). Описание связи в файле .TXD представлено далее диаграммой, в которой столбцами записаны альтернативные параметры.
RELATION,	ONE:MANY, UPDATE( RESTRICT ),DELETE( RESTRICT )
	MANY:ONE CASCADE CLEAR CASCADE CLEAR
filename	FILE( {key} )
filename	RELATED_FILE( {key} )
{FILE_TO_RELATED_KEY	
FIELD(	filefieldname,relatedfilefieldname )
	NOLINK, NOLINK
повторяется	
END}	
{RELATED_FILE_TO_KEY	
FIELD(	relatedfilefieldname,filefieldname )
	NOLINK, NOLINK
повторяется	
END}	
END	
RELATION	Начало определения связи
ONE:MANY   MANY:ONE	
UPDATE	Тип связи (обязательный признак). Действия, предпринимаемые для поддержания целостности ссылок в базе данных, когда в основном файле изменяются значения ключей используемых для связи с дочерними файлами (необязательный атрибут). Может быть указано RESTRICT, CASCADE или CLEAR.

RESTRICT	Если существуют порожденные записи в связанных файлах, то изменения значения связывающего ключа не допускаются.
CASCADE	Если существуют порожденные записи в связанных файлах, то изменения значения связывающего ключа автоматически выполняются и в них.
CLEAR	Автоматически выполняются изменения значения связывающего ключа во всех зависимых записях на нулевое или пробельное значение.
DELETE	Действия, предпринимаемые для поддержания целостности ссылок в базе данных, когда в основном файле удаляется запись (необязательный атрибут). Может быть указано RESTRICT, CASCADE или CLEAR.
RESTRICT	Если существуют связанные записи, то удаление “записи-родителя” запрещено.
CASCADE	Если существуют связанные записи, то они тоже удаляются.
CLEAR	Во всех зависимых записях значения ключа обнуляется или становится пробельным.
Имя_файла FILE( {ключ} )	Ключ в первом файле в связи, который содержит поле(я), общие для обоих файлов (необязательный атрибут). Это или порождающий файл для связи “ОДНА ко МНОГИМ” или порожденный файл для связи “МНОГИЕ к ОДНОЙ”. Для однозначных связей (только с одним файлом по этому ключу) и для связей типа “просмотр в справочнике” параметр ключ необязателен.
Имя_файла СВЯЗАННЫЙ_ФАЙЛ( {ключ} )	Ключ во втором файле из связи, который содержит общие для обоих файлов поля (необязательный атрибут). Это или порожденный файл в связи “ОДНА ко МНОГИМ” или главный файл для связи “МНОГИЕ к ОДНОЙ”. Для однозначных связей (только с одним файлом по этому ключу) и для связей типа “просмотр в справочнике” параметр ключ необязателен.
RELATION, ONE:MANY,UPDATE( RESTRICT ),DELETE( RESTRICT )	
MANY:ONE CASCADE	
CLEAR	
filename FILE( {key} )	
filename RELATED_FILE( {key} )	
{FILE_TO_RELATED_KEY	
FIELD( filefieldname,relatedfilefieldname )	повторяемое
NOLINK, NOLINK	
END}	
{RELATED_FILE_TO_KEY	
FIELD( relatedfilefieldname,filefieldname )	повторяемое
NOLINK, NOLINK	



END}  
END



### КЛЮЧ\_ДЛЯ СВЯЗИ\_ФАЙЛОВ

Определяет связи между двумя файлами (для однозначных связей - только с одним файлом по этому ключу - и для связей типа “просмотр в справочнике” этот подраздел необязателен). Сравняются связывающие поля и записи в которых значения этих полей совпадают считаются связанными.

### ПОЛЕ

Задает пару связывающих файлы полей (обязательный компонент).

Имя\_файла-поле | NOLINK

Имя\_файла-поле - это метка поля, в первом файле, которое связывает (сравнивается) с полем в связанном файле. Значение NOLINK означает, что сравнения с полем в связанном файле не выполняется.

Имя\_связанного\_файла-поле | NOLINK

Имя\_связанного\_файла-поле - это имя поля в связанном файле, которое связывает (сравнивается) с полем в первом файле. Значение NOLINK означает, что сравнения с полем в связанном файле не выполняется.

### КЛЮЧ\_ДЛЯ СВЯЗИ\_ФАЙЛОВ

Определяет связи между двумя файлами (для однозначных связей - только с одним файлом по этому ключу - и для связей типа “просмотр в справочнике” этот подраздел необязателен). Сравняются связывающие поля и записи в которых значения этих полей совпадают считаются связанными.

### ПОЛЕ

Задает пару связывающих файлы полей (обязательный компонент).

Имя\_связанного\_файла-поле | NOLINK

Имя\_связанного\_файла-поле - это имя поля в связанном файле, которое связывает (сравнивается) с полем в первом файле. Значение NOLINK означает, что сравнения с полем в связанном файле не выполняется.

Имя\_файла-поле | NOLINK

Имя\_файла-поле - это метка поля, в первом файле, которое связывает (сравнивается) с полем в связанном файле. Значение NOLINK означает, что сравнения с полем в связанном файле не выполняется.

### **Пример группы определения связи**

[RELATIONS]

```

                                RELATION,MANY:ONE,UPDATE(RESTRICT),DELETE(RESTRICT)
OrderDetails      FILE(DTL:KeyProductNumber)
Products          RELATED_FILE(PRO:KeyProductNumber)
                  FILE_TO_RELATED_KEY
                  FIELD(DTL:ProductNumber,PRO:ProductNumber)
                  END
                  RELATED_FILE_TO_KEY
                  FIELD(PRO:ProductNumber,DTL:ProductNumber)
                  END
                  END

                                RELATION,MANY:ONE
Customers         FILE()
States           RELATED_FILE(STA:KeyStateCode)
                  FILE_TO_RELATED_KEY
                  FIELD(CUS:State,STA:StateCode)
                  END
                  END

                                RELATION,ONE:MANY,UPDATE(CASCADE),DELETE(RESTRICT)
Customer         FILE(CUS:KeyCustNumber)
Orders           RELATED_FILE(ORD:KeyCustDate)
                  FILE_TO_RELATED_KEY
                  FIELD(CUS:CustNumber,ORD:CustNumber)
                  FIELD(NOLINK,ORD:Date)
                  END
                  RELATED_FILE_TO_KEY
                  FIELD(ORD:CustNumber,CUS:CustNumber)
                  END
                  END
```

## **[VIEWS] - раздел описания структур VIEW**

Этот раздел имеется в файле .TXD только в лиственном экземпляре. Причем он не является обязательным. Здесь определяются все структуры VIEW.

### **Группа определения структуры VIEW**

Группа определения структуры VIEW представляет собой последовательность подразделов файла .TXD и ключевых слов, которые полностью описывают одну структуру VIEW в словаре данных. Для каждой структуры VIEW эта последовательность повторяется и, таким образом все структуры VIEW целиком определяются. Группа определения структуры VIEW содержит следующие ключевые слова и подразделы:

[VIEWS]

[LONGDESC]

Определение View

!!> список ключевых слов

Группа определения структуры x  
VIEW -повторяемая

[LONGDESC]

Определение VIEW

См. ниже Общие подразделы..(необязательная опция)

Операторы языка Clarion для объявления структуры VIEW

(обязательный элемент). Может включать в себя необязательный комментарий длиной до 40 символов.

Кроме того, определение структуры VIEW включает в себя список ключевых слов. Например:

CustView VIEW(Customers),FILTER(State="FL") !Florida only

!!> IDENT(7)

еще операторы объявления структуры VIEW

END

Список ключевых слов

Список внутренних ключевых слов, которые описывают опции, не задаваемые в операторе объявления структуры VIEW

(необязательный элемент). Список начинается с символов !!>.

Для раздела VIEW допустимо единственное ключевое слово IDENT.

Оно представляет внутренний номер-указатель, по которому Среда Clarion для Windows идентифицирует структуры VIEW. Например:

!!> IDENT(7)

### **Пример группы определения структуры VIEW**

[VIEWS]

[LONGDESC]

! это структуры View, фильтрующие всех кроме покупателей из Флориды

! из телефонного справочника включаются имя покупателя и номер телефона

CustView VIEW(Customers),FILTER(CUS:State = "FL") !View Florida only

PROJECT(CUS:ZipCode,CUS:State,CUS:City,CUS:CompanyName,CUS:Address)

!!> IDENT(7)

JOIN(PHO:KeyCustNumber,PHO:CustNumber)

PROJECT(PHO:PhoneNumber,PHO:AreaCode,PHO:FirstName,PHO:LastName)

END

END

## Общие подразделы

Эти подразделы появляются в файле .TXD множество раз. Их структура и синтаксис везде одинакова.

### [LONGDESC]

---

На нескольких строках приводится необязательный описательный текст длиной до 1000 символов. Каждая строка текста начинается восклицательным знаком и содержит до 75 символов. Текст вводится на вкладке Comments соответствующего окна Properties для словаря, файла, алиаса, связи поля или ключа. Например:

[LONGDESC]

!До 1000 символов текста, описывающего этот элемент.

!Каждая строка длиной до 75 символов и начинается восклицательным знаком

### [USEROPTION]

---

Содержит до 1000 символов текста, доступного в шаблонах (необязательный элемент). Каждая строка начинается с восклицательного знака и содержит до 75 символов.

Этот текст доступен в шаблонах, которые работают с файлами, алиасами, полями, ключами и т.д. Смотрите функцию EXTRACT, описанную Языке шаблонов.

Этот текст вводится на вкладке Options соответствующего окна Properties для файла, алиаса, связи, поля или ключа. Например:

[USEROPTION]

!ThirdPartyTemplateAttribute(on)

!DefaultWindowSize=Max

### [SCREENCONTROLS]

---

Отмечает начало необязательного подраздела, который описывает оконные объекты, используемые по умолчанию для работы полями из словаря, полями MEMO или BLOB.

Следующая за [SCREENCONTROLS], комбинация восклицательного знака и пробела отмечает начало объявления экранного объекта. Оно представляет собой оператор языка Clarion, объявляющий экранный объект, который будет использоваться для работы с этим элементом данных. Элементу данных можно поставить в соответствие различные экранные объекты, поэтому может быть несколько объявлений объектов, сразу после [SCREENCONTROLS] и вплоть до начала следующего подраздела, обычно подраздела [REPORTCONTROLS]. Например:

```
[SCREENCONTROLS]
! PROMPT('CurrentTab:'),USE(?CurrentTab:Prompt)
! ENTRY(@s80),USE(CurrentTab)
[REPORTCONTROLS]
```

### **[REPORTCONTROLS]**

---

Отмечает начало необязательного подраздела, который описывает объекты в печатном документе, используемые по умолчанию для работы полями из словаря, полями MEMO или BLOB.

Следующая за [REPORTCONTROLS], комбинация восклицательного знака и пробела отмечает начало объявления объекта для документа. Оно представляет собой оператор языка Clarion, объявляющий объект в печатном документе, который будет использоваться для работы с этим элементом данных. Элементу данных можно поставить в соответствие различные объекты, поэтому может быть несколько объявлений объектов, сразу после [REPORTCONTROLS] и вплоть до начала следующего подраздела. Например:

```
[REPORTCONTROLS]
! STRING(@s80),USE(CurrentTab)
```



## Формат файла .ТХА

### Файлы.ТХА: Описание приложения Clarion в формате ASCII

Файл .ТХА представляет собой просто текстовый вариант файла описания приложения Clarion для Windows (.APP). В этом файле Среда Clarion хранит всю относящуюся к приложению информацию, необходимую для генерации его исходного текста. В файле .ТХА в текстовом формате, пригодном для работы в любом текстовом редакторе, содержится все хранящиеся в файле . APP данные (за исключением данных, относящихся к системе описания проекта).

Файл .ТХА легко создать из файла .APP, загрузив приложение в Среду Clarion для Windows и затем выбрав в меню File д Export Text.

Зачем может понадобиться файл .ТХА? По крайней мере по трем причинам:

Копирование	Поскольку файл .ТХА также легко можно импортировать, он может служить резервной копией текущей версии Файла приложения.
Массовые изменения	В связи с тем, что с файлом . ТХА можно работать привычным для вас текстовым редакторе, всю мощь которого можно использовать для проведения массовых изменений в приложении, или подобным же образом любых изменений, которые легче сделать в текстовом редакторе.
Первая помощь	Иногда с файлом описания приложения .APP может быть связано несколько странное поведение в Среде разработки. Экспорт в .ТХА файл, а затем импорт обратно, может выявить и исправить эти проблемы.

## Структура файла .ТХА

Организация и синтаксис файла .ТХА отражает парадигму Приложение-Программа-Модуль-Процедура, на которой базируется генерация приложения в Clarion для Windows. Эту парадигму определяют шаблоны класса Clarion.

Файл .ТХА состоит из четырех основных разделов, приблизительно соответствуют выходным файлам Генератора приложений.

- ◆ Раздел [APPLICATION] соответствует файлу Имя\_прилож.APP.
- ◆ Раздел [PROGRAM] соответствует файлу Имя\_прилож.CLW.
- ◆ Раздел [MODULE] соответствует файлу Имя\_прилож00n.CLW.
- ◆ Раздел [PROCEDURE] соответствует процедурам, описанным в дереве приложения и хранящимся в файлах Имя\_прилож 00n.CLW.

### Схема файла .ТХА

---

Далее приводится упорядоченный список разделов файла .ТХА, подразделов и ключевых слов. Он приведен для того, чтобы дать вам общее представление о структуре и организации файла .ТХА в целом.

Каждый раздел начинается с заголовка, заключенного в квадратные скобки, и заканчивается [END], или началом следующего раздела. В каждом разделе могут содержаться подразделы и ключевые слова.

Начало подраздела похоже на начало основного раздела - заголовок, заключенный в квадратные скобки.

Ключевые слова записываются большими буквами, а следом - его значение. Значение для ключевых слов приводятся в различных форматах, которые описываются ниже.

Пробелы и отступы в схеме сделаны для улучшения читаемости и в реальном .ТХА файле отсутствуют.

Следом за схемой подробно обсуждается каждый раздел, подраздел и ключевое слово.

#### [APPLICATION]

VERSION

необязательно

HLP

необязательно

DICTIONARY

необязательно

PROCEDURE

необязательно



[COMMON]		
DESCRIPTION	необязательно	
LONG	необязательно	
FROM		
[DATA]		
Общая часть		
[FILES]	необязательно	
[PROMPTS]		
[EMBED]	необязательно	
[ADDITION]	необязательно, может повторяться	
[PERSIST]		
[PROGRAM]		
NAME	необязательно	
INCLUDE	необязательно	
NOPOPULATE	необязательно	
[COMMON]		
DESCRIPTION	необязательно	
LONG	необязательно	
FROM		
[DATA]		
Общая часть		
[FILES]	необязательно	
[PROMPTS]		
[EMBED]	необязательно	
[ADDITION]	необязательно, может повторяться	
[PROCEDURE]	необязательно, может повторяться	
NAME	необязательно	
PROTOTYPE	необязательно	
[COMMON]		
DESCRIPTION	необязательно	
LONG	необязательно	
READONLY	необязательно	только для процедур
FROM		
[DATA]		
Общая часть		
[FILES]	необязательно	
[PROMPTS]		
[EMBED]	необязательно	
[ADDITION]	необязательно	может повторяться
[CALLS]	необязательно	

[WINDOW]	необязательно	
[REPORT]	необязательно	
[FORMULA]	необязательно	
[END]		
[MODULE]	необязательно	может повторяться
NAME	необязательно	
INCLUDE	необязательно	
NOPOPULATE	необязательно	
[COMMON]		
DESCRIPTION	необязательно	
LONG	необязательно	
FROM		
[DATA]		
Общая часть		
[FILES]	необязательно	
[PROMPTS]		
[EMBED]	необязательно	
[ADDITION]	необязательно	может повторяться
[PROCEDURE]	необязательно	может повторяться
.		
.		
.		
[END]		

## Разделы файла .TXA

### [APPLICATION]

Этот раздел является обязательным и имеется в одном экземпляре в начале каждого файла .TXA (если только не экспортируется часть приложения т.е. файл может содержать отдельный модуль или процедуру и в этом случае файл будет соответственно начинаться с [MODULE] или [PROCEDURE]. Раздел “приложение” начинается с [APPLICATION] и заканчивается с началом программной секции ([PROGRAM]). Этот раздел содержит следующие подразделы и ключевые слова и относится ко всему приложению.

[APPLICATION]	
VERSION	необязательно
HLP	необязательно
DICTIONARY	необязательно
PROCEDURE	необязательно
[COMMON]	
DESCRIPTION	необязательно

LONG	необязательно	
FROM		
[DATA]		
[FILES]	необязательно	
[PROMPTS]		
[EMBED]	необязательно	
[ADDITION]	необязательно	может повторяться
[PERSIST]		

**VERSION** Номер версии приложения (необязательный). Это значение отражает версию Генератора и изменения в формате файла TXA. Например: VERSION 10.

**HLP** Путь и имя файла системы помощи Windows, к которому обращается данное приложение (необязательно). Если полный путь к файлу не указан, то Clarion ищет сначала в текущем каталоге, системных путях, затем в каталогах заданных в Redirection-файле (C:\CW15\BIN\CW15.RED).

Например:

HLP 'C:\CW15\APPS\MY.APP.HLP'

**DICTIONARY** Необязательный файл словаря данных, используемый приложением. Имя файла можно задать вместе с путем или без него. Если путь не задан, то Clarion просматривает пути, заданные в файле переназначений (C:\CW15\BIN\CW15.RED).

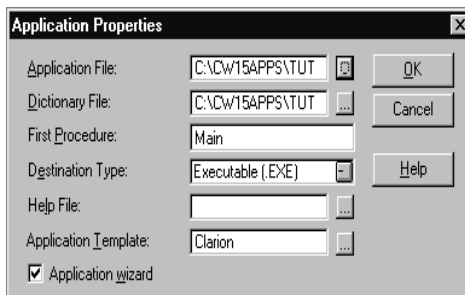
Например:

DICTIONARY 'TUTORIAL.DCT'

**PROCEDURE** Имя первой процедуры в приложении (необязательно - для библиотек это не имеет смысла). Из этой процедуры прямо или косвенно происходит обращение к другим процедурам приложения.

Например:

PROCEDURE Main



[COMMON]	Общий подраздел обязательно имеется в разделах [APPLICATION], [PROGRAM], [MODULE] и [PROCEDURE]. Этот подраздел начинается с [COMMON] и заканчивается с началом следующего подраздела. В зависимости от того, в каком разделе он находится и из каких под- подразделов состоит, этот подраздел может значительно изменяться по длине и наполнению. Полное описание смотрите далее в разделе [COMMON].
[PERSIST]	Сведения о приложении, которые хранятся в промежутках между сеансами работы (обязательный раздел). Несмотря на то, эти элементы хранятся в файле .TXA в формате [PROMPT], они никогда не видны разработчику как приглашения. Они объявлены в приложении оператором #DECLARE с атрибутом SAVE, что приводит к тому, что в файле приложения они хранятся и доступны в каждом последующем сеансе работы. Полное описание синтаксиса смотрите далее в разделе [PROMPT].

### **Пример раздела [APPLICATION]**

```
[APPLICATION]
VERSION 10
HLP 'C:\CW15\APPS\MY.APP.HLP'
DICTIONARY 'TUTORIAL.DCT'
PROCEDURE Main
[COMMON]
```

```

.
.
.
```

### **[PROGRAM]-[END]**

Этот раздел обязателен и имеется в каждом .TXA файле только в одном экземпляре, после секции [APPLICATION]. Программная секция начинается предложением [PROGRAM] и заканчивается предложением [END]. В этом разделе содержатся следующие ключевые слова и подразделы, свойственные файлу исходного текста, содержащему оператор PROGRAM.

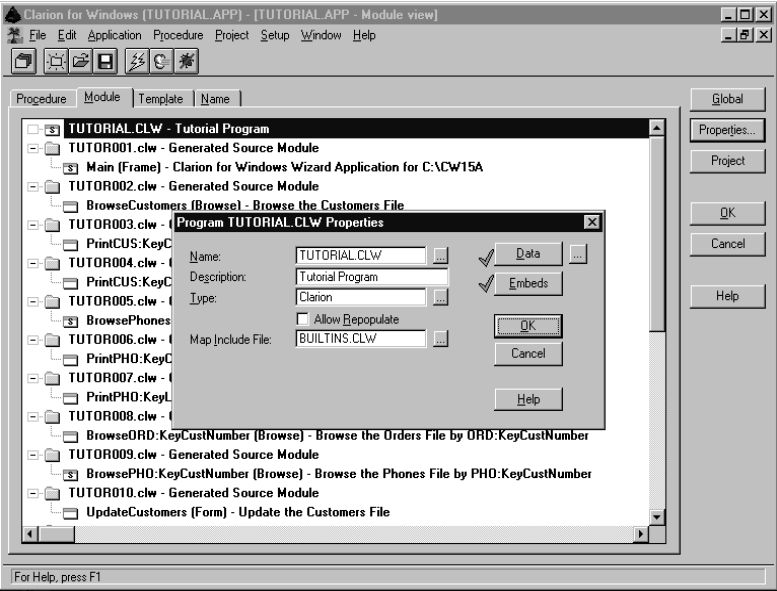
[PROGRAM]	
NAME	необязательный
INCLUDE	необязательный
NOPOPULATE	необязательный
[COMMON]	

DESCRIPTION	необязательный	
LONG	необязательный	
FROM		
[DATA]		
[FILES]	необязательный	
[PROMPTS]		
[EMBED]	необязательный	
[ADDITION]	необязательный	повторяемый
[PROCEDURE]	необязательный	повторяемый
[END]		

NAME                   Имя файла исходного текста, который содержит оператор PRO  
GRAM (необязательный элемент). Если это имя опущено, то по  
умолчанию используется имя приложения.  
Например:                   NAME 'TUTORIAL.CLW'

INCLUDE               Имя файла исходного текста, который включается в раздел  
объявления данных программного исходного модуля  
(необязательный элемент).  
Например:                   INCLUDE 'EQUATES.CLW'

NOPOPULATE           Генератор Программ не должен сохранять в этом файле процедуры  
(необязательный элемент). Обычно указывается для внешних  
модулей.  
Например:                   NOPOPULATE



[COMMON]	Общий подраздел обязательно имеется в разделах [APPLICATION], [PROGRAM], [MODULE] и [PROCEDURE]. Этот подраздел начинается с [COMMON] и заканчивается с началом следующего подраздела. В зависимости от того, в каком разделе он находится и из каких под- подразделов состоит, этот подраздел может значительно изменяться по длине и наполнению. Полное описание смотрите далее в разделе [COMMON].
[PROCEDURE]	Информация, которая полностью описывает процедуру (необязательная). Этот подраздел может повторяться для каждой процедуры в модуле. Данные, хранящиеся в этом подразделе, относятся только к процедуре указанной ключевым словом NAME. Структуру и синтаксис этого раздела смотрите раздел [PROCEDURE].

#### **Пример раздела -[PROGRAM]-[END]**

```
[PROGRAM]
NAME 'TUTORIAL.CLW'
INCLUDE 'EQUATES.CLW'
NOPOPULATE
[COMMON]
.
.
.
[END]
```

#### **[MODULE]-[END]**

---

Структура и синтаксис раздела описания модуля идентична разделу описания программы. Раздел описания модуля не является обязательным и может повторяться нужное количество раз. Раздел начинается с предложения [MODULE] и заканчивается оператором [END].

Будучи идентичными по структуре и синтаксису, подраздел описания модуля отличается по области его применимости. Он повторяется для каждого модуля в приложении, и его содержимое применимо только к исходному модулю, указанному ключевым словом NAME, и таким образом, относится только к процедурам, располагающимся в этом модуле. Определенные в его разделе [DATA] являются общими данными модуля и доступны для всех его процедур.

[PROCEDURE]

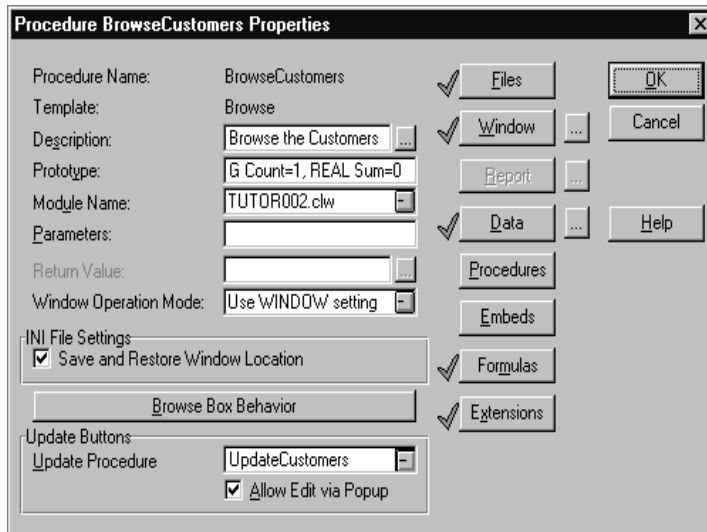
Подраздел описания процедуры не является обязательным и повторяется для каждой процедуры в программе или модуле. Этот подраздел начинается предложением [PROCEDURE] и заканчивается началом следующего подраздела или предложением [END] для описания модуля или программы. Этот подраздел содержит информацию, которая относится только к указанной ключевым словом NAME процедуре.

[PROCEDURE]	необязательный	повторяемый
NAME	необязательный	
PROTOTYPE	необязательный	
[COMMON]		
DESCRIPTION	необязательный	
LONG	необязательный	
READONLY	необязательный	
FROM		
[DATA]		
[FILES]	необязательный	
[PROMPTS]		
[EMBED]	необязательный	
[ADDITION]	необязательный	повторяемый
[CALLS]	необязательный	
[WINDOW]	необязательный	
[REPORT]	необязательный	
[FORMULA]	необязательный	
NAME	Имя процедуры к которой относятся ключевые слова в этом разделе (необязательный элемент). Например: NAME BrowseCustomers	
PROTOTYPE	Необязательный прототип процедуры. Подробнее о прототипах смотрите в Описании языка. Например: PROTOTYPE 'LONG Count, REAL Sum'	
[COMMON]	Общий подраздел обязательно имеется в разделах [APPLICATION], [PROGRAM], [MODULE] и [PROCEDURE]. Общий подраздел начинается предложением [COMMON] и заканчивается с началом нового подраздела. В зависимости от того, в каком разделе он находится и из каких под- подразделов состоит, этот подраздел может значительно изменяться по длине и наполнению. Полное описание смотрите далее в разделе [COMMON].	
[CALLS]	Процедуры, вызываемые из данной процедуры (необязательный	

элемент).

Например:

[CALLS]  
UpdateCustomers  
BrowseOrders



[WINDOW]

Операторы языка Clarion, определяющие окно, к которому относятся действия в этой процедуре (необязательный элемент).

Дополнительно к операторам языка Clarion, определяющим структуру WINDOW, этот подраздел может содержать четыре ключевых слова, которые используются Средой Разработки Clarion в своих внутренних целях.

#SEQ(номер экземпляра)

Все объекты управления, полученные на основе шаблонов объектов, имеют такое ключевое слово, которое присваивает объекту номер экземпляра объекта-шаблона, от которого этот экранный объект произведен.

#ORIG(оригинальное имя объекта)

Оригинальное имя объекта, которое дается ему на основании объекта-шаблона.

#FIELDS(список полей в окне-списка)

Список полей выводимых в объекте типа LIST.

#LINK(метка соответствия связанного поля)

Если по одному шаблону сделано несколько экранных полей, то они связываются в “кольцо”, каждое связывается со следующим. Если экранный объект вводит значение в поле из словаря данных, то с ним также связывается поле-приглашение (объект типа PROMPT).



Например:

[WINDOW]

```
QuickWindow WINDOW('Browse the Customers File'),AT(,358,188),SYSTEM,GRAY,MDI
LIST,AT(8,20,342,124),USE(?Browse:1),IMM,HVSCROLL,FORMAT('16L|M~Cust' &|
'Number~@n4@80L|M~Company Name~@S20@80L|M~Address~@S20@80L|M~City~@S20'
&|'@8L|M~State~@S2@20L|M~Zip~@S5@'),FROM(Queue:Browse:1),#SEQ(1),#ORIG(?List),
|#FIELDS(CUS:CustNumber,CUS:CompanyName,CUS:Address,CUS:City,CUS:State,CUS:Zip)
BUTTON('&Insert'),AT(207,148,45,14),USE(?Insert:2),#SEQ(2),#ORIG(?Insert),#LINK(?Change:2)
BUTTON('&Change'),AT(256,148,45,14),USE(?Change:2),DEFAULT,#SEQ(2),#ORIG(?Change),#LINK(?Delete:2)
BUTTON('&Delete'),AT(305,148,45,14),USE(?Delete:2),#SEQ(2),#ORIG(?Delete),#LINK(?Insert:2)
END
```

[REPORT]

Операторы языка Clarion, определяющие печатный отчет, к которому относятся действия в этой процедуре (необязательный элемент). Дополнительно к операторам языка Clarion, определяющим структуру REPORT, этот подраздел может содержать четыре ключевых слова, о которых шла речь выше.

[FORMULA]

Описывает каждую формулу, определенную для этой процедуры с помощью Редактора Формул (необязательный элемент). Заметьте, что ключевые слова и их значения точно соответствуют диалоговому окну Formula Editor. Подробнее о создании формул смотрите Руководство пользователя и Систему интерактивной подсказки.

Пример:

[FORMULA]

DEFINE OrderTax

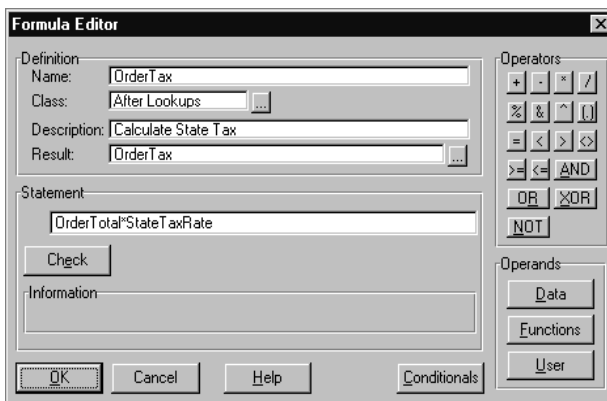
ASSIGN OrderTax

CLASS After Lookups

DESCRIPTION Calculate State Tax

= OrderTotal \* StateTaxRate

[END]



### Пример подраздела - [PROCEDURE]

```
[PROCEDURE]
NAME BrowseCustomers
PROTOTYPE 'LONG Count, REAL Sum'[CALLS]
[COMMON]
.
.
.
[CALLS]
UpdateCustomers
BrowseOrders
[WINDOW]
QuickWindow WINDOW('Browse Customers'),AT(,358,188),SYSTEM,GRAY,MDI
LIST,AT(8,20,342,124),USE(?Browse:1),IMM,FORMAT('16L|M~Cust'      &|
Number~@n4@80L|M~Name~@S20@80L|M~Address~@S20@80L|M~City~@S20' &|
'@8L|M~St~@S2@20L|M~Zip~@S5@'),FROM(Queue:Browse:1),#SEQ(1),#ORIG(?List),|
#FIELDS(CUS:CustNumber,CUS:CompanyName,CUS:Address,CUS:City,CUS:State,CUS:Zip)
BUTTON('&Insert'),AT(207,148,45,14),USE(?Insert:2),#SEQ(2),#ORIG(?Insert),#LINK(?Change:2)
BUTTON('&Change'),AT(256,148,45,14),USE(?Change:2),DEFAULT,#SEQ(2),#ORIG(?Change),#LINK(?Delete:2)
BUTTON('&Delete'),AT(305,148,45,14),USE(?Delete:2),#SEQ(2),#ORIG(?Delete),#LINK(?Insert:2)
END
[FORMULA]
DEFINE OrderTax
ASSIGN OrderTax
CLASS After Lookups
DESCRIPTION Calculate State Tax
= OrderTotal * StateTaxRate
```

## **Общие подразделы**

### **[COMMON]**

Подразделы общего описания имеются в разделах [APPLICATION], [PROGRAM], [MODULE] и [PROCEDURE]. Этот подраздел начинается с предложения [COMMON] и заканчивается с началом следующего подраздела. Подраздел общего описания является обязательным, несмотря на то, многие из содержащихся в нем ключевых слов обязательными не являются. В зависимости от того, в каком разделе он находится и из каких под- подразделов состоит, этот подраздел может значительно изменяться по длине и наполнению. Подраздел [COMMON] содержит следующие ключевые слова и подразделы.

## [COMMON]

DESCRIPTION	необязательный	
LONG	необязательный	
FROM		
[DATA]		
[FILES]	необязательный	
[PROMPTS]		
[EMBED]	необязательный	
[ADDITION]	необязательный	повторяющийся

DESCRIPTION Текст длиной до 40 символов, описывающий приложение, программу, модуль или процедуру (необязательный элемент). Например:

DESCRIPTION 'Print dynamic label report'

LONG Текст длиной до 1000 символов, описывающий приложение, программу, модуль или процедуру (необязательный элемент). Например: Этот текст в действительности располагается на нескольких строках, которые при выводе соединяются вместе.

Например:

LONG 'Print dynamic label report. At runtime, the ' LONG 'user selects (or adds a new description of) '

LONG 'a label paper from the LAB file. This proced'

LONG 'ure then makes property assignments to adjus'

LONG 't the size and location of the label text to'

LONG ' fit the selected label paper.'

READONLY Данную процедуру в Среде Разработки Clarion можно просматривать, но нельзя изменять (необязательный элемент). Допустим только в подразделе [PROCEDURE]. Сейчас это ключевое слово не добавляется Средой к процедуре, но зарезервировано для дальнейшего использования, чтобы разработчики фирмы TopSpeed могли реализовать многопользовательскую Среду Разработки, в которой можно было бы "включать" и "выключать" возможность изменения процедуры в целях обеспечения целостности программного кода. Пример:

READONLY

FROM Имя класса шаблонов для приложения или имя класса шаблонов и конкретный шаблон по которым производятся программа, модуль или процедура (необязательный элемент - может опускаться для процедур типа ToDo). Пример:

```

[APPLICATION]
.
.
.
FROM Clarion
or
[PROCEDURE]
.
.
FROM Clarion Report

```

### **Пример подраздела [COMMON]**

```

[COMMON]
DESCRIPTION      'Печать различных этикеток'
LONG              'Печать различных этикеток. Во время выполнения пользователь
                  выбирает из файла LAB (или добавляет новое описание) этикетку.
                  Затем эта процедура присваивает значения свойствам, чтобы
                  настроить размеры и расположение текста в выбранной этикетке
READONLY
FROM Clarion Report

```

### **[DATA]**

---

Подраздел данных - это необязательная часть подраздела [COMMON]. В свою очередь он может содержать несколько подразделов и ключевых слов, которые описывают определенные для этой процедуры, модуля, программы или приложения переменные в памяти. Обсуждение синтаксиса такого описания смотрите в главе Формат файла .TXD в этом руководстве. Кроме того, смотрите главу Определение данных процедуры в Руководстве пользователя.

Каждой объявленной переменной соответствует серия необязательных подразделов и ключевых слов, которые целиком описывают переменную, а также различные используемые по умолчанию указания по форматированию для Генератора программ. Полный список возможных подразделов и ключевых слов:

[DATA]	
[LONGDESC]	необязательный
[USEROPTION]	необязательный
[SCREENCONTROLS]	необязательный
[REPORTCONTROLS]	необязательный
field definition	
keyword list	необязательный

- [LONGDESC]      До 13 строк текста, каждый длиной до 75 символов (необязательный элемент). Каждая строка начинается восклицательным знаком (!). Данные заносятся на вкладке Comments в окне Field Properties.  
Пример:  
[LONGDESC]  
!CurrentTab используется в сгенерированном  
! по шаблону кода для хранения номера/идентиф-ра  
! объекта TAB, имеющего сейчас фокус.
- [USEROPTION]      До 13 строк текста, каждый длиной до 75 символов (необязательный элемент). Каждая строка начинается восклицательным знаком (!). Этот текст доступен в шаблоне. Подробнее смотрите функцию Extract в Описании языка шаблонов. Данные заносятся на вкладке Options в окне Field Properties.  
Пример:  
[USEROPTION]  
!ThirdPartyTemplateAttribute:Details(on)  
!ThirdPartyTemplateAttribute:WizardHelp(off)
- [SCREENCONTROLS]      Начало подраздела, описывающего экранные объекты, используемые по умолчанию для отображения переменных из памяти (необязательный элемент). Для установки этих “объектов по умолчанию” используется вкладка Window в диалоговом окне Field Properties  
Следующий за предложением [SCREENCONTROLS] восклицательный знак (!) отмечает начало объявления объектов. Эти объявления представляют собой операторы языка Clarion, описывающие оконные объекты управления. С переменной в памяти могут быть связаны различные оконные объекты, поэтому может быть несколько объявлений, начинающихся сразу после предложения [SCREENCONTROLS] и продолжающихся до начала следующего подраздела, обычно это подраздел [REPORTCONTROLS].  
Пример:  
[SCREENCONTROLS]  
! PROMPT(‘CurrentTab:’),USE(?CurrentTab:Prompt)  
! ENTRY(@s80),USE(CurrentTab)
- [REPORTCONTROLS]      Начало подраздела, описывающего объекты, используемые по умолчанию для вывода переменных из памяти в печатных документах (необязательный элемент). Для установки этих “объектов по умолчанию” используется вкладка Report в диалоговом окне Field

## Properties

Следующий за предложением [REPORTCONTROLS] восклицательный знак (!) отмечает начало объявления объектов. Эти объявления представляют собой операторы языка Clarion, описывающие оконные объекты управления. С переменной в памяти могут быть связаны различные оконные объекты, поэтому может быть несколько объявлений, начинающихся сразу после предложения [REPORTCONTROLS] и продолжающихся до начала подраздела определения поля.

Пример:

```
[REPORTCONTROLS]
```

```
! STRING(@s80),USE(CurrentTab)
```

## Field Definition

Обязательный список объявлений полей на языке Clarion. Включает необязательное текстовое описание длиной до 40 символов. Текстовое описание начинается с восклицательного знака (!).

Например:

```
CurrentTab STRING(80) !выбранная вкладка
```

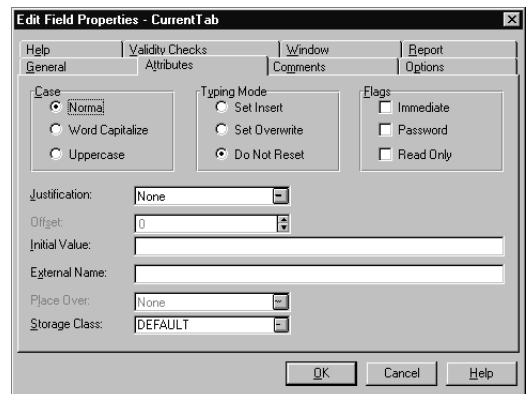
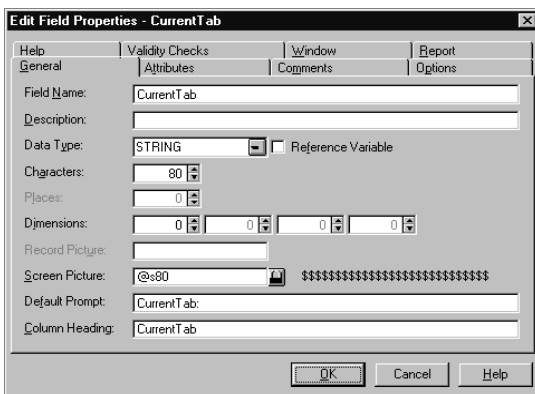
## Keyword List

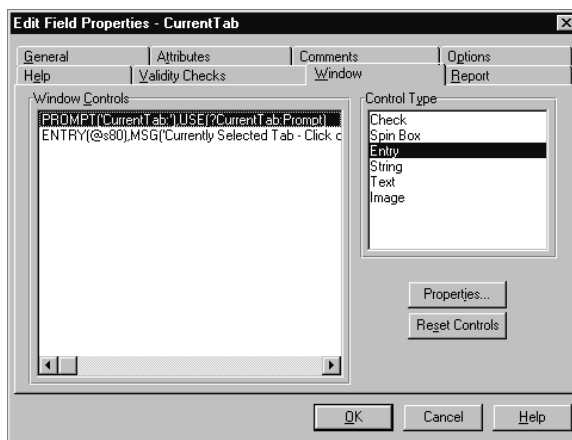
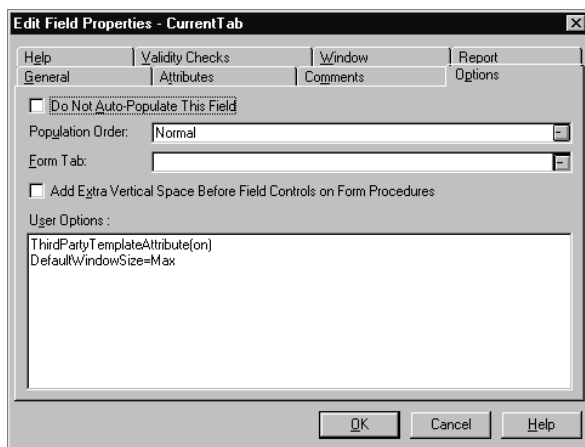
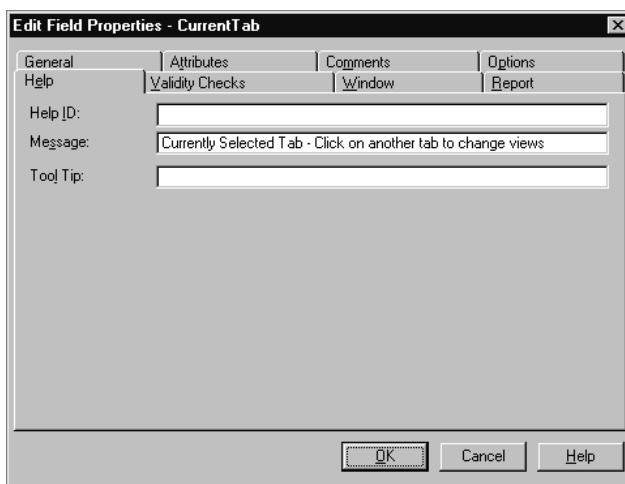
Ключевые слова, задающие различные атрибуты переменных в памяти (необязательный элемент). Список начинается с символов !!>.

Ключевые слова из этого списка устанавливаются в диалоговом окне Field Properties. Многие из них напрямую соответствуют ключевым словам языка Clarion. Более подробно о них смотрите в Описании языка.

Подробное обсуждение ключевых слов и их действия смотрите в главе Формат файла .TXD в этом руководстве. Относительно конкретных ключевых слов языка Clarion смотрите также Руководство пользователя, Описание языка и интерактивную систему помощи.

Заметим, что ключевые слова в подразделе [DATA] тесно связаны со вкладками и приглашениями в диалоговом окне Field Properties. Там их значения и устанавливаются.





### Пример подраздела [DATA]

Обычно, для любой переменной вы увидите только часть из возможных подразделов и ключевых слов, так как некоторые из них взаимоисключающие, а многие другие могут просто не потребоваться. Единственный обязательный элемент это описание на языке Clarion.

Чтобы показать как действует каждый подраздел, давайте рассмотрим построчно следующий типовой пример:

```
[DATA]
[SCREENCONTROLS]
! PROMPT('CurrentTab:'),USE(?CurrentTab:Prompt)
! ENTRY(@s80),USE(CurrentTab)
[REPORTCONTROLS]
! STRING(@s80),USE(CurrentTab)
CurrentTab          STRING(80) ! выбранная пользователем вкладка
```

```
!!> IDENT(4294967206),PROMPT('CurrentTab:'),HEADER('CurrentTab'),  
PICTURE(@s80)  
[SCREENCONTROLS]
```

·  
·  
·

Этот подраздел, описывающий все переменные в памяти для данного приложения, программы, модуля или процедуры, начинается предложением [DATA].

Предложение [SCREENCONTROLS] отмечает начало подраздела, описывающего, какого типа экранные объекты используются по умолчанию для отображения первой переменной.

Следующий за [SCREENCONTROLS] восклицательный знак (!) обозначает начало объявления экранного объекта. Объявление объекта представляет собой операторы языка Clarion, определяющие экранный объект, который будет использоваться для вывода значения переменной. С переменной в памяти могут быть связаны различные оконные объекты, поэтому может быть несколько объявлений, начинающихся сразу после предложения [SCREENCONTROLS] и продолжающихся до начала следующего подраздела, обычно это подраздел [REPORTCONTROLS]. Пример:

Предложение [REPORTCONTROLS] отмечает конец подраздела [SCREENCONTROLS] и начало подраздела описывающего, какого типа объекты используются по умолчанию для вывода значения переменной в печатных документах. Эти объявления объектов представляют собой операторы языка Clarion, описывающие оконные объекты управления. С переменной в памяти могут быть связаны различные объекты, поэтому может быть несколько объявлений, начинающихся сразу после предложения [REPORTCONTROLS] и продолжающихся до начала подраздела определения поля.

За объявлением объекта для печатного документа следует объявление переменной на языке Clarion: "CurrentTab String(80)." Это единственный обязательный элемент в каждом описании переменной в файле .TXA. За ним может идти необязательный восклицательный знак (!) и краткое описание переменной.

И наконец, символы "!!>" отмечают начало списка ключевых слов, связанных с этой переменной. Ключевые слова разделяются запятыми и список при необходимости продолжается на следующих строках, вплоть до начала следующего подраздела. Подробное обсуждение ключевых слов и их действия смотрите в главе Формат файла .TXD в этом руководстве.



**[FILES]**

Подраздел описания файлов это необязательная часть подраздела [COMMON]. Он в свою очередь может состоять из нескольких подразделов и ключевых слов, описывающих файлы, которые используются в этой процедуре, модуле, программе или приложении. В генерируемых приложениях класса Clarion подраздел [FILES] чаще всего появляется в подразделе [PROCEDURE], поскольку чаще всего доступ к файлам осуществляется в процедурах, тогда как приложение, программа и модуль более ориентированы на организацию среды для пользователя.

Для каждого используемого в процедуре, модуле, программе или приложении файла имеется последовательность подразделов и ключевых слов, которые идентифицируют файл, ключ и любые связанные файлы, которые также используются. Полный список возможных подразделов и ключевых слов:

[FILES]	необязательный
[PRIMARY]	необязательный, повторяющийся
[INSTANCE]	
[KEY]	необязательный
[SECONDARY]	необязательный, повторяющийся
[OTHERS]	необязательный
.	
.	
.	

[FILES] Начало подраздела, который несет в себе необходимую информацию об используемых этой процедурой файлах (необязательный элемент).

[PRIMARY] Начало подраздела, который описывает одну первичную файловую иерархическую структуру для управляющего шаблона в этой процедуре (обязательный элемент). Может быть подраздел [PRIMARY] для каждого управляющего шаблона в процедуре. Термин “первичная” просто означает, что это главный файл, обрабатываемый этим управляющим шаблоном. Все другие файлы, обрабатываемые этим управляющим шаблоном являются зависимыми. В строке, следующей за предложением [PRIMARY], перечисляются имена файлов.

Например:  
[PRIMARY]  
Customers

[INSTANCE] Вводит номер экземпляра для управляющего шаблона, для которого этот файл является первичным (обязательный элемент). Подробнее

о номере экземпляра смотрите описание “Общие подразделы - [ADDITION]”.

Например:  
[INSTANCE]

6

[KEY] Вводит ключ, используемый для доступа к этому файлу (необязательный элемент). Например:

[KEY]  
CUS:KeyCustNumber

[SECONDARY] Вводит порожденный и порождающий файлы, к которым осуществляется доступ в этом управляющем шаблоне (необязательный элемент). Этот подраздел повторяется для каждого связанного файла.

Оба файла указываются для того, чтобы избежать какой бы то ни было двусмысленности в том случае, когда есть связь более чем с одним файлом. Первым всегда перечисляется порожденный файл, а вторым - порождающий.

Подраздел [SECONDARY] является частью подраздела [PRIMARY] и таким образом предложение [SECONDARY] не означает окончания подраздела [PRIMARY], а продолжает его. Пример:

[SECONDARY]  
Phones Customers

[OTHERS] Вводит другие файлы, к которым осуществляется доступ в этой процедуре, но не с помощью управляющего шаблона (необязательный элемент). Это предложение отмечает также конец предыдущего подраздела [PRIMARY]. Единственный код, генерируемый для файлов из раздела [OTHERS], это код, необходимый для того чтобы открыть файл в начале процедуры, а затем закрыть файл в конце. Пример:

[OTHERS]  
Labels

Пример подраздела [FILES]

[FILES]  
[PRIMARY]  
Customers  
[INSTANCE]

```
6
[KEY]
CUS:KeyCustNumber
[SECONDARY]
Phones Customers
[SECONDARY]
Orders Customers
[OTHERS]
Labels
.
.
.
```

Чтобы показать как действует каждый подраздел, давайте рассмотрим построчно следующий типовой пример:

Предложение [FILES] отмечает начало подраздела, который указывает необходимую информацию об используемых в этой процедуре файлах.

Предложение [PRIMARY] отмечает начало подраздела, который описывает одну первичную файловую иерархическую структуру для управляющего шаблона. Термин “первичная” просто означает, что это главный файл, обрабатываемый этим управляющим шаблоном. Все другие файлы, обрабатываемые этим управляющим шаблоном являются зависимыми.

Строка сразу следом за [PRIMARY] указывает имя, то есть Customers.

Предложение [INSTANCE] означает, что следующая строка задает номер экземпляра управляющего шаблона, для которого этот файл является первичным. Среда разработки использует этот номер экземпляра для того, чтобы связать соответствующий файл с соответствующим управляющим шаблоном.

Предложение [KEY] означает, что следующая за ним строка задает ключ, используемый для доступа к первичному файлу, то есть CUS:KeyCustNumber.

Предложение [SECONDARY] означает, что следующая за ним строка задает связанный файл, к которому в этом управляющем шаблоне тоже осуществляется доступ: Phones. Заметим, что эта строка кроме вторичного файла задает и первичный: Customers. Это сделано для того, чтобы избежать какой бы то ни было двусмысленности, если имеется связь с более чем одним файлом. Например, если бы файл Orders был указан как вторичный файл наряду с файлом Phones, то важно было бы знать связан ли файл Orders с файлом Phones или с файлом Customers.

Точно так же важен порядок, в котором перечислены файлы. Первым всегда указывается

вторичный файл, а вторым - всегда первичный.

И наконец, [OTHERS] означает, что на следующей за ним строке задаются другие файлы, к которым осуществляется доступ в этой процедуре, но не в этом управляющем шаблоне. Он отмечает также конец подраздела [PRIMARY].

## **[PROMPTS]**

---

Подраздел [PROMPTS] является частью общего подраздела. В нем перечисляются шаблонные приглашения, связанные с приложением, программой, модулем или процедурой, кроме того, значения для подсказок, заданные разработчиком. Более подробную информацию смотрите в Справочнике по языку шаблонов и в интерактивной системе помощи по шаблонным подсказкам.

Шаблонные подсказки можно найти во многих подразделах, включая [PERSIST] (см. выше [APPLICATION]), [PROMPTS] и [FIELDPROMPT] (см. далее [ADDITION]).

Шаблонные приглашения и связанные с ними значения появляются в двух различных форматах: простом и зависимом.

### **Простые приглашения**

Оба формата начинаются с названия приглашения. Название приглашения начинается со знака процента (%). В простом формате за процентом следует название, тип и заключенное в скобки значение.

Возможные типы приглашений: @picture, LONG, REAL, STRING, FILE, FIELD, KEY, COMPONENT, PROCEDURE и DEFAULT. Тип приглашения может (но необязательно) уточняться ключевыми словами UNIQUE или MULTI. Последнее означает, что у приглашения есть множество значений. UNIQUE тоже означает множество значений, однако значения упорядочены по возрастанию и не повторяются.

Синтаксис простого формата представлен следующей диаграммой, в которой вертикальные колонки образованы из альтернативных параметров. В фигурных скобках необязательные параметры, а значение представляет значение для приглашения:

%name	{UNIQUE}	@picture	('value')
	{MULTI}	LONG	('value1','value2','valuen')
		REAL	
		STRING	
		FILE	
		FIELD	
		KEY	
		COMPONENT	

## PROCEDURE DEFAULT

Далее описание возможных типов:

@picture	Значением является шаблон.
LONG	Значением является число в формате LONG (4 байта целое без знака).
REAL	Значением является число в формате REAL (8 байт с плавающей точкой).
STRING	Значением является строка символов.
FILE	Значением является имя файла данных.
FIELD	Значением является имя поля в файле данных.
KEY	Значением является имя ключа.
COMPONENT	Значением является имя компоненты ключа.
PROCEDURE	Значением является имя процедуры.
DEFAULT	Обычно тоже самое, что и STRING, но отличается внутренним формат хранения. DEFAULT переменные не имеют явного типа. При присвоении они принимают тип присваиваемого значения.

Пример:

[PROMPTS]

```
%LastProgramExtension DEFAULT ('EXE')
%SizePreferences MULTI LONG (3,3)
%RangeField COMPONENT (PHO:CustNumber)
%UpdateProcedure PROCEDURE (UpdatePhones)
```

Рассмотрим каждую строку этого примера.

%LastProgram	Extension представляет собой имя шаблона для хранения значения. DEFAULT означает, что хранится в строке (поскольку 'EXE' это строка). ('EXE') задает значение для %LastProgramExtension.
%SizePreferences	представляет собой имя шаблона для хранения значения. Значение хранится в формате LONG, а MULTI означает наличие нескольких значений. (3,3) означает, что %SizePreferences присваиваются два значения 3 и 3.
%RangeField	представляет собой имя шаблона для хранения значения. Значение хранится как имя компоненты ключа и эта компонента называется PHO:CustNumber.
%UpdateProcedure	представляет собой имя шаблона для хранения значения. Значение хранится как имя процедуры и это имя UpdatePhones.

**Зависимые приглашения**

Как и простые приглашения зависимые начинаются с имени. А имя начинается со знака процента (%). В формате зависимого приглашения следом за именем идет ключевое слово **DEPEND** и приглашение распространяется на несколько строк.

Формат зависимого приглашения представлен далее диаграммой, в которой в столбик записаны альтернативные параметры, в фигурных скобках необязательные параметры, а строки **WHEN** представляют все возможные значения и для **%Prompt** и для **%ParentSymbol**.

%Prompt	DEPEND	%ParentSymbol	{UNIQUE} {MULTI}	@picture LONG REAL STRING FILE FIELD KEY COMPONENT DEFAULT	TIMES n
WHEN	(‘ParentSymbolValue’)		(‘promptvalue’)1		
WHEN	(‘ParentSymbolValue’)		(‘promptvalue’)2		
WHEN	(‘ParentSymbolValue’)		(‘promptvalue’)n		

Следующее за **DEPEND** имя **%ParentSymbol** представляет шаблонный символ, от значения которого зависит значение **%Prompt**. Таким образом **%ParentSymbol** может имеет множество различных значений, а значение **%Prompt** зависит от текущего значения **%ParentSymbol**.

Пример:

```
[PROMPTS]
%GenerationCompleted DEPEND %Module DEFAULT TIMES 4
WHEN (‘TUTORIAL.clw’) (‘1’)
WHEN (‘TUTOR001.clw’) (‘1’)
WHEN (‘TUTOR002.clw’) (‘1’)
WHEN (‘TUTOR003.clw’) (‘1’)
.
.
.
```

Давайте опять рассмотрим каждую строку примера. **%GenerationCompleted** это имя шаблона для хранения значения. Значение **%GenerationCompleted** зависит от значения **%Module**. Это значение хранится как строка. **TIMES 4** означает, что **%Module** имеет 4 разных возможных значения.

Из следующих 4-х строк по одной отводится на каждое возможное значение **%Module**. **WHEN** задает возможные значения **%Module** - ‘**TUTORIAL.CLW**’, за которым идет

соответствующее значение для %GenerationCompleted ('1').

### **Вложенные зависимые приглашения**

Зависимые приглашения могут появляться более чем на одном уровне зависимости. В каждом вложенном уровне есть свое предложение WHEN. Пример:

```
%ForegroundNormal DEPEND %Control DEPEND %ControlField LONG TIMES 2
WHEN ('?Browse:1') TIMES 2
WHEN ('CUS:CustNumber') (4294967295)
WHEN ('CUS:CompanyName') (4294967295)
WHEN ('?Browse:2') TIMES 4
WHEN ('CUS:Address') (4294967295)
WHEN ('CUS:City') (4294967295)
WHEN ('CUS:State') (4294967295)
WHEN ('CUS:ZipCode') (4294967295)
```

В этом примере значение %ForegroundNormal зависит от значения %Control, и затем от значения %ControlField. Для %Control есть два возможных значения: ?Browse:1 и ?Browse:2. Для каждого из них есть WHEN...TIMES, которые указывают число возможных значений %ControlField связанных с этим значением %Control.

Затем, каждая следующая строка WHEN...TIMES показывает возможное значение %ControlField, за которым следует соответствующее значение для %ForegroundNormal. Отметим старшинство, значение %controlfield зависит от текущего значения %control.

### **[EMBED]-[END]**

Подраздел [EMBED] представляет собой необязательную составляющую общего подраздела. В нем могут содержаться различные подразделы и ключевые слова, которые описывают вставку исходного текста, определенную в этой процедуре, модуле или программе с помощью диалогового окна Embedded Source. Смотрите в Руководстве пользователя раздел Определение вставляемого текста.

В подразделе [EMBED] могут быть следующие подразделы и ключевые слова:

[EMBED]	необязательно
EMBED	может повторяться
[INSTANCES]	необязательно, может повторяться
WHEN	
[DEFINITION]	
[SOURCE]	необязательно, может повторяться
[TEMPLATE]	необязательно, может повторяться
[PROCEDURE]	необязательно, может повторяться
[GROUP]	необязательно, может повторяться
INSTANCE 4	
[END]	

[END]

[END]

[EMBED]-[END]

Отмечает начало этого подраздела, описывающего каждую точку вставки, определенную в этой процедуре (необязательный элемент). Следующая за этим ключевым словом строка идентифицирует точку вставки (обязательный элемент). Предложение EMBED указывается для каждой “заполненной” точки вставки.

EMBED

Пример:

EMBED %ControlPreEventHandling

[INSTANCES]-[END]

Указывает что существует более чем один экземпляр этой точки вставки (необязательный элемент). Смотри ниже пример.

WHEN

Указывает, в какой экземпляр точки вставки вставляются данные операторы (обязательный элемент).

Пример:

[INSTANCES]

WHEN ‘?Change:2’

[DEFINITION]-[END]

Отмечает начало подраздела, который определяет вставляемые операторы (обязательный элемент).

[SOURCE]

Обозначает введенный в свободной форме в текстовом редакторе текст (необязательный элемент).

Пример:

[SOURCE] ! Это точка вставки исходного текста

[TEMPLATE]

Обозначает операторы исходного текста в свободной форме, которые включают в себя операторы языка шаблонов (необязательный элемент).

Например:

[TEMPLATE]!Это точка вставки кода на языке шаблонов

#FOR(%LocalData)

! %LocalData

#ENDFOR

[PROCEDURE]

Обозначает операторы исходного текста, представляющие обращения к процедурам, вводимые в диалоговом окне Procedure to Call.

Например:

[PROCEDURE]

EmbeddedProcedureCall

[GROUP]

Обозначает вставляемый текст как кодовый шаблон.

INSTANCE

Задает номер экземпляра вставленного кодового шаблона. См. далее

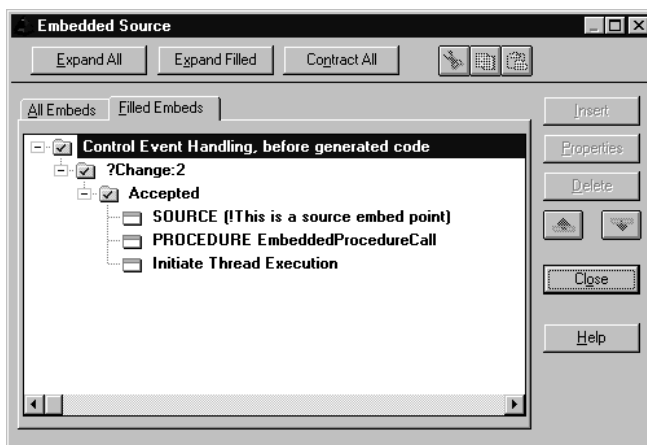
[ADDITION].



Пример:  
 [GROUP]  
 INSTANCE 4

### **Пример подраздела - [EMBED]-[END]**

Давайте рассмотрим обстоятельный пример, иллюстрирующий каждый компонент подраздела [EMBED]. Во-первых, отметим, как текст в файле .TXA очень похож на текст в диалоговом окне Embedded Source:



```
[EMBED]
EMBED          %ControlPreEventHandling
[INSTANCES]
WHEN ' ?Change:2'
[INSTANCES]
WHEN 'Accepted'
[DEFINITION]
[SOURCE]      !This is a source embed point
[PROCEDURE]
EmbeddedProcedureCall
[GROUP]
INSTANCE 4
[END]
[END]
[END]
EMBED          %ControlEventHandling
...
[END]
```

Предложение [EMBED] отмечает начало подраздела. Этот подраздел всегда заканчивается предложением [END].

EMBED %ControlPreEventHandling идентифицирует точку, где вставляется текст. Заметим, что название точки вставки в подразделе [EMBED] слегка отличается от названия в диалоговом окне Embedded Source. Там оно для большей ясности максимально расширено.

Предложение [INSTANCES] показывает, что существует не единственная точка вставки %ControlPreEventHandling, то есть для каждого объекта в процедуре есть экземпляр такой точки вставки. Каждый экземпляр заканчивается предложением [END].

Предложение WHEN ‘?Change:2’ показывает, к какой реализации точки вставки относятся вставляемые операторы. Во время генерации кода имеется переменная (или макрос) с именем %Control, которой присваивается значение ‘?Change:2’, указывающее какой объект, и соответственно какая точка вставки обрабатывается.

Пятая и шестая строки представляют еще один уровень реализации этой точки вставки. Точка вставки имеет свой номер экземпляра не только для каждого объекта, но и для каждого события связанного с этим объектом. Во время генерации кода имеется переменная (или макрос) с именем %ControlEvent, значение которой, ‘Accepted’, показывает, какое связанное с объектом событие, и соответственно, какая точка вставки обрабатывается.

Предложение [DEFINITION] отмечает начало подраздела, который определяет вставляемые операторы. Заканчивается этот подраздел предложением [END].

Предложение [SOURCE] указывает тип вставляемого текста: произвольный текст из текстового редактора. Произвольным текст идет на следующей строке и продолжается на последующих до тех пор, пока не начнется следующий подраздел файла .TXA.

Предложение [PROCEDURE] указывает другой тип вставляемого текста: обращение к процедуре. Сам вызов процедуры идет на следующей строке.

Предложение [GROUP] задает третий тип вставляемого текста: программный шаблон. Программный шаблон описывается в подразделе [ADDITION]. Предложение INSTANCE 4 означает, что номер подраздела экземпляра в подразделе [ADDITION], в котором описывается вставляемый программный шаблон (см. далее подраздел [ADDITION]). Название [GROUP] более правильно, чем исторически сложившееся [CODE].

## **[ADDITION]**

---

Подраздел [ADDITION] представляет собой необязательную часть общего подраздела и появляется по одному разу для каждого используемого программного шаблона. В этом подразделе могут быть различные подразделы и ключевые слова, описывающие каждый шаблон объекта, программный шаблон или шаблон расширения, определенный для данной

процедуры, модуля или программы. См. в Руководстве пользователя главу Использование шаблона объекта, программного шаблона и шаблона расширения.

Подраздел [ADDITION] может содержать следующие подразделы и ключевые слова:	
[ADDITION]	повторяемый
NAME	
[FIELDPROMPT]	необязательный
[INSTANCE]	повторяемый
INSTANCE	
PARENT	необязательный
PROCPROP	необязательный
[PROMPTS]	необязательный
[ADDITION]	Отмечает начало подраздела (необязательный элемент). Появляется по одному разу для каждого использованного типа шаблона. Таким образом, если шаблон “BrowseBox” использовался в процедуре дважды, то в разделе [PROCEDURE] есть только один подраздел BrowseBox [ADDITION], но несколько подразделов [INSTANCE] (см. ниже).
NAME	Указывает класс шаблона и конкретный вызываемый шаблон (обязательный элемент). Имеется в каждом подразделе [ADDITION] в одном экземпляре. Например: NAME Clarion BrowseUpdateButtons
[FIELDPROMPT]	Указывает приглашение и соответствующие ему тип и значение (необязательный элемент). Генерируется, только если вы использовали в шаблоне оператор #FIELD. Само приглашение начинается на следующей строке. См. выше описание подраздела [PROMPTS]. Пример: [FIELDPROMPT] %MadeItUp LONG (1)
[INSTANCE]	Вводит номер экземпляра ( на следующей строке). Для каждого шаблона объекта, программного шаблона и шаблона расширения в приложении, программе, модуле или процедуре приводится один раз. См. выше NAME.
INSTANCE	Указывает номер (идентификационный номер) этого конкретного шаблона расширения. Пример: [INSTANCE] INSTANCE 2
PARENT	Показывает, что этот шаблон объекта зависит от другого шаблона

объекта (необязательный элемент). Следом за PARENT идет номер шаблона объекта, от которого зависит данный шаблон.

Пример:

PARENT 1

PROCPROP

Означает, что приглашение для данного шаблона объекта высвечивается в диалоговом окне Procedure Properties (необязательный элемент). Если PROCPROP отсутствует, то в окне Procedure Properties это приглашение не выводится.

Пример:

PROCPROP

[PROMPTS]

Отмечает начало списка приглашений для этого шаблона объекта, и значения, заданные разработчиком для каждого из них (необязательный элемент). Сами приглашения начинаются на следующей строке и идут вплоть до начала следующего подраздела файла .TXA. См. выше подраздел [PROMPTS].

Пример:

[PROMPTS]

%UpdateProcedure PROCEDURE (UpdatePhones)

%EditViaPopup LONG (1)

### **Пример подраздела [ADDITION]**

И снова проиллюстрируем каждый компонент подраздела [ADDITION] исчерпывающим примером:

[ADDITION]

NAME Clarion BrowseUpdateButtons

[FIELDPROMPT]

%MadeItUp LONG (1)

[INSTANCE]

INSTANCE 2

PARENT 1

PROCPROP

[PROMPTS]

%UpdateProcedure PROCEDURE (UpdatePhones)

%EditViaPopup LONG (1)

.

.

.

[INSTANCE]

INSTANCE 4

PARENT 3

.

Предложение [ADDITION] отмечает начало подраздела.

Предложение NAME задает класс шаблонов (Clarion) и указывает конкретный вызываемый шаблон (BrowseUpdateButtons).

Предложение [FIELDPROMPT] указывает приглашение и связанные с ним тип и значение. Эта часть генерируется, только если в шаблонах использовано предложение #FIELD. Приглашения начинаются на следующей строке. Подробное обсуждение синтаксиса приглашений смотрите выше в подразделе [PROMPTS].

Предложение [INSTANCE] вводит номер экземпляра. INSTANCE 2 на следующей строке указывает номер реализации данного конкретного шаблона в подразделе “addition”.

PARENT 1 означает, что этот шаблон объекта зависит от другого шаблона, чей номер равен 1. Таким образом, шаблон BrowseUpdateButtons ощущается, только если есть также шаблон BrowseBox. Далее, шаблон BrowseUpdateButtons связывается с конкретным BrowseBox, чей номер экземпляра равен 1. Когда в процедуре несколько окон-таблиц (с шаблоном BrowseBox), очень важно их различать.

PROCPROP означает, что приглашение для данного шаблона объекта выводятся в диалоговом окне Procedure Properties. Если PROCPROP отсутствует, то данные приглашения не будут выводиться в диалоге Procedure Properties.

Предложение [PROMPTS] отмечает начало списка приглашений для данного шаблона объекта и значений, присвоенных разработчиком каждому приглашению. Сами приглашения начинаются со следующей строки и продолжаются до начала следующего подраздела файла .TXA. См. выше подраздел [PROMPTS].



## **Система поддержки проекта TOPSPEED**

### **Введение**

Система поддержки проекта TopSpeed интегрируется со Средой Разработки Clarion для Windows. Она представляет собой мощный последовательный язык, в котором сочетаются функциональность командного процессора, компоновщика, и интеллектуальной системы компиляции и компоновки.

В диапазоне от создания простого .EXE модуля и вплоть до сложного проекта из нескольких библиотек .DLL система поддержки проекта TopSpeed обеспечивает вам полный контроль над процессами компиляции и компоновки.

Основными преимуществами использования системы поддержки проекта TopSpeed являются автоматизация, эффективность и точность. Одной командой можно заново собрать весь проект независимо от его сложности и при этом можно быть уверенным, что в процессы компиляции и компоновки будут включены нужные исходные и объектные модули. При этом не нужные компоненты не будут вовлечены в обработку. Кроме того, простой установкой переключателя можно сделать различные версии проекта: основной релиз, отладочная версия, демонстрационно-оценочная версия и т.д.

Вот простой образец описания некоторого проекта, сгенерированного Генератором приложений Clarion для Windows.

```
#noedit
#system win
#model clarion dll
#pragma debug(vid=>full)
#compile QWKTU_RD.CLW- GENERATED
#compile QWKTU_RU.CLW- GENERATED
#compile QWKTU_SF.CLW- GENERATED
#compile QWKTUTOR.clw /define(GENERATED=>on)- GENERATED
#compile QWKTU001.clw /define(GENERATED=>on)- GENERATED
#compile QWKTU002.clw /define(GENERATED=>on)- GENERATED
#compile QWKTU003.clw /define(GENERATED=>on)- GENERATED
#pragma link(C%L%TPS%S%.LIB)- GENERATED
#link QWKTUTOR.EXE
```

## **Компоненты языка**

Ключевые слова начинаются с символа фунта ( # ). В приведенном выше примере для улучшения читаемости каждое ключевое слово начинается с новой строки. Но это вовсе не обязательно.

Компоненты начинаются с двойного знака переноса ( — ) и заканчиваются возвратом каретки или переводом строки.

Макросы заключены между знаками процента (%). Можно рассматривать макросы как переменные - как только система встречает заключенное в проценты имя макроса она подставляет вместо него его текущее значение. Смотрите ниже раздел Макросы системы поддержки проекта.

Все остальное, что вы видите в примере - это параметры ключевых слов. Параметры и их синтаксис рассматриваются вместе с каждым ключевым словом.

Система поддержки проекта TopSpeed распознает следующие ключевые слова.

<u>#abort</u>	<u>#expand</u>	<u>#older</u>
<u>#and</u>	<u>#file</u>	<u>#or</u>
<u>#autocompile</u>	<u>#if</u>	<u>#pragma</u>
<u>#compile</u>	<u>#ignore</u>	<u>#prompt</u>
<u>#declare_compiler</u> <u>#implib</u>	<u>#run</u>	
<u>#dolink</u>	<u>#include</u>	<u>#rundll</u>
<u>#else</u>	<u>#link</u>	<u>#set</u>
<u>#elsif</u>	<u>#message</u>	<u>#split</u>
<u>#endif</u>	<u>#model</u>	<u>#system</u>
<u>#error</u>	<u>#noedit</u>	<u>#then</u>
<u>#exemod</u>	<u>#not</u>	<u>#to</u>
<u>#exists</u>		

## **Файлы и редактирование**

Для Среды Clarion для Windows команды системы поддержки проекта содержатся или в файле с .PRJ или файле .APP. Изменения в файл приложения (.APP) вносятся только самой Средой Разработки, а файлы проектной системы (.PRJ) являются просто текстовыми файлами. А изменения в них можно вносить как установками параметров в Среде (см. Руководство пользователя, глава Использование системы ведения проекта), так и привычным для вас текстовым редактором.

### **#noedit**

Команду #noedit можно поместить в начале файла проектной системы, для того, чтобы



запретить его редактирование Средой TopSpeed. На Среду Clarion для Windows она не оказывает никакого действия.

## Макросы проектной системы

Макросы представляют собой специальные строки, которые означают, что в этом месте требуется подстановка переменного значения. Можно трактовать макросы как просто переменные.

Последовательность символов, заключенная между знаками процента (%), обозначает имя макроса. В имени допустимы следующие символы:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 _
```

Заключительный процент можно опустить, если следующий за именем символ не из приведенного выше списка.

Всякий раз когда встречается ограниченное процентами имя макроса, оно заменяется строкой, связанной с этим именем, или, если имени не назначено никакого значения, пустой строкой. Строка для подстановки назначается имени с помощью команды #set.

Когда в подставляемой строке требуется символ процента, то следует поставить два символа процента подряд. Этот способ (два процента подряд) используется и когда необходимо осуществить вложенную подстановку. Например:

```
#set echo = '#message %%мумас'- Строка '#message %мумас'
                               связывается с именем echo
```

```
#set мумас = 'Hello'
— Строка 'Hello' связывается с именем мумас
```

```
%echo — Вместо %echo подставляется '#message %мумас'
— а вместо %мумас подставляется 'Hello'
```

```
#set мумас = 'World'
— Строка 'World' связывается с именем мумас
```

```
%echo — Вместо %echo подставляется '#message %мумас'
— а вместо %мумас подставляется 'World'
```

Если бы в приведенном выше примере было указан один процент, то макрос %мумас был бы заменен пустой строкой. Два процента подряд в приведенном выше примере дадут в результате следующие команды системы поддержки проекта:

```
#message Hello
#message World
А один символ процента:
#message ""
#message ""
```

## Установка значения макроса

---

### #set

#### **#set macroname = строка**

Команда #set связывает имя макроса со строкой. Любое присвоенное ранее значение утрачивается. Имя макроса в команде #set не должно заключаться между символами процента. Если внутри имени содержатся пробелы или ключевые слова системы поддержки проекта, то оно должно быть взято в одинарные кавычки.

Например:

```
#set cwindow = TopSpeed
#set linkit = '#link myfile'
...
#if '%cwindow'= TopSpeed #then
  #pragma link(CS_GRAPH.LIB)
#endif
%linkit
```

### #expand <file-name>

Имя файла является предметом анализа при переопределении каталогов и для этого определены следующие макросы:

%cpath     Содержит полную спецификацию файла, который должен создаваться.  
%opath     Содержит полную спецификацию файла, который должен открываться.  
%ext     Содержит расширение файла.  
%tail     Содержит имя файла без расширения, пути и диска.  
%cdir     Содержит каталог, в котором должен создаваться файл.

%odir     Содержит каталог, в котором файл открывается для чтения (если файл не существует, то %opath содержит то же, что и %cpath).

Например, предположим в redirection-файле есть строка:

```
*.def : . ; c:\ts\include
существует файл c:\ts\include\io.def, и текущим является каталог d:\test
#expand io.def
эквивалентно:
#set opath = d:\test\io.def
#set cpath = c:\ts\include\io.def
```

```
#set ext  = .def
#set tail = io
#set odir = d:\test\
#set cdir = c:\ts\include\
```

### **#split <filename>**

Имя файла разделяется на собственно имя и расширение. Для этого определены макросы:

%ext	Содержит расширение файла.
%name	Содержит собственно имя файла.

Например:

```
#split d:\name.exe
```

эквивалентно следующему:

```
#set ext  = exe
#set name = d:\name
```

## **Специальные макросы проектной системы**

---

Ряд макросов используется проектной системой для особых нужд, и поэтому следует избегать случайного определения макросов с такими же именами. Также следует избегать определения макросов со знаком подчеркивания в конце имени.

Список в алфавитном порядке всех специальных макросов:

%action	В зависимости от запроса устанавливает создание (make), компоновку (link), компиляцию (compile) или выполнение (run).
%cdir	Значение устанавливается командой #expand.
%compile_src	В режиме компиляции содержит имя компилируемого файла, вместе с путем и расширением, когда это возможно. В других режимах содержит пустую строку.
%cpath	Значение устанавливается командой #expand.
%devsys	Устанавливается Средой Разработки Clarion.
%editfile	Содержит имя файла, редактируемого в “самом верхнем” окне. Если нет открытого окна редактирования, или установлен пакетный режим, то содержит пустую строку.
%editwin	Содержит номер (0-9) самого верхнего окна. Если нет открытого окна редактирования, или установлен пакетный режим, то содержит пустую строку.
%errors	Содержит число ошибок, найденных в предыдущем процессе компиляции, или установленное командой #file adderrors.
%ext	Значение устанавливается командами #split и #expand.
%filetype	Значение устанавливается командой #system равным ее второму аргументу, и проверяется командой #link.

%jpicall	Значение устанавливается командой #model равным ее второму аргументу, и проверяется командой #link.
%L	Значение устанавливается командой #model равным L или W, что означает компоновку одного EXE модуля или компоновку с динамической библиотекой. В команде #link это значение используется для того, чтобы произвести имя файла, какой-либо требуемой библиотеки.
%link	Содержит текущий список компоновки.
%link_arg	Значение устанавливается командой #link равным ее аргументу.
%main	Содержит подразумеваемое имя главного исходного модуля. В режиме создания или компоновки, если используется файл проекта с именем отличным от UNNAMED.PR, значение этого макроса производится от имени файла проекта без расширения и пути. В противном случае макрос содержит заданное имя файла дополненное путем и расширением, если они были указаны.
%make	Устанавливается командами #compile, #link и #dolink к значению on или off, для того, чтобы указать, что результирующий файл соответствовал (или не соответствовал) по дате.
%manual_export	Для того, чтобы указать, что при компоновки динамической библиотеки DLL командой #link не должен создаваться файл .LIB. Если значение этого макроса не установлено, то библиотечный файл .LIB автоматически будет создаваться из соответствующего файла .EXP, если он имеется (смотри далее раздел Файл описания модуля) или из объектных файлов из списка компоновки.
%model	Устанавливается командой #model равным ее первому аргументу и проверяется командой #link.
%name	Устанавливается командой #split
%obj	Устанавливается равным имени объектного файла в команде #compile.
%odir	Устанавливается командой #expand
%opath	Устанавливается командой #expand
%pragmastring	Всегда будет расширяться до текущего состояния установок макроса #pragma - используется в целях отладки.
%prjname	Устанавливается к подразумеваемому имени проекта - обычно производится от имени файла описания проекта, но без пути и расширения. Если используется проектный файл UNNAMED.PR, то значение этого макроса производится от имени главного исходного модуля без расширения.
%remake	Используется внутри макроса declare_compiler для определения того, требуют ли повторного создания зависимые исходные/объектные модули.
%remake_jpi	Используется внутри макроса declare_compiler для определения того, требуют ли повторного создания зависимые исходные/объектные

модули. Макрос `%remake_jri` следует использовать для объектных файлов, созданных компиляторами TopSpeed, и которые содержат дополнительную информацию.

`%reply`

Устанавливается командой `#prompt`

`%S`

Устанавливается командой `#system` равным 16 или 32, что указывает на набор команд, используемый для создания проекта. Команда `#link` использует этот макрос для того, чтобы создать имя какой-либо требуемой библиотеки.

`%src`

В команде `#compile` этому макросу присваивается имя исходного модуля.

`%system`

Устанавливается командой `#system` к значению равному ее первому аргументу и проверяется командами `#model` и `#link`.

`%tail`

Устанавливается командой `#expand`.

`%tsc`

Устанавливается в “on”, если компилируется исходный файл на C или C++.

`%tscpp`

Устанавливается в “on”, если компилируется исходный файл на C++.

`%tsm2`

Устанавливается в “on”, если компилируется исходный файл на Modula-2.

`%tspas`

Устанавливается в “on”, если компилируется исходный файл на Паскале.

Приведенные выше макросы проверяются командой `#link` для того, чтобы определить, какие включать библиотеки, а затем устанавливается в “off”.

`%warnings`

Равно числу предупреждений, выданных компилятором, или установленному командой `#file adderrors`.

## ОСНОВЫ КОМПИЛЯЦИИ И КОМПОНОВКИ

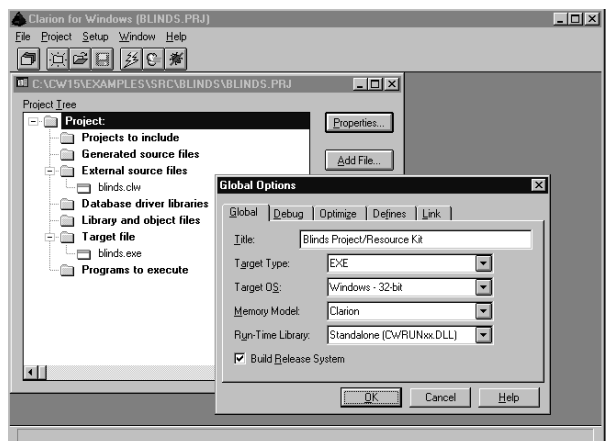
### Установка опций компиляции и компоновки

Опции компиляции и компоновки задаются в файле проекта посредством команд `#system`, `#model` и `#pragma`.

#### `#system`

`#system operating_system [ target_type ]`

Команда `#system` используется для того, чтобы задать операционную систему, для которой создается файл, и вид файла. Макросы `%system` и `%filetype` задают первый и второй аргументы. Смотрите далее раздел Специальные макросы системы поддержки



проекта.

Первый аргумент задает операционную систему, для которой создается приложение, и может принимать значения “win” и “win32”.

Второй аргумент обозначает тип создаваемого файла и может иметь значение “exe”, “lib”, или “dll”. Если второй аргумент опущен, то подразумевается “exe”.

Команда `#system` влияет на работу последующих команд `#model` и `#link`. Поэтому, она должна указываться прежде любой из них. Если в проекте окажется несколько команд `#system`, то за каждой из них должна идти команда `#model`, для того чтобы команда `#system` влияла на процесс создания файла.

### **#model**

#### **#model memory\_model [ linking\_convention ]**

Команда `#model` используется для указания модели памяти, которая должна использоваться при последующей компиляции и компоновке. Эта модель памяти будет продолжать использоваться до тех пор, пока не будет изменена явно, прагмами или другой командой `#model`.

Команда `#model` устанавливает значения макросов `%model` и `%jpicall` равным, соответственно, ее первому и второму аргументам.

Например:

```
#model clarion dll
```

равнозначно:

```
#set %model = 'clarion'
```

```
#set %jpicall = 'dll'
```

Первый аргумент задает модель памяти, которая для проектов Clarion для Windows равна ‘clarion’. Второй аргумент означает способ компоновки, который может быть `dll`, `lib`, или `owndll`. Если этот аргумент опущен, то подразумевается `dll`.

Установка второго аргумента в `dll` указывает, что вы создаете `exe`-модуль или `dll`-библиотеку, из которых происходит обращения к стандартным `dll`-библиотекам Clarion. Установка второго аргумента в `lib` означает, что вы создаете `exe`-модуль или библиотеку `lib` или `dll`, которые включают в себя все компоненты стандартных библиотек Clarion (и файловых драйверов). Использование `owndll` означает, что производится перекомпоновка `dll`, созданного ранее по соглашениям `lib`, и поэтому повторно не включаются стандартные `dll`-библиотеки Clarion.

Команда `#system` должна указываться прежде первой команды `#model`.

## #pragma

**#pragma <#pragma> { , <#pragma> }**

Команда #pragma изменяет состояние опций #pragma, влияющих на поведение компиляторов и компоновщика TopSpeed. Синтаксис и значения всех опций #pragma, описываются далее в разделе Прагмы TopSpeed.

Особый макрос %pragmastring расширяет текущее состояние всех опций #pragma, которые находятся не в принятом по умолчанию состоянии - это может быть полезным для точного определения, какие опции используются при данной компиляции. Например:

```
#message '%pragmastring'
```

## **Команды компиляции и компоновки**

---

При каждой компиляции и компоновке текущие установки компилятора и компоновщика сравниваются с теми, что использовались при последней обработке этого модуля, чтобы выяснить не было ли изменений. Если компиляция или компоновка необходима, то текущие установки передаются компилятору или компоновщику.

## #compile

**#compile<source> [ #to <object> ] [ / <#pragma> { , <#pragma> } ]  
{ , <source> [ #to <object> ] [ / <#pragma> { , <#pragma> } ] }**

Команда #compile приводит к тому, что каждый упомянутый в ней файл компилируется (если необходимо). Используя команду #to, можно задать имя объектного файла. Если команда #to опущена, то имя производится из имени исходного модуля, но с расширением .obj.

Любые прагмы, указанные в команде #compile относятся только к одному исходному модулю, имя которого предшествовало символу /.

Если компиляция необходима, то значение макроса %make устанавливается в “on”, в противном случае - в “off”. Макросам %src и %obj присваиваются имена исходного и объектного файлов.

Каждый объектный файл добавляется в список компоновки, т.е. указывается явно:  
#pragma link( %obj )

Например:

```
#compile fred.c #to fred.obj
```

```
#compile george.cpp /debug(vid=>full)
```

Используя команду #declare\_compiler, можно переопределить поведение системы поддержки проекта при компиляции файлов с заданным расширением. Ее можно также

использовать для того, чтобы объявить действия, которые следует выполнить с файлами с другим расширением - например, для использования сторонних компиляторов и препроцессоров. Смотрите далее раздел Прочие команды.

### **#link**

#### **#link <target\_filename>**

Команда **#link** компоует вместе (если нужно) все файлы из списка компоновки в указанный исполняемый модуль или библиотеку. Тип файла определяется по расширению назначенного результирующего файла, или, если не указано расширение по типу файла, заданного в последней команде **#system**. Если ничего не указано, то по умолчанию создается исполняемый модуль. Действие команды **#link** состоит в том, что макросу **%link\_arg** присваивается заданное имя.

Система ведения проекта поддерживает список тех файлов, которые должны при следующем создании исполняемого модуля или библиотеки использоваться для компоновщика как входные. Этот список называется список компоновки. С помощью команды **#pragma link** имя файла можно добавить в список компоновки.

Например:

```
#pragma link (mylib.lib)
```

Однако, необходимость использовать **#pragma link** явно возникает редко, так как все компиляторы TopSpeed при каждой компиляции исходного файла с помощью **#compile** добавляют полученный объектный файл в список компиляции. К тому же когда встречается команда **#link** все необходимые стандартные библиотеки и другие объектные файлы, импортируемые теми модулями, которые уже находятся в списке, также добавляются в него. После каждой компоновки список очищается.

Команда **#link** отличается от похожей команды **#dolink**. Это отличие в том, что (насколько это может определить Система поддержки проекта) любые дополнительно необходимые объектные файлы, перед компоновкой добавляются в список компоновки. Это относится и к библиотекам TopSpeed и также к (благодаря явному указанию команды **#autocompile**) ко всем модулям, импортированным предложением **IMPORT** в TopSpeed Модулао-2 или оператором **#pragma link** в исходных модулях TopSpeed C или TopSpeed C++. Кроме того, команда **#link** по типу результирующего файла определит любую дополнительную обработку, которую необходимо выполнить над выходным файлом.

При определенных обстоятельствах использование команды **#link** может оказаться затруднительным - например, если требуется специализированный начальный файл ("startup file") или когда строится библиотека, для которой предпочтителен непосредственный контроль над тем, какие файлы включаются в библиотеку. В этих случаях следует использовать команду **#dolink**.



**#dolink <target\_filename>**

Команда #dolink берет объектные файлы, которые предварительно были добавлены в список компоновки, и компоует их в исполняемый модуль или библиотеку (в зависимости от расширения указанного результирующего файла), сохраняя, если требуется новейший файл. В список компоновки ничего не добавляется, поэтому все нужные файлы должны быть предварительно указаны с помощью команд #pragma link, #pragma linkfirst, #compile и #autocompile. Для простых проектов использование #link предпочтительнее, потому что системой ведения проекта динамически поддерживается список компоновки, освобождая разработчика от этой заботы.

После окончания компоновки команда #dolink очищает список компоновки.

Смотри также:

#pragma link\_options (link)

**#autocompile**

Команда #autocompile проверяет объектные файлы, присутствующие в списке компоновки, на предмет необходимости перекомпиляции и включения в процесс компоновки. Это относится к объектным модулям, указанным в исходных файлах TopSpeed С или С++ прагмой #pragma link, или в случаях модульно-ориентированных языков, таких как TopSpeed Modula-2.

Каждый получившийся в результате этой команды объектный файл, который еще не указан в списке компоновки, при необходимости перекомпилируется и добавляется в список компоновки. Если какому-либо объектному модулю могут соответствовать несколько исходных модулей, то выдается сообщение об ошибке. Процесс проверки объектных модулей повторяется до тех пор, пока не будут обработаны все изменения в списке компоновки.

В простых проектах нет необходимости использовать команду #autocompile, в них вместо #dolink чаще используется команда #link, которая неявно выполняет #autocompile.

**#ignore****#ignore <filename>****#ignore #pragmastring**

Команда #ignore имеет две формы. Первая, в которой задается имя файла, предписывает Системе ведения проекта при принятии решения о том, выполнять компиляцию или нет, игнорировать данные, находящиеся в упомянутом файле. Эта возможность полезна для того, чтобы предотвратить массовую перекомпиляцию модулей, когда в широко используемый заголовок вносятся “незначительные” изменения.

Специальная форма #ignore #pragmastring предписывает Системе ведения проекта при решении вопроса компилировать или нет файл игнорировать установки #pragma. Это может быть полезным, например, когда определяется новое макрорасширение, но не нужно

перекомпилировать все.

### **#implib**

#### **#implib <libfilename>**

Команда #implib используется для создания (если это необходимо) файла динамической библиотеки. У этой команды есть две формы, которые работают немного по-разному. Если указано имя файла, то это имя библиотеки, создаваемой на основе списка компоновки и объектных файлов. Объектные файлы просматриваются и экспортируются все общие функции и переменные.

Например:

```
#pragma link( fred.obj, joe.obj )  
#implib mylib.lib
```

Во второй форме команды #implib задаются и имя библиотеки импорта и файла описания модуля (.exp - смотрите ниже Файл описания модуля), и на основании символических имен в файле описания модуля создается библиотека. Эта форма команды эквивалентна использованию утилиты tsimplib, поставляемой вместе с TopSpeed C, C++, and Modula-2.

#### **#implib <expfilename> <libfilename>**

Использование команды #implib требует от вас ручного создания и поддержания списка экспорта, тогда как первая форма экспортирует все общие имена автоматически. Если вам нужно обеспечивать совместимость с предыдущими версиями программного интерфейса, то использование “файла описания модуля” является несомненным преимуществом - он также позволяет вам экспортировать только необходимые функции.

Когда команда #implib используется совместно с файлом описания модуля, то список компоновки очищается.

## ***Условная обработка и управление последовательностью действий***

Можно выполнять команды проектного файла в зависимости от неких условий, используя команды #if, #then, #elsif, #else и #endif. Кроме того, при возникновении определенных условий, с помощью команд #error и #abort можно остановить обработку.

### **#if**

Синтаксис команды #if следующий:

```
#if <логическое выражение> #then  
    команды  
#elsif < логическое выражение > #then
```

```
команды  
#else  
команды  
#endif
```

Ветвь `#elseif` можно опускать или повторять много раз. Ветвь `#else` можно опустить.

Выражения вычисляются по порядку, пока одно из них не окажется истинным, затем выполняется идущая за ним последовательность команд. Если ни одно из выражений не окажется истинным и имеется ветвь `#else`, то выполняются идущие в ней команды. Все другие команды игнорируются.

Синтаксис и семантика логических выражений описываются далее в разделе Логические выражения.

### **#error**

#### **#error <строка>**

Эта команда заканчивает работу с текущим проектом. В Среде Разработки Clarion для Windows открывается окно текстового редактора, в котором проектный файл позиционирован на команду `#error` и высвечивается соответствующее сообщение.

Например:

```
#if "%name"="" #then  
#error "name not set up"  
#endif
```

### **#abort**

#### **#abort [ on | off ]**

Эта команда используется для того, чтобы определить будет ли прерываться обработка проекта при неудачном завершении команд `#compile` или `#run`. Если режим прерывания включен, то как только будет обнаружена неудачная компиляция или команда возвратит отличный от нуля код, обработка проекта будет прекращена. Если же режим прерывания выключен, то обработка проекта прекращается только если неуспешно выполнится какая-либо внутренняя команда, включая `#link`, `#implib` or `#exemod`.

Команда `"#abort on"` включает режим прерывания обработки проекта, а `"#abort off"` - выключает его. Команда `#abort` без аргументов вызовет немедленное прерывание обработки текущего проекта.

По умолчанию в Среде Разработки Clarion режим прерывания обработки проекта включен.

## Интерфейс пользователя

---

Следующие команды позволяют в процессе создания проекта получить информацию и обеспечить обратную связь с пользователем.

### **#message**

#### **#message <строка>**

Эта команда выводит указанную в ней строку в окне процесса создания. Тем самым можно отображать стадии выполнения проектного файла или выводить сообщения о состоянии выполнения этапов проекта.

Например:

```
#message "finished making %prjname"
```

### **#prompt**

#### **#prompt <promptstring> [ <defaultstring> ]**

Эта команда, высвечивая “строку приглашения”, предлагает пользователю ввести строку и ожидает ввода с клавиатуры. А введенная пользователем строка возвращается в виде значения макроса %reply. Если указана <defaultstring> (строка по умолчанию), и с клавиатуры ничего не введено, то эта строка и возвращается макросом %reply.

Например:

```
#prompt "Command line: " %cline  
#set cline = %reply
```

## Логические выражения

---

Используемые в командах #if и #elsif логические выражения строятся следующих логических операторов (перечисляемых в порядке старшинства):

```
#or  
#and  
#not  
=  
#exists  
#older  
( )
```

### **#or**

```
boolean-expression = <factor> { #or <factor> }
```

Логическое выражение, содержащее один или более операторов #or дает значение “истина”, если истинно значение любого из составляющих выражений.

### **#and**

```
<factor> = <term> { #and <term> }
```

Выражение, содержащие один или более операторов `#and`, дает в результате “ложь”, если любое из составляющих выражений дает в результате “ложь”.

### **#not**

`<term> = #not <term>`

Выражение, которому предшествует оператор `#not` дает в результате значение “истина”, если без этого оператора оно дает значение “ложь” и наоборот.

### **= (сравнение)**

`<term> = string = string`

Выражение, в котором имеется оператор сравнения дает в результате значение “истина”, если строки идентичны, в противном случае результат - “ложь”. Вместо символа `=` можно использовать удвоенный этот символ (`==`)

Оператор `=` и вторая строка могут быть опущены, в этом случае первая строка сравнивается со строкой `“on”`. Таким образом:

DemoSwitch =

равнозначно

DemoSwitch = “on”

Первую строку можно заменить выражением вида `“name1(name2)”`, в котором `name2` указывает прагму (`#pragma`) класса `name1`. В этом случае выражение перед сравнением заменяется текущей строкой, заданной в `#pragma`

### **#exists**

`<term> = #exists <имя-файла>`

Выражение, содержащее оператор `#exists`, дает в результате истину, если файл существует (после выполнения подстановок из `redirection-файла`). В противном случае выражение ложно.

### **#older**

`<term> = <file-name> #older <file-name> { , <file-name> }`

Выражение, содержащее оператор `#older`, дает в результате истину, если первый указанный файл имеет более раннюю дату, чем хотя бы один из остальных файлов, в противном случае результат “ложь”. При поиске файлов учитываются указания в `Redirection-файле` (см. раздел `Redirection-файлы` далее в этом руководстве). Этот оператор часто используется для определения того, нужно ли выполнять какие-либо пост- или препроцессорные действия. Например:

`#if mydll.lib #older mydll.exp #then`

...

### **() логические выражения в скобках**

`<term> = ( <boolean-expression> )`

Чтобы изменить порядок выполнения логических операций, выражение может содержать

логические выражения в скобках. Выражение истинно, если истинно выражение в скобках. С помощью скобок можно построить логическое выражение произвольной сложности.

## Redirection - файл

Среда Разработки Clarion устанавливает в качестве рабочего (текущего) каталога один из каталогов, в которых находится файл .APP или .PRJ. Дополнительно для указания каталогов, содержащих отдельные компоненты среды и проекта, Clarion для Windows использует Redirection - файл (файл переназначений - CW15.RED). При поиске файлов после текущего каталога, установленного при загрузке файла .PP или .PRJ, используются каталоги, указанные в CW15.RED, а затем в переменной окружения DOS PATH. Файл переназначений сообщает Среде где искать файлы и где создавать новые. Каждая строка в нем имеет следующий формат:

шаблон\_файла = каталог1 [;каталог2]... [;каталогn]

Шаблон\_файла представляет собой имя файла или шаблон имени, в котором используются стандартные для DOS символы шаблона \* и ?.

Например:

\*.dbd = c:\cw15\obj

\*.dll = ;c:\cw15\bin

\*.lib = c:\cw15\obj;c:\cw15\lib

\*.res = c:\cw15\obj;c:\cw15\lib

\*.obj = c:\cw15\obj;c:\cw15\lib

\*.rsc = c:\cw15\obj

\*.ico = ;c:\cw15\template

\*.bmp = ;c:\cw15\template

\*.tpl = c:\cw15\template

\*.tpw = c:\cw15\template

\*.trf = c:\cw15\template

\*.\* = ; c:\cw15\examples; c:\cw15\libsrc

QCKSTART.TXA = c:\cw15\TEMPLATE

QCKSTART.TXD = c:\cw15\TEMPLATE

Первый параметр каталог это каталог, в котором создается новый файл такого типа, какой задан в шаблоне\_файла. Это справедливо только для файлов, создаваемых и сохраняемых Средой, таких как .OBJ, .DBD, .LIB, .EXE или .CLW. Остальные параметры каталог задают пути к каталогам, в которых Clarion будет искать существующие файлы. При этом точка (.) указывает рабочий каталог, то есть один из тех, в которых находится файл .APP или .PRJ текущего проекта.

**Замечание:** Фалы - резервные копии всегда создаются в том каталоге, где находятся исходные файлы.

Для редактирования файла переназначений выберите в меню Setup д Edit Redirection File. Шаблон файла выводится слева, каталоги и пути справа.

## **Команды управления файлами**

---

Все эти команды предваряются ключевым словом #file реализуют интерфейс с файловой системой DOS/Windows через файл переназначений TopSpeed.

### **Имена файлов и анализ Redirection-файла**

Имена файлов могут быть заданы полностью т.е. C:\CW15\ORDERS\ORDER.TPS и в этом случае анализ Redirection-файла не производится. В других случаях имена файлов могут задаваться не полностью, как например, ORDERS.TPS, и тогда выполняется анализ Redirection-файла.

Анализ Redirection-файла означает, что система ведения проекта сопоставляет имя файла с шаблонами имен в текущем Redirection-файле до тех пор, пока не найдет совпадения. Затем система ведения проекта осуществляет поиск файла только в тех каталогах, которые указаны для совпавшего шаблона. Смотри раздел выше Файл переназначений.

При создании нового файла система ведения проекта в первом же каталоге из указанных для совпавшего шаблона.

### **Команды #file**

Имеются следующие команды для работы с файлами:

- #file adderrors
- #file append
- #file copy
- #file delete
- #file move
- #file redirect
- #file touch

### **#file copy <исх-имяфайла> <назн.-имяфайла >**

Эта команда копирует файл из < исх-имяфайла > в <назн.-имяфайла >. Оба эти параметра должны быть именами файлов без шаблонных символов (\* и ?). К обоим файлам применяются принципы поиска по Redirection-файлу.

### **#file delete < имяфайла >**

Эта команда удаляет указанный в ней файл. Имя файла должно быть без шаблонных символов (\* и ?). К файлу применяются принципы поиска по Redirection-файлу.

**#file move <исх-имяфайла> <назн.-имяфайла >**

Эта команда перемещает (переименовывает) файл, указанный параметром <исх-имяфайла> в файл, заданный параметром <назн.-имяфайла >. Оба имени должны указывать на один и тот же диск. К файлу применяются принципы поиска по Redirection-файлу.

**#file touch <имяфайла >**

Эта команда устанавливает дату и время файла, заданного параметром <имяфайла> на текущую дату и время.

**#file append <имяфайла > <строка>**

Эта команда добавляет заданную <строку> с добавленными символами возврат каретки/перевод строки (CR/LF) к указанному параметром <имяфайла> файлу. Если файл не существует, то он будет создан. Эту команду можно использовать для ведения файла-протокола и т.п.

**#file redirect [ <имяфайла > ]**

Эта команда изменяет текущий файл переназначений на указанный параметром <имяфайла >. Если не задан никакой файл, то команда изменяет текущий файл на файл, с которого начинался проект.

По концу project-файла восстанавливается файл переназначений на тот, с которого начинался проект.

**#file adderrors <имяфайла >**

Эта команда обрабатывает сообщения об ошибках в указанном файле и добавляет их к сообщениям об ошибках, которые выводятся по концу обработки проекта.

Каждое сообщение должно быть в одном из следующих форматов:

(имя\_файла номер\_строки,номер\_колонки): строка сообщения

(имя\_файла номер\_строки): строка сообщения

имя\_файла(номер\_строки): строка сообщения

При сборе ошибок от программ с различающимися форматами сообщений для приведения их к единому виду можно использовать фильтр.

Например:

```
#run 'masm %f; > %f.err'
```

```
#file adderrors %f.err
```

```
#run 'myprog %f; | myfilter > %f.err'
```

```
#file adderrors %f.err
```



Если обнаружены ошибки, и включен режим прерывания процесса, то обработка проекта заканчивается, и ошибки будут выведены в окне состояния процесса создания проекта.

Значения макросов %errors и %warnings соответствуют числу обнаруженных ошибок и выданных предупреждений.

## Прочие команды

### #run <командная\_строка>

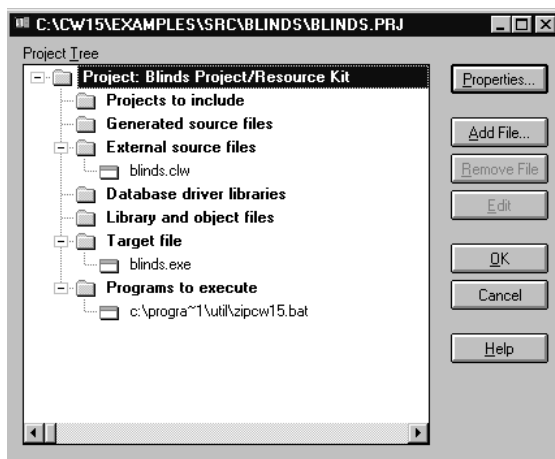
Этой командой выполняется заданная ее параметром <командная\_строка>. Команда #run генерируется для проекта, когда вы добавляете файл в папку Programs to execute (программы, подлежащие выполнению) в диалоговом окне Project Tree.

Пример:

```
#run "dir > dir.log"
```

```
#run "myprog"
```

**Замечание:** Имена файлов в командной строке (за исключением имени самого исполняемого файла) не попадают под действия правил системы переназначений - если нужно чтобы при этом также использовался файл переназначений, то перед #run нужно выполнить команду #expand



### #include <имя-файла>

Копия содержимого указанного файла включается во входной поток. Параметр <имя-файла> должен либо задавать полностью квалифицированное имя, либо не полностью указанное имя, тогда к нему будут применяться правила поиска задаваемые в файле переназначений (см. выше раздел Файл переназначений).

В команде #include полностью доступны список компоновки, значения прагм и макросов. Другими словами, включаемые предложения обрабатываются так, как если бы они

находились внутри включаемого файла проекта (.prj).

### **#call <имя-файла>**

Копия содержимого указанного файла включается во входной поток. Параметр <имя-файла> должен либо задавать полностью квалифицированное имя, либо не полностью указанное имя, тогда к нему будут применяться правила поиска задаваемые в файле переназначений (см. выше раздел Файл переназначений).

В команде #call не доступны список компоновки, значения прагм и макросов и, поэтому, в операторах вызываемых по #call нельзя изменять их значения. Другими словами операторы, к которым обратились с помощью #call обрабатываются как процесс, полностью отдельный от вызвавшего процесса.

### **#declare\_compiler <расширение> = <исполняемый\_макрос>**

Этой командой определяется макрос, который вызывается когда компилируется исходный файл с расширением, совпадающим с указанным в первом параметре. Макросы %src и %obj содержат имена исходного и объектного файлов.

Обычно не нужно использовать эту команду явно, поскольку все компиляторы TopSpeed предварительно объявляются в системе ведения проекта. Например, таким образом вызывается MASM:

```
#declare_compiler asm=  
'#set make=%%remake  
#if %%make #then  
#edit save %%src  
#expand %%src  
#set _masmsrc=%%opath  
#expand %%obj  
#set _masmobj=%%cpath  
#run "masm %%_masmsrc,%%_masmobj/MX/e; >masmtmp.$$$"  
#file adderrors masmtmp.$$$  
#file delete masmtmp.$$$  
#endif  
#pragma link(%%obj)'
```

### **#rundll <имя\_dll> <имя\_исх.файла> <имя\_вых.файла>**

Этой командой вызывается интегрированная утилита/компилятор TopSpeed. Первый параметр это имя DLL библиотеки, второй - имя исходного файла, третий - имя выходного файла.

Никогда не следует использовать эту команду явно, непосредственно, поскольку все

компиляторы TopSpeed предварительно объявляются в системе ведения проекта.

### **#exemod**

**#exemod <имя-файла> <имя-файла> <имя-файла>**

Эта команда равнозначна использованию утилиты “tsexemod”, которая поставляется с компиляторами TopSpeed C, C++ и Modula-2. Команда #exemod требуется для создания программ с оверлеями, программ для Windows и DLL библиотек для DOS. Однако при создании программ для Windows нет необходимости в явном использовании этой команды.

TSEXEMOD используется для того, чтобы, используя данные из файла описания модуля (.EXP), модифицировать заголовок и информацию о сегментах в исполняемом файле нового формата (.EXE or .DLL).

Например:

```
TSEXEMOD binfile.* expfile.exp mapfile.map
```

## ***Прагмы TopSpeed***

Во всех языках TopSpeed и системе ведения проекта TopSpeed используется один и тот же общий набор опций компилятора, называемых “прагмами”. Вообще прагмы могут указываться и исходном тексте и в файле проекта и действие их будет одинаковым. Не могут указываться прагмы только в исходном тексте на языке Clarion.

Прагмы можно использовать или в языке описания проекта или тексте на C++ и Modula-2. Некоторые работают только при указании в определенном месте. Символ ‘Р’ справа от прагмы, означает что ее можно использовать в языке описания проекта, ‘С’ указывает, что прагму можно задавать в тексте на C++, а ‘М’ означает использование в тексте на языке Modula-2.

## **Синтаксис прагм**

### **Синтаксис прагм Modula-2**

Прагмы в TopSpeed Modula-2 задаются в специальной форме комментария, которая начинается с символов ‘(\*#’.

Например:

```
(*# check( index => off ) *)
```

### **Директивы компилятора в старом стиле**

В первых версиях TopSpeed Modula-2 при задании режимов компилятора директивы

начинались с символа \$. Такие директивы по прежнему воспринимаются компиляторами поздних версий, за исключением:

- ◆ \$B (обработчик Ctrl-Break). Эта директива больше не поддерживается. Используйте вместо нее Lib.EnableBreakCheck

- ◆ \$D (имя сегмента данных). Эта директива поддерживается, но к именам вместо префикса D\_ добавляется суффикс \_BSS (для неинициализированных данных) или \_DATA (для инициализированных данных).

- ◆ \$J (вместо команды RET использовать IRET). Эта директива больше не поддерживается. Используйте вместо нее прагму:

```
(*# call( interrupt => on ) *)
```

Однако может оказаться, что вы должны сделать еще какие-то изменения, поскольку результат прагмы отличается от директивы \$J:

- ◆ \$K (соглашение о связях по типу C). Эта директива больше не поддерживается. Вместо нее следует использовать прагму:

```
(*# call( c_conv => on ) *)
```

- ◆ \$M (имя сегмента кода). Эта директива поддерживается, но к именам вместо префикса C\_ добавляется суффикс \_TEXT.

- ◆ \$P (внешнее имя для локальной процедуры). Эта директива больше не поддерживается.

- ◆ \$Q (трассировка процедур). Эта директива больше не поддерживается. Вместо нее следует использовать прагму:

```
(*# debug( proc_trace => on ) *)
```

Она предоставляет другой способ трассировки процедур. Подробнее смотрите прагму proc\_trace.

- ◆ \$X (80x87 сброс стека). Эта директива больше не поддерживается (в ней нет больше необходимости).

- ◆ \$Z (проверка нулевых указателей). Эта директива по прежнему включает проверку нулевых указателей, но не очищает больше память.

- ◆ \$@ (сохранить DS). Эта директива больше не поддерживается.

Поддержка этих директив включена в последние версии компиляторов, поэтому ваши старые модули и программы будут компилироваться с минимальными изменениями. Однако, в новых программах следует избегать использования устаревших директив и использовать вместо них прагмы.

### **Синтаксис прагм C и C++**

Прагмы представляют собой неотъемлемую часть языков C и C++ и реализуются как директивы компилятора:

```
#pragma check( index => off )
```

### **Синтаксис прагм Системы поддержки проекта**

В прагмах Системы поддержки проекта TopSpeed используется синтаксис похожий на

синтаксис директив C и C++.

```
#pragma check(index => off)
```

В Системе поддержки проекта с целью выполнения единой компиляции можно указывать прагмы в виде команд `#compile`.

```
#compile mandel.mod /debug(vid=>on)
```

## Классы прагм

---

Прагмы принимают вид `#pragma класс(имя=>значение)`. Есть следующие классы прагм:

**Call #pragmas**

**Check #pragmas**

**Data #pragmas**

**Debug #pragmas**

**Define #pragmas**

**Expr #pragmas**

**Link and Linkfirst #pragmas**

**Link\_option #pragmas**

**Module #pragmas**

**Name #pragmas**

**Optimize #pragmas**

**Option #pragmas**

**Project #pragmas**

**Save and Restore #pragmas**

**Warn #pragmas**

## Класс Call #pragmas

---

Прагмы, имеющие имя класса “call” воздействуют на все стороны соглашения о связях, сегментов кода и кодовых указателей. Текущие установки `call #pragmas` в тот момент, когда встретится определение процедуры или функции, определяют соглашения, которые будут использоваться при обращении к функции. Компиляторы TopSpeed распознают соответствующие ли соглашения используются при обращении к функции. Компоновщик выдает сообщение об ошибке, если соглашения о связях, провозглашенное для данной функции, не выполняется в каком-то из объектных файлов.

Имеются следующие прагмы класса `call #pragmas`:

```
#pragma call(c_conv => on | off)
```

```
#pragma call(ds_entry => identifier)
```

```
#pragma call(ds_eq_ss => on | off)
```

```
#pragma call(inline => on | off)
```

```
#pragma call(inline_max => Number)
```

```
#pragma call(near_call => on | off)
#pragma call(o_a_copy => on | off)
#pragma call(o_a_size => on | off)
#pragma call(opt_var_arg => on | off)
#pragma call(reg_param => RegList)
#pragma call(reg_return => RegList)
#pragma call(reg_saved => RegList)
#pragma call(result_optional => on | off)
#pragma call(same_ds => on | off)
#pragma call(seg_name => identifier)
#pragma call(set_jmp => on | off)
#pragma call(standard_conv => on | off)
#pragma call(var_arg => on | off)
```

**#pragma call(near\_call => on | off)****cpm**

Указывает является ли обращение к функции ближним или дальним. Когда установлено on, компилятор генерирует ближний вызов функции. Только когда вызывающая и вызываемая функции располагаются в одном и том же сегменте компилятор может ближние обращения. Компилятор проверяет соблюдается ли это условие.

По умолчанию устанавливается “off”. В примере включаются ближние обращения:

```
#pragma call(near_call=>on)
```

**#pragma call(same\_ds => on | off)****cpm**

Задаёт загружается ли регистр сегмента данных при входе в функцию. Если установлено “on”, то регистр DS не будет загружаться в прологе (служебной части кода, выполняющей инициализирующие действия) функции. Это будет правильно только тогда, когда значение в регистре DS вызывающей функции соответствует значению, требуемому для вызываемой функции. Компилятор проверяет соблюдается ли это условие.

По умолчанию устанавливается “off”. Например:

```
#pragma call(same_ds => on)
```

Отменяет загрузку регистра DS в прологе функции.

**#pragma call(c\_conv => on | off)****cpm**

Включает соглашение о вызовах по типу Microsoft C. По этим соглашениям передаваемые функции параметры помещаются в стек, начиная с самого правого и заканчивая левым, а удаляет эти параметры из стека вызывающая функция.

Этот механизм не используется по умолчанию, поэтому при взаимодействии с программными кодами, скомпилированными Microsoft C, следует использовать эту прагму. Пример:

```
#pragma call(c_conv=>on)
```

Для достижения точно такого же. Можно также использовать ключевое слово “cdecl” в C и C++.

Смотрите также прагму “standard\_conv”, которая имеет такое же действие для С и С++, но игнорируется компилятором Modula-2. По умолчанию прагма “standard\_conv” установлена в “off”.

### **#pragma call(inline => on | off)** **cm**

Если эта прагма установлена в “on” до определения функции, то компилятор вставляет в этом месте копию ее программного кода, а не команду перехода. По умолчанию установлено значение “off”.

Этот механизм можно использовать для любых функций, однако в основном эта прагма используется в сочетании с прагмой “reg\_param” для простых функций в машинном коде. Например:

```
#pragma save
#pragma call(inline => on, reg_param => (dx,ax))
static void outp(int port, unsigned char byt) =
{
    0xEE, /* out dx,al */
};
#pragma restore
```

делает функцию outp вставляемой в текст функцией, и поэтому обращение к ней представляет собой простую команду процессора 80x86: out dx,al.

### **#pragma call(seg\_name => identifier)** **cpm**

Задаёт имя кодового сегмента. call(seg\_name => pnn) означает, что компилятор поместит код функции в сегмент pnn\_TEXT. Значение по умолчанию зависит от модели памяти. В моделях small и compact по умолчанию устанавливается null. В других моделях значение равно имени исходного файла. Например, кодовый сегмент \_TEXT задается так:

```
#pragma call(seg_name => null)
а кодовый сегмент MYCODE_TEXT так:
#pragma call(seg_name => MYCODE)
```

Значение по умолчанию зависит от языка и не определяется Системой ведения проекта.

### **#pragma call(ds\_entry => identifier)** **cpm**

Эта прагма указывает имя сегмента, на который при выполнении функции будет указывать регистр DS. Если идентификатор пустой, то компилятор поименует сегмент \_DATA. Если идентификатор отсутствует, то подразумевается не фиксация значения регистра DS во время выполнения функции, а использование его как регистра общего назначения подобно регистру ES.

```
#pragma call(ds_entry => MYDATA)
```

В этом примере устанавливается, что при входе в функцию, регистр DS будет указывать на сегмент с именем MYDATA\_DGROUP.

### **#pragma call(reg\_param => RegList)** **cm**

В реализациях языков фирмы TopSpeed параметры в функции передаются в регистрах процессора, а не через стэк. Это дает возможность генерировать более компактный и быстрый код. Эта прагма дает возможность тонкой настройки обращений к отдельным функциям на максимальную производительность. Другие поставщики компиляторов используют менее эффективные соглашения о связях. Поэтому, при обращениях к откомпилированным объектам, написанным для других компиляторов, вы должны выключить соглашение TopSpeed (смотрите руководство TopSpeed Advanced Programmer's Guide, поставляемое с компиляторами TopSpeed C, C++, and Modula-2, глава 5: Multi-language Programming). Эта прагма не влияет на структурные параметры, которые всегда передаются через стэк.

Аргументом прагмы “reg\_param” является список регистров, задающий какие регистры следует использовать. Регистры для параметров распределяются слева направо по списку. В таблице показано, как компилятор распределяет параметры по регистрам в зависимости от типов:

1 байт	ax, bx, cx, dx
2 байта	ax, bx, cx, dx, si, di
4 байта	ax, bx, cx, dx, si, di для младшего слова. ax, bx, cx, dx, si, di, es, ds для старшего слова.
плавающая точка	st0, st1, st2, st3, st4, st5, st6

Отметим, что регистры es и ds будут использоваться только для длинных 4-х байтовых параметров, типа указателей. Если младшее или старшее слово нельзя поместить в регистр, то весь параметр передается через стэк.

Когда компилятор исчерпает список регистров, то параметры передаются через стэк. Если вы задали пустой список, то компилятор будет использовать стэк для всех параметров.

Установка по умолчанию для соглашения о связях по типу TopSpeed:

```
#pragma call(reg_param=>(ax,bx,cx,dx,st0,st6,st5,st4,st3))
```

Установка по умолчанию для передачи параметров через стэк:

```
#pragma call(reg_param => ( ))
```

### **#pragma call(reg\_saved => RegList)**

**cm**

Эта прагма задает, какие регистры функцией сохраняются. Аргумент RegList представляет собой список, который задает набор регистров.

Установка по умолчанию для соглашения о связях по типу TopSpeed:

```
ccall(reg_saved=>(ax,bx,cx,dx,si,di,ds,st1,st2))
```

Установка по умолчанию для передачи параметров через стэк:

```
#pragma call(reg_saved=>(si,di,ds,st1,st2))
```

### **#pragma call(o\_a\_size => on | off)**

**m**

Когда эта прагма включена, то через стэк передается размер массива с неопределенной длиной:

```
(*# call( o_a_size => on ) *)
```



Если прагма `o_a_coru` не установлена в `off`, то прагма `o_a_size` не влияет на параметр значение массива.

По умолчанию устанавливается в `on`.

**#pragma call(o\_a\_copy => on | off)** **m**

При установке значения этой прагмы в `on`, параметры - массивы с неопределенной длиной в прологе функции копируются в стек. Если в `off`, то передаются только ссылка на массив. Заметим, что для того, чтобы параметры - массивы с неопределенной длиной можно было скопировать, может передаваться размер массива, см. прагму `call(o_a_size)`. По умолчанию, прагма `call(o_a_coru => on)`.

**#pragma call(ds\_eq\_ss => on | off)** **m**

Эта прагма определяет, используют ли параметры типа VAR 16-ти битные указатели или 32-х битные. По умолчанию устанавливается в “`on`” для моделей памяти `small` и `medium`, а для остальных в “`off`”.

**#pragma call(var\_arg => on | off)** **m**

Когда эта прагма установлена в “`on`”, подразумевается, что следующие процедуры принимают переменное число аргументов. Это эффективный способ избавиться от ошибки “`too many arguments`” (слишком много аргументов), которую в обычных условиях выдал бы компилятор. Однако следствием этой установки прагмы является то, что компилятор не может выполнить проверку типов аргументов.

Эту прагму следует использовать при вызове функций языка C (таких как `printf`), у которых переменное число аргументов:

```
(*# call(var_arg => on,
                    reg_param=>()),
   c_conv=>on ) *)
```

По умолчанию устанавливается в `off`.

**#pragma call(reg\_return => RegList)** **cm**

Эта прагма используется для того, чтобы указать регистр, который должен использоваться для возвращаемого значения типа целое, указатель или с плавающей точкой. Пример:

```
#pragma call(reg_return => (bx,cx))
```

По умолчанию устанавливается:

```
#pragma call(reg_return=>(ax,dx,st0))
```

**#pragma call(result\_optional => on | off)** **m**

Эту прагму можно использовать при обращении к процедуре как к правильной процедуре, не генерируя сообщений об ошибках. Например:

```
(*# save *)
```

```
(*# module( result_optional => on ) *)
PROCEDURE FuncProc(x: CHAR): CARDINAL;
(*# restore *)
```

После такого объявления допустимо употреблять любую из этих двух форм:

```
i := FuncProc('a');
FuncProc('a');
```

Это полезно, когда вызываемая процедура имеет побочный эффект, который более важен, чем возвращаемое значение. В частности это полезно при обращении к библиотечным процедурам TopSpeed C.

По умолчанию установлено в off.

**#pragma call(set\_jump => on | off)** **cm**

Эту прагму следует использовать только для библиотечных подпрограмм, которые реализуют дальние переходы. Эффект в том, что компилятор узнает о нестандартном сохранении регистров этими подпрограммами.

**#pragma call(inline\_max => Number)** **cpm**

Эта прагма управляет очень большими встроенными (inline) функциями. Устанавливаемое по умолчанию значение равно 12, что соответствует минимальному размеру кода в большинстве программ. Увеличение значения увеличивает размер программного кода и может ускорить его выполнение.

Эта прагма влияет на каждый вызов функции, так что к функции можно обращаться в разных местах по-разному.

**Замечание.** Функция не встраивается, если до обращения к ней она еще не была скомпилирована.

**#pragma call(standard\_conv => on | off)** **c**

Эта прагма влияет на программы на C и C++ точно также как прагма “call(c\_conv)”. Для Modula-2 она значения не имеет. По умолчанию значение “off”.

**#pragma call(opt\_var\_arg => on | off)** **cp**

Эта прагма определяет, генерируется ли для процедур с переменным числом параметров оптимизированная входная последовательность. По умолчанию включено.

## **Прагмы Data (Data #pragmas)**

Прагмы, для которых указано имя класса “data”, влияют на сегментацию данных, на указатели на данные и все аспекты расположения данных. Текущие установки прагм “data” в момент объявления переменных повлияют на способ обращения к этим переменным.

Есть следующие прагмы “data”

```
#pragma data(c_far_ext => on | off)
#pragma data(class_hierarchy => on | off)
```

```
#pragma data(compatible_class => on | off)
#pragma data(const_assign => on | off )
#pragma data(const_in_code => on | off)
#pragma data(cpp_compatible_class => on | off )
#pragma data(ext_record => on | off )
#pragma data(far_ext => on | off )
#pragma data(near_ptr => on | off)
#pragma data(packed => on | off )
#pragma data(seg_name => identifier)
#pragma data(stack_size => Number)
#pragma data(threshold => Number)
#pragma data(var_enum_size => on | off)
#pragma data(volatile => on | off)
#pragma data(volatile_variant => on | off )
```

### **#pragma data(seg\_name => identifier)                      cpm**

Эта прагма #pragma data(seg\_name=>xxx) указывает, что компилятор должен размещать глобальные инициализированные объекты в сегменте с именем xxx\_DATA, а неинициализированные глобальные объекты сегменте с именем xxx\_BSS. Оба эти сегмента имеют групповое имя xxx и относятся к классу FAR\_DATA. Если размер объекта данных превышает предельный для глобальных данных, то компилятор помещает этот объект в отдельный сегмент.

Следующий пример устанавливает по умолчанию имена сегментов: MYDATA\_DATA и MYDATA\_BSS. Эти сегменты входят в группу MYDATA и относятся к классу FAR\_DATA:

```
#pragma data(seg_name => MYDATA)
```

Можно задать пустое значение, это означает имена \_BSS и \_DATA. Для всех моделей памяти за исключением extra large и multi-thread используемое по умолчанию значение - пустое.

Пример:

```
#pragma data(seg_name => null)
```

### **#pragma data(far\_ext => on | off )                      cp**

Если эта прагма установлено в “on”, генератор кода предполагает, что внешние переменные находятся не в сегменте, указанном прагмой. По умолчанию эта прагма установлена в “on”. В примере:

```
#pragma(seg_name=>MYDATA, far_ext=>off)
```

устанавливается имя сегментов MYDATA\_DATA и MYDATA\_BSS в группе MYDATA. Компилятор предполагает, что внешние данные должны быть в том же самом сегменте.

### **#pragma data(c\_far\_ext => on | off)                      pm**

Если эта прагма установлено в “on”, генератор кода предполагает, что внешние

переменные находятся не в сегменте, указанном прагмой `seg_name`. Для всех моделей памяти по умолчанию эта прагма установлена в “off”. В примере:

```
(*# data(seg_name => MYDATA, c_far_ext => off) *)
```

устанавливается по имя сегмента по умолчанию `MYDATA_DATA` и `MYDATA_BSS` в группе `MYDATA`. Обычно эта прагма в Modula-2 не нужна, за исключением случаев построения программного интерфейса с C.

**#pragma data(near\_ptr => on | off)** **cm**

Задаёт, являются ли указатели на данные ближними или данными. Кроме того, эта прагма влияет на указатели, генерируемые посредством оператора `&` и посредством неявного преобразования массива в указатели в C и C++. Например:

```
#pragma data(near_ptr => on)
```

Указатели на данные делаются ближними.

**#pragma data(volatile => on | off)** **m**

Переменные, объявленные, когда эта прагма установлено в “on”, рассматриваются как изменчивые и всегда должны храниться в памяти, а не в регистрах.

Значение по умолчанию “off”. Эта прагма не употребляется в project-файле и неприменима в C и C++, где следует использовать ключевое слово “volatile”.

**#pragma data(volatile\_variant => on | off)** **m**

Действие этой прагмы такое же как `data(volatile)`, но применяется она только к переменным из вариантных записей. Значение по умолчанию “off”.

**#pragma data(ext\_record => on | off)** **pm**

Обычно в TopSpeed не допускается, чтобы два поля из различных альтернатив вариантной записи имели одинаковое имя. Использование этой прагмы:

```
(*# data( ext_record => on) *)
```

разрешил использовать одно и то же имя в различных альтернативах, если поля располагаются с одинаковым смещением в вариантной записи и имеют одинаковый тип данных.

По умолчанию устанавливается “off”.

**#pragma data(var\_enum\_size => on | off)** **pm**

Перечисляемые константы, имеющие менее чем 256 альтернативных значений, обычно хранятся в одном байте. Переключение этого режима в “off”:

```
(*# data( var_enum_size => off) *)
```

приведет к тому, что компилятор будет сохранять их как двухбайтовые величины. Это особенно полезно для организации взаимодействия со сторонними библиотеками и для обращений к операционной системе, в которых предполагается передача данных в формате машинного слова. Без этой прагмы, перечислимые типы занимали бы байт, а не слово.

Значение по умолчанию “on”.

#pragma data(stack size => Number) cm

Задаёт размер стека. Нужно помещать эту прагму в файл, содержащий основную функцию (главный модуль в Modula-2). Если нужный размер стека не установлен, компилятор будет использовать максимально возможный стек. По умолчанию размер стека равен 16К. Пример:

```
#pragma data(stack_size => 0x6000)
main()
{
    /* операторы */
}
```

определяет стэк во время выполнения программы размером в 0x6000 байт (24K).

```
#pragma data(packed => on | off )
```

Эта прагма управляет тем, упаковываются ли поля записи на уровне битов. По умолчанию устанавливается “off”.

**#pragma data(const in code => on | off)**      p

Эта прагма управляет тем, помещаются ли константы в кодовый сегмент или сегмент данных. По умолчанию устанавливается “on”.

```
#pragma data(class hierarchy => on | off )
```

Эта прагма управляет тем, включается ли информация об иерархии классов в дескриптор класса (таблицу методов). Эта информация используется оператором IS и TypeGuards. По умолчанию устанавливается “on”.

```
#pragma data(cpp compatible class => on | off ) pm
```

Эта прагма управляет тем, включается ли для совместимости с C дополнительная информация в дескриптор класса. По умолчанию устанавливается “on”.

```
#pragma data(compatible class => on | off)      cp
```

Эта прагма управляет тем, включается ли для совместимости с Modula-2 соответствующая информация в дескриптор класса. По умолчанию устанавливается "off".

```
#pragma data(threshold => Number)                                cpm
```

Эта прагма устанавливает предел глобальных данных. Он определяет при каком размере объекта данных он помещается в свой собственный сегмент. По умолчанию устанавливается 16384 байта.

```
#pragma data(const assign => on | off )      pm
```

Эта прагма управляет тем, возможно ли присвоение структурированным константам. По умолчанию устанавливается “off”.

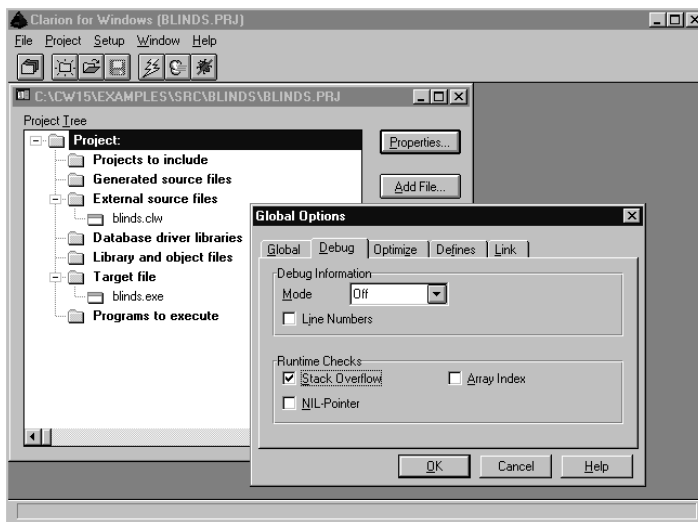
**Примечание:** Если указана прагма `const in code=>on`, то присвоение

константе приведет нарушению защиты кода.

## Прагмы класса Check (Check #pragmas)

Прагмы с именем класса “check” управляют обнаружением ошибок во время выполнения. Это может помочь обнаружить ошибки в логике работы программы, но ценой снижения производительности. По умолчанию все эти прагмы устанавливаются в “off”.

Когда процедуры контроля во время выполнения обнаруживают ошибку, то по умолчанию предпринимаемым действием является прекращение процесса и создание файла CWLOG.TXT.



Есть следующие прагмы, управляющие контролем ошибок времени выполнения:

```
#pragma check(guard => on | off)
#pragma check(index => on | off)
#pragma check(nil_ptr => on | off)
#pragma check(overflow => on | off)
#pragma check(range => on | off)
#pragma check(stack => on | off)
```

#pragma check(stack => on | off)

cpm

Когда эта прагма включена, то исполняющая система проверяет, чтобы программа не выходила за пределы стека. Используя прагму data(stack\_size) можно изменить размеры стека.

#pragma check(nil\_ptr => on | off)

cpm

Когда эта прагма включена, то исполняющая система проверяет инициализацию

указателей (NULL или NIL указатели).

**#pragma check(range => on | off)** **pm**

Когда эта прагма включена, то исполняющая система при присвоении значений переменным проверяет, чтобы присваиваемое значение соответствовало поддиапазону перечислимого типа. Дополнительно, значения, присваиваемые во время компиляции, проверяется на предмет соответствия присущему данному типу данных диапазону.

**#pragma check(overflow => on | off)** **pm**

Когда эта прагма включена, то исполняющая система проверяет, чтобы числовое значение не выходило за границы допустимого диапазона.

**#pragma check(index => on | off)** **cpm**

Когда эта прагма включена, то исполняющая система проверяет, чтобы индекс при обращении к массиву не превышал максимального значения.

**#pragma check(guard => on | off)** **pm**

Эта прагма управляет тем, выполняются ли проверки по некоторым контрольно-защитным (checked-guard ) операторам.

## Прагмы класса Name

Прагмы с именем класса “name” управляют отдельными сторонами образования имен, используемых при компоновке. Однако, программист на С должен также владеть принципами механизма образования имен и объявления внешних имен в С.

Есть следующие прагмы класса “name”:

**#pragma name(prefix => (none | modula | c | os2\_lib | windows))**

**#pragma name(prefix => string)**

**#pragma name(upper\_case => on | off)**

**#pragma name(upper\_case => on | off)** **cp**

Эта прагма доступна только для С и С++. Она задает должны ли общие (public) имена преобразовываться к верхнему регистру букв. Это приходится делать при реализации программного интерфейса с модулем на Паскале или библиотеками на С сторонних фирм. ПО умолчанию устанавливается в “off”.

**#pragma name(prefix)** **cpm**

У этой прагмы есть две формы:

В Modula-2 name(prefix => (none | modula | c | os2\_lib | windows))

В С и С++ name(prefix => строка)

#pragma name(prefix => (none | modula | c | os2\_lib | windows)) mp

Эта прагма относится к Modula-2 (для C и C++ синтаксис немного другой - см. прагму `name(prefix => строка)`).

Прагма `name(prefix)` задает используемый компилятором префикс и регистр общих (public) имен. Общие имена - это имена нестатических процедур и внешних объектов данных. По умолчанию в TopSpeed Modula-2 все внешние имена предваряются именем модуля, за которым идет символ '@' для данных и символ '\$' для процедур. При организации программного интерфейса с TopSpeed C нужно будет использовать эту прагму.

Прагма <code>name(prefix =&gt;</code> задает, какую префиксную схему использовать:	
<code>modula</code>	использовать принятые в TopSpeed Modula-2 префиксные соглашения для всех внешних имен т.е. имя модуля и символ '@' или '\$';
<code>none</code>	не предварять внешние имена префиксом;
<code>c</code>	использовать принятые в C префиксные соглашения для всех внешних имен (ко всем внешним именам добавлять знак подчеркивания '_');
<code>os2_lib</code>	использовать принятый для библиотек OS/2 стандарт (имя модуля в качестве префикса добавляется ко всем внешним именам).
<code>windows</code>	использовать принятые в Microsoft Windows префиксные соглашения.

### **#pragma name(prefix => строка)**

**ср**

Эта прагма относится только к C и C++. В Modula-2 синтаксис слегка отличается - см. прагму `name(prefix => (none | modula | c | os2_lib | windows))`.

Значение представляет собой строку, задающую префикс для всех внешних имен. Пустая строка означает отсутствие префикса. По умолчанию префиксом служит знак подчеркивания.

Если вы хотите реализовать программный интерфейс с TopSpeed Modula-2, то для того, чтобы указать префикс с символом доллара в конце, можно использовать эту прагму. Например, чтобы использовать процедуру `WrStr` из модуля `IO`:

### **#pragma name(prefix => "IO\$")**

```
void WrCard(unsigned);
```

В спецификациях по компоновке в C, Паскале или Modula2 можно внутри спецификации задать имя модуля и в этом случае отпадает необходимость в этой прагме.

Значение по умолчанию для этой строки зависит от языка. Система ведения проекта не устанавливает для этой прагмы значения по умолчанию.



## Прагмы класса Optimize

---

Прагмы с именем класса “optimize” управляют оптимизацией, которую выполняет генератор кода фирмы TopSpeed. По умолчанию включены все методы оптимизации. Выключение оптимизации ухудшает качество кода и вряд ли оказывает какое-либо существенное влияние на время компиляции.

Есть следующие прагмы класса “optimize”:

```
#pragma optimize(alias => on | off)
#pragma optimize(const => on | off)
#pragma optimize(cpu => 86 | 286 | 386 | 486)
#pragma optimize(cse => on | off)
#pragma optimize(jump => on | off)
#pragma optimize(loop => on | off)
#pragma optimize(peep_hole => on | off)
#pragma optimize(regass => on | off)
#pragma optimize(speed => on | off)
#pragma optimize(stk_frame => on | off)
```

#pragma optimize(cse => on | off) **cpm**

Когда эта прагма включена, компилятор минимизирует время вычисление всего выражения, сохраняя промежуточные результаты в регистрах. По умолчанию эта прагма установлена в “on”.

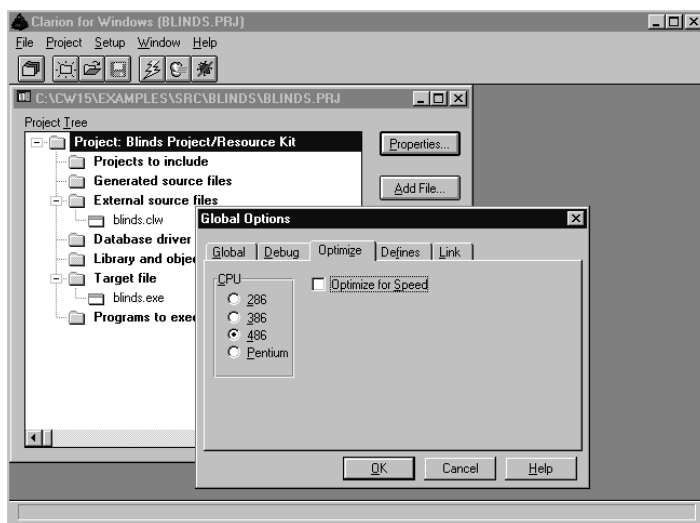
#pragma optimize(const => on | off) **cpm**

Когда эта прагма включена, компилятор с целью генерации более быстрого кода будет держать часто используемые константы в регистрах. По умолчанию эта прагма установлена в “on”.

#pragma optimize(speed => on | off) **cpm**

Когда эта прагма включена, то компилятор TopSpeed пытается сгенерировать максимально быстрый код, не взирая на его размеры. Если значение этой прагмы “off”, то компилятор пытается генерировать максимально компактный код.

Хорошим примером различия оптимизации “по скорости” и “по памяти” служит реализация цикла. При оптимизации по скорости компилятор может использовать команду “por” (нет операции) чтобы поместить метку перехода внутри цикла на четный адрес. В архитектуре процессоров 80x86 такие переходы осуществляются гораздо быстрее, чем на нечетный адрес, но каждое такое выравнивание командой “por” увеличивает на один байт размер кода. Это значит, что при оптимизации по памяти компилятор исключает такие излишние команды. По умолчанию эта прагма установлена в “on”.



### **#pragma optimize(stk\_frame => on | off)**

**cpm**

Когда эта прагма включена, то компилятор TopSpeed будет делать фрейм стека только, где это требуется, исключая тем самым необходимость в загрузке регистра BP. Такая оптимизация может быть выполнена только когда все параметры и локальные переменные функции можно разместить в регистрах процессора.

Когда эта прагма установлена в “off”, то компилятор всегда устанавливает значение регистра BP, разрешая тем самым полную активизацию стека, представляемую во время отладки. По умолчанию устанавливается значение “on”.

### **#pragma optimize(regass => on | off)**

**cpm**

Когда эта прагма включена, то компилятор тратит некоторое время для нахождения оптимального размещения переменных в регистрах. Это приводит к тому, что получается быстрый и компактный код, но увеличивается время компиляции. По умолчанию устанавливается значение “on”.

### **#pragma optimize(peep\_hole => on | off)**

**cpm**

Когда эта прагма включена, то компилятор выполняет различные преобразования машинного кода, генерируя меньший по размеру и более быстрый код. По умолчанию устанавливается значение “on”.

### **#pragma optimize(jump => on | off)**

**cpm**

Когда эта прагма включена, то компилятор будет перестраивать циклы, чтобы исключить насколько это возможно переходы. Это дает в результате более быстрый код. По умолчанию устанавливается значение “on”.

### **#pragma optimize(loop => on | off)**

**cpm**

Когда эта прагма включена, то компилятор использует всю глубину вложенных циклов при исключении общих подвыражений и оптимизации переходов. В результате этой оптимизации получается более быстрый, но, возможно больший по размеру код. По умолчанию устанавливается значение “on”.

**#pragma optimize(alias => on | off)**

**cpm**

Когда эта прагма включена, то это позволяет компилятору считать, что к переменным в этой функции не будет косвенных обращений с помощью указателей. Такое предположение строго недопустимо в ANSI C, однако справедливо во многих программах. По умолчанию устанавливается значение “on”.

**#pragma optimize(cpu => 86 | 286 | 386 | 486)**

**cpm**

Объявляя тип используемого процессора, эта прагма управляет набором команд, который использует генератор кода. По умолчанию процессор 286-й.

## **Прагмы класса Debug**

---

Прагмы с именем класса “ debug” управляют размером дополнительных данных, включаемых компилятором для облегчения процесса отладки программы.

Есть следующие отладочные прагмы:

**#pragma debug(line\_num => on | off)**

**#pragma debug(proc\_trace => on | off)**

**#pragma debug(public => on | off)**

**#pragma debug(vid => off | min | full)**

**#pragma debug(vid => off | min | full)**

**cpm**

Когда установлено значение full, компилятор помещает информацию для визуального интерактивного отладчика TopSpeed (Visual Interactive Debugger - VID) в файл .DBD. При отладке программы с помощью этого отладчика используйте этот режим.

**Замечание:** Эта прагма отменяет использование регистров и оптимизацию кадра стэка, обеспечивая всесторонний доступ к переменным в отладчике. Все локальные переменные трактуются как изменяемые, чтобы обеспечить их хранение в процессе выполнения оператора в памяти, а не в регистрах, таким образом обеспечивается доступ к ним из отладчика в любое время.

Если установлено min, компилятор выполняет описанную выше оптимизацию и не трактует локальные переменные как изменяемые. Отладчик по-прежнему можно использовать, но нельзя ссылаться на локальные переменные и к некоторым фреймам стэка.

Если установлено off, то компилятор не генерирует отладочной информации и, таким образом, ускоряя компиляцию, генерируя оптимальный код, и экономя дисковое

пространство.

По умолчанию устанавливается значение “off”.

**#pragma debug(proc\_trace => on | off)** **pm**

Когда эта прагма включена, компилятор генерирует команды обращения к процедурам EnterProc и ExitProc при входе в каждую процедуру и выходе соответственно. Эти процедуры затем могут выполнять любые запрошенные вами действия по трассировке.

**Замечание: Следует быть уверенным, что эта прагма установлена в “off” для самих процедур EnterProc и ExitProc, в противном случае возникает бесконечная рекурсия и программа несомненно “свалится”.**

Эти две процедуры должны быть видимы в модуле, для которого эта прагма установлена в on. Это означает, что в самом модуле должны определяться процедуры ExitProc и EnterProc или модуль должен импортировать их.

По умолчанию установлена в “off”.

**#pragma debug(line\_num => on | off)** **cpm**

Эта прагма приводит к тому, что для отладчиков типа “symdeb” генерируется информация о номерах строк программы. Она сохраняется в объектных файлах и печатается в тар-файле. По умолчанию установлена в “off”.

**#pragma debug(public => on | off)** **cpm**

Действие этой прагмы в том, что личные объекты становятся общими, облегчая использование таких отладчиков как “symdeb”. Это может приводить к появлению повторяющихся предупреждений во время компоновки с такими языками как С и С++, которые не имеют модульной структуры. Эти предупреждения можно спокойно проигнорировать, во избежание возможных конфликтов рекомендуется переименовать такие функции. По умолчанию установлена в “off”.

## **Прагмы класса Module**

---

Прагмы с именем класса “module” управляют режимами, которые применимы ко всему исходному модулю. Эти прагмы нужно указывать в начале исходного файла, к которому они относятся, или в проектном файле.

Есть следующие прагмы класса module:

**#pragma module(implementation => on | off )**

**#pragma module(init\_code => on | off )**

**#pragma module(init\_prio => Number)**

**#pragma module(smart\_link => on | off)**

**#pragma module(init\_code => on | off )** **pm**

Когда эта прагма установлена в “on”, это означает, что модуль содержит инициализационный код, который должен быть выполнен после загрузки программы, но перед началом ее выполнения. Выключение этого режима полезно для модулей, написанных на других языках, так как это может устранять предупреждения компоновщика о неопределенном символическом имени:

```
(*# module( init_code => off ) *)
```

**Замечание:** Если в `implementation`-модуле эта прагма устанавливается в “off”, то это имеет распространительный эффект, т.е. во всех импортируемых модулях прагма `init_code` должна быть выключена.

По умолчанию устанавливается в “on”.

**#pragma module(implementation => on | off )** **pm**

Эта прагма задает, имеет или нет файл определений (.DEF или .ITF) соответствующий ему объектный файл. Если файл определений определяет интерфейс с подпрограммами на других языках, то чтобы предупредить попытки создания проектной системой соответствующего объектного файла, эта прагма должна быть выключена. По умолчанию устанавливается в “on”.

Эту прагму также можно использовать в реализационной части модуля перед исходным текстом любого из модулей. В этом случае он переопределяет используемый по умолчанию способ образования имени соответствующего объектного файла. Обычно, имя файла .OBJ, соответствующего модулю берется от имени модуля. Если эта прагма установлена в “off”, то имя соответствующего объектного модуля производится от имени файла, а не от имени модуля.

**#pragma module(smart\_link => on | off)** **cpm**

Установка этой прагмы в “off” выключает возможность интеллектуальной компоновки, чтобы или все или никакие объекты в каждом сегменте, полученном в процессе компилятора, были включены в процесс компоновки. Это приводит к ускорению процесса компоновки, а также позволяет использовать другие компоновщики (как например, от фирмы Microsoft). Попытка использовать не “топспидовский” компоновщик несет в себе множество потенциальных проблем, и, определенно, не рекомендуется это делать. По умолчанию эта прагма устанавливается в “on”.

**#pragma module(init\_prio => Число)** **cp**

Эта прагма относится только к языку С. Она определяет приоритет для инициализирующего кода для статического объекта. Обычно, порядок инициализации неопределен, а эта прагма позволяет вам контролировать порядок инициализации, в котором файлы с большим приоритетом инициализируются прежде модулей с меньшим приоритетом. Число может находиться в диапазоне от 0 до 32. По умолчанию 16, для библиотеки С используется значение между 25 и 32 (поэтому не рекомендуется

использовать значения из этого диапазона, в противном случае часть библиотеки может быть неинициализирована до того, как начнется выполнение пользовательского кода).

## Прагмы класса Option

Прагмы с именем класса “Option” управляют режимами, зависимиыми от языка, такими как расширения TopSpeed.

Имеются следующие прагмы класса Option:

```
#pragma option(ansi => on | off)
#pragma option(bit_opr => on | off)
#pragma option(incl_cmt => on | off)
#pragma option(lang_ext => on | off)
#pragma option(min_line => on | off)
#pragma option(nest_cmt => on | off)
#pragma option(pre_proc => on | off)
#pragma option(uns_char => on | off)
```

**#pragma option(ansi => on | off)** **cp**

Эта прагма применима только в С и С++. Если ее значение установлено в “on”, то допустимы только ключевые слова стандарта ANSI. По умолчанию устанавливается “off”.

**#pragma option(lang\_ext => on | off)** **cp**

Эта прагма применима только в С и С++. Когда эта прагма не установлена в “on”, по стандарту ANSI С следующие конструкции неприменимы, но включаются в TopSpeed С и С++:

- если операнд представляет собой именуемое выражение, то приведение типов дает именуемое выражение;
- функции не могут инициализироваться с помощью двоичного машинного кода;
- в операциях отношения (>, >=, <=, <) допускается смешение целочисленных операндов и операндов-указателей;
- битовые поля в С могут иметь тип char и unsigned char.
- относительные указатели.

По умолчанию эта прагма устанавливается в “on”.

**#pragma option(nest\_cmt => on | off)** **cp**

Эта прагма применима только в С и С++. Когда она установлена в “on”, комментарии могут быть вложенными.

Например:

```
/* Это тестовый комментарий
/* Это вложенный комментарий */
*/
```

Когда она выключена, вложение комментариев приводит к выдаче сообщения об ошибке. По умолчанию устанавливается “off”, позволяя компилятору, легко отслеживать неоконченные комментарии и обеспечивая соответствие с ANSI C.

**#pragma option(uns\_char => on | off)** **cp**

Эта прагма применима только в С и С++. Когда она установлена в “on”, значения типов, объявленных как char лежат в диапазоне между 0 и 256. Если установлено “off”, то между -127 и 128. По умолчанию устанавливается “off”.

**#pragma option(pre\_proc => on | off)** **cp**

Эта прагма применима только в С и С++. Когда она установлена в “on”, компилятор осуществляет вывод результата работы препроцессора в файл с тем же именем, что и исходный, но с расширением .i. Это облегчает отладку макрорасширений. По умолчанию устанавливается в “off”.

**#pragma option(incl\_cmt => on | off)** **cp**

Эта прагма применима только в С и С++. Когда она установлена в “on”, в выведенных результатах работы препроцессора сохраняются комментарии. По умолчанию устанавливается в “off”.

Если прагма #pragma pre\_proc не установлена в “on”, то данная прагма не действует.

**#pragma option(min\_line => on | off)** **cp**

Эта прагма применима только в С и С++. Когда она установлена в “on”, препроцессор минимизирует пустых строк в выводимых результатах работы. По умолчанию устанавливается в “on”.

Если прагма #pragma pre\_proc не установлена в “on”, то данная прагма не действует.

**#pragma option(bit\_opr => on | off)** **pm**

Эта прагма применима только в Modula-2. Она разрешает побитовые операции над числительными:

( a AND/OR b, NOT a ).

По умолчанию устанавливается в “off”.

## **Warn #pragmas**

---

Прагмы класса “warn” управляют генерацией сообщений компилятора. Эти прагмы относятся только к С и С++.

Получаемые от компилятора TopSpeed С и С++ помогают вам (насколько это возможно) избегать наиболее распространенных ошибок кодирования. Поскольку нет компиляторов, способных угадывать ваши намерения, иногда вы можете получать от компилятора предупреждения, даже когда ваш код правилен. При компиляции вашей программы одни предупреждения могут выдаваться чаще чем другие, поэтому TopSpeed предоставляет вам

возможность настроить, какие проверки выполняются.

Каждую опцию предупреждения можно установить в “on”, “off” и “err”. Когда установлено в “on”, TopSpeed выполняет данную проверку кода и сообщает о наличии проблемы, но не останавливает процесс компиляции и компоновки. Когда установлено “off”, то выдача предупреждения не производится. Когда установлено “err”, то TopSpeed выполняет данную проверку кода, сообщает о наличии проблемы, и останавливает процесс компиляции и компоновки, пока вы не исправите ситуацию.

**Замечание:** Компилятор TopSpeed C и C++ проверяет код и выдает сообщение по основательной причине. В самом деле, для того, чтобы использовать ваш несоответствующий ANSI C код, используется по умолчанию минимальный набор предупреждений. Поэтому, следует крепко подумать прежде чем выключать какое-либо из выдаваемых по умолчанию предупреждений. Мы советуем, чтобы все эти предупреждения оставались в состоянии “on” или “err”.

Есть следующие прагмы класса “warn”

```
#pragma warn(wacc => on | off | err)
#pragma warn(wait => on | off | err)
#pragma warn(wall => on | off | err)
#pragma warn(watr => on | off | err)
#pragma warn(wcic => on | off | err)
#pragma warn(wcld => on | off | err)
#pragma warn(wclt => on | off | err)
#pragma warn(wcne => on | off | err)
#pragma warn(wcor => on | off | err)
#pragma warn(wert => on | off | err)
#pragma warn(wdel => on | off | err)
#pragma warn(wdne => on | off | err)
#pragma warn(wdnu => on | off | err)
#pragma warn(wetb => on | off | err)
#pragma warn(wfnd => on | off | err)
#pragma warn(wftn => on | off | err)
#pragma warn(wnid => on | off | err)
#pragma warn(wnre => on | off | err)
#pragma warn(wnrw => on | off | err)
#pragma warn(wntf => on | off | err)
#pragma warn(wovl => on | off | err)
#pragma warn(wovr => on | off | err)
#pragma warn(wpcv => on | off | err)
#pragma warn(wpic => on | off | err)
#pragma warn(wpin => on | off | err)
#pragma warn(wpnd => on | off | err)
```



```
#pragma warn(wpnu => on | off | err)
#pragma warn(wprg => on | off | err)
#pragma warn(wral => on | off | err)
#pragma warn(wrfp => on | off | err)
#pragma warn(wsto => on | off | err)
#pragma warn(wtxt => on | off | err)
#pragma warn(wubd => on | off | err)
#pragma warn(wvnu => on | off | err)
```

**#pragma warn(wall => on | off | err)** **cp**

Эта прагма влияет на установку всех предупреждений. Если установлено “on” или “err”, то все проверки выполняются.

**#pragma warn(wpcv => on | off | err)** **cp**

Pointer conversion.

Преобразование указателей. Когда установлено “on” или “err”, компилятор проверяет преобразование между двумя, несовместимыми типами указателей или между указателем и целочисленным типом. По умолчанию установлено “on”.

**#pragma warn(wdne => on | off | err)** **cp**

Declaration has no effect.

Бессмысленные объявления. Когда установлено “on” или “err”, компилятор проверяет наличие бессмысленных объявлений, например, long int;. Объявление должно содержать декларатор переменной, метку структуры или объединения, или членов перечисления. По умолчанию установлено “on”.

**#pragma warn(wsto => on | off | err)** **cp**

Storage class redeclared.

Переобъявление класса памяти. Когда установлено “on” или “err”, компилятор проверяет, чтобы вы не объявили одну и ту же переменную по-разному в различных частях программы. Например:

```
int x;           /* External linkage */
static int x;    /* Internal linkage */
```

Статический класс памяти всегда имеет преимущество. По умолчанию установлено “on”.

**#pragma warn(wtxt => on | off | err)** **cp**

Unexpected text in preprocessor command.

Неожиданный текст в команде препроцессора. Когда установлено “on” или “err”, компилятор ищет завершающий команду препроцессора символ новой строки. По умолчанию установлено “on”.

**#pragma warn(wprg => on | off | err)** **cp**

Unknown #pragma.

Неизвестная прагма. Когда установлено “on” или “err”, компилятор выполняет проверку допустимости прагм или ошибок в прагмах TopSpeed C. Если при создании текста вы использовали только прагмы TopSpeed C или C++, то следует установить значение этой прагмы равным “on” или “err”. По умолчанию установлено “on”.

**#pragma warn(wfnd => on | off | err)** **cp**

Function not declared.

Необъявленная функция. . Когда установлено “on” или “err”, компилятор проверяет, объявлена ли функция к которой происходит обращение. Если такая функция встретится, то компилятор TopSpeed C предполагает, что эта функция внешняя и возвращает значение типа int. По умолчанию установлено “off”.

**#pragma warn(wpnd => on | off | err)** **cp**

Function prototype not declared.

Прототип функции не объявлен. Когда установлено “on” или “err”, компилятор проверяет, имеет ли функция соответствующий ей прототип. Для компилятора TopSpeed прототип важен, поскольку без него он не может выполнить проверку соответствия типа. Лучше сохранить установку этой прагмы в “on” или “err”. По умолчанию установлено “off”.

**#pragma warn(wnre => on | off | err)** **cp**

No expression in return statement.

В операторе return не задано выражение. Когда установлено “on” или “err”, компилятор для функций, не объявленных как void, проверяет возвращает ли она значение. Если вы компилируете старый программный текст на C, в котором нет прототипов, то следует установить эту прагму в “off”. По умолчанию установлено “off”.

**#pragma warn(wnrv => on | off | err)** **cp**

No return value in function.

Функция не возвращает значение. Когда установлено “on” или “err”, компилятор для функций, не объявленных как void, проверяет возвращает ли она значение. По умолчанию установлено “off”.

**#pragma warn(watr => on | off | err)** **cp**

Different const attributes.

Различающиеся константные атрибуты. Когда установлено “on” или “err”, компилятор проверяет, получает ли функция, ожидающая указатель на переменную, вместо этого указатель на константу. По умолчанию установлено “on”.

**#pragma warn(wftn => on | off | err)** **cp**

Far to near pointer conversion.

Преобразование дальнего в ближний указатель. Когда установлено “on” или “err”, компилятор контролирует преобразование 32-х битных указателей в 16-ти битные. По умолчанию установлено “on”.

**#pragma warn(wntf => on | off | err)** **cp**

Near to far pointer conversion.

Преобразование ближнего в дальний указатель. Когда установлено “on” или “err”, компилятор контролирует преобразование 16-ти битных ближних указателей в 32-х битные дальние. По умолчанию установлено “on”.

**#pragma warn(wubd => on | off | err)** **cp**

Possible use of variable before assignment.

Возможно использование переменной до присвоения значения. Когда установлено “on” или “err”, компилятор проверяет, чтобы прежде чем использовать локальную переменную, ей было присвоено значение. Компилятор TopSpeed выполняет эту проверку простым сканированием функции, в которой может выполняться оператор goto и т.п., что может приводить к выдаче ложных предупреждений.

**#pragma warn(wpnu => on | off | err)** **cp**

Parameter never used in function.

Параметр функции никогда не используется. Когда установлено “on” или “err”, компилятор проверяет все ли параметры используются в коде функции, поэтому объявление пустых параметров приводит к появлению этого предупреждения. По умолчанию установлено “off”.

**#pragma warn(wdnu => on | off | err)** **cp**

Variable declared but never used.

Переменная объявлена, но не используется. Когда установлено “on” или “err”, компилятор проверяет все ли объявленные переменные используются в коде функции. По умолчанию установлено “on”.

**#pragma warn(wcne => on | off | err)** **cp**

Code has no effect.

Код не влияет на работу. Когда установлено “on” или “err”, компилятор проверяет операторы и оставшиеся операнды, перечисленные через запятую, чтобы выяснить, какие из них не используются. По умолчанию установлено “on”. Пример:

```
x y; /* выражение не влияет */
f, x; /* оставшийся операнд не влияет */
```

**#pragma warn(wcld => on | off | err)** **cp**

Conversion may lose significant digits.

При преобразовании возможна потеря значимости. Когда установлено “on” или “err”, компилятор выполняет проверку преобразования из long или unsigned long в int или unsigned int. По умолчанию установлено “on”.

**#pragma warn(wait => on | off | err)** **cp**

Assignment in test expression.

Присвоение в выражении сравнения. Когда установлено “on” или “err”, компилятор проверяет возможную ошибку в C: оператор сравнения (он состоит из двух знаков равенства “= “). Пример:

```
if (x=y) printf("X equals Y"); /* ошибка */
```

По умолчанию установлено “on”.

**#pragma warn(wetb => on | off | err)** **cp**

Value of escape sequence is too large.

“Эскейп последовательность” превышает предел. Когда установлено “on” или “err”, компилятор проверяет, чтобы значение “эскейп последовательности” находилось в диапазоне от 0 до 255. По умолчанию установлено “on”.

**#pragma warn(wcor => on | off | err)** **cp**

Value of constant is out of range.

Значение константы за границами допустимого диапазона. Когда установлено “on” или “err”, компилятор проверяет, находится ли целочисленная константа в пределах диапазона допустимых значений для unsigned long или константа с плавающей в диапазоне для long double. По умолчанию установлено “on”.

**#pragma warn(wclt => on | off | err)** **cp**

Constant is long.

Константа типа long. Когда установлено “on” или “err”, компилятор выполняет проверку целочисленные константы, по значению которых их следует отнести к типу long, но суффикса L у них нет. По умолчанию установлено “off”.

**#pragma warn(wral => on | off | err)** **cp**

Returns address of local variable.

Возвращается адрес локальной переменной. Когда установлено “on” или “err”, компилятор проверяет не возвращает ли оператор return адрес локальной переменной. Это может послужить причиной ошибки, поскольку в C память, занимаемая локальной переменной по завершении функции повторно используется для других целей. Таким образом, указатель ссылается на неопределенные данные. По умолчанию установлено “on”.

**#pragma warn(wpin => on | off | err)****cp**

Default type promotion on parameter.

Распространение на параметр умалчиваемого типа. Когда установлено “on” или “err”, компилятор сравнивает объявление параметра в объявлении функции, сделанном в устаревшем стиле, с прототипом на предмет несовместимости аргументов. Пример:

```
int func(char); /* параметр объявлен как char */
```

```
int func(IntegerByPromotion);
```

```
char IntegerByPromotion;
```

```
/* НЕСОВМЕСТИМОСТЬ */
```

```
{  
    ...  
}
```

Замечание: Это нарушение совместимости со стандартом ANSI совместимости объявлений функций. По умолчанию устанавливается в “on”.

**#pragma warn(wpic => on | off | err)****cp**

Parameter list inconsistent with previous call.

Список параметров не совпадает с предыдущим обращением к функции. Это предупреждение выдается, если объявление параметра не совместимо с соответствующим параметром в предыдущем обращении к функции. По умолчанию устанавливается в “on”.

**#pragma warn(wnid => on | off | err)****cp**

Address for local variable not in DGROUP.

Адрес локальной переменной не указывает внутрь DGROUP. Когда установлено “on” или “err”, компилятор проверяет, чтобы адрес локальной переменной не был в соответствии с моделью памяти small, при том, что используется прагма #pragma data(ss\_in\_dgroup => off). По умолчанию устанавливается в “on”.

**#pragma warn(wrfp => on | off | err)****cp**

Function redeclared with fixed parameters.

Функция переобъявлена с фиксированным числом параметров. Когда установлено “on” или “err”, компилятор проверяет не указано ли в прототипе переменное число параметров, тогда как в соответствующем объявлении функции задано фиксированное число аргументов. Это работает в TopSpeed C, но является нарушением правил ANSI C по совместимости объявлений функций и, следовательно, нарушает переносимость кода. По умолчанию устанавливается в “on”.

**#pragma warn(wvnu => on | off | err)****cp**

Local variable never used.

Локальная переменная никогда не используется. Когда установлено “on” или “err”, компилятор проверяет ситуацию, когда локальная переменная объявлена и ей присвоено значение, но тем не менее она больше нигде не используется. По умолчанию устанавливается в “on”.

**#pragma warn(wovr => on | off | err)** **cp**

Overflow in constant expression.

Переполнение в константном выражении. Это предупреждение выдается, когда происходит переполнение целочисленного константного выражения. По умолчанию устанавливается в “on”.

**#pragma warn(wacc => on | off | err)** **cp**

Default access specifier used for base class.

Для базового класса используется спецификатор доступа по умолчанию. Это предупреждение выдается, когда в спецификации базового класса не задан спецификатор доступа и используется спецификатор по умолчанию (т.е. public для struct и private для класса). По умолчанию устанавливается в “on”.

**#pragma warn(wdel => on | off | err)** **cp**

Expression in delete[] is obsolete.

Выражение в delete[] имеет устаревший синтаксис. Это предупреждение выдается, когда задано выражение в квадратных скобках в выражении delete. Такое выражение игнорируется. Этот элемент синтаксиса C считается устаревшим. По умолчанию установлено “on”.

**#pragma warn(wovl => on | off | err)** **cp**

Keyword ‘overload’ is not required.

Ключевое слово ‘overload’ не требуется. Это предупреждение выдается, когда в C задано ключевое слово ‘overload’. Этот элемент синтаксиса C считается устаревшим. По умолчанию установлено “on”.

**#pragma warn(wcic => on | off | err)** **cp**

Constant in code segment requires initialization.

Константа в кодовом сегменте должна быть инициализирована. Это предупреждение выдается, когда во время выполнения требуется инициализация помещаемой в кодовый сегмент константы, как это может быть для объекта, объявленного в C константным, и который инициализируется выражением, и при этом установлена в “on” прагма “const\_in\_code”. В защищенном режиме в OS/2 или Window 3 это может привести к нарушению защиты памяти, поэтому при появлении этого сообщения следует установить прагму const\_in\_code в “off”. По умолчанию установлено “on”.

**#pragma warn(wcrt => on | off | err)****ср**

Class definition as function return type, missing ‘;’ after ‘}’?

Типом возвращаемого функцией значения является класс, после ‘}’ может быть пропущено ‘;’ ? Это предупреждение выдается, когда в качестве типа возвращаемого значения указано определение класса. Такая конструкция допустима, но необычна, и часто возникает из-за пропуска точки с запятой между определением функции и объявлением класса. По умолчанию установлено “on”.

## Прагма класса Project

---

Прагма с именем класса “project” используются для того, чтобы передать информацию от компилятора в Систему ведения проекта. Значением этой прагмы является стока, которая затем сохраняется в объектном файле для последующего использования проектной системой. Когда же объектный файл добавляется в список компоновки, то текст, заданный этой прагмой, выполняется как команда Системы ведения проекта. Например, если заголовочный файл включается строкой:

```
#pragma project(“#set myflag=on”)
```

то как только исходный файл, который включает в себя этот заголовок, добавляется в проект, макрос проектной системы “myflag” будет установлен в “on”.

Эта прагма указывается только в исходном файле, а не в файле описания проекта.

## Прагмы Save/Restore

Прагмой “save” сохраняется состояние всех прагм, так что потом можно восстановить их с помощью прагмы “restore”. Прагмы Save/Restore работают по типу стека, поэтому допускается их вложенность. Например:

```
/*save the #pragma state and enable      the interrupt convention*/  
#pragma save  
#pragma call(interrupt => on)  
/* interrupt functions are specified here */
```

```
#pragma restore
```

На число прагм “save” ограничений нет, кроме объема доступной памяти. Эти прагмы можно использовать в исходном файле или в проектном файле.

## Прагмы Link

---

```
#pragma link( <filename> {,<filename> } )  
#pragma linkfirst( <filename> )
```

Эти прагмы можно задавать в project-файле, в этом случае указанные в них файлы немедленно добавляются в список компоновки. Кроме того, прагмы класса “link” могут

указываться в исходном файле С и С++, тогда указанные в них файлы добавляются в список компоновки во время выполнения проектной системой команды `autocompile`, если в каком-либо файле, уже находящемся в списке компоновки имеется эта прагма.

Например:

```
#pragma link( file1.obj, file2.obj, file3.lib )
#pragma linkfirst (initexe.obj)
```

Если расширение не задано, то подразумевается `.obj`. Файлы задаваемые прагмой “`link`” добавляются в конец списка компоновки (если только уже не находились в этом списке). Используя прагму `linkfirst`, можно задавать только один файл.

## Прагмы класса `Link_Option`

Прагмы с именем класса “`link_option`” используются для того, чтобы задать опции компоновщика. Эти прагмы можно указывать только в проектном файле.

Есть следующие прагмы класса “`link_option`”:

```
#pragma link_option(case => on | off )
#pragma link_option(decode => on | off )
#pragma link_option(link=> <string>)
#pragma link_option(map => on | off)
#pragma link_option(overlay => on | off )
#pragma link_option(pack => on | off )
#pragma link_option(shift => num)
#pragma link_option(share_const => on | off)
#pragma link_option(icon => iconname)
```

**`#pragma link_option(map => on | off)`                      p**

Эта прагма определяет, генерируется ли map-файл с информацией о размерах сегментов, public-именах и т.д. По умолчанию этот файл генерируется.

**`#pragma link_option(case => on | off )`                      p**

Эта прагма определяет, различает ли компоновщик регистры букв в именах. По умолчанию установлено “`off`”.

**`#pragma link_option(pack => on | off )`                      p**

Эта прагма определяет, упаковываются ли сегменты вместе. По умолчанию установлено `pack=>on`.

**`#pragma link_option(decode => on | off )`                      p**

Значение этой прагмы показывает, должен ли компоновщик расшифровывать имена в



MAP-файле также как public-имена. Эта опция устанавливается в “on”, только если в проект включаются исходные файлы на C, в противном случае “off”.

**#pragma link\_option(shift => num)** **p**

Эта прагма указывает счетчик сдвигов при выравнивании сегментов для исполняемых модулей нового формата. По умолчанию равно 4, означающее, что данный сегмент выравнивается на границу, кратную 16-ти.

**#pragma link\_option(link => <string> )** **p**

Эта прагма задает команду проектной системы, которая должна выполняться по команде #dolink.

**#pragma link\_option(share\_const=> on | off)** **p**

Эта прагма определяет, объединяет ли 16-ти битный компоновщик идентичные константы. По умолчанию устанавливается в “on”, уменьшая тем самым, однако программисту на C может потребоваться выключить эту прагму, чтобы константы имели различные адреса.

**#pragma link\_option(icon => iconname)** **p**

Эта прагма задает имя файла - пиктограммы для этого приложения (icon => MyIcon.ico).

## **Прагмы класса Define**

---

Прагма с именем класса “define” используется для того, чтобы определить символическое имя для условной компиляции. Это имя можно опросить директивами компилятора OMIT и COMPILE. Смотрите Описание языка.

Эту прагму можно использовать только в проектном файле.

Прагма “define” имеет форму:

**#pragma define(идентификатор=>значение)**

где “идентификатор” это определяемое символическое имя, а “значение” задает присваиваемое этому имени значение.

Для Modula-2 данный идентификатор определяется как булева константа имеющая значение ИСТИНА, если имя определено и ЛОЖЬ - если нет.

Для C и C++ данный идентификатор определяется как макрос. Если задано значение “on”, то значением макроса определяется как 1. Если задано значение “off”, то макрос не определен. Любое другое значение приводит к тому, что определяемый как макрос идентификатор расширяется до указанного значения. Может быть задана только одна

лексема языка C или C++, в противном случае компилятор выдаст сообщение об ошибке. Для того, чтобы определить макрос, значением которого является строковый литерал, используйте такую форму прагмы: `define (name => "fred")`.

**`#pragma define(maincode => on | off )`**                    **p**

Эта прагма разрешает (“on” - по умолчанию) или запрещает (“off”) генерацию инициализирующего кода. Выключайте ее при генерации общих или библиотечных модулей.

**`#pragma define(zero_divide => on | off )`**                    **p**

Эта прагма задает действие, предпринимаемое при делении на ноль. Когда установлено “on” (по умолчанию), деление на ноль дает в результате 0, а когда установлено “off”, то в результате возникает прерывание.

**`#pragma define(logical_round => on | off )`**                    **p**

Эта прагма задает действие, предпринимаемое при “усечении” REAL до LONG. Когда установлено “on”, результат округляется до ближайшего большего целого числа. Когда установлено “off” (по умолчанию), округления не производится.

**`#pragma define(stach_threshold => size )`**                    **p**

Эта прагма задает размер (в байтах) предела, при котором любой структуре данных, превышающей эту величину (по умолчанию 16384) выделяется память из кучи, а не в стеке.

**`#pragma define(BCD_Arithmetic => on | off )`**                    **p**

Эта прагма задает использование двоично-десятичной арифметики, когда установлена в “on” (по умолчанию), или арифметики с плавающей точкой, в случае, когда установлено в “off”.

**`#pragma define(BCD_ULONG => on | off )`**                    **p**

Эта прагма задает использование для переменных типа ULONG двоично-десятичной арифметики, когда установлена в “on” (по умолчанию), или арифметики с плавающей точкой, в случае, когда установлено в “off”.

**`#pragma define(BCD_Large => on | off )`**                    **p**

Эта прагма разрешает или запрещает использование переменных DECIMAL и PDECIMAL больше чем пятнадцать (15) цифр. По умолчанию устанавливается “on” (разрешено).

## **Предопределенные флажки.**

Каждый раз, когда командой `#compile` компилируется программа, система ведения проекта автоматически определяет ряд флажков, установленных в “on” или “off”, в

зависимости от назначения. Для управления процессом создания программы можно использовать эти предопределенные флажки. Есть четыре флажка, которые автоматически создаются компилятором, и которые можно использовать в операторах OMIT и COMPILE для организации условной компиляции.

<u>_WIDTH32_</u>	Устанавливается в “on” для 32-х разрядных приложений и в “off” для 16-ти разрядных.
<u>_CW_</u>	Устанавливается в “on” для Clarion для Windows .
<u>_CLW20_</u>	Устанавливается в “on” для Clarion для Windows версии 2.0.
<u>_CDD_</u>	Устанавливается в “on” для Clarion для DOS.

## ***Project System Examples***

Далее приводится пример некоторого набора команд проектной системы, который используется в TopSpeed для создания файловых драйверов. В нем используется широкий набор операторов системы поддержки проекта и демонстрируется, как можно использовать проектную систему для управления точностью и завершенностью даже при реализации самых сложных проектов.

Команды этого примера разделены на четыре файла, что демонстрирует поддержку проектной системой структурного программирования, модульности повторного использования кода. Это файлы: ALLDRV.PR, из которого происходит обращение к оператором #call к файлу ORACLE.PR (наряду с другими), который включает в себя оператором #include файл SQLFILES.PR, который в свою очередь DRVKIT.PI.

### **ALLDRV.PR**

```
#system win dll
#model clarion

#set drvdebug = full
#set kitdebug = full

#set release = off
#set fromclw = on
#set incbuildno = off
#set demo = off

#if “%release”=“on” #then
#pragma define(_RELEASE=>on)
#set incbuildno = on
#set kitdebug = off
#set drvdebug = off
```

```
#endif
```

```
#pragma define(DEMO_VERSION=>%demo)
```

```
#set domodels=
```

```
{
```

```
#abort on
```

```
#set down32=off #set dolib=off
```

```
#call %%prjfile
```

```
#set down32=off #set dolib=on
```

```
#call %%prjfile
```

```
#set down32=off #set dolib=off
```

```
#abort off
```

```
#set down32=on #set dolib=off
```

```
#call %%prjfile
```

```
#set down32=on #set dolib=on
```

```
#call %%prjfile
```

```
#abort on
```

```
{
```

```
#if #exists btrieve.pr #then #set prjfile=btrieve.pr %domodels #endif
```

```
#if #exists odbc.pr #then #set prjfile=odbc.pr %domodels #endif
```

```
#if #exists cla21.pr #then #set prjfile=cla21.pr %domodels #endif
```

```
#if #exists tps.pr #then #set prjfile=tps.pr %domodels #endif
```

```
#if #exists dos.pr #then #set prjfile=dos.pr %domodels #endif
```

```
#if #exists ascii.pr #then #set prjfile=ascii.pr %domodels #endif
```

```
#if #exists basic.pr #then #set prjfile=basic.pr %domodels #endif
```

```
#set domodels=
```

```
{
```

```
#set down32=off #set dolib=off
```

```
#call %%prjfile
```

```
#set down32=off #set dolib=on
```

```
#call %%prjfile
```

```
#set down32=off #set dolib=off
```

```
{
```

```
#file redirect ts.red
```

```
#if #exists sql400.pr #then #set prjfile=sql400.pr %domodels #endif
```

```
#if #exists oracle.pr #then #set prjfile=oracle.pr %domodels #endif
```

**ORACLE.PR:**

```
#noedit
```

```
-----
-----
— ORACLE.PR проектный файл по созданию драйвера Oracle
-----
-----
```

```
#system win dll — Windows - ОС для которой делаем dll
#model clarion — модель памяти - clarion
```

```
— установим значения макросов по умолчанию. Эти “переключатели”
— будут управлять процессом создания.
```

```
#set drv = ORA
#set trace = off
#set heapchk = off
#set drvdebug = full
#set sqldebug = full
#set kitdebug = off
— #set release = on
```

```
#expand ORACLEIN.CPP —установим %spath - путь к создаваемому файлу
                      —%opath -путь в кат., где открываются файлы
                      — %ext расширение CPP
                      — %tail равным ORACLEIN
                      — %cdir - каталог, где создаются файлы
                      — %odir - каталог, где открываются файлы
```

```
#set drvdir = %odir —запомним значение %odir
#set sql_type = O —установим %sql_type равным O
```

```
— Определим символ условной компиляции для последующих процессов
— Этот символ - DRVSPEC и его значение равно “oraclesp.h”
```

```
— DRVSPEC доступен для прочтения в операторах OMIT и COMPILE
— подробнее смотрите в Описании языка Clarion.
#pragma define(DRVSPEC=>”oraclesp.h”)
```

```
— компилируем модули sql с соответствующим уровнем отладочного кода.
#include SQLFILES.PR —Здесь выполняются операторы файла SQLFILES.PR
                   — Все макросы, прагмы и список компоновки
                   — доступны во включаемых операторах.
```

```
#compile ORACLEIN.CPP
```

```
—
```

компилируем исходный текст на C++

#compile ORAIMPOR.CW /define (maincode=>off) —и текст на Clarion.

—оба добавляются в список компоновки.

#pragma link(ORAIMPOR.RSC) —добавить ORAIMPOR.RSC в список компоновки

#pragma link (ORA7WIN.LIB) —добавить ORA7WIN.LIB в список компоновки

#pragma link (%lnkpfx%asc%S%.LIB) —добавить CWASC16.LIB в список

—%lnkpfx% установлен в CW,

—%S% установлен в “16”

— серия операторов помещенных в drv\_Link выполняется в самом конце

— файла DRVKIT.PI file. Эти операторы предназначены для компоновки

— и изменений в драйвере

%drv\_Link

### **SQLFILES.PR:**

— Окончательная версия: выключим всю отладку и трассировку

```
#if “%release”=“on” #then
```

```
#set drvdebug = off
```

```
#set kitdebug = off
```

```
#set sqldebug = off
```

```
#set trace = off
```

```
#set heapchk = off
```

```
#endif
```

— убедимся, что переключатель sql\_type явно установлен (no default)

```
#if ‘%sql_type’=’ ’ #then
```

```
#error “sql_type must be set”
```

```
#endif
```

— по умолчанию минимизация кода. Установим символ DRIVER\_COMPACT

— на основании значения макроса %compact

```
#if “%compact”=”” #then #set compact=on #endif
```

```
#if ‘%compact’=’off’ #or ‘%heapchk’=’on’ #then
```

```
#pragma define(DRIVER_COMPACT=>off)
```

```
#else
```

```
#pragma define(DRIVER_COMPACT=>on)
```

```
#endif
```

```
#include DRVKIT.PI
```

— тут идут команды из файла DRVKIT.PI  
 — Все макросы, прагмы и список компоновки  
 — целиком доступны во включаемых операторах.

```
#pragma save
```

— сохраним текущие установки прагм, так что  
 — их можно будет потом восстановить прагмой  
 — restore.

— отладка: вкл. отладочную информацию и проверки во время выполнения

```
#if "%sqldebug"="" #then #set sqldebug=off #endif —default is off
#pragma debug(vid=>%sqldebug)
#if "%sqldebug"="full" #then
```

— для всесторонней отладки

```
#pragma check(index=>on,range=>on,overflow=>on) —вкл проверки во
```

— время выполнения

```
#pragma debug(line_num=>on) —и нумерацию строк
#endif
```

```
#pragma warn(wall=>on)
```

—вкл. Все сообщения об ошибках

```
#set srcfile = SAFESTR.CPP
```

—установим %srcfile в SAFESTR.CPP

```
#set dstfile = %sql_type%SAFE.CPP
```

— установим %dstfile в OSAFE.CPP

— выполним серию операторов, присвоенных makesrc в середине файла  
 — DRVKIT.PI file. Эти операторы разработаны для того, чтобы получить  
 — точку входа в конструктор C++, чтобы иметь различные имена для каждого драйвера.

```
%makesrc
```

```
#set srcfile = SQLOPEN.CPP
```

— повторение

```
#set dstfile = %sql_type%SQLOPE.CPP
%makesrc
```

```
#set srcfile = SQLUPDAT.CPP
```

— повторение

```
#set dstfile = %sql_type%SQLUPD.CPP
%makesrc
```

```
#set srcfile = SQLRETRI.CPP
```

— повторение

```
#set dstfile = %sql_type%SQLRET.CPP
%makesrc
```

```
#set srcfile = SQLGLOB.CPP
```

— повторение

```
#set dstfile = %sql_type%SQLGLO.CPP
%makesrc
```

```

    #if %system=win #then
— по условию
    #set srcfile = SQLVIEW.CPP           — повторение
    #set dstfile = %sql_type%SQLVIEW.CPP
    %makesrc
    #endif

    #pragma restore           —восстановить ранее сохраненные установки прагм
    #pragma warn(wall=>on)    —включить все виды предупреждений

```

### **DRVKIT.PI:**

```

#noedit
-----
-----
—  DRVKIT.PI  Включаемый файл из инструментального набора Driver Kit
-----
-----

— DrvKit.Pi содержит команды проектной системы общие для всех
— драйверов, реализуемых на основе Driver Kit. Перед включением
— этого файла должны быть сделаны следующие установки:
—
—  #set drv      = символ 3 или 4 : идентификатор
—                драйвера (т.е C21)
—  #pragma define(DRVSPEC="c21specs.h") — или соответств.файл
—
—  Любые другие установки, определяющие спецификации драйвера
—
—  Необязательно:
—
—  #set trace    = on      — включить трассировку
—  #set heapchk  = on      — включить проверку кучи
—  #set common   = on      — объединение одинакового кода
—
—  Окончательная версия: отключить всю отладку и трассировку
#if "%release"="on" #then
#set drvdebug = off
#set kitdebug = off

```



```
#set trace    = off
#set heapchk  = off
#endif
```

— Среда Windows: убедиться, что приняты соглашения Clarion 4 Windows

```
#pragma warn(wall=>on)           —вкл. все предупреждения
#pragma define(__CLARION__=>on)  —установить переключатель директив
```

```
#if #not %system=dos #then      — dos,
#if "%dowin32"=on #then
#system win32 dll              — win32, или
#else
#system win dll                — win16
#endif
#model clarion
```

```
#if %filetype=dll #then
#pragma define(_WINDLL=>on)     —установить переключатель директив
#endif
```

— Для этого набора установить соответствующие имена файлов.

```
#if "%dolib"="" #then
#set dolib=off                 —по умолчанию dolib = off
#endif
#if "%dowin32"="" #then
#set dowin32=off              — по умолчанию dowin32 = off
#endif
#if %dolib=on #then            —установить префикс
#set lnkpfx = CL
#else
#set lnkpfx = CW
#endif
#if "%pfx"="" #then
#set pfx    = %drv%           — установить префикс в имени драйвера
#endif
#else
#set dolib=off
#if %model=extendll #then
#pragma define(_XTDDL=>on)
#endif
```

```
#set pfx    = %M%%drv%
#if '%RWMODE%' = 'on' #then
```

```
#set lnkpfx=drw
#else
#set lnkpfx=%clapfx%
#endif
#set S = "" —установить суффикс в null
#endif
```

```
#set drvname = %lnkpfx%%drv%%S% —собрать имя
#message "Making %drvname% File Driver" —вывести сообщение
#if #not "%inbrowser" #then
#if #exists %drvname%.ver #then —по условию...
#compile %drvname%.ver — компилировать драйвер
#endif
#endif
```

```
— проверка кучи: компилировать и включить Heap Checker (без отладки)
#if "%heapchk"="on" #or "%heapdbg"="on" #then
#set heapchk = on
#pragma define(HEAPCHK=>on) — установить переключатель директив.
#if "%heapdll"="on" #then —если dll, то
#pragma link(%clapfx%hchk.lib) —доб. библи. heapchk в список комп.
#else
#pragma save
— сохраним установки прагм.
#if "%heapdbg"="on" #then —по условию...
#pragma debug(vid=>full) — включим отладочный код.
#endif
#compile HEAPCHK.C #to %pfx%HCHK.OBJ —компил. проверку кучи.
#compile STRCHK.C #to %pfx%SCHK.OBJ
#compile NEW.CPP #to %pfx%NEW.OBJ
#pragma restore —восстановить установки прагм.
#endif
#endif
```

```
— отладка: инф. для отладчика и проверки во время выполнения
#if "%drvdebug"="" #then #set drvdebug=full #endif —по умолч. "full"
#pragma debug(vid=>%drvdebug)
#if "%drvdebug"="full" #then
#pragma check(index=>on,range=>on,overflow=>on) —вкл. проверки
#pragma debug(line_num=>on) —вкл. номера строк
```

```
#endif
```

—общий код: некоторые куски кода Driver Kit включаются несколько раз

— библиотеки файлового драйвера

```
#if "%common"="on" #then
```

— по условию...

```
#pragma define(COMMON_CODE=>on)
```

— определим переключатель

```
#endif
```

```
#if '%drvdir' = ' ' #then
```

```
#error "drvdir must be set to build lib versions of the drivers"
```

```
#endif
```

— Чтобы получить точку входа в конструктор C++ с целью иметь разные

— имена для каждого драйвера, если нужно компилировать разные

— исходные модули C++ :

```
#set makesrc = ‘
```

```
  #if (#not #exists %%%drvdir%%dstfile) #or (%%drvdir%%dstfile #older %%srcfile
) #or (%%srcfile #older %%%drvdir%%dstfile ) #then
```

```
  #expand %%srcfile
```

```
  #run "copy %%opath %%%drvdir%%dstfile > NUL "
```

```
  #endif
```

```
  #compile %%dstfile %%defs
```

```
‘
```

— Driver Kit: компиляция исходных текстов их Driver Kit

```
#pragma save
```

— сначала сохраним установки прагм

```
#if #not "%kitdebug"="" #then
```

— по условию...

```
#pragma debug(vid=>%kitdebug) — зададим степень отладочн. проверок
```

```
#else
```

```
#pragma debug(vid=>off)
```

```
#endif
```

```
#set srcfile = DRVL1.C
```

Система поддержки проекта TOPSPEED

— установить имя исходного файла

```
#set dstfile = %pfx%L1.C
```

— зададим в dstfile имя с правил. префикс.

```
set dstfile name w correct prefix
```

```
%makesrc
```

—выполнить определенные выше операторы.

```
#if #not "%nocommon"="on" #then
```

—если общий код

```
#set srcfile = DRVSTATE.C
```

—сделать DRVSTATE

```
#set dstfile = %pfx%STAT.C
```

```
%makesrc
```

```
#if #not (%system=dos) #then
#set srcfile = DRVVIEW.CPP
#set dstfile = %pfx%VIEW.CPP
%makesrc

#if #not (%filetype=dll) #then
#pragma define(DRV_HAS_LIBMAIN=>on)
#endif

#set srcfile = DRVW.C
#set dstfile = %pfx%W%dolib%.C
#set defs = '/define(_LIB_TARGET=>%dolib%)'
%makesrc
#set defs = "

#set srcfile = DRVWUTIL.C
#set dstfile = %pfx%WUTI.C
%makesrc

#if #not %dolib% #then
#set srcfile = DRVDIAL.CLW
#set dstfile = %pfx%DIAL.CLW
#set defs = '/define(maincode=>off)'
%makesrc
#set defs = "
#pragma link(%pfx%dial.rsc)
#endif
#endif
#endif
#if "%trace"="on" #then
#pragma save, define(TRACE=>on)
#set srcfile = DRVTRACE.C
#set dstfile = %pfx%TRAC.C
%makesrc
#endif

#set srcfile = DRVPIPE.C
#set dstfile = %pfx%P%dolib%.C
#set defs = '/define(_LIB_TARGET=>%dolib%)'
%makesrc
#set defs = "
```

—если не для DOS  
—сделать DRVVIEW

—по условию...  
— установить переключатель

—сделать DRVWUTIL

—по условию  
— сделать DRVDIAL

—доб. DIAL в список компоновки

—по условию...  
— сохранить установки...  
— сделать DRVTRACE

—сделать DRVPIPE

```
#if "%trace"="on" #then
#pragma restore                                —восстановить установки прагм
#endif
#pragma restore
```

— создать макрос drv\_Link, чтобы позже использовать его

— для компоновки и исправлений в файловом драйвере:

```
#set drv_Link =
(
#pragma link_option(share_const=>on)

#if #not (%%system=dos) #then
#if "%dolib"="on" #then
#dolink %%drvname%%.lib
#else
#implib %%drvname%%.lib %%drvname%%.exp
#if define(_CW15)=on #then
#pragma linkfirst(idll%%S%%w.obj)
#else
#pragma linkfirst(icwdll.obj)
#endif
#pragma link(win%%S%%.lib)
#pragma link(cwrun%%S%%.lib)
#pragma link_option(decode=>off)
#dolink %%drvname%%.dll
#endif
#if "%make" #and #not "%dolib"="on" #then
#exemod %%drvname%%.dll %%drvname%%.exp %%drvname%%.map
#endif
#else
#if %%filetype=dll #then
#implib %%drvname%%.lib %%drvname%%.exp
#endif
#set tscla = on
#set tscpp = off
#link %%drvname%%

#if "%make" #and (%%filetype=dll) #then
#expand %%drvname%%.dll
#run "mkdriver %%cpath > NUL"
#endif
#endif
)
```

## Модуль определения файлов (.EXP)

Файл определения модуля описывает имя, атрибуты, экспортные и другие характеристики динамической (DLL) библиотеки для Microsoft Windows. Этот файл обязателен для использования ее в Windows.

Этот файл (с расширением .EXP) генерируется всякий раз, когда создается новый проект или проект, у которого изменен тип создаваемого модуля, операционная система или библиотека времени выполнения.

### 16-Bit

---

#### Синтаксис файла определения модуля

Файл определения модуля содержит один или несколько операторов. Каждый оператор определяет атрибут исполняемого файла, такой как имя модуля, атрибуты программных сегментов и число и именно экспортируемых символов. Ниже перечислены эти операторы и определяемые ими атрибуты:

Оператор	Атрибут
NAME	Имя приложения
LIBRARY	Имя dll-библиотеки
CODE	Дает кодовым сегментам атрибуты по умолчанию
DATA	Дает сегментам данных атрибуты по умолчанию
SEGMENTS	Задаёт конкретному сегменту атрибуты.
STACKSIZE	Задаёт размер локального стека в байтах.
EXPORTS	Определяет экспортируемые функции.
HEAPSIZE	Задаёт размер кучи.
EXETYPE	Идентифицирует операционную систему.

Использование этих операторов регулируется следующими правилами:

- Если используется оператор NAME или LIBRARY, то он должен идти перед остальными операторами в файле определения модуля.
- В файл определения модуля можно включать построчный комментарий, начиная строку точкой с запятой (;). Утилиты такие строки комментария игнорируют.
- Ключевые слова определения модуля, такие как NAME, LIBRARY и SEGMENTS должны быть написаны заглавными буквами.

**Пример файла определения модуля**

Следующий пример содержит описание модуля для динамической библиотеки: LIBRARY MyDLL

; Пример экспортируемого файла

EXPORTS

Func1 @1

Var1 @2

Func2 @3

Func3 @4

Func4 @5

**Оператор NAME**

Оператор NAME идентифицирует файл как исполняемое приложение (в отличие от библиотеки) и кроме того, может определять имя и тип приложения.

**NAME [имя\_приложения] [тип\_приложения]**

имя_приложения	Если задан параметр имя_приложения, то его значение становится именем, под которым это приложение известно операционной системе. Если этот параметр не задан, то именем приложения становится имя исполняемого файла без расширения.
тип_приложения	Используется для управления поведением программы в среде Windows. Эта информация хранится в заголовке исполняемого файла. Этот параметр может иметь одно из следующих значений:
WINDOWAPI	Приложение использует Интерфейс прикладного программирования (API), обеспечиваемый Windows, и должно выполняться в этой среде.
GUI	То же, что и WINDOWAPI.
CUI	Данная программа использует текстовый интерфейс подобный интерфейсу DOS.
	Если в файле определения модуля указан оператор NAME, то нельзя задавать оператор LIBRARY.

Если в файле определения модуля не указан ни оператор NAME, ни LIBRARY, то подразумевается использование оператора NAME.

В следующем примере определяемому приложению присваивается имя wdemo:

NAME wdemo GUI

**Оператор LIBRARY**

Оператор LIBRARY идентифицирует файл как динамическую библиотеку. Кроме того, могут быть указаны имя библиотеки и тип требуемой инициализации библиотечного

модуля.

### **LIBRARY [имя\_библиотеки][инициализация]**

имя_библиотеки	Если задан параметр имя_библиотеки, то его значение становится именем, под которым эта библиотека известна операционной системе. Если этот параметр не задан, то именем библиотеки становится имя исполняемого файла без расширения.
инициализация	Поле инициализация не является обязательным и может иметь одно из двух перечисленных выше значений. Если не задано ничего, то по умолчанию инициализация INITINSTANCE.
INITGLOBAL	Подпрограмма инициализации вызывается только тогда, когда библиотечный модуль первоначально загружается в память.
INITINSTANCE	Подпрограмма инициализации вызывается каждый раз, когда к библиотечному модулю обращается новый процесс.

Если в файле определения модуля указан оператор LIBRARY, то нельзя задавать оператор NAME.

В следующем примере определяемой dll-библиотеке присваивается имя mydll и указывается, что инициализация выполняется каждый раз, когда к библиотеке myDLL обращается новый процесс.

### **LIBRARY myDLL INITINSTANCE**

#### **Оператор CODE**

Оператор CODE определяет атрибуты, используемые по умолчанию для сегментов кода в библиотеке.

### **CODE [атрибут1] [атрибут2] ... [атрибутn]**

атрибут	Каждый указанный атрибут должен соответствовать одному из следующих полей-атрибутов. Каждое поле можно указать только один раз, а их последовательность не имеет значения. Ниже представлены поля-атрибуты вместе с их допустимыми значениями. В каждом случае значение, используемое по умолчанию, приведено последним.
---------	--

Поле	альтернативные значения
load	PRELOAD
executeonly	EXECUTEONLY
iopl	IOPL
conforming	CONFORMING
shared	SHARED
	LOADONCALL
	EXECUTEREAD
	NOIOPL
	NONCONFORMING
	NONSHARED



moveable	MOVEABLE		FIXED
discard	NONDISCARDABLE		DISCARDABLE

Поле load определяет, когда загружается сегмент. В этом поле указывается одно из следующих ключевых слов:

PRELOAD	Сегмент загружается автоматически, в начале выполнения программы, и, при использовании системы оверлеев TopSpeed никогда не выгружается.
LOADONCALL	Пока к сегменту нет обращения он не загружается в память. Поле executeonly определяет можно ли сегмент кода читать и выполнять. В нем указывается одно из следующих ключевых слов:
EXECUTEONLY	Сегмент можно только выполнять.
EXECUTEREAD	Сегмент можно и выполнять и читать (по умолчанию). Поле iopl определяет, имеет или нет сегмент привилегии по вводу-выводу (т.е. можно или нет из него обращаться напрямую к аппаратным средствам компьютера). В этом поле указывается одно из следующих ключевых слов:
IOPL	Этот сегмент кода имеет привилегии по вводу-выводу.
NOIOPL	Этот сегмент кода не имеет привилегий по вводу-выводу (по умолчанию). Поле conforming определяет, соответствует или нет данный сегмент кода сегментам 286 процессора. Соответствие сегмента связано с уровнем привилегий (набором команд, которые может выполнять процессор в этом сегменте) и касается только тех пользователей, которые пишут драйверы устройств и процедуры системного уровня. К сегменту, который соответствует сегментации 286-го процессора можно обращаться с Уровня 2 или Уровня 3, а данный сегмент выполняется на уровне привилегий пользователя. В этом поле указывается одно из следующих ключевых слов:
CONFORMING	Означает, что сегмент соответствует сегментации 286 процессора.
NONCONFORMING	Означает несоответствие кодового сегмента (по умолчанию). Поле shared определяет, могут ли все экземпляры программы разделять данный сегмент кода. Это поле обеспечивает использование сегмента в real-режиме Windows. Поле shared может содержать одно из следующих ключевых слов:
SHARED	Все экземпляры программы могут разделять данный сегмент кода.
NONSHARED	Данный сегмент кода нельзя разделять между экземплярами программы (по умолчанию). Поле moveable определяет, можно ли перемещать сегмент в памяти. Это поле обеспечивает использование сегмента в real-режиме Windows. Поле moveable может содержать одно из следующих ключевых слов:

MOVEABLE FIXED	Данный кодовый сегмент можно перемещать в памяти. Данный кодовый сегмент в памяти перемещать нельзя (по умолчанию). Поле discard определяет может ли операционная система выгружать данный сегмент на диск, когда в нем нет необходимости. Этот атрибут обеспечивает использование сегмента в real-режиме Windows. Поле discard может содержать одно из следующих ключевых слов:
DISCARDABLE NONDISCARDABLE	Этот кодовый сегмент можно выгружать на диск. Этот кодовый сегмент нельзя выгружать на диск (по умолчанию). В следующем примере для кодовых сегментов модуля устанавливаются значения, используемые по умолчанию, поэтому они не загружаются в память пока к ним нет обращений и им присущи привилегии по аппаратному вводу-выводу:

CODE LOADONCALL IOPL

Оператор DATA

Оператор DATA определяет значения по умолчанию для атрибутов сегментов данных в данном приложении или библиотеке.

**DATA [атрибут1] [атрибут 2] ... [атрибут n]**

атрибут                      Каждый указанный атрибут должен соответствовать одному из следующих полей-атрибутов. Каждое поле можно указать только один раз, а их последовательность не имеет значения. Ниже представлены поля-атрибуты вместе с их допустимыми значениями. В каждом случае значение, используемое по умолчанию, приведено последним.

**Поле                      альтернативные значения**

load	PRELOAD		LOADONCALL
readonly	READONLY		READWRITE
instance	NONE		SINGLE                 MULTIPLE
shared	SHARED		NONSHARED
moveable	MOVEABLE		FIXED
discard	DISCARDABLE		NONDISCARDABLE

Поле load определяет, когда загружается сегмент. В этом поле указывается одно из следующих ключевых слов:

PRELOAD	Сегмент загружается автоматически, в начале выполнения программы, и, при использовании системы оверлеев TopSpeed никогда не выгружается.
LOADONCALL	Пока к сегменту нет обращения он не загружается в память (по

	умолчанию).
READONLY	Поле readonly определяет права доступа к сегменту данных. В нем указывается одно из следующих ключевых слов:
READWRITE	Сегмент данных можно только читать
	Можно и читать и писать в сегмент данных (по умолчанию).
	Поле instance влияет на атрибуты совместного использования автоматического сегмента DATA, который является физическим сегментом, представленным групповым именем DGROUP. Эта сегментная группа составляет физический сегмент, содержащий локальный стек и кучу для всего приложения. Поле instance может содержать одно из следующих ключевых слов:
NONE	Не создается автоматического сегмента DATA.
SINGLE	Создается единственный автоматический сегмент DATA для всех экземпляров модуля. В этом случае модуль, как говорится, имеет “сольные” данные. Это ключевое слово по умолчанию используется для динамических библиотек.
MULTIPLE	Для каждого экземпляра модуля создается свой автоматический сегмент DATA. В этом случае модуль, как говорится, имеет свой экземпляр данных. Это ключевое слово по умолчанию используется для приложений.
	Поле shared определяет, могут ли все экземпляры программы разделять сегмент данных READWRITE DATA.
SHARED	Загружается одна копия сегмента данных и она будет использоваться всеми процессами, обращающимися к данному модулю.
NONSHARED	Сегмент нельзя использовать совместно и для каждого процесса должна загружаться отдельная копия (по умолчанию).
	Поле moveable определяет, можно ли перемещать сегмент в памяти. Это поле обеспечивает использование сегмента в real-режиме Windows. Поле moveable может содержать одно из следующих ключевых слов:
MOVABLE	Данный сегмент данных можно перемещать в памяти.
FIXED	Данный кодовый сегмент нельзя перемещать в памяти (по умолчанию).
	Необязательное поле discard определяет может ли операционная система выгружать данный сегмент на диск, когда в нем нет необходимости. Этот атрибут обеспечивает использование сегмента в real-режиме Windows. Поле discard может содержать одно из следующих ключевых слов:
DISCARDABLE	Сегмент данных можно выгружать на диск.
NONDISCARDABLE	Сегмент данных нельзя выгружать на диск (по умолчанию).

**Замечание:** Нужно быть внимательным и не задавать противоречивых

## атрибутов.

В следующем примере определяется значения по умолчанию для атрибутов сегментов данных, так что они загружаются только при обращении к ним, не могут использоваться более чем одной копией программы:

```
DATA LOADONCALL NONSHARED
```

По умолчанию, сегмент DATA можно читать и записывать в него, а для каждого экземпляра модуля делается копия автоматического сегмента данных.

## Оператор SEGMENTS

Оператор SEGMENTS определяет значения по умолчанию атрибутов для одного или более отдельных сегментов в приложении или посегментно скомпонованной библиотеке. Атрибуты, указанные этим оператором, перекрывают атрибуты, установленные для кодового сегмента или сегмента данных.

## SEGMENTS

### определения\_сегмента

Ключевое слово SEGMENTS начинает определения сегментов. За ним могут идти одно или более определений сегментов, каждое на отдельной строке. Синтаксис определения сегмента следующий:

```
segname [CLASS 'classname'] [attribute1] [attribute2] ... [attributen]
```

segname	Начинает определение сегмента. Может быть указано в кавычках необязательное имя сегмента. Кавычки требуются, если имя сегмента конфликтует с ключевым словом, используемым при определении модуля, как например CODE или DATA.
CLASS	Указывает класс сегмента. Имя класса обязательно заключается в одинарные кавычки. Если класс не указан, то подразумевается CODE.
attribute	Должен соответствовать одному из полей-атрибутов, описанных выше для операторов CODE и DATA. Каждое поле может указываться не более одного раза, а прыжок не имеет значения.

В следующем примере задаются сегменты с именами pr1\_TEXT, pr2\_TEXT и pr3\_TEXT. Каждому сегменту присваиваются различные атрибуты.

```
SEGMENTS
```

```
    pr1_TEXT IOPL
```

```
    pr2_TEXT EXECUTEONLY PRELOAD
```

```
    pr3_TEXT LOADONCALL READONLY
```

### **Оператор STACKSIZE**

Оператор STACKSIZE определяет размер в байтах для стека приложения.

#### **STACKSIZE число**

число	Задаёт размер стека в байтах. Число должно быть целым и по умолчанию трактуется как десятичное, однако можно использовать нотацию принятую в С, чтобы задать шестнадцатеричное или восьмеричное число. Пример: STACKSIZE 4096
-------	---

### **Оператор HEAPSIZE**

Оператор HEAPSIZE определяет в байтах размер локальной кучи приложения. Эта величина влияет на размер автоматического сегмента данных.

#### **HEAPSIZE число**

число	Задаёт размер кучи в байтах. Число должно быть целым и по умолчанию трактуется как десятичное, однако можно использовать нотацию принятую в С, чтобы задать шестнадцатеричное или восьмеричное число. Пример: HEAPSIZE 4000
-------	---

### **Оператор EXPORTS**

Оператор EXPORTS определяет имена и атрибуты функций, экспортируемых в другие модули, и функций, выполняемых с привилегиями по вводу-выводу. Термин “экспорт” означает процесс обеспечения доступа к данной функции из других модулей. По умолчанию, во время выполнения функции невидимы для других модулей.

#### **EXPORTS**

##### **определения\_экспорта**

Ключевое слово EXPORTS отмечает начало определений экспорта. За ним могут идти до 3072-х определений, каждое на отдельной строке. Для каждой динамически подключаемой подпрограммы, которую вы хотите сделать доступной из других модулей, следует указать определение экспорта. Синтаксис определений экспорта следующий:

**entryname [pwords] @number | ? [NODATA]**

entryname	Определяет имя функции, под которым она известна в других модулях.
pwords	Задаёт общий размер параметров функции, измеренный в словах (общее число байт, деленное на 2). Это поле обязательно только, если функция выполняется с привилегиями ввода-вывода. При обращении к функции с привилегиями ввода-вывода OS/2 использует это поле для определения того, сколько слов копировать из стека вызвавшей процедуры в стек функций с привилегиями ввода-вывода.
@number   ?	Определяет порядковый номер функции в таблице определения модуля. Перед номером может идти символ @ или за номером может следовать знак вопроса. Номер должен быть порядковым.
NODATA	Приводится для использования в real-режиме Windows (необязательный элемент).

Операторы EXPORTS много значат для функций в библиотеке DLL, функций с привилегиями ввода-вывода и “call back” функций в программах для Windows.

Пример:

EXPORTS

Func1 @?

Func2 @?

CharTest @?

### **Оператор EXETYPE**

Оператор EXETYPE указывает, в какой операционной системе будет выполняться приложение или библиотека. Этот оператор необязательный и обеспечивает дополнительную степень защиты от запуска программы в несоответствующей ей операционной системе.

#### **EXETYPE [WINDOWS | DOS4]**

Действие оператора EXETYPE состоит в том, что просто устанавливаются в заголовке биты, которые идентифицируют тип операционной системы. Загрузчики операционной системы могут проверять, а могут и не проверять эти биты. При написании программ для Microsoft Windows должно быть указано EXETYPE WINDOWS

## **32-Bit**

### **Синтаксис файла определения модуля**

Файл определения модуля содержит один или несколько операторов. Каждый оператор определяет атрибут исполняемого файла, такой как имя модуля, атрибуты программных сегментов и число и имена экспортируемых символов. Ниже перечислены эти операторы и определяемые ими атрибуты:

Оператор	Атрибут
NAME	Имя приложения
LIBRARY	Имя dll-библиотеки
HEAP_COMMIT	Величина зафиксированной кучи
HEAP_RESERVE	Величина зарезервированной кучи
STACK_COMMIT	Величина зафиксированного стека
STACK_RESERVE	Величина зарезервированного стека
IMAGE_BASE	Module base memory location
DEBUG	Включение отладочных данных
LINENUMBERS	Включение номеров строк
SECTION_ALIGNMENT	Кратно только 4096
FILE_ALIGNMENT	Кратно только 512
EXPORTS	Определяет экспортируемые функции

Использование этих операторов управляется следующими правилами:

- Если используется оператор NAME или LIBRARY, то он должен идти перед остальными операторами в файле определения модуля.
- В файл определения модуля можно включать построчный комментарий, начиная строку точкой с запятой (;). Утилиты такие строки комментария игнорируют.
- Ключевые слова определения модуля, такие как NAME, LIBRARY и SEGMENTS должны быть написаны заглавными буквами.
- Оператор EXPORTS должен идти последним.

### **Пример файла определения модуля**

Следующий пример содержит описание модуля для динамической библиотеки:

```
LIBRARY MyDLL
```

```
; Sample export file
```

```
EXPORTS
```

```
    Func1  @1
```

```
    Var1   @2
```

```
    Func2  @3
```

```
    Func3  @4
```

```
    Func4  @5
```

### **Оператор NAME**

Оператор NAME идентифицирует файл как исполняемое приложение (в отличие от DLL) и дополнительно определяет имя и тип приложения.

**NAME [имя\_приложения] [тип\_приложения]**

имя_приложения	Если задан параметр имя_приложения, то его значение становится именем, под которым приложение известно операционной системе. Если этот параметр не задан, то для этой цели служит имя исполняемого файла без расширения.
тип_приложения	Используется для управления поведением программы в среде Windows. Эта информация хранится в заголовке исполняемого файла. Это параметр может иметь одно из следующих значений:
WINDOWAPI	Приложение использует Интерфейс прикладного программирования (API), обеспечиваемый Windows, и должно выполняться в этой среде.
GUI	То же самое, что и WINDOWAPI.
CUI	Данная программа использует текстовый интерфейс подобный интерфейсу DOS.

Если в файле определения модуля указан оператор NAME, то нельзя задавать оператор LIBRARY.

Если в файле определения модуля не указан ни оператор NAME, ни LIBRARY, то подразумевается использование оператора NAME.

В следующем примере определяемому приложению присваивается имя wdemo:  
NAME wdemo WINDOWAPI

**Оператор LIBRARY**

Оператор LIBRARY идентифицирует файл как динамическую библиотеку. Кроме того, могут быть указаны имя библиотеки и тип требуемой инициализации библиотечного модуля.

**LIBRARY [имя\_библиотеки][инициализация]**

имя_библиотеки	Если задан параметр имя_библиотеки, то его значение становится именем, под которым эта библиотека известна операционной системе. Это может быть любое допустимое имя файла. Если этот параметр не задан, то именем библиотеки становится имя исполняемого файла без расширения.
инициализация	Поле инициализация не является обязательным и может иметь одно из двух перечисленных выше значений. Если не задано ничего, то по умолчанию инициализация INITINSTANCE.
INITGLOBAL	Подпрограмма инициализации вызывается только тогда, когда библиотечный модуль первоначально загружается в память.
INITINSTANCE	Подпрограмма инициализации вызывается каждый раз, когда к



библиотечному модулю обращается новый процесс.

Если в файле определения модуля указан оператор `LIBRARY`, то нельзя задавать оператор `NAME`.

В следующем примере определяемой `dll`-библиотеке присваивается имя `mydll` и указывается, что инициализация выполняется каждый раз, когда к библиотеке `myDLL` обращается новый процесс.

```
LIBRARY myDLL INITINSTANCE
```

### **Оператор HEAP\_COMMIT**

Задаёт подтвержденное количество памяти в куче. Этот оператор не генерируется средой `Clarion` для `Windows`, но его можно указать вручную, редактируя файл `.EXP` и выполняя компоновщик отдельно. Синтаксис оператора `HEAP_COMMIT` следующий:

**HEAP\_COMMIT число**

### **Оператор STACK\_COMMIT**

Задаёт подтвержденное количество памяти в стеке. Этот оператор не генерируется средой `Clarion` для `Windows`, но его можно указать вручную, редактируя файл `.EXP` и выполняя компоновщик отдельно. Синтаксис оператора `STACK_COMMIT` следующий:

**STACK\_COMMIT число**

### **Оператор HEAP\_RESEERVE**

Задаёт зарезервированное количество памяти в куче. Этот оператор не генерируется средой `Clarion` для `Windows`, но его можно указать вручную, редактируя файл `.EXP` и выполняя компоновщик отдельно. Синтаксис оператора `HEAP_RESEERVE` следующий:

**HEAP\_RESERVE число**

### **Оператор STACK\_RESERVE**

Задаёт зарезервированное количество памяти в стеке. Этот оператор не генерируется средой `Clarion` для `Windows`, но его можно указать вручную, редактируя файл `.EXP` и выполняя компоновщик отдельно. Синтаксис оператора `STACK_RESERVE` следующий:

**STACK\_RESERVE число**

### **Оператор IMAGE\_BASE**

Задаёт расположение модуля в памяти. Этот оператор не генерируется средой Clarion для Windows, но его можно указать вручную, редактируя файл .EXP и выполняя компоновщик отдельно. Синтаксис оператора IMAGE\_BASE следующий:

**IMAGE\_BASE**

### **Оператор DEBUG**

Указывает, что включается отладочная информация для отладчика TopSpeed. Этот оператор не генерируется средой Clarion для Windows, но его можно указать вручную, редактируя файл .EXP и выполняя компоновщик отдельно. Синтаксис оператора DEBUG следующий:

**DEBUG**

### **Оператор LINENUMBERS**

Указывает, что включается нумерация строк для отладчика CodeView. Этот оператор не генерируется средой Clarion для Windows, но его можно указать вручную, редактируя файл .EXP и выполняя компоновщик отдельно. Синтаксис оператора LINENUMBERS следующий:

**LINENUMBERS**

### **Оператор SECTION\_ALIGNMENT**

Задаёт, что кратность адреса выравнивания секций должна быть 4096. Этот оператор не генерируется средой Clarion для Windows, но его можно указать вручную, редактируя файл .EXP и выполняя компоновщик отдельно. Синтаксис этого оператора следующий:

**SECTION\_ALIGNMENT**

### **Оператор FILE\_ALIGNMENT**

Задаёт, что кратность адреса выравнивания файла должна быть 512. Этот оператор не генерируется средой Clarion для Windows, но его можно указать вручную, редактируя файл .EXP и выполняя компоновщик отдельно. Синтаксис этого оператора следующий:

**FILE\_ALIGNMENT**

## **Оператор EXPORTS**

Оператор EXPORTS определяет имена и атрибуты функций, экспортируемых в другие модули, и функций, которые выполняются с привилегиями ввода-вывода. Термин “экспорт” означает процесс обеспечения доступа к данной функции из других модулей. По умолчанию, во время выполнения функции невидимы для других модулей.

### **EXPORTS**

#### **определения\_экспорта**

Ключевое слово EXPORTS отмечает начало определений экспорта. За ним могут идти до 3072-х определений, каждое на отдельной строке. Для каждой динамически подключаемой подпрограммы, которую вы хотите сделать доступной из других модулей, следует указать определение экспорта. Синтаксис определений экспорта следующий:

**entryname [pwords] @number | ? [NODATA]**

entryname	Определяет имя функции, под которым она известна в других модулях.
pwords	Задаёт общий размер параметров функции, измеренный в словах (общее число байт, деленное на 2). Это поле обязательно только, если функция выполняется с привилегиями ввода-вывода. При обращении к функции с привилегиями ввода-вывода OS/2 использует это поле для определения того, сколько слов копировать из стека вызвавшей процедуры в стек функций с привилегиями ввода-вывода.
@number   ?	Определяет порядковый номер функции в таблице определения модуля. Перед номером может идти символ @ или за номером может следовать знак вопроса. Номер должен быть порядковым.
NODATA	Приводится для использования в real-режиме Windows (необязательный элемент).

Операторы EXPORTS много значат для функций в библиотеке DLL, функций с привилегиями ввода-вывода и “call back” функций в программах для Windows.

Пример:

EXPORTS

Func1	@?
Func2	@?
CharTest	@?



## **Программирование на нескольких языках**

### **Предисловие**

У TopSpeed есть и 16-ти битные и 32-х битные (по мере реализации) компиляторы языков C++ и Modula-2, которые могут интегрироваться в среду Clarion для Windows. Дополнительно к ним, есть 16-ти битные компиляторы языков С и Паскаль. 16-ти битные компиляторы генерируют объектный код, ориентированный на традиционные среды Windows (3.1 и 3.11), тогда как 32-х битные компиляторы генерируют объектный код, предназначенный для выполнения в среде Windows 95 или Windows NT. Для того, чтобы создавать 32-х разрядные приложения, вы должны запускать среду разработки в 32-битовой операционной системе.

Clarion для Windows также обладает возможностью генерировать объектный код, который конкурирует со многими компиляторами С. Более того, вы можете усилить ваше приложение любыми нужными вам низкоуровневыми функциями. Вышеупомянутые компиляторы языков 3-го поколения позволяют разработчику включить исходные модули, написанные на них, непосредственно в кларионовский проект, обеспечивая беспримечную функциональность и разнообразие. Эта смесь языка скоростной разработки приложений 4-го поколения и традиционных компиляторов, делает Clarion для Windows исключительным инструментом разработки.

Зачем вообще использовать текст на С, C++, Pascal или Modula-2 в кларионовском приложении? Потому что имеются библиотеки (статистические, финансовые, графические, коммуникационные и так далее и так далее), которые могут значительно сократить время разработки приложения, в котором требуются такие возможности. Многие из этих библиотек написаны на С, доступны также мощные библиотеки на C++, Pascal и Modula-2. Clarion позволяет вам использовать эти библиотеки в их “естественной” форме, “не изобретая велосипеда”.

**При изложении этой главы, мы предполагаем, что вы обладаете хорошими знаниями в вопросах Среды разработки Clarion, языка Clarion и языков 3-го поколения.** В примерах программ предполагается, что вы используете компилятор TopSpeed (при этом рассматриваются вопросы использования других компиляторов). Поскольку Clarion для Windows использует технологии генерации кода компоновки фирмы TopSpeed, легче вставить в кларионовское приложение код, создаваемый компиляторами TopSpeed. Можно встраивать и код, производимый компиляторами других фирм, однако, при этом нужна некоторая осторожность и ясное понимание работы обоих компиляторов. В общем случае, (по причинам, изложенным далее в этой главе) нельзя непосредственно вставить код, написанный на C++ для других, не TopSpeed, компиляторов.

## Интеграция компиляторов

При наличии установленных компиляторов 3-го поколения фирмы TopSpeed Среда разработки Clarion для Windows выполняет все действия, необходимые для вызова соответствующего компилятора для каждого исходного модуля в приложении. Нельзя смешивать разные языки в одном исходном модуле, однако, приложение может содержать любое количество исходных модулей, написанных на любых языках 3-го поколения или на языке Clarion.

На основании расширения имени исходного модуля Среда разработки во время компиляции вызывает соответствующий компилятор, как в следующем примере:

### Расширение исходного файла

.CLW  
.CPP или .C  
.CPP  
.C  
.MOD  
.PAS

### Вызываемый компилятор

Clarion (16 и 32-бит)  
C++ (32-бит)  
C++ (16-бит)  
C (16-бит)  
Modula-2 (16 и 32-бит)  
Pascal (16-бит только)

Исходные файлы с любыми другими расширениями будут вызывать появление во время компиляции сообщения 'Unknown compiler for ...'.

Среда разработки будет, кроме того, обеспечивать, чтобы все модули компоновались правильно и, чтобы компоновщику TopSpeed SmartLinker передавалась вся необходимая информация.

## Включение модулей на языках 3-го поколения в проект на языке Clarion

---

### Использование Генератора приложений

1. Создать "включаемый файл", содержащий прототипы функций для компилятора Clarion.

Если вы намерены обращаться к вашим функциям в тексте на Clarion, вы ДОЛЖНЫ представить их прототипы (см раздел Прототипы процедур и функций в Описании языка). Включаемый файл должен содержать прототипы для всех функций на языках 3-го поколения, к которым есть обращения в коде на языке Clarion. Каждый модуль текста на языке 3-го поколения должен иметь свой включаемый файл.

На шаге 3 рассматриваемого процесса включаемый файл без каких-либо изменений вставляется в генерируемый на языке Clarion текст. Сгенерированный код на языке Clarion для включаемого файла выглядит в глобальной структуре Map приблизительно так:

```
MODULE('module name')  
    INCLUDE(' YourInc.Inc ')  
END
```

Генератор приложений создает предложения MODULE и END. Ошибка в прототипе ваших функций почти определенно приведет к ошибке General Protection Fault во время компиляции.

2. В текстовом редакторе напишите текст функции на языке 3-го поколения. Убедитесь что вы сохранили текст в файле с расширением, которое компилятор может распознать (.C, .CPP, .MOD или .PAS).
3. Добавьте модуль в приложение как External Source Module. Выберите в главном меню пункт Application д Insert Module. Затем в окне Select Module Type выберите класс External Source. Введите имя модуля в поле Name, затем введите имя включаемого файла в поле Map Include File диалоговом окне Module Properties.
4. Откомпилируйте и выполните приложение.

#### **В приложении, написанном вручную:**

1. Сделайте прототип функции для компилятора Clarion.

Если вы намерены обращаться к вашим функциям в тексте на Clarion, вы ДОЛЖНЫ представить их прототипы (см раздел Прототипы процедур и функций в Описании языка). Эти прототипы должны содержаться в глобальной структуре MAP.

Для каждого модуля на языке 3-го поколения его прототипы должны содержаться в отдельной структуре MODULE, как в примере:

```
MODULE('module name')  
    MyFunc(*CSTRING),CSTRING,RAW,PASCAL,NAME('_MyFunc')  
END
```

Полные структуры MODULE для всех модулей на других языках должны включаться в структуру MAP. Ошибка в прототипе ваших функций почти определенно приведет к ошибке General Protection Fault во время компиляции.

2. В текстовом редакторе напишите текст функции на языке 3-го поколения, сохранив его в файле с расширением, которое компилятор может распознать (.C, .CPP, .MOD или .PAS).
3. Добавьте модуль в проект как External Source File. В главном меню выберите пункт Project д Edit. Выделите External Source Files, а затем нажмите кнопку Add File... В появившемся стандартном окне открытия файла

выберите модуль на языке 3-го поколения.

4. Откомпилируйте и выполните приложение.

## **Разрешение типов данных**

В языке Clarion определяются типы данных: BYTE, SHORT, USHORT, LONG, ULONG, SREAL, REAL и STRING, которым довольно легко находится соответствие в C, C++, Pascal и Modula-2. Кроме этих типов в Clarion определяются типы данных DATE и TIME, и структуры GROUP, которые можно в каждом языке отобразить структурами. Типы данных CSTRING и PSTRING введены в язык Clarion специально для упрощения взаимодействия с внешними функциями на C и Паскале.

Типы данных DECIMAL, PDECIMAL, BFLOAT4 и BFLOAT8 не рассматриваем, поскольку они очень непохожи на типы переменных, обычно используемые в программах на C, C++, Pascal и Modula-2. Если понадобится передать в функцию на C, C++, Pascal и Modula-2 данные какого-либо из этих типов просто присвойте значение этой переменной типа REAL или SREAL и передайте в качестве параметра ее (при присвоении в Clarion автоматически выполняется преобразование типа данных).

В таблице ниже приведены соответствия типов параметров, поддерживаемых компиляторами Clarion, C++ и Modula-2. Как в ней показано, некоторые параметры для корректной передачи требуют применения дополнительных прагм. Различие типов “со знаком” и “без знака” (SIGNED и UNSIGNED) в Clarion равнозначно изменению типа с SHORT на USHORT и с LONG на ULONG в зависимости от 16-ти или 32-х битной компиляции.

### **Clarion**

BYTE  
BYTE  
\*BYTE  
USHORT  
\*USHORT  
SHORT  
\*SHORT  
LONG  
\*LONG  
ULONG  
\*ULONG  
SREAL  
\*SREAL  
REAL

### **C++**

unsigned char  
unsigned char  
unsigned char \*  
unsigned short  
unsigned short \*  
short  
short \*  
long  
long \*  
unsigned long  
unsigned long \*  
float  
float \*  
double

### **Modula-2**

BOOLEAN  
SHORTCARD  
var SHORTCARD  
CARDINAL  
var CARDINAL  
INTEGER  
var INTEGER  
LONGINT  
var LONGINT  
LONGCARD  
var LONGCARD  
REAL  
var REAL  
LONGREAL



*REAL	double *	var LONGREAL
STRING	не передаются значением	не передаются значением
*STRING	unsigned int, char *	CARDINAL, ARRAY OF CHAR
*STRING (with RAW)	char[]	var ARRAY OF CHAR
*CSTRING (with RAW)	char[] or char *	var ARRAY OF CHAR
*PSTRING	char[] or Char *	ARRAY OF CHAR
GROUP	struct	var record type
*GROUP (with RAW)	struct *	var record type
*?	void far*	FarADDRESS
UNSIGNED	unsigned int	
SIGNED	int	

Переменные типа STRING обычно передаются двумя параметрами: первый UNSIGNED, содержит длину буфера, второй - адрес данных. Строки типа CSTRING передаются точно таким же образом как и PSTRING: в первом байте буфера содержится число символов в строке. В прототипах на языке Clarion для того, чтобы указать передачу во внешнюю функцию на языках 3-го поколения только адреса строки, можно использовать атрибут RAW (прототипы функций на языке Clarion не нуждаются в атрибуте RAW).

### Соответствие типов данных с С и С++

В программном коде на языке С можно использовать следующие определения эквивалентности типов. Предложения typedef должны находиться в файле заголовка .H, на который есть ссылка в коде на С или С++. Для предотвращения конфликта имен с библиотеками сторонних фирм используется префикс CLA.

typedef unsigned char	CLABYTE;
typedef short	CLASHORT;
typedef unsigned short	CLAUSHORT;
typedef long	CLALONG;
typedef unsigned long	CLAULONG;
typedef float	CLASREAL;
typedef double	CLAREAL;

Типы данных языка Clarion DATE и TIME можно передавать в функции С как CLALONG, для представления элементов данных из значения CLALONG можно использовать объединения CLADATE и CLATIME.

```
typedef union {
    CLALONG
    struct {
        n;
```

```

        CLABYTE
        CLABYTE
        CLAUSHORT
    }
}    CLADATE;
typedef union {
    CLALONG
    struct {
        CLABYTE
        CLABYTE
        CLABYTE
        CLABYTE
    }
} CLATIME;

```

```

ucDay;
ucMonth;
usYear;
s;

n;

ucHund;
ucSecond;
ucMinute;
ucHour;
s;

```

Из-за того, что в Clarion передает строки посредством двух параметров (длина и адрес строки), для внутренних нужд можно использовать структуру CLASTRING, но нельзя ее использовать для приема параметров из Clarion.

```

    typedef struct {
        char *pucString;
        CLAUSHORT usLen
    }    CLASTRING;

```

В языке Clarion переменные типа STRING не оканчиваются нуль-символом, до конца отведенного ей буфера строка заполняется пробелами. Эти пробелы в конце переменной можно удалить с помощью встроенной функции CLIP. В следующем примере объявляется строка длиной 20 символов, ей присваивается некоторое значение, и оно передается как параметр в функцию на языке C или C++.

```

StringVar  STRING(20)
CODE
StringVar = 'Hello World...'
C_Write_Function(StringVar)

```

Функция на C или C++ должна определяться так:

```

extern void C_Write_Function(CLAUSHORT usLen, char *bData)
{
    CLAUSHORT usNdx = 0;

    while (usNdx < usLen)
#ifdef __cplusplus
        cout << bData[usNdx++];
#else

```

```

        putchar(bData[usNdx++]);
    #endif
}

```

В приведенном выше примере переменная `usLen` получит значение 20, а переменная `bData` будет содержать в конце себя пробелы. Они будут выводиться на экран функций `C_Write_Function()`. Многие функции на С ожидают, что им будет передано оканчивающееся нулевым кодом строка. Для разрешения этого вопроса в Clarion введен тип данных `CSTRING`. При присвоении такой строке значения, после него автоматически заносится код - ноль. Это делает возможным работать с данными для уже откомпилированным (или библиотечным функциям). на С.

Для хранения связанных данных можно объявить оператором `GROUP` группу. Группа приблизительно соответствует структуре `struct` в С или С++. При передаче в качестве параметра в процедуру группа обычно передается как три параметра: первый типа `UNSIGNED` содержит размер группы; второй, адрес структуры; третий - адрес буфера, содержащего дескриптор типа для этой группы. Содержание дескриптора типа рассматривать здесь не будем, он может меняться от версии к версии Clarion. Группы могут быть вложенными, а другие группы могут подразумеваться имеющими такую же структуру как ранее объявленные группы. В Clarion есть несколько форм объявления групп:

```

Struct1      GROUP      !Struct1 определяется такой как группа
ul1          ULONG      !содержащая две переменные типа ULONG
ul2          ULONG
                                END

```

В этой форме определения резервируется память для структуры `Struct1`, и она, эта форма эквивалентна следующему определению в С:

```

struct {
    CLAULONG  ul1;
    CLAULONG  ul2;
} Struct1;

```

В следующем примере в объявлении объявляется `Struct2` подобная определению `Struct1`, однако память не резервируется. На практике не нужно определять экземпляры объявленной структуры `Struct2`.

```

Struct2      GROUP,TYPE  ! Struct2 объявлена как группа
ul3          ULONG      ! содержащая два поля ULONG
ul4          ULONG
                                END

```

Соответствующее определение в С:

```
typedef struct {
    CLAULONG ul3;
    CLAULONG ul4;
} Struct2
```

В следующем примере определение структуры Struct3 и Struct4 объявляет их такими же как Struct2, т.е. такой же внутренней структуры. Для того, чтобы различать члены структур Struct3 и Struct4 от таких же членов структуры Struct2 нужно использовать префиксы S3 и S4. Struct3 и Struct4 определяют экземпляры структуры Struct2, которые нет необходимости нигде определять. В обоих случаях резервируется память.

```
Struct3      LIKE(Struct2)
Struct4      LIKE(Struct2)
```

Соответствующие определения в С:

```
typedef Struct2  Struct3;
typedef Struct2  Struct4;

Struct3 S3;
Struct4 S4;
```

Объявления GROUP в языке Clarion могут быть вложенными, например:

```
Struct5      GROUP,TYPE      ! Struct5 определяется как группа
Struct6      GROUP          ! содержащая другую группу
ul5  ULONG
ul6  ULONG

                                END
                                END
```

Соответствующие определения в С:

```
typedef struct {
    struct {
        CLAULONG ul5;
        CLAULONG ul6;
    } Struct6;
} Struct5;
```

## Соответствие типов данных с Modula-2

При сопряжении с кодом на Modula-2 используются следующие соответствия типов. Эти предложения должны помещаться в модуль определений Modula-2, на который ссылается программный код на этом языке. Их следует использовать для описания параметров и возвращаемого значения в процедурах, к которым будет обращение из программного кода на языке Clarion для Windows.

```
CONST
    BYTE          ::=    BYTE;
    SHORT         ::=    INTEGER (16-bit);
    USHORT        ::=    CARDINAL (16-bit);
    LONG          ::=    LONGINT;
    ULONG         ::=    LONGCARD;
    SREAL         ::=    REAL;
    REAL          ::=    LONGREAL;
```

Кларионовские типы DATE и TIME можно передавать в функции на Modula-2 как LONG. Для выделения отдельных элементов времени или даты из числа типа LONG можно использовать записи (RECORD) DATE и TIME языка Modula-2.

```
DATE = RECORD
    CASE : BOOLEAN OF
    | TRUE:
        1                                     : LONG;
    ELSE
        ucDay          : BYTE;
        ucMonth        : BYTE;
        usYear         : SHORT;
    END
END;
TIME = RECORD
    CASE : BOOLEAN OF
    | TRUE:
        1                                     : LONG;
    ELSE
        ucHund         : BYTE;
        ucSecond       : BYTE;
        ucMinute       : BYTE;
        ucHour         : BYTE;
    END
END;
```

Кларионовские строки передаются таким же образом как в Modula-2 принято передавать параметры ARRAY OF CHAR с неопределенной длиной (с установленным значением прагмы call(o\_a\_copy=>off), т.е. передаются длина и адрес строки).

В следующем примере кода объявляется строка из 20-ти символов, присваиваются некие данные и передаются в процедуру на Modula-2.

```

MAP
  MODULE('M2_Code')
    M2_Write_Proc(*STRING), NAME('M2_Code$M2_Write_Proc')
  END
END
StringVar  STRING(20)
CODE
StringVar = 'Hello World...'
M2_Write_Proc(StringVar)

```

Процедура на Modula-2 должна быть описана так:

```

DEFINITION MODULE M2_Code;
(*# save, call(o_a_copy=>off) *)
PROCEDURE M2_Write_Proc(StringVar: ARRAY OF CHAR);
(*# restore *)
END M2_Code.

```

Заметим, что строки в Clarion не оканчиваются нуль-символом, до конца отведенного буфера строка заполняется пробелами. В приведенном выше примере переменная StringVar будет дополнена пробелами до 20-ти символов. В переменные типа CSTRING при присвоении им данных автоматически заносится нуль-символ как признак конца данных. Это обеспечивает возможность работы с этими данными в подпрограммах на языке Modula-2.

Группы в языке Clarion приблизительно соответствуют структурам RECORD в Modula-2. Для групп в Clarion существуют несколько форм объявления. Следующая форма соответствует приведенному выше объявлению типа DATE в Modula-2:

```

DateType  GROUP
n          LONG
d          GROUP,OVER(n)
ucDay     BYTE
ucMonth   BYTE
usYear    SHORT
          END
        END

```

Для того, чтобы `p` и `d` занимали одну и ту же область памяти, используется атрибут `OVER`, общий размер этой группы равен размеру переменной `p`. При передачи в качестве параметра группа передается с помощью трех параметров: первый, `UNSIGNED` передает размер группы, второй - адрес группы, а третий адрес буфера, содержащего описатель типа для данной группы. Содержание дескриптора типа рассматривать здесь не будем, он может меняться от версии к версии `Clarion`. Для того, чтобы указать компилятору, что нужно передавать только адрес группы, можно использовать в “кларионовском” прототипе процедуры на языке `Modula-2` атрибут `RAW`, в противном случае нужно определять в `Modula-2` процедуру как принимающую два дополнительных параметра:

```
MAP
  MODULE('M2_Code')
    M2_Proc1(*GROUP)
    M2_Proc2(*GROUP), RAW
  END
END
```

Соответствующий модуль определения на `Modula-2` содержит:

```
DEFINITION MODULE M2_Code;
  TYPE
    GROUP = RECORD
      (* Members *)
    END;
  PROCEDURE M2_Proc1(Len: USHORT; VAR Data: GROUP; TypeDesc:
ADDRESS);
  PROCEDURE M2_Proc2(VAR Data: GROUP);
END M2_Code.
```

## Соответствие типов данных с Pascal

При сопряжении с кодом на Паскале используются следующие соответствия типов. Эти предложения должны помещаться в интерфейсный `unit`, на который ссылается программный код на этом языке. Их следует использовать для описания параметров и возвращаемого значения в процедурах, к которым будет обращение из программного кода на языке `Clarion` для `Windows`.

```
ALIAS
  SHORT      =      INT16;
  USHORT     =      INT16;
  LONG       =      INTEGER;
  ULONG      =      INTEGER;
  SREAL      =      SHORTREAL;
```

Кларионовские типы DATE и TIME можно передавать в функции на Паскале как LONG. Для выделения отдельных элементов времени или даты из числа типа LONG можно использовать записи (RECORD) DATE и TIME языка Паскаль.

```
DATE = RECORD
CASE BOOLEAN OF
TRUE:
    (n                :    LONG);
FALSE:
    (ucDay            :    BYTE;
     ucMonth          :    BYTE;
     usYear           :    SHORT);
END;
```

```
TIME = RECORD
CASE BOOLEAN OF
TRUE:
    (n                :    LONG);
FALSE:
    (ucHund           :    BYTE;
     ucSecond         :    BYTE;
     ucMinute         :    BYTE;
     ucHour           :    BYTE);
END;
```

Из-за того, что в Clarion строки передаются с помощью двух параметров, переменные типа STRING полезны для внутреннего использования в программе на Clarion, но не для передачи как параметры в процедуры на других языках:

```
TYPE
STRING = RECORD
    usLen            :    USHORT;
    pucString        :    ^CHAR;
END;
```

Переменные типа PSTRING передаются в языке Clarion точно также как STRING в Паскале, если установлена прагма call(s\_cору=>off) (передаются длина и адрес строки).

В следующем примере объявляется строка длиной 20 символов, ей присваивается некоторое значение, и она передается как параметр в процедуру на Паскале:

```
MAP
MODULE('Pas_Code')
    Pas_Write_Proc(*PSTRING), NAME('Pas_Code$Pas_Write_Proc')
```



```

        END
    END
StringVar  PSTRING(20)
CODE
StringVar = 'Hello World...'
Pas_Write_Proc(StringVar)

```

В Паскале процедура должна быть определена следующим образом:

```

INTERFACE UNIT Pas_Code;

(*# save, call(s_copy=>off) *)
PROCEDURE Pas_Write_Proc(StringVar: STRING[HIGH]);
(*# restore *)
END.

```

Группы в Clarion приблизительно эквивалентны записям в Паскале. Для групп в Clarion существуют несколько форм объявления. Следующая форма соответствует приведенному выше объявлению типа DATE в Паскале:

```

DateType    GROUP
n            LONG
d            GROUP,OVER(n)
ucDay       BYTE
ucMonth     BYTE
usYear      SHORT
            END
            END

```

Для того, чтобы n и d занимали одну и ту же область памяти, используется атрибут OVER, общий размер этой группы равен размеру переменной n. При передачи в качестве параметра группа передается с помощью трех параметров: первый, USHORT передает размер группы, второй - адрес группы, а третий адрес буфера, содержащего описатель типа для данной группы. Содержание дескриптора типа рассматривать здесь не будем, он может меняться от версии к версии Clarion. Для того, чтобы указать компилятору, что нужно передавать только адрес группы, можно использовать в “клариионовском” прототипе процедуры на языке Паскаль атрибут RAW, в противном случае нужно определять в Паскале процедуру как принимающую два дополнительных параметра:

```

MAP
MODULE('Pas_Code')
    Pas_Proc1(*GROUP)
    Pas_Proc2(*GROUP), RAW
END
END

```

Соответствующий интерфейсный unit в Паскале:

```
INTERFACE UNIT Pas_Code;
  TYPE
    GROUP = RECORD
      (* Members *)
    END;
  PROCEDURE Pas_Proc1(Len: USHORT; VAR Data: GROUP; VAR
TypeDesc);
  PROCEDURE Pas_Proc2(VAR Data: GROUP);
END.
```

## ***Прототипирование в Clarion функций на языках 3-го поколения***

Единственное, что нужно сделать для того, чтобы использовать программе на языке Clarion любые стандартные библиотечные функции на С, это добавить в структуру MAP приложения “кларионовские” прототипы функций. Прототипом компилятору и компоновщику сообщается, какого типа параметры передаются в функцию на С и какого типа значение возвращается (если возвращается). Синтаксис и атрибуты, требуемые для создания в Clarion прототипа функции или процедуры, обсуждаются в разделе Прототипы процедур и функций в главе 2 Описания языка. Тот же самый синтаксис используется для создания прототипов функций, написанных на языке С.

С созданием прототипов функций, написанных на языке С, связаны четыре момента: соглашения о вызове, соглашения об именах, передача параметров и тип возвращаемого функцией значения.

Соглашения о вызовах для всех стандартных библиотечных функций TopSpeed С те же самые, с передачей параметров в регистрах, которые используются и в Clarion. Поэтому нет необходимости использовать в кларионовских прототипах стандартных библиотечных функций на С атрибуты С или PASCAL.

Используемые компилятором TopSpeed С соглашения об именах - это обычные соглашения, принятые в С. Это значит, что при компиляции к имени функции спереди добавляется знак подчеркивания ( \_ ). Для того, чтобы в тексте на языке Clarion использовать имена без знака подчеркивания спереди, в прототипах обычно используют атрибут NAME. Например, С функции “access” компилятор присваивает имя “\_access”. Поэтому, если вы не хотите в процедурах на языке Clarion обращаться к функции как к “\_access”, в прототипе нужно указать атрибут NAME(‘\_access’).

Каждый передаваемый в функцию на С параметр, должен быть представлен в кларионовском прототипе типом данных передаваемого параметра. В Clarion параметры

передаются “адресом” или “значением”.

Когда параметр передается “значением”, функция получает копию передаваемых данных. Передаваемый параметр представляется в прототипе типом данных параметра. При передаче параметра “адресом” в функцию передается адрес данных в памяти. Такой параметр представляется в прототипе типом данных со звездочкой (\*) спереди. Это соответствует передаче в С функцию указателя на данные.

## Типы данных параметров

Трансляция типа данных параметра является “ключом” к прототипированию функций на С в Clarion. В следующей таблице приводятся типы данных С и соответствующие им типы, указываемые в прототипе в Clarion:

Тип данных С	Тип данных Clarion
---	-----
char	BYTE (вызывает предупреждение компоновщика)
unsigned char	BYTE
int	SHORT
unsigned int	USHORT
short	SHORT
unsigned short	USHORT
long	LONG
unsigned long	ULONG
float	SREAL
double	REAL
unsigned char *	*BYTE
int *	*SHORT
unsigned int *	*USHORT
short *	*SHORT
unsigned short *	*USHORT
long *	*LONG
unsigned long *	*ULONG
float *	*SREAL
double *	*REAL
char *	*CSTRING с атрибутом RAW
struct *	*GROUP с атрибутом RAW

Поскольку в языке Clarion нет типа данных BYTE со знаком, когда прототипируемая функция получает параметр типа char, компоновщик выдаст предупреждение (“несовместимость типов”). Если вы знаете, что С функция ожидает значение со знаком, и правильно устанавливаете битовое значение однобайтового поля, чтобы передать

величину в диапазоне от -128 до 127, то это предупреждение можно безболезненно игнорировать.

Когда функция ожидает передачи ей адреса параметра типа CSTRING или GROUP, то нужно использовать в прототипе атрибут RAW. По умолчанию данные типа STRING, CSTRING, PSTRING и GROUP передаются (внутренне) в другие кларионовские процедуры и функции посредством адреса и длины строки. Поэтому атрибут RAW перекрывает это умолчание.

Если С функция не возвращает никакого значения, то оно подпадает под определение процедуры в Clarion. Если же функция возвращает значение, то она прототипируется так, чтобы указать тип возвращаемого значения, и она подпадает под определение кларионовской функции. Обращение к процедуре в Clarion выглядит как отдельный оператор, тогда как вызов функции может быть частью условия, присвоения или списка параметров.

## Типы возвращаемых значений

Возвращаемые С функциями типы данных почти те же, что и для передаваемых параметров:

<u>Возвращаемый С тип</u>	<u>Возвращаемый Clarion тип</u>
char	BYTE (вызывает предупреждение компоновщика)
unsigned char	BYTE
int	SHORT
unsigned int	USHORT
short	SHORT
unsigned short	USHORT
long	LONG
unsigned long	ULONG
float	SREAL
double	REAL
unsigned char *	*BYTE
int *	*SHORT
unsigned int *	*USHORT
short *	*SHORT
unsigned short *	*USHORT
long *	*LONG
unsigned long *	*ULONG
float *	*SREAL
double *	*REAL
char *	CSTRING (указатель автоматич. Разыменовыв.)
struct *	ULONG (вызывает предупреждение компонов-ка)

Как можно видеть для `char *` в Clarion возвращается `CSTRING` (а не `*CSTRING` как можно было бы ожидать). Это происходит потому, что компилятор Clarion для возвращаемого значения автоматически разыменовывает указатель на данные (как он делает со всеми возвращаемыми типами-указателями).

Заметим, что возвращаемое в Clarion значение для `struct *` - это `ULONG`. Он вызовет появление предупреждения компоновщика “type inconsistency”. Это происходит потому, что в Clarion не используются указатели-адреса, а `ULONG` представляет собой 4-х байтовое целое, которое может служить заменителем для указателя на возвращаемое значение. Предупреждение не имеет значения и можно спокойно не обращать на него внимания. Для того, чтобы получить возвращаемое значение, пожалуй, можно использовать функцию `memstr()`.

## Передаваемые параметры

---

Clarion для Windows предлагает два отдельных способа передачи параметров в процедуру: “передача значением” и “передача адресом”.

“Передача значением” означает, что вызывающий код, передает копию данных в вызываемую функцию или процедуру. Вызванный программный код работает с этими данными, не оказывая влияния на исходную копию данных. Такая передача указывается в прототипах типами данных параметров.

“Передача адресом” означает, что вызывающий код, передает в вызываемую функцию или процедуру адрес данных. При этом вызванный программный код может изменить исходные данные. Такая передача указывается в прототипах звездочкой (\*) перед типами данных параметров.

```
MAP
  MODULE('My_C_Lib')
    Var_Parameter(*USHORT)      ! параметр передаваемый адресом
    Val_Parameter(USHORT)      ! параметр передаваемый значением
  END
END
```

Эти объявления представляют собой интерфейс на Clarion для функций на C, содержащихся в библиотеке `My_C_Lib`. Этот пример эквивалентен объявлениям на C:

```
void Var_Parameter(CLAUSHORT *uspVal);
void Val_Parameter(CLAUSHORT usVal);
```

Кларионовские параметры, “передаваемые адресом”, эквивалентны указателям на

соответствующие типы в С. “Передаваемые значением” параметры в Clarion передаются точно таким же образом, что и параметры-значения в С и С++.

Соответствующее определение модуля в Modula-2 было бы таким:

```
DEFINITION MODULE M2_Code;

  IMPORT Cla;

  PROCEDURE Var_Parameter(VAR us: Cla.USHORT);
  PROCEDURE Val_Parameter(us: Cla.USHORT);
END M2_Code.
```

Соответствующий интерфейсный unit для Паскаля будет таким:

```
INTERFACE UNIT Pas_Code;

  IMPORT Cla;

  PROCEDURE Var_Parameter(VAR us: Cla.USHORT);
  PROCEDURE Val_Parameter(us: Cla.USHORT);
END.
```

“Посредством значения” нельзя передавать из Clarion данные типа STRING и GROUP. По этой причине вы должны передавать их “посредством адреса”.

## **Соглашения о вызовах**

---

Clarion для Windows использует генератор объектного кода фирмы TopSpeed, поэтому в нем используется тот же самый эффективный механизм передачи параметров в регистрах, эксплуатируемый во всех языках TopSpeed. Если компиляторами сторонних фирм используются отличающиеся соглашения о вызовах, то результаты могут быть непредсказуемыми. Обычно приложение во время выполнения падает.

Для того, чтобы использовать объектный код, полученный компиляторами других фирм, нужно быть уверенным что:

1) другой компилятор генерирует коды, используя принятый в Clarion (TopSpeed) метод передачи параметров, или,

2) что Clarion генерирует код, используя принятый в другом компиляторе метод.

Вы также должны убедиться, что никакая из функций не возвращает данные с плавающей точкой. В этом вопросе нет стандарта на совместимость компиляторов. Например, Microsoft

С возвращает значения с плавающей точкой в глобальной переменной, тогда как Borland C - через стек (TopSpeed тоже возвращает их через стек, но гарантии совместимости нет). Поэтому, любые функции, полученные не компилятором TopSpeed, которым нужно обращаться к данным с плавающей точкой и изменять их, должны принимать значения “посредством адреса” и напрямую модифицировать значение - не возвращая значение.

Большинство других компиляторов не обеспечивают совместимых с Clarion соглашений о передаче параметров, зато обеспечивают стандартный механизм передачи по типу C или Паскаля (передачи через стек). Для того, чтобы указать передачу параметров через стек, в Clarion для Windows есть атрибуты C и PASCAL в прототипе функции,

Большинство компиляторов C и C++ (не фирмы TopSpeed) используют соглашения о вызовах, в которых параметры помещаются в стек справа налево (как они указываются в списке параметров). Этот тип соглашения и задает в прототипе функции в Clarion атрибут C. Многие компиляторы C и C++ предлагают паскалевское соглашение о вызовах, по которому параметры помещаются из списка параметров в стек слева направо. Такие же соглашения используются также во многих других языках на персональных компьютерах. Этот тип соглашения задает в прототипе функции в Clarion атрибут PASCAL.

В большинстве случаев атрибуты C и PASCAL используются в сочетании с атрибутом NAME. Это потому, что многие компиляторы предваряют знаком подчеркивания имя функции, в которой используется соглашение о вызове типа C, или записывают большими буквами имя функции, для которой используется паскалевское соглашение о вызове (Clarion для Windows также переводит в верхний регистр имена процедур). Пример:

```
MAP
    MODULE('My_C_Lib')
        StdC_Conv(UNSIGNED, ULONG), C, NAME('_StdC_Conv')
        StdPascal_Conv(UNSIGNED,        ULONG),        PASCAL,
NAME('STDPASCAL_CONV')
    END
END
```

При обращении к процедуре StdC\_Conv параметр типа ULONG помещается в стек первым, а следом за ним помещается параметр типа UNSIGNED. При вызове же функции StdPascal\_Conv наоборот, первым в стек помещается параметр типа UNSIGNED, а за ним - ULONG. Следует быть очень внимательным, нужно чтобы соглашения о вызовах совпадали, иначе программа может повести себя непредсказуемым образом. При организации взаимодействия между подпрограммами, откомпилированными компиляторами TopSpeed, атрибуты C и PASCAL не нужны, поскольку Clarion для Windows использует соглашения о вызовах TopSpeed, в основе которого лежит передача параметров в регистрах.

При написании функций на TopSpeed C, которые будут вызываться из программы на Clarion, для выбора правильного соглашения об образовании имен следует использовать обсуждавшийся макрос CLA\_CONV. Наилучший способ сделать это - объявить интерфейсные функции в отдельном файле заголовке (.H) и применить к этим объявлениям соответствующие соглашения. Функции на C++ должны объявляться с модификатором "Pascal" (также обсуждавшимся ранее). Соглашения по образованию имен в Modula-2 и Pascal лучше всего задавать, используя в прототипе атрибут NAME.

## Реализация соглашения об именах

---

При компоновке кодов, сгенерированных различным программным инструментарием, необходимо убедиться, что используются соответствующие правила образования внешних имен. Если эти соглашения для разных модулей будут различными, компоновщик не сможет разрешить ссылки на имена внутри кода, полученного одним компилятором, и определения этих имен внутри кода, выработанного другим компилятором. В этом случае исполняемый модуль не будет скомпонован.

Многие компиляторы языка C добавляют спереди к именам функций и переменных знак подчеркивания. Атрибут NAME в языке Clarion упрощает задачу организации взаимодействия с этими компиляторами, явно указывая компилятору Clarion какое имя следует присвоить процедуре или функции для компоновки. Такой подход позволяет прямо кодировать в Clarion прототипы, соответствующие соглашениям языка C. Например:

```
MAP
    MODULE('My_C_Lib')
        StdStr_Parm(STRING), NAME('_StdStr_Parm')
    END
END
```

Когда компилятор Clarion встречается процедуру StdStr\_Parm(), в объектном коде он генерирует имя \_StdStr\_Parm. Хотя имена в Clarion не различаются по регистру букв, имена, заданные атрибутом NAME, переносятся в объектный код точно так, как записаны.

Приведенный далее макрос языка C определяет соглашение об образовании имен для Clarion. Этот макрос можно использовать при определении функций организующих интерфейс с Clarion, для того, чтобы компилятор C генерировал имена, соответствующие принятому в Clarion соглашению (имя большими буквами и без знака подчеркивания).

```
#define CLA_CONV    name(prefix=>"", upper_case=>on)
```

Компиляторы C++ дополняют имя функции типом возвращаемого значения и типами данных параметров. И в таком виде имена функций и появляются в объектном коде. Поэтому, компилируемые C++ функции, которые могут вызываться из Clarion, можно



объявить с модификатором ‘extern “Pascal” {...};’, который эквивалентен макросу CLA\_CONV в С, который не влияет на образование имен, выполняемое компилятором С++). Например:

```
extern “Pascal” void Clarion_Callable_Proc(void);
```

Более гибкая форма приведенного выше объявления, позволяющая компилировать его и компилятором С, и компилятором С++, следующая:

```
#ifdef __cplusplus
extern “Pascal” {           /* вкл. соглашения Clarion для С++ */
#else
#pragma save, CLA_CONV      /* вкл. соглашения Clarion для С */
#endif

void Clarion_Callable_Proc(void); /* объявление для С или С++ */

#ifdef __cplusplus
}                             /* восстановим соглашения С++ */
#else
#pragma restore             /* восстановим соглашения С */
#endif
```

Такая форма объявления обычно приводится в заголовочном файле, который должен включаться в интерфейсный код. Она обеспечивает то, что и при компиляции текста на С, и при компиляции компилятором С++ будут использоваться правильные соглашения, и то, что отпадает необходимость использовать атрибут NAME.

Clarion является языком не различающим различие в регистрах букв, компилятор преобразует имена всех процедур в регистр прописных букв. В противоположность ему в Modula-2 и Pascal регистр букв в имена различается, и, кроме того, ко именам всех процедур спереди добавляется имя модуля в виде: MyModule\$MyProcedure. Способ обойти это различие - это указать атрибутом NAME компилятору Clarion полное имя процедуры на языке Modula-2 или Pascal:

```
MAP
  MODULE('M2_Code')
    M2_Proc1(*GROUP), RAW, NAME('M2_Code$M2_Proc2')
  END
  MODULE('Pas_Code')
    Pas_Proc1(*GROUP), RAW, NAME('Pas_Code$Pas_Proc2')
  END
END
```

Соответствующий модуль определений в Modula-2 будет иметь вид:

```
DEFINITION MODULE M2_Code;
```

```

TYPE
    GROUP = RECORD
        (* Members *)
    END;
PROCEDURE M2_Proc1(VAR Data: GROUP);
END M2_Code.

```

Соответствующий интерфейсный unit в Паскале будет иметь вид:

```

INTERFACE UNIT Pas_Code;
TYPE
    GROUP = RECORD
        (* Members *)
    END;
PROCEDURE Pas_Proc1(VAR Data: GROUP);
END.

```

Соглашение об образовании имен для данных в Clarion отличается от принятых для процедур и функций и гораздо сложнее. Поэтому, для любых данных, к которым нужно обращаться из разных языков, для генерации имен, совместимых с языками Modula-2 и Паскаль, следует использовать атрибут NAME(). В Modula-2 и Паскале регистр букв в именах данных различается, а сами имена снабжаются префиксом из имени модуля и символа '@' вот таким образом: MyModule@MyProc.

## Атрибуты EXTERNAL и DLL

---

В языке Clarion атрибут EXTERNAL используется для того, чтобы объявить переменную, которая определена во внешней библиотеке. А атрибут DLL объявляет, что внешняя переменная или функция определена в динамически присоединяемой библиотеке (DLL). Этот атрибут необходим для 32-х разрядных проектов и игнорируется для 16-ти битных.

Эти атрибуты снабжают программы на языке Clarion сведениями о том, как осуществлять доступ к общим данным во внешних библиотеках. Компилятор не будет резервировать память для переменной, объявленной с атрибутом EXTERNAL. Например:

```

typedef struct    {
    unsigned long    ul1;
    unsigned long    ul2;
}    StructType;
#ifdef __cplusplus
extern "C" {    /* Исполыз. соглашения об именах по C, кот. требуют */
#endif    /* в прототипе На Clarion атрибута NAME */

```

```

StructType Str1;    /* Определение Str1      */
StructType Str2;    /* Определение Str2      */
#ifdef __cplusplus
}                  /* Восст. соглашения C++ */
#endif

```

Следующие операторы объявления на языке Clarion - это все, что нужно сделать, чтобы функции Str1 и Str2 стали доступными в программе на Clarion.

```

тип      StructType      GROUP,TYPE      ! объявить определенный пользователем

ul1      ULONG
ul2      ULONG
          END

! объявить Str1 и Str2, которые определяются в С модуле
          Str1    LIKE(StructType),NAME('_Str1'),EXTERNAL
          Str2    LIKE(StructType),NAME('_Str2'),EXTERNAL

```

Для того, чтобы позволить компоновщику использовать С-соглашения об именах при обращении к функциям Str1 и Str2.

## Вопросы программирования

### Использование библиотек классов C++

---

При обращении из Clarion для Windows к коду и данным на C++ существуют некоторые ограничения. C++ это объектно-ориентированный язык и имеет языковые средства для поддержки классов и объектов, полиморфизма, перегрузки операторов и функций, наследования классов. Все эти возможности поддерживаются в Clarion не так как в C++. Это не запрещает нам использовать их положительные стороны при разработке смешанного приложения на Clarion и C++, однако, это определяет характер интерфейсного программного кода.

Clarion для Windows не может напрямую обращаться к классам или объектам некоторого класса в C++. Поэтому, программы на языке Clarion для Windows не имеют прямого доступа к данным и функциям, содержащимся внутри этих классов. Для того, чтобы обращаться к ним, нужно обеспечить С-подобный интерфейс к функциональным возможностям C++. Функция в стиле С может вызываться из Clarion, который в этом случае может обращаться к классам и объектам C++, включая их общие данные и методы.

Следующий демонстрационный пример программного кода показывает, как написать функцию на C++, которая обращается к библиотеке классов C++. Функцию MakeFileList

можно вызвать прямо из Clarion, а конструкторы DirList и члены класса - нет. С помощью класса DirList реализуется список файлов в каталоге, который может быть упорядочен по именам файлов, размерам и датам. Так как показано далее объявляются точки входа для вызова из Clarion и определения этого класса (отметим, что для принятия соглашения об образовании имен по типу C используется спецификатор 'extern "C"'):

```

    /*** определение класса DirList
class DirList: public List {
    public:
        DirList(char *Path, CLAUSHORT Attr, CLAUSHORT Order);
        DirList();
        void ReOrder(int Order);
};

    /*** объявления точек входа
extern "C" {
    void MakeFileList(char *Path, CLAUSHORT Attr, CLAUSHORT Order);
}

```

Следующий программный код не нужен не для чего больше, чем для того, чтобы обеспечить точки входа для обращения из Clarion к функциональным возможностям библиотечного класса DIRLIST. Поскольку, Clarion не образует принятые в C++ составные имена и не может обращаться напрямую к классам и их членам, этот программный интерфейс довольно прост:

```

DirList *FileList = NULL;           // объект - список файлов

void MakeFileList(char *Path, CLAUSHORT Attr, CLAUSHORT Order)
{
    if (FileList != NULL)           // если уже имеем список
    {
        delete FileList;           // вызовем деструктор класса
        FileList = NULL;           // и сможем начать сначала
    }
    FileList = new DirList(Path, Attr, Order);
}

```

Далее идет соответствующая структура MAP с прототипом, который позволяет Clarion обращаться к интерфейсной функции MakeFileList:

```

MAP
    MODULE('DirList')

```

```

MakeFileList(*CSTRING,USHORT,USHORT),RAW,NAME('_MakeFileList')
END
END

```

В таком подходе есть только один недостаток: кажется, что создание интерфейса для большой библиотеки связано с большим объемом дополнительной работы. Однако на практике необходимо только обеспечить очень небольшой интерфейс, чтобы начать извлекать преимущества от использования существующей библиотеки классов C++.

Обращаться из Clarion к коду на C++, скомпилированному компиляторами других фирм (не TopSpeed) невозможно. Для модулей C++ обычно требуется специальная инициализация: конструкторы для всех статических объектов должны вызываться в правильном порядке. Этот процесс инициализации должен быть выполнен “start-up”-кодом Clarion для Windows. Этот начальный (“start-up”) код в Clarion автоматически выполняет необходимую инициализацию для любых имеющихся модулей TopSpeed C++, но не инициализирует модули, скомпилированные другими компиляторами. Даже если модули не требуют инициализации, проблема состоит еще и в том, что другие компиляторы используют другие соглашения о вызовах и образовании имен и применяют отличающуюся внутреннюю структуру классов. Это делает невозможным использование в приложениях на Clarion библиотек классов от других компиляторов.

## **Инициализация модулей Modula-2 и Pascal**

---

Многие модули на Modula-2 и Паскале содержат секции инициализации, которые выполняются перед основным программным модулем. Если вы обращаетесь к процедурам Modula-2 или Паскаля из модуля, которому требуется инициализация, обязательно нужно быть уверенным, что каждый модуль перед использованием инициализирован.

Процедура InitModules языка Clarion предназначена для того, чтобы облегчить инициализацию модулей на Modula-2 и Паскале. Интерфейс Clarion для InitModules содержит полное описание ее использования:

```
!*** записи инициализации Modula-2 RTL
_FIO          BYTE,DIM(0),EXTERNAL,NAME('FIO$')
_FloatExc     BYTE,DIM(0),EXTERNAL,NAME('FloatExc$')
_ShtHeap      BYTE,DIM(0),EXTERNAL,NAME('ShtHeap$')
_Str          BYTE,DIM(0),EXTERNAL,NAME('Str$')
_BiosIO       BYTE,DIM(0),EXTERNAL,NAME('BiosIO$')
_FIOR         BYTE,DIM(0),EXTERNAL,NAME('FIOR$')
_FormIO       BYTE,DIM(0),EXTERNAL,NAME('FormIO$')
_Graph        BYTE,DIM(0),EXTERNAL,NAME('Graph$')
_LIM          BYTE,DIM(0),EXTERNAL,NAME('LIM$')
_MATHLIB      BYTE,DIM(0),EXTERNAL,NAME('MATHLIB$')
_MsMouse      BYTE,DIM(0),EXTERNAL,NAME('MsMouse$')
_IO           BYTE,DIM(0),EXTERNAL,NAME('IO$')
_Lib          BYTE,DIM(0),EXTERNAL,NAME('Lib$')
```

```

_SYSTEM          BYTE,DIM(0),EXTERNAL,NAME('SYSTEM$')
_Storage         BYTE,DIM(0),EXTERNAL,NAME('Storage$')
_Window          BYTE,DIM(0),EXTERNAL,NAME('Window$')

!*** записи инициализации Pascal RTL
_PasLib          BYTE,DIM(0),EXTERNAL,NAME('PasLib$')
_PasDos          BYTE,DIM(0),EXTERNAL,NAME('PasDos$')
_PasRed          BYTE,DIM(0),EXTERNAL,NAME('PasRed$')
_PasProc         BYTE,DIM(0),EXTERNAL,NAME('PasProc$')

MAP
    MODULE('InitMod')
        InitModules(LONG, LONG, <LONG>, <LONG>, <LONG>, <LONG>),
        C, RAW, NAME('_InitModules');
    END
END

```

В любом программном модуле Clarion, который использует модули на Modula-2 и Паскале первым вызовом должен быть вызов процедуры InitModules. За одно обращение можно инициализировать до 5 модулей. Последним параметром должен быть 0 (если передается 5 параметров, то 6-м должен быть 0, если 2 - то 3-м должен быть 0 и т.д.). Параметры указывают на записи инициализации.

Объявления перед структурой MAP представляют собой записи инициализации для библиотеки времени выполнения Modula-2 и Pascal. Вы должны объявить подобные записи инициализации и для модулей, написанных на Паскале и Modula-2 вами. Имя записи инициализации должно быть указано в атрибуте NAME() и оканчиваться на '\$':

```

MyModule  BYTE,DIM(0),EXTERNAL,NAME('MyModule$')
CODE
    InitModules(MyModule, 0)

```

## Резюме

Интерфейс прикладного программирования API обеспечивает ряд функций, помогающих разработчику реализовать интерфейс с программным кодом, написанным на других языках программирования. С небольшими мерами предосторожности можно сделать интерфейс для Clarion для Windows к некоторым очень полезным и мощным внешним библиотекам.

В данном документе имеется полный пример, включающий в себя программный код, написанный на TopSpeed C, C++, Pascal и Modula-2. В этом примере реализуются все

концепции, рассматривавшиеся в данном руководстве. Это средство для вас, которое “все расставит по своим местам”.

При написании интерфейса к библиотекам, написанным на других языках, следует учитывать следующие советы:

- Не пишите функции на C, C++, Pascal или Modula-2, возвращающие в Clarion переменную типа CSTRING. Пусть подпрограммы на этих языках помещают значение такой строки в public-переменную или передают параметр \*CSTRING - адрес строки, в подпрограмму на C, для того чтобы принять значение.
- Не вызывайте из других языков функции на языке Clarion, которые возвращают переменную типа STRING. Пусть подпрограмма на Clarion помещает значение такой строки в public-переменную или передают параметр \*CSTRING - адрес строки, в подпрограмму на C.
- Для простоты и увеличения эффективности параметры STRING и GROUP в обычных случаях передавать “адресом” и с атрибутом RAW, чтобы быть уверенным в том, что передается только адрес.
- Тестируйте приложение сначала в модели памяти XLARGE.

### **Вопросы относительно C и C++**

- Если функция на C или C++ принимает в качестве параметра указатель, то соответствующий параметр в прототипе в Clarion для этой функции нужно объявить как передаваемый “посредством адреса”, указав перед типом данных звездочку (\*).
- Если функция на C или C++ принимает в качестве параметра указатель на GROUP, STRING, PSTRING или CSTRING, то в прототипе в Clarion следует использовать атрибут RAW.
- Если функция на C или C++ принимает в качестве параметра строку типа ASCIIZ (заканчивающуюся нулем), то соответствующий параметр в прототипе в Clarion для этой функции должен иметь тип \*CSTRING.
- Если функция на C или C++ принимает в качестве параметра указатель на структуру, то соответствующий параметр в прототипе в Clarion для этой функции должен иметь тип \*GROUP.
- В качестве шаблонов при разработке интерфейса из Clarion к библиотеке C или C++ используйте заголовочные файлы, что исключает необходимость использования в прототипах в Clarion атрибута NAME для указания имен.
- Помните, что имена в C различаются по регистру букв и начинаются со знака подчеркивания ( \_ ), - для указания имен функций в библиотеке, для которой не использовался макрос CLA\_CONV используйте в прототипах в Clarion атрибут NAME

### **Вопросы относительно Modula-2 и Паскаль**

- Если функция на Modula-2 или Паскале принимает параметр типа VAR, то в соответствующем параметре в прототипе в Clarion для этой функции перед типом данных

должна указываться звездочка, что задает “передаваемый посредством адреса” параметр.

- Если функция на Modula-2 или Паскале принимает параметр типа GROUP, STRING, PSTRING или CSTRING, то следует использовать в прототипе в Clarion атрибут RAW.

- Если функция на Modula-2 или Паскале принимает параметр типа VAR record, то в прототипе в Clarion следует использовать \*GROUP и атрибут RAW.

### **Дополнительные вопросы относительно C++**

- Для интерфейсный функций на C++ используйте спецификацию “Pascal”. Это исключит необходимость использования в прототипе на Clarion атрибута NAME.

- Не обращайтесь из текста на Clarion к функциям-членам классов.
- Не пытайтесь обращаться из текста на Clarion к объектам классов C++.
- Не пытайтесь обращаться коду на C++, компилированному не TopSpeed компилятором C++.

### **Дополнительные вопросы относительно Modula-2**

- В качестве шаблонов при разработке интерфейса из Clarion к библиотеке Modula-2 используйте файлы определений (.DEF).

- Если функция на Modula-2 принимает в качестве параметра строку типа ASCIIZ (заканчивающуюся нулем), то соответствующий параметр в прототипе в Clarion для этой функции должен иметь тип \*CSTRING.

- Помните, что к именам функций в Modula-2 спереди добавляются имена модулей через символ-разделитель ‘\$’, регистр букв различается, - для указания имен функций используйте в прототипах в Clarion атрибут NAME.

### **Дополнительные вопросы относительно Pascal**

- В качестве шаблонов при разработке интерфейса из Clarion к библиотеке Pascal используйте интерфейсные файлы (.ITF).

- Помните, что к именам функций в Паскале спереди добавляются имена модулей через символ-разделитель ‘\$’, имена записываются большими буквами, - для указания имен функций используйте в прототипах в Clarion атрибут NAME.



## **Вызовы API и усложненные файлы ресурсов**

### **Прототипы и объявления**

На вашем установочном компакт-диске есть файлы с прототипами, объявлениями и заголовками, которые можно использовать для того, чтобы позволить Clarion для Windows взаимодействовать с Windows, C/C++, Modula-2 и т.д.

#### **Обращения из Clarion к стандартной библиотеке C/C++**

---

Для того, чтобы обращаться к функциям стандартной библиотеки C из Clarion для Windows, вставьте текст `\CLIB.CLW` в точку вставки “Inside the Global Map”.

```
INCLUDE('CLIB.CLW')
```

Этот файл содержит прототипы для разнообразных строковых функций, целочисленной математики, функций для работы с символами, и для работы с файлами на низком уровне. Подробную информацию по отдельным функциям смотрите в Справочном руководстве по библиотеке C/C++.

#### **Обращение из Clarion к API Windows**

---

Для того, чтобы обращаться из Clarion к функциям Интерфейса прикладного программирования Windows (Windows API), нужно в структуру MAP приложения включить прототипы этих функций и глобально объявить стандартные присвоения мнемонических имен и требуемые для этих функций структуры данных.

В состав Clarion для Windows входит утилита WINAPI.EXE, которая создает файл, содержащий все операторы, которые необходимо включить в приложение. Эта программа по умолчанию создает файл WINAPI.CLW, который состоит из двух разделов: раздела “Equates”, содержащего все необходимые операторы EQUATE и раздела “Prototypes”, в котором приведены прототипы использованных вами функций Windows API.

Включите раздел “Equates” файла WINAPI.CLW в точку вставки “After Global INCLUDEs”:

```
INCLUDE('WINAPI.CLW','Equates')
```

А раздел прототипов этого файла включите в точку вставки “Inside the Global Map”:

```
INCLUDE('WINAPI.CLW','Prototypes')
```

Подробную информацию по отдельным функциям следующих категорий смотрите в справочнике по Windows API:

Creating Windows	(создание окон)
Window Support	(поддержка окон)
Message Processing	(обработка сообщений)
Memory Management	(управление памятью)
Bitmaps and Icons	(растровые изображения и пиктограммы)
Color Palette Control	(управление цветовой палитрой)
Sound	(звук)
Character Sets and Strings	(наборы символов и строки)
Communications	(связь)
Metafiles	(метафайлы)
Tool Help Library	(инструментальная библиотека для Help'a)
File Compression	(сжатие файлов)
Installation and Version Information	(установка и версии программ)
TrueType Fonts	(TrueType шрифты)
Multimedia	(мультимедиа)

## **Обращение из Modula-2 к Clarion**

---

### **Библиотека исполняющей системы Clarion**

При обращении к функциям исполняющей системы Clarion для Windows используйте файл \CWRUN.DEF. В нем содержатся объявления для Modula-2 множества функций языка Clarion, а также многих функций стандартной библиотеки C, которые находятся в библиотеке исполняющей системы CWRUNxx.DLL. Функции, которые можно использовать, описываются далее в разделе Функции исполняющей системы Clarion.

### **Функции файлового драйвера Clarion**

Для того, чтобы обращаться к функциям драйвера баз данных Clarion для Windows, используйте файл \CWFILE.DEF. Он содержит объявления на языке Modula-2 для функций, манипулирующих такими объектами Clarion как FILE, RECORD, KEY, INDEX, MEMO и BLOB, а также полное описание используемого в Clarion блока управления файлом.

## **Обращение из C/C++ к Clarion**

---

### **Библиотека исполняющей системы Clarion**

При обращении к функциям исполняющей системы Clarion для Windows используйте файл \CWRUN.H. В нем содержатся прототипы C/C++ для множества функций языка Clarion, а также многих функций стандартной библиотеки C, которые находятся в библиотеке исполняющей системы CWRUNxx.DLL. Функции, которые можно использовать, описываются далее в разделе Функции исполняющей системы Clarion.

## Функции файлового драйвера Clarion

Для того, чтобы обращаться к функциям драйвера баз данных Clarion для Windows, используйте файл \CWFIL.H. Он содержит прототипы C/C++ для функций, манипулирующих такими объектами Clarion как FILE, RECORD, KEY, INDEX, MEMO и BLOB, а также полное описание используемого в Clarion блока управления файлом.

## **Обращение к библиотеке исполняющей системы Clarion из C/C++ или Modula-2**

Далее приведен список функций библиотеки исполняющей системы Clarion, структур данных и переменных, которые можно использовать в программном коде на C/C++ или Modula-2.

### **Структуры и определения типов данных**

---

#### **COLORREF**

C++:                   typedef unsigned long COLORREF;  
Modula-2:             TYPE COLORREF = LONGINT;

### **Переменные времени выполнения**

---

Во время выполнения программы можно прочитать значение следующих переменных:

Cla\$DOSerror	Целочисленная переменная без знака, содержащая код последней ошибки, полученный от DOS.
Cla\$FILEERRCODE	Целочисленная переменная, содержащая код последней ошибки Clarion.
Cla\$FILEERRORMSG	Массив из 80 символов, содержащий текст последнего сообщения об ошибке Clarion.
WSL@AppInstance	Переменная типа unsigned short, содержащая идентификатор экземпляра данного приложения.

### **Встроенные функции Clarion**

---

Приведенные далее функции - это встроенные функции языка Clarion, к которым спокойно можно обращаться во время выполнения приложения. Если не оговорено особо, то подразумевается, что обращение к этим функциям происходит по типу связей языка C.

Cla\$ACOS	Функция ACOS() языка Clarion. Возвращает угол, косинус которого представляет величина параметра.
B C++:	double Cla\$ACOS(double val)
B Modula-2:	Cla\$ACOS(val :LONGREAL):LONGREAL; val:   числовое выражение описывающее угол в радианах.
Cla\$ARC	Оператор ARC языка Clarion. Помещает в текущее окно или отчет

	дугу эллипса, вписанного в прямоугольник, определенный параметрами x, y, wd и ht
B C++:	void Cla\$ARC(int x, int y, int wd, int ht, int start, int end)
B Modula-2:	Cla\$ARC(x,y,wd,ht,start,end: INTEGER);
x:	Целое число, задающее горизонтальную координату начальной точки.
y:	Целое число, задающее вертикальную координату начальной точки.
wd:	Целое число, задающее ширину.
ht:	Целое число, задающее высоту.
start:	Целое число, задающее начало дуги в десятых долях градуса.
end:	Целое число, задающее конец дуги в десятых долях градуса.
Cla\$ASIN	Функция ASIN() языка Clarion. Возвращает арксинус величины, заданной параметром val.
B C++:double	Cla\$ASIN(double val)
B Modula-2:	Cla\$ASIN(val LONGREAL): LONGREAL;
val:	Числовое выражение, описывающее угол в радианах.
Cla\$ATAN	Функция ATAN() языка Clarion. Возвращает арктангенс величины, заданной параметром val.
B C++:	double Cla\$ATAN(double val)
B Modula-2:	Cla\$ATAN(val: LONGREAL):LONGREAL;
val:	Числовое выражение, описывающее угол в радианах.
Cla\$BOX	Оператор BOX языка Clarion. Эта процедура рисует в текущем окне или отчете прямоугольник, закрашенный цветом, который определен структурой COLORREF, левый верхний угол, которого задается координатами x, y, заданной длины и высоты.
B C++:	void Cla\$BOX(int x, int y, int wd, int ht, COLORREF fillcolor)
B Modula-2:	Cla\$BOX(x, y, wd, ht: INTEGER; fillcolor: COLORREF);
x:	Целое число, задающее горизонтальную координату левого верхнего угла.
y:	Целое число, задающее вертикальную координату левого верхнего угла.
wd:	Целое число, задающее ширину.
ht:	Целое число, задающее высоту.
fillcolor:	Структура COLORREF.
Cla\$BSHIFT	Функция BSHIFT() языка Clarion. Эта функция возвращает результат битового сдвига значения параметра val на заданное параметром count число двоичных разрядов. Если значение count положительное, то происходит сдвиг влево, если отрицательное, то

	вправо.
B C++:	long Cla\$BSHIFT(long val, int count)
B Modula-2:	Cla\$BSHIFT(val: LONGINT; count: INTEGER): LONGINT;
	val: Числовое выражение.
	count: Числовое выражение.
Cla\$CHORD	Оператор CHORD языка Clarion. Рисует в текущем окне или отчете дугу эллипса, концы которой соединены отрезком. Эллипс вписан в прямоугольник, заданный параметрами x, y, wd и ht. Цвет закрашивания фигуры определяется структурой COLORREF. Параметры start и end определяют, какая часть эллипса должна прорисовываться.
B C++:	void Cla\$CHORD(int x, int y, int wd, int ht, int start, int end, COLORREF fillcolor)
B Modula-2:	Cla\$CHORD(x, y, wd, ht, start, end: INTEGER; fillcolor: COLORREF);
x:	Целое число, задающее горизонтальную координату левого верхнего угла прямоугольника.
y:	Целое число, задающее вертикальную координату левого верхнего угла прямоугольника.
wd:	Целое число, задающее ширину прямоугольника.
ht:	Целое число, задающее высоту прямоугольника.
start:	Целое число, задающее начало хорды в десятых долях градуса.
end:	Целое число, задающее конец хорды в десятых долях градуса.
fillcolor:	Структура COLORREF.
Cla\$CLOCK	Функция CLOCK() языка Clarion. Возвращает системное время в формате стандартного времени Clarion.
B C++:	long Cla\$CLOCK(void)
B Modula-2:	Cla\$CLOCK(): LONGINT;
Cla\$COS	Функция COS() языка Clarion. Возвращает косинус угла, заданного параметром val.
B C++:	double Cla\$COS(double val)
B Modula-2:	Cla\$COS(val: LONGREAL): LONGREAL;
	val: числовое выражение, задающее угол в радианах.
Cla\$DATE	Функция DATE() языка Clarion. Возвращает стандартную дату Clarion по заданным параметрами day, month и year компонентам.
B C++:	long Cla\$DATE(unsigned mn, unsigned dy, unsigned yr)
B Modula-2:	Cla\$DATE(mn, dy, yr: CARDINAL): LONGINT;
mn:	числовое выражение в диапазоне от 1 до

	12, задающее месяц.
dy:	числовое выражение в диапазоне от 1 до 31, задающее день месяца.
yr	числовое выражение в диапазоне от 1801 до 2099, задающее год.
Cla\$DAY	Функция DAY() языка Clarion. Возвращает день в диапазоне от 1 до 31 из параметра заданного в формате стандартной даты Clarion.
B C++:	long Cla\$DAY(long dt)
B Modula-2:	Cla\$DAY(dt: LONGINT): LONGINT;
dt:	Числовое выражение представляющее дату в формате стандартной даты Clarion.
Cla\$ELLIPSE	Оператор ELLIPSE языка Clarion. Рисует в текущем окне или отчете эллипс. Эллипс вписан в прямоугольник, заданный параметрами x, y, wd и ht. Цвет линии определяется структурой COLORREF.
B C++:	void Cla\$ELLIPSE(int x, int y, int wd, int ht, COLORREF fillcolor)
B Modula-2:	Cla\$ELLIPSE(x, y, wd, ht: INTEGER; fillcolor: COLOREF);
x:	Целочисленное выражение.
y:	Целочисленное выражение.
wd:	Целочисленное выражение.
ht:	Целочисленное выражение.
fillcolor:	Структура COLORREF.
Cla\$INT	Функция INT() языка Clarion. Возвращает целую часть параметра val. Значение усекается до десятичной точки, округления не выполняется.
B C++:	double Cla\$INT(double val)
B Modula-2:	Cla\$INT(val: LONGREAL): LONGREAL;
val:	Целочисленное выражение.
Cla\$LOG10	Функция LOG10() языка Clarion. Возвращает логарифм по основанию 10 от заданного параметром val значения.
B C++:	double Cla\$LOG10(double val)
B Modula-2:	Cla\$LOG10(val: LONGREAL): LONGREAL;
val:	Целочисленное выражение.
Cla\$LOGE	Функция LOGE() языка Clarion. Возвращает натуральный логарифм от заданного параметром val значения.
B C++:	double Cla\$LOGE(double val)
B Modula-2:	Cla\$LOGE(val: LONGREAL): LONGREAL;
val:	Целочисленное выражение.
Cla\$MONTH	Функция MONTH() языка Clarion. Из заданного в формате стандартной даты Clarion возвращает месяц (число в диапазоне от 1 до 12).
B C++:	long Cla\$MONTH(long dt)
B Modula-2:	Cla\$MONTH(dt: LONGINT): LONGINT;
dt:	Целочисленное выражение, содержащее

Cla\$MOUSEX	дату в формате стандартной даты Clarion. Функция MOUSEX() языка Clarion. Возвращает горизонтальную координату указателя мыши.
B C++:	int Cla\$MOUSEX(void)
B Modula-2:	Cla\$MOUSEX(): INTEGER;
Cla\$MOUSEY	Функция MOUSEY() языка Clarion. Возвращает вертикальную координату указателя мыши.
B C++:	int Cla\$MOUSEY(void)
B Modula-2:	Cla\$MOUSEY(): INTEGER;
Cla\$NUMERIC	Функция NUMERIC() языка Clarion. Возвращает 1 (истина), если параметр str содержит правильное представление числа. В противном случае возвращает 0 (ложь).
B C++:	unsigned Cla\$NUMERIC(char *str, unsigned slen)
B Modula-2:	Cla\$NUMERIC(VAR str: ARRAY OF CHAR; slen: CARDINAL): CARDINAL;
	str: Указатель на строку.
	slen: Длина строки.
Cla\$RANDOM	Функция RANDOM() языка Clarion. Возвращает псевдослучайное число, из заданного диапазона.
B C++:	long Cla\$RANDOM(long low, long high)
B Modula-2:	Cla\$RANDOM(low, high: LONGINT): LONGINT;
	low: Целое число, задающее нижнюю границу диапазона.
	high: Целое число, задающее верхнюю границу диапазона.
Cla\$ROUND	Функция ROUND() языка Clarion. Возвращает значение параметра val, округленное до заданного параметром ord десятичного разряда.
B C++:	double Cla\$ROUND(double val, double ord)
B Modula-2:	Cla\$ROUND(val, ord: LONGREAL): LONGREAL;
	val: Целочисленное выражение.
	ord: Целочисленное выражение равное степени 10 (т.е. .001, .0, 1, 10, 100 и т.д.).
Cla\$SETCLOCK	Оператор SETCLOCK языка Clarion. Устанавливает системные часы на время, содержащее параметр dt.
B C++:	void Cla\$SETCLOCK(long dt)
B Modula-2:	Cla\$SETCLOCK(dt: LONGINT);
	dt: Целочисленное выражение в формате стандартного времени Clarion.
Cla\$SETTODAY	Оператор SETTODAY языка Clarion. Устанавливает системную дату DOS, заданную параметром dt.
B C++:	void Cla\$SETTODAY(long dt)
B Modula-2:	Cla\$SETTODAY(dt: LONGINT);

	dt:	Целочисленное выражение, содержащее дату в формате стандартной даты Clarion.
Cla\$SIN		Функция SIN() языка Clarion. Возвращает синус, заданного параметром val, угла.
B C++:		double Cla\$SIN(double val)
B Modula-2:		CLA\$SIN(val: LONGREAL): LONGREAL;
	val:	Целочисленное выражение, задающее угол в радианах.
Cla\$SQRT		Функция SQRT() языка Clarion. Возвращает квадратный корень из заданной параметром val величины.
B C++:		double Cla\$SQRT(double val)
B Modula-2:		Cla\$SQRT(val: LONGREAL): LONGREAL;
	val:	Целочисленное выражение.
Cla\$TAN		Функция TAN() языка Clarion. Возвращает тангенс угла, заданного параметром val.
B C++:		double Cla\$TAN(double val)
B Modula-2:		Cla\$TAN(val: LONGREAL): LONGREAL;
	val:	Целочисленное выражение, задающее угол в радианах.
Cla\$TODAY		Функция TODAY() языка Clarion. Возвращает системную дату в формате стандартной даты Clarion.
B C++:		long Cla\$TODAY(void)
B Modula-2:		Cla\$TODAY(): LONGINT;
Cla\$YEAR		Функция YEAR() языка Clarion. Извлекает год в диапазоне от 1801 до 2099 из даты в формате стандартной даты Clarion.
B C++:		long Cla\$YEAR(long dt)
B Modula-2:		Cla\$YEAR(dt: LONGINT): LONGINT;
	dt:	Целочисленное выражение задающее дату в формате стандартной даты Clarion.

### **Функции Clarion, управляющие стэковой обработкой строк**

В следующем разделе описывается использование внутренних функций Clarion выполняющих во время исполнения программы работу со строками. В Clarion используется такой же как в ЛИСП подход к работе со строками, при котором параметры помещаются в вершину стэка, а операции выполняются над самыми верхними элементами стека. Предполагается, что если не оговорено особо, то использованные элементы функции удаляют (или выталкивают) из стэка.

Заметьте, пожалуйста, что в некоторые из следующих процедур, требуются передавать в качестве параметров указатели на строки, оканчивающиеся двоичным нулем. Программистам на Modula-2 для преобразования строк к этому виду следует использовать библиотечную функцию Str.StrToC. Кроме того, для того, чтобы в библиотечную функцию не передавался размер массива символов, нужно использовать прагму call(o\_a\_size=>off,o\_a\_copy=>off).



Cla\$PopCString	Берет верхний элемент стэка и копирует его по адресу, указанному параметром s, параметр len задает длину копируемой строки.
B C++:	void Cla\$PopCString(char *s, unsigned len)
B Modula-2:	Cla\$PopCString(s: POINTER TO CHAR; len: CARDINAL);
s:	Указатель на оканчивающуюся двоичным нулем строку.
len:	Длина строки s.
Cla\$PopPString	Берет верхний элемент стэка и копирует его по адресу, указанному параметром s, параметр len задает длину копируемой строки. При копировании данная строка преобразуется в формат, принятый в Паскале (т.е. первый байт содержит длину).
B C++:	void Cla\$PopPString(char *s, unsigned len)
B Modula-2:	Cla\$PopPString(VAR s: ARRAY OF CHAR; len: CARDINAL);
s:	Указатель на строку.
len:	Длина строки s.
Cla\$PopString	Выбирает верхний элемент стэка и копирует его в строку s.
B C++:	void Cla\$PopString(char *s, unsigned len)
B Modula-2:	Cla\$PopString(VAR s: ARRAY OF CHAR; len: CARDINAL);
s:	Указатель на оканчивающуюся двоичным нулем строку.
len:	Длина строки s.
Cla\$PushCString	Помещает строку s в вершину стэка.
B C++:	void Cla\$PushCString(char *s)
B Modula-2:	Cla\$PushCString(VAR s: ARRAY OF CHAR);
s:	Указатель на оканчивающуюся двоичным нулем строку.
Cla\$PushString	Помещает строку s в вершину стэка. Параметр Len задает длину строки s.
B C++:	void Cla\$PushString(char *s, unsigned len)
B Modula-2:	Cla\$PushString(VAR s: ARRAY OF CHAR; len: CARDINAL);
s:	Указатель на строку.
len:	Длина строки s.
Cla\$StackALL	Функция ALL() языка Clarion. Вытаскивает верхний элемент из стэка и заменяет его строкой, содержащей первоначальную строку, повторенную столько раз, сколько нужно, чтобы получить строку длиной len.
B C++:	void Cla\$StackALL(unsigned len)
B Modula-2:	Cla\$StackALL(len: CARDINAL);
len:	Целое без знака.
Cla\$StackCENTER	Функция CENTER() языка Clarion. Вытаскивает верхний элемент из стэка и заменяет его строкой с добавленными спереди пробелами так, чтобы получился центрированный текст в строке длиной len

В C++: `void Cla$StackCENTER(unsigned len)`  
 В Modula-2: `Cla$StackCENTER(len: CARDINAL);`  
           len: Целое без знака.  
 Cla\$StackCLIP Функция CLIP() языка Clarion. Удаляет пробелы в конце строки, являющейся верхним элементом стэка.  
 В C++: `void Cla$StackCLIP(void)`  
 В Modula-2: `Cla$StackCLIP();`  
 Cla\$StackCompare  
           Сравнивает верхний элемент стэка (s1) со вторым элементов (s2) и возвращает одно из следующих значений:  
           -1: если s1 < s2  
           0: если s1 = s2  
           1: если s1 > s2

После выполнения сравнения строки s1 и s2 автоматически удаляются из стэка.

В C++: `int Cla$StackCompare(void)`  
 В Modula-2: `Cla$StackCompare(): INTEGER;`  
 Cla\$StackCompareN  
           Сравнивает верхний элемент стэка с нулевым значением. Возвращает значение “истина”, если верхний элемент есть пустая строка, и “ложь” в противном случае.  
 В C++: `int Cla$StackCompareN(void)`  
 В Modula-2: `Cla$StackCompareN(): INTEGER;`  
 Cla\$StackConcat Два верхних элемента выбираются из стэка, присоединяются друг к другу и получившаяся в результате строка помещается обратно в стэк.  
 В C++: `void Cla$StackConcat(void)`  
 В Modula-2: `Cla$StackConcat();`  
 Cla\$StackINSTRING  
           Функция INSTRING() языка Clarion. Производит поиск внутри первого элемента стэка подстроки совпадающей со вторым элементом стэка. Поиск начинается с байта с порядковым номером start и производится с шагом step до тех пор, пока не будет достигнут конец строки. Возвращается число итераций, потребовавшихся для выполнения поиска подстроки или 0, если подстрока не найдена.  
 В C++: `unsigned Cla$StackINSTRING(unsigned step, unsigned start)`  
 В Modula-2: `Cla$StackINSTRING(step, start: CARDINAL): CARDINAL;`  
           step: Целое без знака, сдвиг по строке при поиске.  
           start: Целое без знака, позиция, с которой начинается поиск.  
 Cla\$StackLEFT Функция LEFT() языка Clarion. Заменяет самую верхнюю строку стэка на ее же, но выровненную по левому краю. Замещающая строка будет иметь длину len.  
 В C++: `void Cla$StackLEFT(unsigned len)`

B Modula-2:	Cla\$StackLEFT(len: CARDINAL);
len:	Целое без знака.
Cla\$StackLen	Возвращает длину самого верхнего элемента стэка. Сам элемент из стэка не выталкивает.
B C++:	unsigned Cla\$StackLen(void)
B Modula-2:	Cla\$StackLen(): CARDINAL;
Cla\$StackLen2	Возвращает длину самого верхнего элемента стэка. После определения длины элемента выталкивает его из стэка.
B C++:	unsigned Cla\$StackLen2(void)
B Modula-2:	Cla\$StackLen2(): CARDINAL;
Cla\$StackLOWER	Функция LOWER() языка Clarion. Замещает самый верхний элемент стэка его эквивалентом, записанным маленькими буквами.
B C++:	void Cla\$StackLOWER(void)
B Modula-2:	Cla\$StackLOWER();
Cla\$STACKpop	Выталкивает верхний элемент из стэка.
B C++:	void Cla\$STACKpop(void)
B Modula-2:	Cla\$STACKpop();
Cla\$StackNUMERIC	Возвращает значение “истина”, если самый верхний элемент стэка содержит строковое представление числового значения, в противном случае возвращает значение “ложь”.
B C++:	unsigned Cla\$StackNUMERIC(void)
B Modula-2:	Cla\$StackNUMERIC(): CARDINAL;
Cla\$StackPRESS	Оператор PRESS языка Clarion. Помещает символы из самой верхней строки стэка в буфер клавиатуры Windows .
B C++:	void Cla\$StackPRESS(void)
B Modula-2:	Cla\$StackPRESS();
Cla\$StackRIGHT	Функция RIGHT() языка Clarion. Замещает самый верхний элемент стэка его эквивалентом, выровненным на правый край. Заменяющая строка будет иметь длину len символов.
B C++:	void Cla\$StackRIGHT(unsigned len)
B Modula-2:	Cla\$StackRIGHT(len: CARDINAL);
len:	Целое без знака.
Cla\$StackSUB	Функция SUB() языка Clarion. Заменяет самую верхнюю строку стэка подстрокой этой строки начиная с символа pos и длиной len.
B C++:	void Cla\$StackSUB(unsigned pos, unsigned len)
B Modula-2:	Cla\$StackSUB(pos, len: CARDINAL);
pos:	Целое без знака; начало подстроки.
len:	Целое без знака; длина подстроки.
Cla\$StackVAL	Функция VAL() языка Clarion. Возвращает код первого символа самой верхней строки в стэке.
B C++:	unsigned char Cla\$StackVAL(void)

В Modula-2:	Cla\$StackVAL(): BYTE;
Cla\$StackUPPER	Замещает самую верхнюю строку стека ее эквивалентом, записанным большими буквами.
В C++:	void Cla\$StackUPPER(void)
В Modula-2:	Cla\$StackUPPER();

## **Стандартные функции C, находящиеся в библиотеке исполняющей системы Clarion**

К следующим функциям, составляющим подмножество стандартной библиотеки TopSpeed, можно обращаться из программного кода на Clarion, C/C++ или Modula-2. Все эти функции описаны в Справочном руководстве по библиотечным функциям TopSpeed (или в любом справочнике по библиотеке стандарта ANSI) и поэтому, здесь подробно не описываются. Если прямо не утверждается иное, то подразумевается, что функция работает точно как описано в справочниках.

Назначение этого списка - просто показать вам, какие стандартные библиотечные функции C доступны и привести правильные прототипы для каждого из языков.

### **Функции преобразований**

Заметьте, что для некоторых из приводимых функций в качестве параметров нужно передавать указатели на оканчивающиеся двоичным нулем строки. Программистам на Modula-2 для преобразования строк к этому виду следует использовать библиотечную функцию Str.StrToC. Кроме того, для того, чтобы в библиотечную функцию не передавался размер массива символов, нужно использовать прагму call(o\_a\_size=>off,o\_a\_copy=>off).

atof	C++:	double atof(const char *_nptr)
	Modula-2:	atof( VAR _nptr: ARRAY OF CHAR): LONGREAL;
	Clarion:	AToF(*cstring),real,raw,name('_atof')
atoi	C++:	int atoi(const char *_nptr)
	Modula-2:	atoi(VAR _nptr: ARRAY OF CHAR): INTEGER;
	Clarion:	AToI(*cstring),short,raw,name('_atoi')
atol	C++:	long atol(const char *_nptr)
	Modula-2:	atol( VAR _nptr: ARRAY OF CHAR): LONGINT;
	Clarion:	AToL(*cstring),long,raw,name('_atol')
atoul	C++:	unsigned long atoul(const char *_nptr)
	Modula-2:	atoul(VAR _nptr: ARRAY OF CHAR): LONGCARD;
	Clarion:	AToUL(*cstring),ulong,raw,name('_atoul')

## Целочисленная арифметика

---

abs

```
C++:  int abs(int _num)
Modula-2:  abs(_num: INTEGER): INTEGER;
Clarion:  API_Abs(short),short,name('_abs')
          ! переименована во избежание конфликта с ! Builtins.Clw
```

labs

```
C++:  long labs(long _j)
Modula-2:  labs(_i: LONGINT): LONGINT;
Clarion:  LAbs(long),long,name('_labs')
```

## Символьные функции

---

Следующие функции протестированы только при реализации их как функций. Мы не советуем определять `_CT_MTF` с целью реализации их как макросов.

toupper

```
C++:  int toupper(int c)
Modula-2:  toupper(c: INTEGER): INTEGER;
Clarion:  ToUpper(short),short,name('_toupper')
```

tolower

```
C++:  int tolower(int c)
Modula-2:  tolower(c: INTEGER): INTEGER;
Clarion:  ToLower(short),short,name('_tolower')
```

isalpha

```
C++:  int isalpha(int c)
Modula-2:  isalpha(c: INTEGER): INTEGER;
Clarion:  API_IsAlpha(short),short,name('_isalpha')
          !Переименована во избежание конфликта с Builtins.Clw
```

islower

```
C++:  int islower(int c)
Modula-2:  islower(c: INTEGER): INTEGER;
Clarion:  API_IsLower(short),short,name('_islower')
          !Переименована во избежание конфликта с Builtins.Clw
```

isupper

```
C++:  int isupper(int c)
Modula-2:  isupper(c: INTEGER): INTEGER;
Clarion:  API_IsUpper(short),short,name('_isupper')
          !Переименована во избежание конфликта с Builtins.Clw
```

isascii

```
C++:  int isascii(int c)
Modula-2:  isascii(c: INTEGER): INTEGER;
Clarion:  IsAscii(short),short,name('_isascii')
```

isctrl

C++: int isctrl(int c)  
Modula-2: isctrl(c: INTEGER): INTEGER;  
Clarion: IsCntrl(short),short,name('\_isctrl')

isdigit

C++: int isdigit(int c)  
Modula-2: isdigit(c: INTEGER): INTEGER;  
Clarion: IsDigit(short),short,name('\_isdigit')

isgraph

C++: int isgraph(int c)  
Modula-2: isgraph(c: INTEGER): INTEGER;  
Clarion: IsGraph(short),short,name('\_isgraph')

isprint

C++: int isprint(int c)  
Modula-2: isprint(c: INTEGER): INTEGER;  
Clarion: IsPrint(short),short,name('\_isprint')

ispunct

C++: int ispunc(int c)  
Modula-2: ispunc(c: INTEGER): INTEGER;  
Clarion: IsPunct(short),short,name('\_ispunct')

isspace

C++: int isspace(int c)  
Modula-2: isspace(c: INTEGER): INTEGER;  
Clarion: IsSpace(short),short,name('\_isspace')

isxdigit

C++: int isxdigit(int c)  
Modula-2: isxdigit(c: INTEGER): INTEGER;  
Clarion: IsXDigit(short),short,name('\_isxdigit')

## Служебные функции

---

rand

C++: int rand(void)  
Modula-2: rand(): INTEGER;  
Clarion: Rand(),short,name('\_rand')

srand

C++: void srand(unsigned \_seed)  
Modula-2: srand(\_seed: CARDINAL);  
Clarion: SRand(ushort),name('\_srand')

## Строковые функции

---

strcat

C++: char \*strcat(char \*\_dest, const char \*\_source)  
Modula-2: Не использовать

Clarion: StrCat(\*cstring,\*cstring),cstring,raw,name('\_strcat')

strcmp  
C++: int strcmp(const char \*\_s1, const char \*\_s2)  
Modula-2: Не использовать  
Clarion: StrCmp(\*cstring,\*cstring),short,raw,name('\_strcmp')

chrncmp  
C++: int chrncmp(char \_c1, char \_c2)  
Modula-2: chrncmp(\_c1,\_c2: CHAR): INTEGER;  
Clarion: ChrCmp(byte,byte),short,name('\_chrncmp')

strequ  
C++: int strequ(const char \*\_s1, const char \*\_s2)  
Modula-2: Не использовать  
Clarion: StrEqu(\*cstring,\*cstring),short,raw,name('\_strequ')

strcpy  
C++: char \*strcpy(char \*\_dest, const char \*\_source)  
Modula-2: Не использовать  
Clarion: StrCpy(\*cstring,\*cstring), cstring, raw,| name('\_strcpy')

strlen  
C++: unsigned strlen(const char \*\_s)  
Modula-2: strlen(VAR \_s: ARRAY OF CHAR): CARDINAL;  
Clarion: StrLen(\*cstring),ushort,raw,name('\_strlen')

strchr  
C++: char \*strchr(const char \*\_s, int \_c)  
Modula-2: Не использовать  
Clarion: StrChr(\*cstring,short),cstring,raw,name('\_strchr')

strcspn  
C++: unsigned strcspn(const char \*\_s1, const char \*\_s2)  
Modula-2: Не использовать  
Clarion: StrCSpn(\*cstring,\*cstring), ushort, raw,| name('\_strcspn')

strerror  
C++: char \* strerror(int \_errno)  
Modula-2: Не использовать  
Clarion: StrError(short),cstring,raw,name('\_strerror')

strspn  
C++: unsigned strspn(const char \*\_s1, const char \*\_s2)  
Modula-2: Не использовать  
Clarion: StrSpn(\*cstring,\*cstring),ushort,raw,name('\_strspn')

strstr  
C++: char \*strstr(const char \*\_s1, const char \*\_s2)  
Modula-2: Не использовать  
Clarion: StrStr(\*cstring,\*cstring),cstring,raw,name('\_strstr')

strtok  
C++: char \*strtok(char \*\_s1, const char \*\_s2)

Modula-2: Не использовать  
Clarion: StrTok(\*cstring,\*cstring),cstring,raw,name('\_strtok')

strpbrk  
C++: char \*strpbrk(const char \*\_s1, const char \*\_s2)  
Modula-2: Не использовать  
Clarion: StrPBrk(\*cstring,\*cstring), cstring, raw,| name('\_strpbrk')

strrchr  
C++: char \*strrchr(const char \*\_s, int \_c)  
Modula-2: Не использовать  
Clarion: StrRChr(\*cstring,short),cstring,raw,name('\_strrchr')

strlwr  
C++: char \*strlwr(char \*\_s)  
Modula-2: Не использовать  
Clarion: StrLwr(\*cstring),cstring,raw,name('\_strlwr')

strupr  
C++: char \*strupr(char \*\_s)  
Modula-2: Не использовать  
Clarion: StrUpr(\*cstring),cstring,raw,name('\_strupr')

strdup  
C++: char \*strdup(const char \*\_s)  
Modula-2: Не использовать  
Clarion: StrDup(\*cstring),cstring,raw,name('\_strdup')

strrev  
C++: char \*strrev(char \*\_s)  
Modula-2: Не использовать  
Clarion: StrRev(\*cstring),cstring,raw,name('\_strrev')

strncat  
C++: char \*strncat(char \*\_dest, const char \*\_source, unsigned \_n)  
Modula-2: Не использовать  
Clarion: StrNCat(\*cstring,\*cstring, ushort), cstring, raw,| name('\_strncat')

strncmp  
C++: int strncmp(const char \*\_s1, const char \*\_s2, unsigned \_n)  
Modula-2: Не использовать  
Clarion: StrNCmp(\*cstring,\*cstring, ushort), short, raw,| name('\_strncmp')

strncpy  
C++: char \* strncpy(char \*\_dest, const char \*\_source, unsigned \_n)  
Modula-2: Не использовать  
Clarion: StrNCpy(\*cstring,\*cstring, ushort), cstring, raw,| name('\_strncpy')

strnicmp  
C++: int stricmp(const char \*\_s1, const char \*\_s2, unsigned \_n)  
Modula-2: Не использовать  
Clarion: StrNICmp(\*cstring,\*cstring, ushort), short, raw,| name('\_strnicmp')



## Работа с файлами на низком уровне

---

access	<p>Проверяет существует ли заданный параметром path файл или каталог, и, если это не каталог, возможно ли к нему обращение в указанном режиме.</p> <p>C++: <code>int _access(const char *path, int access)</code></p> <p>Modula-2: <code>_access(VAR path: ARRAY OF CHAR; access: INTEGER): INTEGER;</code></p> <p>Clarion: <code>Access(*cstring,short),short,raw,name('_access')</code></p>
chmod	<p>C++: <code>int _chmod(const char *path, int mode)</code></p> <p>Modula-2: <code>_chmod(VAR path: ARRAY OF CHAR; mode: INTEGER): INTEGER;</code></p> <p>Clarion: <code>ChMod(*cstring,short),short,raw,name('_chmod')</code></p>
remove	<p>Удаляет заданный параметром path файл.</p> <p>C++: <code>int _remove(const char *_path)</code></p> <p>Modula-2: <code>_remove(VAR _path: ARRAY OF CHAR): INTEGER;</code></p> <p>Clarion: <code>API_Remove(*cstring),short,raw,name('_remove')</code> !Переименована во избежание конфликта с Bultins.Clw</p>
rename	<p>Изменяет имя файла или каталога, заданное параметром oldname.</p> <p>C++: <code>int _rename(const char *_oldname, const char *_newname)</code></p> <p>Modula-2: <code>rename(VAR _oldname, VAR _newname: ARRAY OF CHAR): INTEGER;</code></p> <p>Clarion: <code>API_Rename(*cstring, *cstring), short, raw, name('_rename')</code> !Переименована во избежание конфликта с Bultins.Clw</p>
fnmerge	<p>Из компонентов: drive, directory, filename и extension строит полное имя файла.</p> <p>C++: <code>void _fnmerge(char *_path, const char *_drive, const char *_dir, const char *_name, const char *_ext)</code></p> <p>Modula-2: <code>_fnmerge(VAR _path, VAR _drive, VAR _dir, VAR _name, VAR _ext: ARRAY OF CHAR);</code></p> <p>Clarion: <code>FnMerge(*cstring, *cstring, *cstring, *cstring,   *cstring), raw, name('_fnmerge')</code></p>
fnsplit	<p>Эта функция разбивает полное имя файла на составляющие.</p> <p>C++: <code>int _fnsplit(const char *_path, char *_drive, char *_dir, char *_name, char *_ext)</code></p> <p>Modula-2: <code>_fnsplit(VAR _path, VAR _drive, VAR _dir, VAR _name, VAR _ext: ARRAY OF CHAR): INTEGER;</code></p> <p>Clarion: <code>FnSplit(*cstring, *cstring, *cstring, *cstring,   *cstring), short, raw, name('_fnsplit')</code></p>

mkdir	Создает новый каталог с именем, заданным параметром path. C++: int _mkdir(const char *_path) Modula-2: _mkdir(VAR _path: ARRAY OF CHAR): INTEGER; Clarion: Mkdir(*cstring),short,raw,name('_mkdir')
rmdir	Удаляет каталог с именем, заданным параметром path. C++: int _rmdir(const char *_path) Modula-2: _rmdir(VAR _path: ARRAY OF CHAR): INTEGER; Clarion: Rmdir(*cstring),short,raw,name('_rmdir')
getcurdir	C++: int _getcurdir(int drive, char *_buf) Modula-2: _getcurdir(drive: INTEGER; VAR _buf: ARRAY OF CHAR): INTEGER; Clarion: GetCurDir(short, *cstring), short, raw, name('_getcurdir')
chdir	C++: int _chdir(const char *_path) Modula-2: _chdir(VAR _path: ARRAY OF CHAR): INTEGER; Clarion: ChDir(*cstring),short,raw,name('_chdir')
getdisk	C++: int _getdisk(void) Modula-2: _getdisk(): INTEGER; Clarion: GetDisk(),short,name('_getdisk')
setdisk	C++: int _setdisk(int _drive)  Modula-2: _setdisk(_drive: INTEGER): INTEGER; Clarion: SetDisk(short),short,name('_setdisk')

# Содержание

## ЧАСТЬ I

### ЯЗЫК ПРОГРАММИРОВАНИЯ CLARION

#### **ПРЕДИСЛОВИЕ ..... 1**

#### **Глава I Язык программирования CLARION ..... 3**

##### ***Clarion-как язык программирования ..... 3***

*Событийно-управляемые программы ..... 3*

*Привет Windows ..... 4*

*Окно Hello Windows с объектами в нем ..... 9*

*Добавление процедуры ..... 13*

*Добавление функции ..... 15*

*Первый шаг к реальному приложению - добавим меню ..... 17*

*Еще шаг в реальный мир - добавим чтение файла и форму для заполнения полей ..... 19*

*Что делать дальше ..... 35*

#### **Глава 2 Структура программы ..... 37**

##### ***Структурное программирование ..... 37***

*ПРОЦЕДУРЫ и ФУНКЦИИ ..... 37*

*Объявление локальных данных, размещаемых в стеке ..... 37*

*Структура MAP программы ..... 38*

*МОДУЛЬ ..... 39*

*MEMBER-модуль ..... 39*

*MAP в MEMBER-модуле ..... 40*

*Структуры MODULE в структуре MAP MEMBER-модуля ..... 41*

*Резюме ..... 43*

#### **Объектно-ориентированное программирование (ООП) .... 45**

##### ***Введение в объекты ..... 45***

*Что такое объекты? ..... 45*

*Почему объекты? ..... 45*

*Что делает объект объектом? ..... 46*

##### ***Объектно-ориентированное расширение языка Clarion ..... 49***

Структура CLASS - инкапсуляция .....	49
Порожденные классы - наследование .....	54
Виртуальные методы - полиморфизм .....	58
Резюме .....	60
<b>РАЗРАБОТКА БАЗЫ ДАННЫХ .....</b>	<b>61</b>
<b>Разработка базы данных .....</b>	<b>61</b>
Разработка Реляционной базы данных .....	61
Взаимосвязь файлов .....	63
Переложение теории на язык Clarion .....	65
Целостность ссылок .....	67
Резюме .....	73
<b>Работа с файлами .....</b>	<b>75</b>
Методы доступа .....	75
Ключи и Индексы .....	75
Последовательный доступ к файлу .....	77
Произвольный доступ к файлу .....	80
Резюме .....	81
<b>Совместное использование файлов .....</b>	<b>85</b>
Открытие файлов .....	85
Проверка на вмешательство .....	86
Блокировка и освобождение записей .....	92
Блокирование и разблокирование файлов .....	95
Взаимная блокировка .....	97
Резюме .....	99

## ЧАСТЬ II

### РЕСУРСЫ ДЛЯ УГЛУБЛЕННОГО ПРОГРАММИРОВАНИЯ

<b><u>Настройка и расширение среды .....</u></b>	<b><u>104</u></b>
<b><u>Введение .....</u></b>	<b><u>104</u></b>
<b><u>Неизменяемые разделы файла CW20.INI .....</u></b>	<b><u>105</u></b>
User Information (данные о пользователе) .....	105
Paths (пути к каталогам) .....	105
Опции Среды .....	105
Аплеты Clarion .....	105
<b><u>Изменение Среды Разработки Clarion .....</u></b>	<b><u>107</u></b>
Указание определенных пользователем приложений .....	107
Добавление пунктов в меню Clarion .....	108
Добавление пунктов в меню Clarion Setup .....	108
Добавление масок файлов выпадающий список типов файлов .....	109
Добавление карточек-закладок в диалоговые окна New, Open и Pick File .....	110
Задание расширения проектного файла .....	112
Спецификации печати .....	113
<b><u>Установки опций Среды .....</u></b>	<b><u>114</u></b>
Опции автоматического размещения полей .....	114
Опции словаря .....	114
Параметры системы управления проектом .....	116
Опции Генератора программ .....	116
Опции синхронизации словаря .....	118
Опции реестра шаблонов .....	121
Опции форматера окон .....	122
Опции форматера отчетов .....	123
Опции текстового редактора .....	124
Табуляция в редакторе .....	124
<b><u>CLARION как DDE сервер .....</u></b>	<b><u>127</u></b>
<b><u>Введение .....</u></b>	<b><u>127</u></b>
Соединение с Clarion для Windows как с DDE сервером .....	128
Отсоединение от DDE сервера Clarion для Windows .....	129
Экспорт словаря в текстовый (TXD) формат .....	129
Импорт словаря из текстового (TXD) формата .....	131

Экспорт данных из файла .APP в текстовый формат (TXA).....	132
Импорт данных в файл .APP из текстового формата (TXA) .....	133
Загрузка файла описания приложения .....	134
Генерировать текст приложения .....	135
Выполнить проект или приложение .....	136
Выполнение шаблонов-утилит .....	138
Регистрация класса шаблона .....	139
Получение сообщений об ошибках DDE .....	141
Ошибки при динамическом обмене данными с Clarion.....	142

## **Формат файла .TXD ..... 145**

<b>Файлы.TXD: словари Clarion в формате ASCII .....</b>	<b>145</b>
---	------------

<b>Структура файла .TXD .....</b>	<b>146</b>
-----------------------------------	------------

Схема файла .TXD .....	146
------------------------	-----

<b>Разделы файла .TXD .....</b>	<b>148</b>
---------------------------------	------------

[DICTIONARY] .....	148
--------------------	-----

[FILES] .....	149
---------------	-----

[ALIASES] (алиасы) .....	165
--------------------------	-----

[RELATIONS] .....	166
-------------------	-----

[VIEWS] - раздел описания структур VIEW.....	171
--	-----

<b>Общие подразделы .....</b>	<b>172</b>
-------------------------------	------------

[LONGDESC].....	172
-----------------	-----

[USEROPTION] .....	172
--------------------	-----

[SCREENCONTROLS] .....	172
------------------------	-----

[REPORTCONTROLS] .....	173
------------------------	-----

## **Формат файла .TXA ..... 175**

<b>Файлы.TXA: Описание приложения Clarion в формате ASCII .....</b>	<b>175</b>
---	------------

<b>Структура файла.TXA .....</b>	<b>176</b>
----------------------------------	------------

Схема файла .TXA .....	176
------------------------	-----

<b>Разделы файла .TXA .....</b>	<b>178</b>
---------------------------------	------------

[APPLICATION] .....	178
---------------------	-----

[PROGRAM]-[END].....	180
----------------------	-----

[MODULE]-[END] .....	182
----------------------	-----

[PROCEDURE] .....	183
-------------------	-----

<b>Общие подразделы .....</b>	<b>186</b>
-------------------------------	------------

[COMMON] .....	186
----------------	-----

[DATA].....	188
-------------	-----

[FILES] .....	193
---------------	-----

[PROMPTS].....	196
----------------	-----

[EMBED]-[END].....	199
--------------------	-----

[ADDITION] .....	202
<b><u>Система поддержки проекта TOPSPEED .....</u></b>	<b><u>207</u></b>
<b>Введение .....</b>	<b>207</b>
<b>Макросы проектной системы .....</b>	<b>209</b>
Установка значения макроса .....	210
Специальные макросы проектной системы .....	211
<b>Основы компиляции и компоновки .....</b>	<b>213</b>
Установка опций компиляции и компоновки .....	213
Команды компиляции и компоновки .....	215
<b>Условная обработка и управление последовательностью действий</b>	<b>218</b>
Интерфейс пользователя .....	220
Логические выражения .....	220
Redirection - файл .....	222
Команды управления файлами .....	223
Прочие команды .....	225
<b>Прагмы TopSpeed .....</b>	<b>227</b>
Синтаксис прагм .....	227
Классы прагм .....	229
Класс Call #pragmas .....	229
Прагмы Data (Data #pragmas) .....	234
Прагмы класса Check (Check #pragmas) .....	238
Прагмы класса Name .....	239
Прагмы класса Optimize .....	241
Прагмы класса Debug .....	243
Прагмы класса Module .....	244
Прагмы класса Option .....	246
Warn #pragmas .....	247
Прагма класса Project .....	255
Прагмы Link .....	255
Прагмы класса Link_Option .....	256
Прагмы класса Define .....	257
Предопределенные флажки .....	258
<b>Project System Examples .....</b>	<b>259</b>
<b>Модуль определения файлов (.EXP) .....</b>	<b>270</b>
16-Bit .....	270
32-Bit .....	278
<b><u>Программирование на нескольких языках .....</u></b>	<b><u>285</u></b>
<b>Предисловие .....</b>	<b>285</b>
<b>Интеграция компиляторов .....</b>	<b>286</b>

Включение модулей на языках 3-го поколения в проект на языке Clarion .....	286
<b>Разрешение типов данных .....</b>	<b>288</b>
Соответствие типов данных с C и C++ .....	289
Соответствие типов данных с Modula-2 .....	293
Соответствие типов данных с Pascal .....	295
<b>Прототипирование в Clarion функций на языках 3-го поколения ....</b>	<b>298</b>
Типы данных параметров .....	299
Типы возвращаемых значений .....	300
Передаваемые параметры .....	301
Соглашения о вызовах .....	302
Реализация соглашения об именах .....	304
Атрибуты EXTERNAL и DLL .....	306
<b>Вопросы программирования .....</b>	<b>307</b>
Использование библиотек классов C++ .....	307
Инициализация модулей Modula-2 и Pascal .....	309
Резюме .....	310
<b><u>Вызовы API и усложненные файлы ресурсов .....</u></b>	<b><u>313</u></b>
<b>Прототипы и объявления .....</b>	<b>313</b>
Обращения из Clarion к стандартной библиотеке C/C++ .....	313
Обращение из Clarion к API Windows .....	313
Обращение из Modula-2 к Clarion .....	314
Обращение из C/C++ к Clarion .....	314
<b>Обращение к библиотеке исполняющей системы Clarion из C/C++ или Modula-2 .....</b>	<b>315</b>
Структуры и определения типов данных .....	315
Переменные времени выполнения .....	315
Встроенные функции Clarion .....	315
<b>Стандартные функции C, находящиеся в библиотеке исполняющей системы Clarion .....</b>	<b>324</b>
Функции преобразований .....	324
Целочисленная арифметика .....	325
Символьные функции .....	325
Служебные функции .....	326
Строковые функции .....	326
Работа с файлами на низком уровне .....	329