СИСТЕМА
УПРАВЛЕНИЯ
БАЗАМИ
ДАННЫХ

ЛИНТЕР®

ЛИНТЕР БАСТИОН

ЛИНТЕР СТАНДАРТ

Встроенный SQL

НАУЧНО-ПРОИЗВОДСТВЕННОЕ ПРЕДПРИЯТИЕ

РЕЛЭКС

Товарные знаки

РЕЛЭКС™, ЛИНТЕР® являются товарными знаками, принадлежащими АО НПП «Реляционные экспертные системы» (далее по тексту – компания РЕЛЭКС). Прочие названия и обозначения продуктов в документе являются товарными знаками их производителей, продавцов или разработчиков.

Интеллектуальная собственность

О документе

Материал, содержащийся в данном документе, прошел доскональную проверку, но компания РЕЛЭКС не гарантирует, что документ не содержит ошибок и пропусков, поэтому оставляет за собой право в любое время вносить в документ исправления и изменения, пересматривать и обновлять содержащуюся в нем информацию.

Контактные данные

394006, Россия, г. Воронеж, ул. Бахметьева, 2Б.

Тел./факс: (473) 2-711-711, 2-778-333.

e-mail: market@relex.ru.

Техническая поддержка

С целью повышения качества программного продукта ЛИНТЕР и предоставляемых услуг в компании РЕЛЭКС действует автоматизированная система учёта и обработки пользовательских рекламаций. Обо всех обнаруженных недостатках и ошибках в программном продукте и/или документации на него просим сообщать нам в раздел Поддержка на сайте ЛИНТЕР.

Содержание

Предисловие

Назначение документа

Документ содержит описание языка встроенного SQL.

Документ предназначен для СУБД ЛИНТЕР СТАНДАРТ 6.0 сборка 20.1, далее по тексту СУБД ЛИНТЕР.

Для кого предназначен документ

Документ предназначен для программистов, разрабатывающих приложения на языке программирования C/C++ для информационных систем, построенных на основе СУБД ЛИНТЕР.

Дополнительные документы

- СУБД ЛИНТЕР. Справочник по SQL
- СУБД ЛИНТЕР. Сетевые средства
- СУБД ЛИНТЕР. Процедурный язык
- СУБД ЛИНТЕР. Справочник кодов завершения

Общие сведения

Примечание
Поддержка остановлена, использовать не рекомендуется.

Назначение встроенного SQL

Встроенный SQL предназначен для объединения возможностей языка программирования высокого уровня C/C++ с возможностями языка баз данных SQL СУБД ЛИНТЕР. Он позволяет выполнять любой SQL-оператор из прикладной программы. Для этого SQL-операторы непосредственно встраиваются в текст программы на C/C++ в соответствии с синтаксическими правилами встроенного языка. В результате получение исполняемого кода программы распадается на следующие этапы:

1) прекомпиляция с использованием прекомпилятора (препроцессора), входящего в состав СУБД ЛИНТЕР, исходного текста программы (отдельного модуля), содержащего конструкции встроенного SQL. Прекомпилятор заменяет конструкции встроенного SQL либо операторами языка C/C++, либо вызовами соответствующих функций библиотеки прекомпиляторного интерфейса. Результатом прекомпиляции является исходный текст программы, содержащей только конструкции языка C/C++. Так, например, конструкция встроенного SQL

```
EXEC SQL PREPARE ST FROM :Query;
```

будет заменена на

```
PCI_NewVar((char *)&Query,200,1,0,2,(int *) 0);
#define PCI_STAT_TEXT \
":v1;"
PCI_Prepare(DdbPCI_,4096,&PCISt[0],0x0,PCI_STAT_TEXT);
Fill_SQLca((char *)&sqlca);
if (ErrPCI_ > 0 && ErrPCI_ != ErrPCI_NotFound) goto err_read;
#undef PCI_STAT_TEXT
```

2) компилирование полученного текста программы (модуля) стандартным компилятором C/C++, результатом чего будет объектный код программы (модуля). Если программа (модуль) не содержит конструкции встроенного SQL, то они компилируются только компилятором C/C++;

3) компоновка всех объектных модулей программы (создание библиотеки) совместно с библиотекой встроенного SQL (поставляемой в дистрибутиве СУБД ЛИНТЕР) и системными библиотеками, результатом будет исполняемый код программы или библиотека.

Синтаксические правила

Директивы прекомпилятора предназначены для выделения в исходной C/C++ программе предложений, относящихся непосредственно к встроенному SQL. Прекомпилятор встроенного SQL распознает и обрабатывает только выделенные директивами строки (блоки), не затрагивая остальной текст программы.

Для программы со встроенным SQL установлены следующие общие синтаксические правила:

- допускается произвольно смешивать в исходном тексте строки синтаксически полных (законченных) конструкций встроенных SQL-предложений и операторов языка C/C++;
- нельзя в одной строке размещать одновременно предложение встроенного SQL и оператор C/C++;
- предложения встроенного SQL записываются в тексте программы в свободном формате, т.е. их текст можно переносить на следующие строки, разделяя отдельные элементы синтаксической конструкции, за исключением префикса, т.е. конструкция EXEC /n SQL не распознается как префикс предложения встроенного SQL;
- именование переменных встроенного SQL должно соответствовать правилам языка C/C++;
- в предложения встроенного SQL допускается включать комментарии в формате языка C/C++ (/* ... */) в те места, где разрешены пробелы (кроме ключевых префиксов EXEC SQL, EXEC LINTER), например:

```
EXEC SQL SELECT ENAME, SAL
 INTO :emp_name, :salary /* выходные главные переменные */
 FROM EMP
 WHERE DEPTNO = :dept_number;
```

- ключевые слова встроенного SQL-предложения являются регистронезависимыми, т.е. допускается использование нижнего и/или верхнего регистра. Например, указанные ниже SQL-предложения с точки зрения прекомпилятора являются идентичными:

```
EXEC SQL DECLARE BEGIN;
Exec sql declare begin;
Exec SQL Declare Begin;
```

Отличие ограничителей (одинарных и двойных кавычек) прекомпилятора встроенного SQL и компилятора C/C++ приведено в таблице 1.

Таблица 1. Отличие ограничителей прекомпилятора встроенного SQL и компилятора языка C/C++

Ограничитель	Использование в компиляторе C/C++	Использование в прекомпиляторе встроенного SQL
' ' (одинарные кавычки)	Символьный литерал (см. пример 1)	Представление строкового литерала (см. пример 2)
" " (двойные кавычки)	Представление строкового литерала (см. пример 3)	Представление строкового литерала (см. пример 2) и представление в SQL-предложениях идентификаторов объектов БД, содержащих малые английские буквы или большие/малые русские буквы (см. пример 4)

Примеры, показывающие семантические различия одинарных и двойных кавычек:

1) Символьный литерал в C/C++

```
ch = getchar();
switch (ch)
{
case 'U': update(); break;
case 'I': insert(); break;
...
```

2) Именование объектов БД в SQL-предложении и представление строковых литералов

```
EXEC SQL SELECT ENAME, SAL FROM EMP WHERE JOB = 'MANAGER';
EXEC SQL DECLARE ;
NAME char[50];
EXEC SQL END DECLARE ;
NAME="Банк 'Менатеп'"
```

3) Представление строковых литералов

```
printf("\nПлатежное поручение");
```

4) Именование объектов БД и их элементов

```
EXEC SQL CREATE TABLE "Банк_документы" ("название"
 varchar(50), ...);
```

Элементы языка встроенного SQL

Встроенные SQL-предложения

Термин *встроенное SQL-предложение* относится к SQL-предложениям, размещенным в исходном тексте программы на языке программирования C/C++. Так как модуль на языке C/C++ позволяет включать в себя SQL-предложения, то он называется *модулем основного языка*, а язык, на котором он написан, – *основным языком*. Например, модули основного языка C/C++ позволяют встраивать в себя модули встроенного языка (встроенного SQL), SQL-предложения СУБД ЛИНТЕР и директивы прекомпилятора встроенного SQL СУБД ЛИНТЕР.

Исполняемые и декларативные предложения

Встроенный SQL поддерживает все SQL-предложения СУБД ЛИНТЕР и набор дополнительных предложений, которые обеспечивают обмен данными между СУБД и прикладной программой. Существует два типа встроенных SQL-предложений: *исполняемые* и *декларативные*. Исполняемые предложения после прекомпиляции заменяются вызовами функций исполняемой библиотеки встроенного SQL, поэтому они должны размещаться в исполняемой части главной программы.

Декларативные предложения используются для объявления переменных встроенного SQL, создания коммуникационных областей между главной программой и СУБД ЛИНТЕР, а также для описания других объектов встроенного SQL. После прекомпиляции все они заменяются допустимыми переменными и структурами данных языка C/C++, поэтому декларативные предложения должны размещаться в секциях объявлений основного языка или в любом месте, допускаемом синтаксисом основного языка для объявлений переменных.

Предложение встроенного SQL имеет следующий синтаксис:

```
<предложение встроенного SQL>::=
   EXEC SQL <оператор встроенного SQL>;
```

Пример

```
EXEC SQL drop table TEST;
```

Директивы прекомпилятора

Помимо исполняемых и декларативных предложений прекомпилятор встроенного SQL распознает и обрабатывает ряд встроенных предложений, предназначенных непосредственно самому прекомпилятору и управляющих его работой. Директивы прекомпилятора являются чисто декларативными предложениями и после прекомпиляции никаких следов в основной программе не оставляют, поэтому они могут размещаться в произвольном месте исходной программы со встроенным SQL.

Директива прекомпилятора встроенного SQL имеет следующий синтаксис:

```
<директива прекомпилятора>::=
   EXEC LINTER <оператор директивы>
```

Примеры

```
EXEC LINTER Num_Version;
```

```
EXEC LINTER EndIf;
```

Статические и динамические встроенные SQL-предложения

Большинство прикладных программ разрабатывается таким образом, что в момент написания программы разработчик знает, какие объекты БД будут задействованы при реализации алгоритма обработки данных (т.е. какие таблицы и какие именно столбцы этих таблиц будут выбираться из БД, какие транзакции будут выполняться и какие SQL-операторы будут включены в эти транзакции). Поэтому конструкция таких SQL-предложений известна на момент написания прикладной программы и остается неизменной в процессе выполнения, т.е. такие SQL-предложения являются *статическими SQL-предложениями*. Текст статических SQL-предложений может быть явно вставлен в исходный текст C/C++ программы (кроме, возможно, некоторых подставляемых в запрос значений переменных). Поэтому на этапе прекомпиляции статические SQL-предложения проходят не только синтаксическую проверку, но и семантическую (наличие в БД указанной таблицы и столбцов, соответствие типов данных и др.).

В некоторых случаях до начала выполнения программы и, тем более, при ее написании, невозможно предсказать, какое конкретно SQL-предложение должно быть выполнено. Например, в графической утилите «Рабочий стол СУБД ЛИНТЕР» при просмотре объектов базы данных пользователь может выбрать любую таблицу БД. Поэтому при написании программы имя выбираемой таблицы неизвестно, неизвестно также количество столбцов в ней, их имена и тип данных, следовательно, запрос на выборку данных из такой таблицы может быть сконструирован программным способом только в процессе выполнения программы. Подобный тип SQL-предложений называется *динамическими SQL-предложениями*. Естественно, что синтаксическая и семантическая проверка динамического SQL-предложения возможна СУБД ЛИНТЕР только в момент обработки этого запроса при выполнении программы.

Переменные встроенного SQL

Модуль основного языка со встроенным SQL может содержать три вида переменных:

1) общие для предложений встроенного SQL и операторов основного языка переменные. Эти переменные обеспечивают связь, необходимую для обмена данными между пространством имен модуля основного языка и пространством имен СУБД ЛИНТЕР. Они называются переменными основного языка. Переменные основного языка можно использовать одновременно в предложениях встроенного SQL и в C/C++ операторах либо только в предложениях встроенного SQL, либо только в C/C++ операторах;
2) переменные, используемые только в предложениях встроенного SQL (собственные переменные встроенного SQL – курсоры, соединения, дескрипторы, контексты, имена модулей). Тип и структура таких переменных известна только прекомпилятору. После прекомпиляции они заменяются на соответствующие языку C/C++ структуры данных и становятся доступными основной программе;
3) переменные, используемые только в операторах C/C++ (собственные переменные основного языка: скалярные переменные, массивы, структуры и объединения). Этот вид переменных прекомпилятор встроенного SQL не распознает и не обрабатывает. Они используются только основным языком;
4) переменные прекомпилятора всегда локальны относительно модуля основного языка (исключения составляют неявная переменная – соединение по умолчанию

и псевдопеременные SQLCODE, SQLSTATE...). Подробнее см. в подразделе «Многомодульные приложения»;

5) относительно *модуля встроенного SQL* переменные могут быть локальными и глобальными;

6) *глобальными* являются все переменные, описанные с помощью операторов DECLARE или PREPARE вне модулей встроенного языка;

7) *локальными* являются все переменные, описанные внутри модуля встроенного SQL;

8) все глобальные переменные встроенного SQL в одном модуле основного языка должны иметь уникальные имена. В разных модулях основного языка, объединяемых впоследствии в одну программу (библиотеку), глобальные переменные могут иметь одинаковые имена, но в каждом модуле основного языка будет использоваться определенная именно в этом модуле глобальная переменная встроенного языка. Имена переменных встроенного SQL могут совпадать с зарезервированными словами директив языков SQL и C/C++ и переменных основного языка, но должны отличаться от зарезервированных слов встроенного SQL.

Способы обмена данными между приложением и СУБД

Средства встроенного SQL предоставляют два способа обмена данными между программой и СУБД ЛИНТЕР: метод *переменных* основного языка и метод *дескрипторов*. Метод переменных основного языка применим только к статическим SQL-предложениям, для которых количество и тип входных и выходных переменных определены на момент написания программы, поэтому переменные могут быть объявлены в программе и использованы в SQL-предложениях. Метод дескрипторов применяется, как правило, в динамических SQL-предложениях, для которых количество и тип входных и выходных переменных определяются каждый раз только в процессе выполнения программы и заранее неизвестны. Как частный случай метод дескрипторов может быть применен и для статических SQL-предложений, при этом никаких преимуществ, по сравнению с методом переменных основного языка, он не предоставляет.

Переменные основного языка

Переменные основного языка предназначены для обмена данными между программой и СУБД ЛИНТЕР: выбранная в БД информация передается программе через переменные основного языка, с другой стороны, программа загружает в переменные основного языка ту информацию, которая должна быть записана в БД. Переменные основного языка декларируются в секции объявления переменных основного языка встроенного SQL и могут быть использованы, как в предложениях встроенного SQL, так и в обычных операторах основного языка.

Правила объявления переменных

Ниже приведены основные правила и ограничения объявления переменных:

- переменные объявляются в секциях объявлений переменных основного языка;
- не допускается использование структурных типов, а также объявленных ранее нестандартных типов;
- если переменная – массив, то этот массив должен быть одномерным, а его размерность должна задаваться десятичной константой;
- объявление регистровых переменных допустимо, но использование их в предложениях встроенного SQL невозможно (исключая переменные типа char *A и bit *A);
- в одном модуле основного языка может быть несколько секций объявлений. Видимость имен переменных основного языка в директивах встроенного языка подчиняется тем же правилам, что и в предложениях основного языка;
- переменные основного языка можно использовать только на месте значимых выражений SQL-операторов;
- имена переменных основного языка могут совпадать с именами объектов БД (таблиц, столбцов, имен ролей и т.п.);
- имена переменных основного языка могут совпадать с зарезервированными словами встроенного SQL;
- имена переменных основного языка могут совпадать с именами переменных встроенного языка;
- имена переменных основного языка являются, в конечном итоге, именами переменных C/C++, поэтому должны соответствовать синтаксису C/C++;
- с любой переменной основного языка может быть использована индикаторная переменная.

Область видимости переменных

Область видимости переменных основного языка определяется правилами видимости основного языка.

Атрибуты переменных

Синтаксис

```
<атрибуты переменной>::=
```

```
[<наследование>][<класс памяти>][<модификатор>]<тип данных>
[<указатель>] <имя переменной> [<размер переменной>]
[<размер массива переменных>] [<начальное значение>]
```

Наследование

Только для основного языка C++.

Синтаксис

```
<наследование>::= heir
```

Эта конструкция служит для уведомления прекомпилятора о том, что переменная наследуется из родительского класса. После претрансляции переменная не объявляется, но видна во время претрансляции.

Для описания переменных в языке C++ введен модификатор `heir`. Если в секции объявлений переменных некоторая переменная описана с модификатором `heir`, то прекомпилятор считает, что встретилось объявление переменной, унаследованной из родительского класса. В этом случае в прекомпилированном тексте программы объявленная таким образом переменная будет закомментирована, поэтому переменная с модификатором `heir` должна начинаться с новой строки и быть единственной в строке, либо в этой строке должны находиться только переменные с модификатором `heir`. Проверка на наличие переменной с таким именем в родительском классе прекомпилятором не производится.

Пример

```
Exec SQL Begin Declare Section;
char*p; Heir char *q;
int i;
Exec SQL End Declare Section;
```

После прекомпиляции он будет выглядеть так:

```
/* Exec SQL Begin Declare Section; */
/* char*P; Heir char *Q; */
   int i;
/* Exec SQL End Declare Section; */
```

Класс памяти

Синтаксис

```
<класс памяти>::= {auto | extern | static | register}
```

При описании переменной основного языка можно указать прекомпилятору встроенного SQL, какой тип памяти следует использовать для этой переменной. Назначение класса памяти должно соответствовать правилам используемого компилятора языка C/C++ (например, класс памяти auto можно задавать переменным, используемым только внутри некоторого блока операторов языка C/C++).

Прекомпилятор позволяет объявлять внешними (extern) строковые и битовые переменные с указанием или без указания размера массива. Например:


```
extern char protocol[15],
extern char msg[].
```

Рекомендуется всегда указывать размер массива, чтобы исключить выход за его границы. Например, если некоторая переменная объявлена как внешняя без указания размера массива в одном исходном модуле, а определена в другом, то прекомпилятор не знает ее реальную длину. Во время загрузки в эту переменную данных при выполнении select или fetch-запроса в первом модуле можем получить выход за границу массива. Если внешняя переменная объявлена без указания длины, прекомпилятор выдаст предупреждение.

Класс доступа

Синтаксис

```
<класс доступа>::= {const | volatile}
```

Переменная любого типа может быть объявлена как немодифицируемая. Это достигается добавлением атрибута `const` к спецификатору типа. Объекты с типом `const` представляют собой данные, используемые только для чтения, т.е. этой переменной не может быть присвоено новое значение. Если ключевое слово `const` стоит перед объявлением массива, это приведет к тому, что каждый элемент массива будет немодифицируемым, т.е. значение ему может быть присвоено только один раз.

Атрибут `volatile` обозначает, что переменная может модифицироваться способом, неизвестным компилятору. Компилятор не оптимизирует код с участием этой переменной.

Примеры

```
const double A=2.128E-2;
```

```
const int B[1]={286};
```

Модификатор

Синтаксис

```
<модификатор>::= {signed | unsigned}
```

Атрибуты `signed` и `unsigned` необязательны. Они указывают, как интерпретируется старший бит объявляемой переменной, т.е. если указан `unsigned`, то старший бит интерпретируется как часть числа, в противном случае – как знаковый. При отсутствии атрибута `unsigned` целая переменная считается знаковой.

Примечание
Для переменных типа CHAR модификатор может использоваться при объявлении как одиночных символов, так и строковых литералов (массива символов).

Пример

```
unsigned int n;
unsigned int b;
```

```
int c; (подразумевается signed int c );
```

Именование переменных

Имена переменных основного языка (идентификаторы), в том числе индикаторных переменных (подраздел Индикаторные переменные основного языка) могут содержать:

- прописные и строчные буквы латинского алфавита;
- цифры;
- знак подчеркивания.

Имя должно начинаться с буквы.

Идентификатор имени может быть любой длины, но реально прекомпилятор учитывает только первые 31 символ.

Примечание
Если компилятор языка C/C++ имеет ограничение на длину идентификаторов, то при именовании переменных основного языка следует учитывать это ограничение.

Типы данных переменных

Числовые типы данных

Синтаксис

```
<числовой тип>::= {long | short |  float |  double |  int |
 decimal | bool}
```

Названия типов регистронезависимы. После претрансляции заменяются на `long`, `short`, `float`, `double`, `int`, `decimal`, `unsigned char` соответственно.

Выделяют следующие характеристики числовых типов (см. таблицу 2).

Таблица 2. Характеристики числовых типов

Тип данных	Характеристика	Примечание
long	Как правило, 32-битный знаковый целый	Реальная размерность зависит от архитектуры (например, на Alpha, Sparc v9 это 64 бита)
short	16-битный знаковый целый	
float	4-байтное вещественное число	
double	8-байтное вещественное число	
int	Как правило, 32-битный знаковый целый	Реальная размерность зависит от ОС (например, MS DOS, PalmOS – 16 бит)
decimal	31-разрядное число с фиксированной точкой	Для использования этого типа данных требуется включить в исходный модуль файл

Тип данных	Характеристика	Примечание
		decimals.h и на этапе сборки программы подключить библиотеку специальных типов данных СУБД ЛИНТЕР
bool	Однобайтный, принимает значения 0 (FALSE) или 1 (TRUE)	

Строковые типы данных

Фиксированный строковый тип

Назначение

Задает символьную строку фиксированной длины.

Синтаксис

```
<фиксированный строковый тип>::=
 {CHAR <имя><длина>
 |CHAR * <имя>
 |CHAR <имя>=<начальное значение>}
<длина>::='['<целое положительное число>']'
<начальное значение>::=строковый литерал
```

Описание

1) <Длина> задает размер символьной строки (целое положительное число в диапазоне от 1 до 4000).
2) Объявление фиксированной строковой переменной в виде char* применяется для обозначения строк неизвестной длины. Фактический размер строки определяется на этапе привязки переменной по положению NUL-символа в строке.
3) Если символьная строка содержит символ ' (кавычка) или " (двойная кавычка), то он должен быть экранирован символом \ (обратная косая черта).
4) Чтобы продолжить символьную строку на следующую линию, используется символ \ (обратная косая черта) в последней позиции строки.
5) Строка может содержать символы в любых кодировках, поддерживаемых СУБД ЛИНТЕР.

Примечание
Список установленных кодовых страниц находится в системной таблице $$$CHARSET СУБД ЛИНТЕР.

Переменный строковый тип

Назначение

Задает символьную строку переменной длины.

Синтаксис

1 вариант (в стиле СУБД ЛИНТЕР)

```
<переменный строковый тип>::=
{VARCHAR <имя> <длина>[<начальное значение>}]
```

2 вариант (в стиле СУБД Ingres)

```
<переменный строковый тип>::=
VARCHAR struct
{short length;
char  text <длина>;
}<имя>

<длина>::='['<целое положительное число>']'
<начальное значение>::=см. подраздел «Начальное значение»
```

Описание

1) <Длина> задает максимальный размер символьной строки (целое положительное число в диапазоне от 1 до 4000).
2) Строка, задающая <начальное значение>, должна целиком лежать на одной строке исходного текста (исключая собственно строковый литерал, задающий значение поля text).
3) Если строка, задающая начальное значение, содержит символ ' (кавычка) или " (двойная кавычка), то он экранируется символом \ (обратная косая черта).
4) Чтобы продолжить символьную строку на следующую линию, используется символ \ (обратная косая черта) в последней позиции строки.
5) Строка может содержать символы в любых кодировках, поддерживаемых СУБД ЛИНТЕР.
6) В случае объявления переменных в стиле СУБД ЛИНТЕР переменная типа VARCHAR[n] после претрансляции представляется в модуле основного языка в виде:

```
struct
{
short len;
char arr[n];
}<имя>;
```

7) В случае объявления переменных в стиле СУБД Ingres переменная типа VARCHAR[n] после претрансляции представляется как

```
struct
{ short length;
char text[n];
 }<имя>
```

Пример

```
VARCHAR name_month[8] ;
```


Битовые типы данных

Фиксированный битовый тип

Назначение

Задает строку байт фиксированной длины.

Синтаксис

```
<фиксированный битовый тип>::=
{bit <имя><длина> | bit * <имя>| bit <имя>=<начальное значение>}
<длина>::='['<целое положительное число>']'
<начальное значение>::=байтовый литерал
```

Описание

1) <Длина> задает размер битовой строки (целое положительное число в диапазоне от 1 до 4000).
2) Объявление битовой переменной фиксированной длины в виде `bit *A` применяется для обозначения строк неизвестной длины, в этом случае битовая переменная должна содержать NUL-символ в последней позиции (признак конца строки).
3) Строка, задающая <начальное значение>, может содержать любые символы (включая NUL-символ для переменной с явно заданной длиной).
4) Чтобы продолжить строку на следующей линии, используется символ \ (обратная косая черта) в последней позиции строки.

Пример

```
bit bit_string = '\x07\x56\xff\x00' /* строка байт с неявно
 заданной длиной */
```

Переменный битовый тип

Назначение

Задает строку байт переменной длины.

Синтаксис

1 вариант (в стиле СУБД ЛИНТЕР)

```
<переменный битовый тип>::=
{ VARBIT <имя><длина>
| VARBIT <имя>=<начальное значение>}
```

2 вариант (в стиле СУБД Ingres)

```
<переменный битовый тип>::=
 VARBIT struct
{ short length;
char  text<длина>;
```

```
}<имя>

<длина>::='['<целое положительное число>']'
<начальное значение>::=см. подраздел «Начальное значение»
```

Описание

1) <Длина> задает максимальный размер символьной строки (целое положительное число в диапазоне от 1 до 4000).
2) Строка, задающая <начальное значение>, должна целиком лежать на одной строке исходного текста (исключая собственно строковый литерал, задающий значение поля text).
3) Чтобы продолжить строку на следующей линии, используется символ \ (обратная косая черта) в последней позиции строки.
4) В случае объявления переменных в стиле СУБД ЛИНТЕР переменная типа VARBIT[n] представляется после претрансляции в виде

```
struct
{
short len;
char arr[n];
}<имя>;
```

5) В случае объявления переменных в стиле СУБД Ingres переменная типа VARBIT[n] представляется после претрансляции в виде

```
struct
{ short length;
char text[n];
 }<имя>
```

Примечание
Максимальная длина байтового столбца переменной длины в СУБД ЛИНТЕР 4000 байт.

Тип дата-время

Назначение

Задает тип данных «дата-время».

Синтаксис

```
<дата-время тип>::= {date | datetime | timestamp}
```

<Дата-время тип> предназначен для получения/передачи полей типа DATE СУБД ЛИНТЕР без преобразования их к строковым переменным.

Примечание
Для работы с переменными типа DATETIME должна использоваться библиотека DATETIME СУБД ЛИНТЕР.

Массивы данных

Назначение

Массивы данных используются для пакетного приема/передачи данных (см. подраздел «Исполнение заранее подготовленных предложений»).

Массивы с фиксированной длиной элементов

Синтаксис

```
<массив с фиксированной длиной элементов>::=
  <числовой массив> | <строковый массив>
<числовой массив>::=
{long | short | float |double | int | decimal | date}
 <имя><размер>
<строковый массив>::=
  <строковый массив фиксированной длины>
  | <строковый массив переменной длины>
<строковый массив фиксированной длины>::=
{char | bit}<имя> <размер1> [<размер2>] <длина элемента>
<имя>::=переменная основного языка
<размер1>::='['<значение>']'
<размер2>::='['<значение>']'
<длина элемента>::='['<значение>']'
<значение>::=целое положительное число
```

Массивы с переменной длиной элементов

С переменной длиной элементов массива могут быть только строковые и байтовые массивы.

Объявление этих массивов возможно двумя способами: в стиле СУБД ЛИНТЕР или в стиле СУБД Ingres. Смешение стилей в одном исходном тексте модуля основного языка не допускается.

1 способ (в стиле СУБД ЛИНТЕР)

```
<массив с переменной длиной элементов>::=
{VARCHAR | VARBIT}
<имя> <размер1> [<размер2>] <длина элемента>
```

2 способ (в стиле СУБД Ingres)

```
{[VARCHAR]|VARBIT} struct
{
short length;
char text<длина элемента>;
} <имя><размер1> [<размер2>];
```

```
<имя>::=переменная основного языка
<размер1>::='['<значение>']'
<размер2>::='['<значение>']'
<длина элемента>::='['<значение>']'
<значение>::= целое положительное число
```

Описание

1) <Длина элемента> задает максимальный размер элемента массива.
2) VARCHAR, VARBIT – тип данных элемента массива. Объявление этих массивов возможно двумя способами: в стиле СУБД ЛИНТЕР или в стиле СУБД Ingres. Смешение стилей в одном исходном тексте модуля основного языка не допускается.
3) В случае объявления в стиле СУБД ЛИНТЕР массивы транслируются в виде:
   - VARCHAR[n]][m]:

```
struct
 { short len;
  char arr[m];
 } <имя>[n];
```

   - VARBIT [n][m]:

```
struct
 { short len;
 unsigned char arr[m];
 } <имя>[n];
```

4) В случае объявления в стиле СУБД Ingres массивы транслируются в виде:
   - VARCHAR[n]][m]:

```
struct
 { short length;
 char text[m];
 }<имя>[n]
```

   - VARBIT [n][m]:

```
struct
 { short length;
 unsigned char text[m];
 }<имя>[n]
```

Специальные типы данных

Назначение

Специальные типы данных используются для поддержки возможностей встроенного SQL (их использование описано в соответствующих разделах документа).

Синтаксис

```
<переменные встроенного SQL>::={ CONTEXT | DESCRIPTOR |
 CURSOR }<идентификатор>
```


Описание

1) CONTEXT – контекстная переменная, не может быть массивом.
2) DESCRIPTOR – дескрипторная переменная (фактически указатель на структуру t_sqlda), не может быть массивом.
3) CURSOR – курсорная переменная, не может быть массивом. Курсорная переменная используется для обработки выборки данных, переданной хранимой процедурой.

При попытке объявить массив переменных этих типов на этапе претрансляции будет выдана ошибка E_CANTARR «Недопустим массив».

Указатели

Переменная основного языка типа char или BIT может быть объявлена через механизм указателей, принятый в языке C/C++ . В таком случае размер данных определяется на этапе выполнения как размер строки, содержащейся под этим указателем (до символа NUL).

Передача через указатели значений переменных других типов запрещена. Если это произошло, на этапе претрансляции будет выдана ошибка E_CANTPTR «Недопустим указатель».

Пример

```
char *char_ptr;
int *int_ptr; – ошибка при претрансляции
```

Использование переменной char_ptr в предложении встроенного SQL выполняется стандартным образом:

```
EXEC SQL SELECT char_col INTO :char_ptr FROM ...
```

Начальное значение

Назначение

Спецификация начального значения переменной.

Синтаксис

```
<начальное значение>::=
       {<переменная с фиксированной длиной>
       |<переменная типа VARCHAR>
       |<переменная типа VARBIT>}
<переменная с фиксированной длиной>::=синтаксис C/C++
<переменная типа VARCHAR>::=<строка инициализации>
<переменная типа VARBIT>::=<строка инициализации>
<строка инициализации>::='=' '{'<длина>', '<строка>'}'
```

Описание

Инициализация для переменных типа VARCHAR, VARBIT должна быть задана в одну строку (исключая параметр <строка>). Часть <строки> может быть перенесена на следующую строку (путем указания продолжения строки (символ \ (обратная косая черта) в последней позиции)).

Пример

```
VARCHAR color[10]={8,"белый"}
VARBIT object[15]={5,"ofdd4576cd"}
```

Объявление переменных основного языка

Назначение

Все переменные основного языка, используемые в директивах прекомпилятора, должны быть предварительно описаны в секции объявлений прекомпилятора.

Синтаксис

```
EXEC SQL BEGIN DECLARE SECTION;
<объявление переменной основного языка>
...
<объявление переменной основного языка>
EXEC SQL END DECLARE SECTION;
```

Описание

При использовании переменной основного языка в директиве встроенного SQL ей должно предшествовать двоеточие «:».

Входные и выходные переменные основного языка

Понятия входные и выходные переменные основного языка рассматриваются с точки зрения СУБД. Переменная, в которую СУБД загружает данные при выполнении select-запроса, называется *выходной переменной*. Выходные переменные перечисляются в конструкции INTO встроенного select-предложения, и значение им присваивается только после успешного выполнения этого запроса.

Переменные, которые содержат данные, предназначенные для записи в БД или используемые для формирования условного выражения в SQL-предложениях, называются *входными переменными*. Они используются во встроенных UPDATE, DELETE, INSERT-предложениях и в конструкции WHERE встроенных SQL-предложений. Значение этим переменным присваивается в модуле основного языка перед выполнением соответствующего SQL-предложения.

Индикаторные переменные основного языка

Назначение

С каждой переменной основного языка можно связать ее *индикаторную* переменную. С помощью индикаторных переменных выполняются следующие операции:

- присвоение NULL-значений загружаемой в БД информации (полям таблиц БД) и/или входным переменным хранимых процедур;
- проверка на NULL-значение выбранной из БД информации и/или выходных переменных хранимых процедур;

- проверка усечения данных и/или выходных переменных хранимых процедур при загрузке их в переменные основного языка.

Синтаксис

1) Индикаторная переменная должна иметь тип `short`.
2) Объявление индикаторных переменных, как и переменных основного языка, должно выполняться в секции объявления переменных встроенного SQL.
3) В предложениях встроенного SQL индикаторная переменная (если используется) указывается одновременно со связанной с нею переменной основного языка через символ «двоеточие»:

```
<основная переменная>:<индикаторная переменная>
```

4) Если переменная основного языка – массив, то индикаторная переменная должна быть массивом того же размера.

Описание

1) Взаимосвязь индикаторной и основной переменной (см. таблицу 3).

Таблица 3. Взаимосвязь индикаторной и основной переменной

Значение индикаторной переменной	Значение соответствующей переменной основного языка
нуль	присвоенное значение
не нуль	NULL-значение

Примечание
Если исходный модуль претранслирован в режиме совместимости со старой версией прекомпилятора встроенного SQL (ключ -V), то индикаторная переменная принимает значение 1 для NULL-значений; иначе (по умолчанию) – значение -1.

2) Для передачи NULL-значений необходимо установить индикаторную переменную до исполнения предложения встроенного SQL.
3) Если на этапе выполнения при привязке параметров или при получении результата возникла ошибка ErrPCI_TooLong либо ErrPCI_IncTyp, то индикаторная переменная будет содержать длину данных параметра до усечения (для типов данных CHAR, BIT, VARCHAR, VARBIT).
4) Одна и та же индикаторная переменная может последовательно использоваться с разными переменными основного языка. Однако если она использована в одном и том же SQL-операторе с несколькими переменными одновременно, ее значение не будет соответствовать реально выбранным из БД данным.

Пример

```
SELECT some_col INTO :var:indicator FROM SOME_TBL …
```

Совместимость типов данных

Переменные основного языка могут иметь тип, не совпадающий с типом привязываемого параметра предложения встроенного SQL. В таком случае на этапе выполнения осуществляется преобразование типов (если типы совместимы) от типа основного языка к типу СУБД ЛИНТЕР и наоборот.

Если невозможно выполнить преобразование от типа данных СУБД ЛИНТЕР к типу данных основного языка или наоборот, то на этапе исполнения возникает ошибка `ErrPCI_IncTyp`.

В таблице 4 приведено соответствие типов переменных встроенного языка, типов переменных основного языка (получаемых после претрансляции) и типов данных СУБД ЛИНТЕР, между которыми не требуется преобразование типов при исполнении.

Таблица 4. Соответствие типов данных

Тип данных встроенного языка	Тип данных основного языка	Тип СУБД ЛИНТЕР	Примечание
short	short	SMALLINT	16 бит
unsigned short	unsigned short	SMALLINT	16 бит
int	int	INTEGER	32 бита. В DOS и PalmOS преобразуется в 16 бит
unsigned int	unsigned int	INTEGER	32 бита. В DOS и PalmOS преобразуется в 16 бит
long	long	INTEGER	32 бита. В 64-битных архитектурах (например, Alpha, Sparc v9) преобразуется в 64 бита
unsigned long	unsigned long	INTEGER	32 бита. В 64-битных архитектурах (например, Alpha, Sparc v9) преобразуется в 64 бита
float	float	REAL	4 байта
double	double	DOUBLE	8 байт
decimal	decimal (p,s)	dec (decimal)	Десятичное с фиксированной точкой (30,10)
char	char	CHAR	
char[1]	char[1]	CHAR	
char[n]	char[n]	CHAR(n)	
char*	char*	CHAR()	Заканчивается двоичным нулем
varchar[n]	VARCHAR_PCI(n)	VARCHAR(n)	Символьная строка переменной длины с 2-байтным полем длины
bit	unsigned char	BYTE	
bit[1]	unsigned char [1]	BYTE	
bit[n]	unsigned char [n]	BYTE(n)	
bit*	unsigned char*	BYTE()	Заканчивается двоичным нулем
varbit[n]	VARBIT_PCI(n)	VARBYTE(n)	Байтовая строка переменной длины с 2-байтным полем длины
date	timestamp[2]	DATE	

Тип данных встроенного языка	Тип данных основного языка	Тип СУБД ЛИНТЕР	Примечание
bool	unsigned char	BOOLEAN	unsigned char
		BLOB	В привязанную переменную char[24] возвращается заголовок BLOB
		EXTFILE	Не поддерживается

Объекты встроенного SQL

Коммуникационная область

Назначение

Получение статуса исполнения операторов встроенного SQL в процессе выполнения программы осуществляется через специальную коммуникационную область обмена (см. также подраздел «Информация о результате исполнения запроса»).

Синтаксис

```
<коммуникационная область>::=
   EXEC SQL INCLUDE SQLCA;
```

Данный оператор является неисполняемым и должен размещаться в области неисполняемых операторов до начала исполняемых операторов встроенного SQL (обычно в разделе объявления include-файлов).

Он помещает в претранслированный текст некоторые объявления типов данных, необходимые для компиляции основного модуля, и объявляет структуру данных sqlca.

По умолчанию структура sqlca импортируется из библиотеки PCL (библиотека процедур для сборки кода, полученного в результате работы претранслятора PCI) (подробнее см. «Многомодульные приложения»).

Структура коммуникационной области:

```
struct sqlca
{
char sqlcaid[8];
LONG sqlabc;
LONG sqlcode;
struct
{
unsigned short sqlerrml;
char sqlerrmc[70];
} sqlerrm;
char sqlerrp[8];
LONG sqlerrd[6];
char sqlwarn[8];
char sqlext[8];
}
```

Описание

Описание полей структуры `sqlca` приведено в таблице 5.

Таблица 5. Описание структуры коммуникационной области

Поле	Описание
sqlcaid	8-символьное поле содержит строку 'SQLCA', которая идентифицирует структуру sqlca. Это поле идентифицирует область памяти и используется, как правило, при работе с отладчиком программы (зарезервировано)

Поле	Описание
sqlabc	Содержит длину в байтах структуры SQLCA (зарезервировано)
sqlcode	Результат обработки SQL-запроса: • 0 – нормальное завершение; • 100 – нет данных; • < 0 – код завершения СУБД ЛИНТЕР
sqlerrml	Зарезервировано
sqlerrmc	Зарезервировано
sqlerrp	Зарезервировано
sqlerrd	sqlerrd[1] – код завершения (дубликат поля sqlcode); sqlerrd[2] – количество строк, действительно обработанных при выполнении запроса (для INSERT, UPDATE и DELETE)
sqlwarn	Тип выданного диагностического сообщения при исполнении оператора:'W' – предупреждение, иначе ' '
sqlext	Зарезервировано

Псевдопеременные встроенного языка

Список псевдопеременных встроенного SQL в таблице 6.

Таблица 6. Список псевдопеременных встроенного SQL

Имя	Тип	Описание
SQLCODE	DWORD	Числовое значение кода завершения последнего выполненного оператора встроенного SQL: • 0 – нет ошибки; • <0 – ошибки; • >0 – предупреждение.
SQLSTATE	CHAR(8)	Описание кода завершения или предупреждения (тестовая строка).
ErrPCI_	DWORD	Код завершения, возвращенный оператором встроенного языка: • 0 – нет ошибки; • <0 – ошибка; • >0 – предупреждение. **Примечание** Рекомендуется не использовать.
CntPCI_	DWORD	Число записей, задействованных при обработке последнего выполненного оператора встроенного SQL.
RowsPCI_	DWORD	Текущий номер выбранной записи.
RowIdPCI_	DWORD	RowId последней обработанной записи.

Имя	Тип	Описание
TxtPCI_	CHAR*	Текст последнего выполненного предложения SQL. Только для чтения.
LenPCI_	DWORD	Размер текущей порции BLOB-данных (добавленной или полученной из BLOB-столбца).

Информация о результате исполнения запроса

Назначение

Получить информацию о текущем состоянии последнего выполненного встроенного SQL-запроса.

Синтаксис

```
<состояние запроса>::=
       EXEC SQL INQUIRE_SQL (:<переменная>) =
       {DBMSERROR
        |ENDQUERY
        |ERRORNO
        |ERRORTEXT
        |ERRORTYPE
        |QUERYTEXT
        |ROWCOUNT};
```

Описание

Оператор предоставляет значение запрошенного атрибута последнего выполненного встроенного SQL-запроса.

Тип данных <переменной> должен соответствовать типу данных запрашиваемого атрибута или приводиться к нему (таблица 7).

Таблица 7. Характеристики атрибутов

Имя атрибута	Описание
DBMSERROR	Код завершения последнего выполненного предложения встроенного SQL (значение поля ErrPCI_ структуры sqlca)
ENDQUERY	Признак конца данных исполняемого select-запроса (или цикла FETCH): 1 – выбраны все данные (т.е. данных больше нет)
ERRORNO	Код завершения, возвращенный СУБД ЛИНТЕР
ERRORTEXT	Строка, содержащая расшифровку кода завершения СУБД ЛИНТЕР. В БД СУБД ЛИНТЕР должна быть загружена таблица ERRORS.
ERRORTYPE	Источник порождения кода завершения: • "generic" – библиотека PCL; • "dbmserror" – СУБД ЛИНТЕР
QUERYTEXT	Текст последнего выполненного предложения SQL (значение поля TxtPCI_ структуры sqlca)
ROWCOUNT	Число записей, задействованных при выполнении предложения SQL (значение поля CntPCI_ структуры sqlca)

Модули встроенного языка

Назначение

Операторы объявления начала и окончания модуля встроенного языка предназначены для ограничения области видимости переменных встроенного языка и переменных основного языка, объявленных в секциях деклараций внутри модуля.

Синтаксис

```
<начало модуля>::= EXEC SQL MODULE <имя модуля>;
<конец модуля>::= EXEC SQL END MODULE <имя модуля>;
<имя модуля> – имя переменной встроенного SQL
```

Модуль встроенного SQL имеет следующую структуру:

```
<модуль>::=
  <начало модуля>
  <предложения встроенного языка и/или основного языка>
  . . . .
  <конец модуля>
```

Описание

1) Все переменные встроенного языка, объявленные внутри модуля, являются локальными переменными и видны только в пределах данного модуля.
2) Все переменные основного языка, объявленные в секциях деклараций внутри модуля, являются локальными.
3) Все локальные переменные встроенного языка одного типа в одном модуле должны иметь уникальные имена. Допускается использование этих же имен в других модулях встроенного языка. Если в модуле встроенного языка декларируется переменная с именем, совпадающим с именем глобальной переменной, то в этом модуле используется локальная переменная с таким именем.
4) Вложенность модулей встроенного SQL не допускается.
5) Декларации начала и конца модуля должны находиться в одном и том же блоке основного языка.
6) Каждый модуль должен иметь уникальное имя.

Пример

```
void f(char*query){
strcpy(Query,query);
EXEC SQL MODULE module1;
EXEC SQL BEGIN DECLARE SECTION;
char Query[512];
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE ST FROM :Query;
EXEC SQL DECLARE CR CURSOR_PCI FOR ST;
EXEC SQL OPEN CR;
EXEC SQL UPDATE example SET name='Name' WHERE
```

```
CURRENT OF CR;
EXEC SQL CLOSE CR;
EXEC SQL END MODULE module1;
}
```

Примечание
Следует отличать модули встроенного языка (прекомпилятора) от модулей основного языка.

Переменные встроенного SQL

Переменная типа «соединение»

Назначение

Оператор декларирует наличие переменной типа «соединение» с указанным именем.

Синтаксис

```
<переменная типа 'соединение'>::=
EXEC SQL DECLARE <имя соединения> DATABASE;
<имя соединения>::= <идентификатор> | <строковый литерал>
```

Описание

1) Оператор DECLARE DATABASE является декларативным, поэтому может находиться вне блоков основного языка.
2) В процессе работы пользовательская программа может использовать доступ к нескольким БД. Доступ к конкретной БД выполняется после предварительного *соединения* программы с БД. Каждому соединению можно присвоить уникальное логическое имя (имя БД) для того, чтобы при последующих запросах к этой БД ссылаться уже только на имя соединения, не указывая каждый раз полную информацию, необходимую для соединения. Для этих целей средства встроенного SQL включают тип данных «соединение».
3) Имя соединения известно только программе и не имеет ничего общего с реальным именем БД.
4) Объявленная переменная типа «соединение» (DATABASE) может использоваться в операторе CONNECT и последующих выполняемых операторах для идентификации логической связи с заданной БД.
5) Эта переменная может быть использована в операторах CONNECT и в конструкции AT <имя соединения>.
6) Использование данного оператора для объявления переменной не является необходимым, переменная неявно объявляется оператором CONNECT.

Примечания

1. Имя переменной типа «соединение» может совпадать с именем переменной основного языка.
2. Неявное объявление переменной типа «соединение» выполняется при использовании его в операторе CONNECT.
3. В том случае, когда используется единственное соединение с БД, логическая связь с сервером БД может быть установлена без объявления переменной типа «соединение» (см. использование оператора CONNECT без конструкции AT <имя соединения>).

Пример

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL DECLARE DB1 DATABASE; /* объявление вне блоков основного
 языка */
```

```
void f()
{
/* имя соединения задано идентификатором встроенного языка (case
 insensitive) */
EXEC SQL DECLARE DB2 DATABASE; /* имя соединения задано строковым
 литералом (case sensitive) */
EXEC SQL DECLARE 'DataBase Sale' DATABASE;
…
}
```

Переменная типа «предложение»

Назначение

Оператор декларирует наличие переменной типа «предложение» с указанным именем.

Переменная типа «предложение» позволяет ссылаться на предложения (STATEMENTS) встроенного языка, которые нужно многократно выполнять в процессе работы программы, возможно, с различными значениями входных и выходных параметров.

Предложение предварительно подготавливается к исполнению с помощью оператора PREPARE встроенного языка и затем может выполняться с различными значениями входных и выходных параметров с помощью оператора EXECUTE встроенного языка.

Т.к. имя предложения должно быть известно прекомпилятору, когда он встречает директиву EXECUTE, необходимо объявить переменную типа «предложение» до его обработки прекомпилятором. Неявно переменная типа «предложение» объявляется оператором PREPARE.

Синтаксис

```
<переменная типа «предложение»>::=
EXEC SQL DECLARE <имя предложения> STATEMENT;
<имя предложения>::= <идентификатор> | <строковый литерал>
```

Описание

1) Оператор DECLARE STATEMENT декларативный, поэтому он может находиться вне блоков основного языка.
2) Переменная типа «предложение» может быть использована в операторах PREPARE, EXECUTE.
3) Объявление переменной типа «предложение» не является обязательным, имя предложения неявно создается при исполнении оператора PREPARE.
4) Предложение встроенного SQL не может быть предложением создания или исполнения хранимой процедуры. Для этого существуют специальные операторы встроенного SQL CREATE PROCEDURE, ALTER PROCEDURE.
5) Проверка семантики предложения производится только на этапе выполнения. Исключением является оператор CURRENT OF <имя курсора>, семантика которого проверяется при непосредственном задании предложения. Поэтому, если предложение задано с помощью конструкции <строка>|<главная переменная>, то оно не может содержать CURRENT OF <имя курсора>.

Примечание
<Имя предложения> неявно объявляется при использовании неименованного SQL-оператора в директиве прекомпилятора PREPARE. В некоторых случаях (директива CURSOR FOR) создается неименованное предложение (как правило, в тех случаях, когда к данному SQL-оператору необходимо обратиться только один раз).

Пример

```
EXEC SQL INCLUDE SQLCA; /* декларация предложения вне блока
 основного языка */
EXEC SQL DECLARE ST1 STATEMENT;
void f()
{
EXEC SQL DECLARE ST2 STATEMENT; /* имя предложения задано
 идентификатором встроенного языка (case insensitive) */
EXEC SQL DECLARE 'stmnt_bank' STATEMENT; /* имя предложения задано
 строковым литералом (case sensitive) */
…
}
```

Переменные-параметры

Назначение

Исполняемые предложения встроенного SQL могут содержать параметры (см. документ «СУБД ЛИНТЕР. Справочник по SQL»).

Синтаксис

Параметры бывают 2-х типов: именованные и неименованные.

```
<именованный параметр>::=
:<переменная основного языка>[:<индикаторная переменная>]

<неименованный параметр>::=?
```

Описание

1) Если именованный параметр появляется в разбираемых на этапе претрансляции конструкциях (явно заданные предложения SQL, USING, INTO), его имя должно быть действительным идентификатором переменной основного языка. Если нет переменной основного языка с таким именем, на этапе претрансляции генерируется код завершения «Неопределенное имя».
2) На этапе исполнения:
   - не требуется при каждом исполнении оператора EXECUTE (после PREPARE) повторно связывать именованные операторы с их значениями;
   - можно отменять существующие и устанавливать новые связи между именованными параметрами и их значениями (см. USING, INTO).
3) Неименованный параметр, встречающийся в предложении SQL, требует явного задания фактического параметра на этапе исполнения предложения (см.

конструкции USING, INTO и операторы динамического SQL: EXECUTE USING SQL DESCRIPTOR, OPEN USING SQL DESCRIPTOR, FETCH USING SQL DESCRIPTOR, SET DESCRIPTOR, GET DESCRIPTOR).

Подготовка предложения к выполнению

Назначение

Подготовка предложения SQL к выполнению.

Синтаксис

```
<подготовка предложения>::=
EXEC SQL [ AT<переменная типа «соединение»> ]
  PREPARE <переменная типа «предложение»>
  FROM <предложение>;
  <предложение>::=
  {<предложение SQL>
  |<строковый литерал>
  |<переменная основного языка>}
```

Описание

1) <Переменная типа «предложение»> не обязательно должна быть ранее объявлена в операторе DECLARE STATEMENT.
2) <Предложение SQL> может быть любым исполняемым предложением СУБД ЛИНТЕР. Оно может содержать входные и выходные именованные и неименованные параметры. После выполнения PREPARE именованные параметры будут привязаны к предложению, и станет возможным его дальнейшее выполнение без повторной привязки параметров (т.е. без выполнения операторов USING, INTO).
3) Если <предложение> представлено <строковым литералом> или <переменной основного языка>, то его текст неизвестен во время прекомпиляции. Для работы с такими предложениями используется динамический SQL.
4) В результате выполнения оператора PREPARE с указанной переменной встроенного языка типа «предложение» связываются заданный текст предложения SQL и набор параметров, подготавливая таким образом предложение к дальнейшему выполнению.
5) Подготовленное предложение далее может быть выполнено либо с помощью оператора EXECUTE, либо (если для него объявлен курсор) с помощью комбинации операторов OPEN, FETCH и CLOSE, при этом во фразах USING и INTO можно задавать входные и выходные переменные.
6) Необходимо учитывать, что привязанные переменные основного языка должны быть видны в момент исполнения предложения. Т.е. если секция деклараций переменных основного языка содержится в некотором блоке основного языка, то и оператор исполнения предложения, содержащего привязанные переменные, должен содержаться в том же блоке.

Пример

```
Void f()
{
```

```
EXEC SQL MODULE M1;
EXEC SQL BEGIN DECLARE SECTION;
char*Name = “Vasia”; /* строка задана без явного указания длины */
char*Fname = “Pupkin”;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE ST STATEMENT; /* объявили предложение
 (необязательно) */
EXEC SQL PREPARE ST FROM update person set phone='7779888' where
 name= :Name and firstnam=:Fname; /* подготовили предложение к
 выполнению */
EXEC SQL EXECUTE ST; /* выполнили предложение, используя
 привязанные оператором PREPARE параметры */
EXEC SQL END MODULE M1;
}
```

Доступ к БД

Установление связи с СУБД

Назначение

Перед тем, как начать работать с БД, пользовательская программа обязана выполнить процедуру связи с СУБД и регистрации. Для этого используется оператор CONNECT, устанавливающий соединение с сервером СУБД.

Синтаксис

```
<соединение с СУБД>::=
EXEC SQL CONNECT [ <режим канала>]
:<имя пользователя> [ IDENTIFIED BY :<пароль >]
[AT <переменная типа «соединение» >] [USING : <имя сервера>] ;
<режим канала> : = EXCLUSIVE | OPTIMISTIC | {SHARED | AUTOCOMMIT }
<имя пользователя> :: = <переменная основного языка>
<пароль >::= <переменная основного языка>
<имя сервера >:: =<переменная основного языка>
```

Описание

1) Оператор CONNECT должен быть первым исполняемым SQL-оператором встроенного языка, т.е. все другие SQL-операторы могут физически, но не логически предшествовать оператору CONNECT.
2) <Режим канала> задает способ обработки транзакций в данном соединении. По умолчанию используется режим EXCLUSIVE.
3) <Имя пользователя> – переменная типа char. Задает пользователя, под именем которого будет выполняться доступ к СУБД по данному каналу (соединению). Имя должно содержать не более 18 символов.
4) <Пароль> – переменная типа char. Задает пароль пользователя, указанного в переменной <имя пользователя>. Длина пароля не должна превышать 66 символов. Пароль может указываться совместно с именем пользователя в переменной <имя пользователя> через косую черту (/). Тогда параметр <пароль> должен отсутствовать.

Примечание
Параметр IDENTIFIED BY: <пароль> не задается, если пользователь с именем <имя пользователя> зарегистрирован в БД без пароля.

5) Значение переменных <пароль> и <имя пользователя> должно быть определено до выполнения оператора CONNECT.
6) Программа обязательно должна проверять результат соединения с СУБД (например, посмотреть значение переменной SQLCODE), т.к. в случае неудачного соединения с СУБД дальнейшая ее работа не имеет смысла.
7) <Имя сервера> – переменная типа char, объявленная в секции EXEC SQL DECLARE. Она задает имя ЛИНТЕР-сервера, где находится БД, с которой требуется установить соединение (см. документ «СУБД ЛИНТЕР. Сетевые средства»). Имя ЛИНТЕР-сервера не должно содержать более 8 символов. Если параметр <имя

сервера> не задан, соединение осуществляется с активной БД по умолчанию на локальном сервере (если таковая есть) либо на удаленном ЛИНТЕР-сервере по умолчанию (удаленным ЛИНТЕР-сервером является первый сервер в списке ЛИНТЕР-серверов в файле сетевой конфигурации) (см. документ «СУБД ЛИНТЕР. Сетевые средства»).

8) <Имя соединения> – переменная встроенного языка, необъявленная или объявленная в директиве встроенного языка DECLARE DATABASE, либо символьная константа. Задает имя переменной, связываемой с открываемым каналом соединения с СУБД. В дальнейшем, при необходимости работы с каналом, на него можно ссылаться по этому имени. К одной БД может быть установлено несколько соединений. Если параметр <имя соединения> опущен, можно открыть только один неименованный канал соединения с СУБД.

9) Программа может установить одновременно несколько соединений как с одной, так и с несколькими БД (локальной или удаленной). При этом значение имени соединения в конструкции [AT <переменная типа «соединение»>] для каждого соединения должно быть уникальным (таким образом, в программе может использоваться только один оператор соединения без конструкции [AT <переменная типа «соединение»>]).

10) Для получения информации о ЛИНТЕР-сервере используется оператор EXEC LINTER GET SERVER (см. «Получение характеристик ЛИНТЕР-сервера»).

Примечание
Режим OPTIMISTIC устарел (использовать не рекомендуется).

Отсоединение от СУБД

Назначение

После того, как программа закончила работу с БД по некоторому соединению, необходимо проинформировать СУБД о прекращении доступа к БД по этому соединению и закрыть его. В таком случае используется оператор отсоединения от СУБД.

Оператор служит для закрытия соединения, открытого ранее оператором CONNECT.

Синтаксис

```
<отсоединение от СУБД>::=
EXEC SQL DISCONNECT [ { <имя соединения> | ALL | CURRENT} ];
```

Описание

1) Параметр оператора <имя соединения> задает ранее открытое (установленное) соединение с СУБД. Значение параметра может быть определено переменной типа «соединение» либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, закрывается неименованное соединение с СУБД.

Примечание
Если пользовательская программа завершает свою работу без явного закрытия соединения, то среда исполнения встроенного SQL автоматически выполняет неявное закрытие для всех активных соединений (и их подканалов) в режиме AUTOCOMMIT.

2) `ALL` – закрытие всех открытых ранее соединений.
3) `CURRENT` – закрытие соединения по умолчанию.
4) Перед закрытием соединения по нему выполняется команда `ROLLBACK`.

Транзакции

Транзакция начинается с первого выполняемого SQL-оператора (кроме CONNECT) в пользовательской программе. Когда текущая транзакция заканчивается, следующая начинается с очередного исполняемого SQL-оператора обработки данных.

Декларативные операторы не являются элементами транзакций, поэтому к ним не применяются операции фиксирования или отката.

Операции определения данных всегда выполняются в режиме AUTOCOMMIT, т.е. откат этих операций не поддерживается. Например, если был выполнен оператор CREATE TABLE или ALTER TABLE, то отказаться от созданной таблицы или модификации ее структуры можно только путем явного удаления таблицы оператором DROP TABLE или (если это допустимо) последующей повторной модификацией.

Транзакция заканчивается одним из следующих способов:

- выполнение операции COMMIT либо ROLLBACK, с или без опции RELEASE. Эти операторы фиксируют изменения в БД или выполняют откат к предыдущему состоянию;
- выполнение любого SQL-оператора определения данных (например, ALTER, CREATE, GRANT), которые инициируют автоматическое подтверждение транзакции (COMMIT) перед своим выполнением;
- транзакция автоматически завершается при аварийном завершении пользовательского приложения. В этом случае ядро выполняет операцию ROLLBACK;
- транзакция автоматически завершается при аварийном завершении ядра СУБД ЛИНТЕР (например, при сбое оборудования или отказе операционной системы). В этом случае при рестарте ядра СУБД незавершенная (прерванная) транзакция также выполняет откат.

Фиксация транзакции

Назначение

Для сохранения изменений, произведенных в БД в процессе выполнения транзакции, служит операция подтверждения транзакции.

Синтаксис

```
<подтверждение транзакции>::=
EXEC SQL [AT <имя соединения> ] COMMIT [WORK|RELEASE];
```

Описание

1) <Имя соединения> задает ранее открытое соединение. Значение параметра может быть задано переменной типа «соединение» либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, фиксация транзакции выполняется для неименованного канала. При выполнении оператора осуществляется фиксация изменений в БД как по указанному соединению, так и по всем активным дочерним каналам (курсорам).
2) Если задан режим WORK, то после сохранения изменений работа с БД по данному соединению может быть продолжена.

3) Если задан режим RELEASE, то после сохранения изменений соединение с СУБД закрывается.

4) По умолчанию применяется режим WORK. Для отсоединения от СУБД рекомендуется явно использовать оператор DISCONNECT.

Примечание
Запрос на фиксацию транзакции имеет смысл задавать только в режимах OPTIMISTIC, EXCLUSIVE, т.к. в режиме AUTOCOMMIT изменения сразу фиксируются в БД.

Откат транзакции

Назначение

Отказ от произведенных в БД изменений в процессе выполнения транзакции.

Синтаксис

```
<откат транзакции>::=
   EXEC SQL [AT <имя соединения> ] ROLLBACK [WORK|RELEASE];
```

Описание

1) <Имя соединения> задает ранее открытое соединение. Значение параметра может быть задано переменной типа «соединение» либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, откат транзакции выполняется для неименованного соединения. Оператор выполняет откат транзакции как по соединению, так и по всем активным дочерним каналам (курсорам).

2) Если задан режим WORK, то после отката изменений работа с БД по данному соединению может быть продолжена.

3) Если задан режим RELEASE, то после отката изменений соединение с СУБД закрывается.

4) По умолчанию применяется режим WORK. Для отсоединения от СУБД рекомендуется явно использовать оператор DISCONNECT.

Примечание
Откат транзакции имеет смысл задавать только в режимах OPTIMISTIC, EXCLUSIVE, т.к. в режиме AUTOCOMMIT изменения сразу фиксируются в БД.

Выполнение некурсорных SQL-запросов

Все предложения встроенного SQL, подлежащие выполнению, можно разделить на следующие группы:

1) предложения с формальными параметрами – в этом случае в текст SQL-запроса вставлены имена переменных встроенного языка, значения которым присваиваются в основной программе перед выполнением запроса, либо эти переменные получают значения при выборке данных из БД (в случае select-запросов). Количество и тип параметров остается фиксированным и не изменяется в процессе выполнения программы. Частным случаем таких предложений являются статические SQL-запросы, когда значения переменных запроса известны до выполнения программы и остаются неизменными в течение всего времени ее работы. Конструкции такого рода используются, как правило, для многократного выполнения предопределенных запросов с различными значениями параметров;
2) динамические предложения – в этом случае ни сам текст, ни количество и тип формальных параметров SQL-запроса до начала выполнения программы неизвестны. Конструирование запроса происходит в ходе выполнения программы. Динамические предложения используются, как правило, в сложных самонастраивающихся программах.

Операторы исполнения предложений встроенного языка

Непосредственное исполнение предложений

Назначение

Исполнение предложения SQL без предварительной подготовки его оператором PREPARE (т.е. без создания переменной типа «предложение»).

Синтаксис

```
<непосредственное выполнение предложения>::=
  EXEC SQL [AT <имя соединения>][FOR :<число выполнений>]
  [ EXECUTE IMMEDIATE] ] <предложение>;
  <число выполнений>::= <переменная основного языка>
  <предложение>::=
  <предложение SQL>
  | <строковый литерал>
  |<переменная основного языка>
```

Описание

1) <Имя соединения> задает соединение, по которому должно выполняться предложение. Соединение должно быть предварительно открыто. Значение параметра может быть задано переменной типа «соединение» либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, предложение выполняется для неименованного соединения.
2) Конструкция FOR задает число повторений предложения. Переменная основного языка, задающая параметр цикла <число выполнений>, должна иметь тип данных short (значение в диапазоне от 1 до 32767). Если переменная другого типа, при претрансляции будет выдана ошибка «Ожидалась целая переменная». Циклическое выполнение предложения предназначено для случая, когда часть параметров предложения задана массивами. Размеры массивов при этом должны быть не меньше числа повторений, иначе на этапе исполнения будет выдана ошибка ErrPCI_IncDim. Если конструкция FOR не указана, а параметры – переменные-массивы основного языка, то в качестве числа повторений берется размер массива. При этом размеры массивов входных и выходных переменных должны быть одинаковыми, иначе на этапе исполнения фиксируется ошибка ErrPCI_IncDim. Максимальный размер массива – 32767 элементов.
3) Конструкция EXECUTE IMMEDIATE является необязательной и может использоваться для совместимости программ, написанных на встроенном SQL СУБД ЛИНТЕР, с программами других СУБД (в частности, Oracle). Кроме того, в случае задания числа повторений необходимо явно указать EXECUTE IMMEDIATE, если текст исполняемого предложения SQL содержится в переменной основного языка.
4) Параметр <предложение> задает предложение встроенного SQL, которое должно быть выполнено. Если предложение SQL записано непосредственно, оно может содержать именованные параметры. Привязка их будет выполнена на этапе исполнения оператора. Предложением SQL может быть и select-запрос с указанием выходных параметров (INTO <список переменных основного языка>).
5) Если предложение задано строковым литералом или переменной основного языка, оно не должно содержать параметров, т.к. препроцессору неизвестен его текст.

6) Перед выполнением предложения входные переменные должны иметь необходимые значения; выходным переменным значение присваивается после успешного выполнения предложения. Если в качестве входной переменной используется массив, то при N-ном выполнении в текст предложения подставляется значение N-го элемента переменного массива.
7) Если в качестве выходной переменной используется массив, то при N-ном выполнении предложения выбранное значение заносится в N-ный элемент массива.
8) В одном предложении входные переменные могут одновременно включать скалярные переменные и массивы разной размерности, выходные переменные должны иметь всегда одинаковую структуру – быть либо только скалярными, либо только массивами. При циклическом выполнении предложения для входных скалярных переменных каждый раз подставляется одно и то же значение, для массивов каждый раз подставляется следующий элемент массива.
9) Для select-предложений возможны 2 варианта выполнения:
   - если все входные переменные – скалярные, а все выходные – массивы, то выполняется последовательная выборка записей с помощью одного и того же предложения в выходные массивы; если же все выходные переменные – скалярные, то предложение выполняется один раз;
   - если хотя бы одна входная переменная – массив, то все выходные переменные должны также быть массивами, и каждый раз выполняется выборка одной записи с помощью нового оператора.

Примеры

1) Однократное исполнение явно заданного предложения SQL без параметров

```
EXEC SQL CREATE TABLE test (name CHAR(10), numb INTEGER);
```

2)

```
EXEC SQL BEGIN DECLARE SECTION;

  int I;
  int inp_arr1[10];
  int inp_arr2[12];
  int out_arr1[10];
  int out_arr2[12];
  char *query;
EXEC SQL END DECLARE SECTION;


EXEC SQL SELECT I FROM NUMS INTO :i WHERE S='A'; /* выборка в
 скалярную переменную i. Запрос выполнится один раз */
…
i = 10;
EXEC SQL for :i UPDATE NUMS SET I=:inp_arr1 WHERE S=:inp_arr2; /*
 запрос выполнится 10 раз, каждый раз используя очередные элементы
 inp_arr1 и inp_arr2 */

EXEC SQL UPDATE NUMS SET I=:inp_arr1 WHERE S=:inp_arr2; /* запрос
 вернет ошибку ErrPCI_IncDim */
```

```
EXEC SQL SELECT I, s INTO :out_arr1, out_arr2 FROM NUMS WHERE
 S=:i; /* запрос вернет ошибку ErrPCI_IncDim */

I = 5;
EXEC SQL SELECT I INTO :out_arr1 FROM NUMS WHERE S=:i; /* запрос
 выполнится 10 раз используя каждый раз I= 5 */

query = “DROP TABLE NUM;”;
EXEC SQL :query; /* запрос выполнится, используя текст,
 содержащийся в query. */

i=5;
EXEC SQL FOR :I EXECUTE IMMEDIATE :query; /* запрос выполнится
 один раз, используя текст, содержащийся в query.
Т.е. параметр FOR в этом случае игнорируется. */
EXEC SQL “DROP TABLE NUN;”; /* запрос, заданный строковым
 литералом, выполнится один раз */
```

3)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#if defined(VXWORKS)
#include "vxstart.h"
#endif

#ifdef _DOS_
#include <conio.h>
#endif

EXEC LINTER IFDEF SQL;
EXEC SQL INCLUDE SQLCA;
EXEC LINTER ENDIF;

EXEC SQL BEGIN DECLARE SECTION;
  char *user = "SYSTEM";
  char *pswd = "MANAGER8";
  short i = 10;
  int tid = 0;
  int arr_i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  char buf[51] = {0};
EXEC SQL END DECLARE SECTION;

#if defined(VXWORKS)
MainStart(pcc_sample, 1024*32, UninitLinterClient)
```



```
#else
int main()
#endif
{
   printf("Создание соединения...");
   EXEC SQL WHENEVER SQLERROR GOTO not_conn;
   EXEC SQL CONNECT AUTOCOMMIT :user IDENTIFIED BY :pswd;
   printf("готово.\n");

   printf("Удаление таблицы\n");
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL EXECUTE IMMEDIATE drop table PRAC13_T1;

   printf("Создание таблицы...");
   EXEC SQL WHENEVER SQLERROR GOTO not_created;
   EXEC SQL EXECUTE IMMEDIATE create table PRAC13_T1(id int, name
 char(50));
   printf("готово.\n");

   printf("Вставка данных...");
   EXEC SQL WHENEVER SQLERROR GOTO not_inserted;
   EXEC SQL FOR :i EXECUTE IMMEDIATE insert into PRAC13_T1 (id,
 name) values (:arr_i, 'aaa');
   printf("done.\n");

   printf("Выборка count(*)...");
   EXEC SQL WHENEVER SQLERROR GOTO not_selected;
   EXEC SQL SELECT count(*) from PRAC13_T1 into :i;
   printf("В таблице %d записей.\n", i);

   printf("\nВведите Id >");
   scanf("%d", &tid);
   printf("\nId=%d\n", tid);
   EXEC SQL EXECUTE IMMEDIATE SELECT NAME into :buf from PRAC13_T1
 where ID = :tid;
   printf("(Id, Name) == (%d, %s).\n", i, buf);

   return 0;

not_conn:
  printf("Не удалось создать соединение\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_inserted:
  printf("Не удалось вставить данные\n");
  printf("Код ошибки: %d\n", ErrPCI_);
```

```
  return 1;
not_created:
  printf("Не удалось создать таблицу\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_selected:
  printf("Не удалось создать выборку\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
}
```

Исполнение заранее подготовленных предложений

Назначение

Выполнение подготовленного предложения SQL.

Синтаксис

```
<выполнение подготовленного предложения>::=
EXEC SQL [AT <имя соединения>] [FOR :<число выполнений>]
EXECUTE <предложение>
        <спецификация выходных параметров>
        <спецификация входных параметров>
        <спецификация выходных параметров>::= INTO <список
 параметров>
        <спецификация входных параметров>::= USING <список
 параметров>
<список параметров>::=
   <параметры-аргументы> | <параметр-дескриптор>
<параметры-аргументы>::=
   : <переменная основного языка> [ { , : <переменная основного
 языка> … } ]
<параметр-дескриптор>::=
   SQL DESCRIPTOR <имя дескриптора>
```

Описание

1) <Имя соединения> задает соединение, по которому должно выполняться предложение. Соединение должно быть предварительно открыто. Значение параметра может быть задано переменной типа «соединение» либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, предложение выполняется для неименованного канала.
2) Конструкция FOR задает число повторений предложения. Переменная основного языка, задающая параметр цикла <число выполнений>, должна иметь тип данных short (значение в диапазоне от 1 до 32767). Если переменная другого типа, при претрансляции будет выдана ошибка «Ожидалась целая переменная». Циклическое выполнение предложения предназначено для того случая, когда часть

параметров предложения задана массивами. Размеры массивов при этом должны быть не меньше числа повторений, иначе на этапе исполнения будет выдана ошибка ErrPCI_IncDim. Если конструкция FOR не указана, а параметры – переменные-массивы основного языка, то в качестве числа повторений берется размер массива. При этом размеры массивов входных и выходных переменных должны быть одинаковыми, иначе на этапе исполнения фиксируется ошибка ErrPCI_IncDim. Максимальный размер массива – 32767 элементов.

3) Параметр <предложение> задает переменную типа «предложение» встроенного SQL, которое должно быть выполнено. Переменная, соответствующая параметру <предложение>, должна быть предварительно инициализирована (подготовлена) с помощью оператора PREPARE. Текст предложения может содержать как именованные, так и неименованные формальные параметры. Запрещается задавать имена объектов БД с помощью параметров.
4) Параметр USING <список параметров> устанавливает соответствие между входными формальными и фактическими параметрами предложения. <Список параметров> должен содержать перечисленные через запятую переменные основного языка (возможно, с их индикаторными переменными). Имя каждой переменной в списке должно предваряться двоеточием. В процессе выполнения предложения каждый формальный параметр получает тип и значение соответствующего ему фактического параметра. Соответствие между формальными и фактическими параметрами является порядковым, т.е. первому формальному параметру соответствует первая переменная в конструкции USING, второму формальному параметру – вторая переменная и т.д. Если в процессе выполнения предложения обнаружится, что список формальных параметров длиннее списка переменных, фиксируется ошибка выполнения; если же список фактических параметров превышает список формальных, то лишние фактические параметры игнорируются. Ошибка выполнения предложения в этом случае не фиксируется. Если текст предложения SQL не содержит неименованных параметров, то конструкцию USING можно не использовать, т.к. переменные основного языка уже привязаны оператором PREPARE.
5) Параметр INTO <список параметров> устанавливает соответствие между выходными формальными и фактическими параметрами предложения.
6) Если предложение подготовлено оператором PREPARE, то оно может быть исполнено оператором EXECUTE встроенного языка один и более раз.
7) Чтобы использовать дескриптор для задания параметров и получения результата, его необходимо определить автоматически с помощью оператора SQL DESCRIBE или вручную с помощью оператора SET DESCRIPTOR.
8) Дескриптор надо привязать к переменным основного языка (SET DESCRIPTOR).
9) Если дескриптор задан переменной типа «дескриптор» (DESCRIPTOR) основного языка, то перед любыми действиями его надо инициализировать оператором ALLOCATE DESCRIPTOR.

Примеры

1)

```
EXEC SQL BEGIN DECLARE SECTION;
Char *query;
Int I;
Char s[32];
Char Make[10][32];
EXEC SQL END DECLARE SECTION;
```

```
Exec sql prepape ST_0 from SELECT MAKE INTO :make FROM AUTO;
Exec sql execute ST_0; /* запрос выберет первые 10 записей из
 таблицы AUTO */

Exec sql execute ST using INTO :s; /* запрос выберет первую запись
 из таблицы MAKE */

Query = "SELECT I FROM NUM WHERE S=?;";
exec sql prepare st from :query; /* предложение задано переменной
 основного языка и его значение неизвестно во время
претрансляции */


exec sql execute st using :s INTO :i; /* явно задаем входные и
 выходные параметры при исполнении предложения */
```

2)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#if defined(VXWORKS)
#include "vxstart.h"
#endif

#ifdef _DOS_
#include <conio.h>
#endif

EXEC LINTER IFDEF SQL;
EXEC SQL INCLUDE SQLCA;
EXEC LINTER ENDIF;

EXEC SQL BEGIN DECLARE SECTION;
  char *user = "SYSTEM";
  char *pswd = "MANAGER8";
  int i = 0;
  int n1 = 0, n2 = 0;
  int tid = 0;
  int rowCount = 0;
  int arr_i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  char tname[51] = {0};
EXEC SQL END DECLARE SECTION;

#if defined(VXWORKS)
MainStart(pcc_sample, 1024*32, UninitLinterClient)
#else
```



```
int main()
#endif
{
   char c = 0;
   int I = 5;

   printf("Создание соединения...");
   EXEC SQL WHENEVER SQLERROR GOTO not_conn;
   EXEC SQL CONNECT AUTOCOMMIT :user IDENTIFIED BY :pswd;
   printf("готово.\n");

   printf("Удаление таблицы\n");
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL EXECUTE IMMEDIATE drop table PRAC13_T1;

   printf("Создание таблицы...");
   EXEC SQL WHENEVER SQLERROR GOTO not_created;
   EXEC SQL EXECUTE IMMEDIATE create table PRAC13_T1(id int, name
 char(50));
   printf("готово.\n");

   EXEC SQL WHENEVER SQLERROR GOTO not_st_opened1;
   EXEC SQL PREPARE ST_INS FROM INSERT INTO PRAC13_T1 (id, name)
 VALUES (:tid, :tname);
   EXEC SQL WHENEVER SQLERROR GOTO not_st_opened2;
   EXEC SQL PREPARE ST_SEL FROM SELECT id, name into :tid, :tname
 from (SELECT id, name, rownum as rn FROM PRAC13_T1)
   WHERE Rn = :i;

   /* вставка данных с использованием оператора*/
   EXEC SQL WHENEVER SQLERROR GOTO not_inserted;
   printf("Insert data...");
   do
   {
     printf("\nВведите значения полей Id и Name через пробел \n");
     scanf("%d %s", &tid, tname);
     EXEC SQL EXECUTE ST_INS USING :tid, :tname;
     printf("Данные вставлены в таблицу. Еще? (Y/N) > ");
     c = 0;
     while (!isalpha(c))
       scanf("%c", &c);
   }
   while (c == 'Y' || c == 'y');

   printf("Выборка count(*)...");
   EXEC SQL WHENEVER SQLERROR GOTO not_selected;
```

```
   EXEC SQL SELECT count(*) from PRAC13_T1 into :rowCount;
   printf("В таблице %d записей.\n", rowCount);

   printf("Введите через пробел диапазон выводимых записей n1
 n2\n");
   scanf("%d %d", &n1, &n2);
   if (n1 < 1) n1 = 1;
   if (n2 > rowCount) n2 = rowCount;
   printf("Содержимое таблицы PRAC13_T1. От %d до %d\n", n1, n2);
   for (i = n1; i <= n2; i++)
   {
     EXEC SQL EXECUTE ST_SEL USING :i into :tid, :tname;
     printf("\t%d.\t%d\t%s\n", i, tid, tname);
   }

   return 0;

not_conn:
  printf("Не удалось создать соединение\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_inserted:
  printf("Не удалось вставить данные\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_created:
  printf("Не удалось создать таблицу\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_selected:
  printf("Не удалось создать выборку\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_st_opened1:
  printf("Оператор 1 не открыт\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_st_opened2:
  printf("Оператор 2 не открыт\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
}
```


Работа с курсорами

Понятие курсора

Все select-запросы можно разделить на 2 группы:

1) select-запрос всегда возвращает строго одну запись выборки данных;
2) select-запрос возвращает, в общем случае, множество записей выборки данных.

Если select-запрос к БД возвращает множественную выборку данных, то одним из способов работы с таким запросом является выборка данных в массивы переменных. Этот метод можно использовать, когда объем выборки незначителен (или запрос сконструирован таким образом, что возвращает данные небольшими порциями), количество выбираемых записей заранее известно, а выбираемые записи данных имеют небольшую длину. В этом случае прямой доступ к любой возвращенной записи выборки данных выполняется по индексированному номеру массива. В ситуации, когда возвращается достаточно большое количество записей выборки данных или общее количество записей выборки данных неизвестно, и невозможна порционная выборка записей, метод массивов становится неприемлемым. Для работы с такими запросами встроенный SQL имеет специальную структуру данных, называемую курсор. Курсор, по существу, представляет собой имя многострочного запроса или имя таблицы, получаемой в результате выборки данных. Подобно тому, как по имени и индексу массива можно получить доступ к любой загруженной в массив записи, по имени курсора и по индексу записи в курсоре можно получить доступ к любой записи множественной выборки. Однако если при использовании метода массивов процесс выборки данных из БД не может быть прерван или приостановлен до полного заполнения массива или исчерпания выборки, то курсор можно использовать для обработки отдельных записей выборки данных возвращаемого запросом набора. Записи обрабатываются по одной в том порядке, в котором СУБД ЛИНТЕР возвращает их из БД. Процесс выборки может быть прерван в любой момент времени (например, когда необходимые данные найдены). Механизм курсоров позволяет также реализовать одновременное выполнение нескольких SQL-запросов, возвращающих множественные выборки данных. Одновременное выполнение понимается как поочередное выполнение одного курсора внутри другого. Так как работа с множественными выборками записей при использовании курсоров может представлять собой довольно длительный процесс, то он управляется пользовательской программой, а не СУБД. Для этого встроенный SQL имеет следующие команды управления курсором:

- начать работу с курсором (открыть курсор);
- выбрать указанную запись из множественной выборки данных;
- закончить работу с курсором (закрыть курсор).

Имена курсоров могут задаваться идентификаторами, строками встроенного языка и переменными основного языка.

В последнем случае такие курсоры называются динамическими курсорами и имеют несколько особенностей использования, о которых будет сказано далее.

Имя курсорной переменной, заданной идентификатором встроенного языка или строкой встроенного языка, может использоваться в конструкции `CURRENT OF <имя курсора>` при подготовке предложения встроенного языка (`PREPARE`) в том случае, если предложение SQL задается непосредственно. Это связано с тем, что курсоры встроенного SQL имеют имена, не совпадающие с именами физических курсоров СУБД ЛИНТЕР.

Курсор с именем <имя курсора> должен быть описан ранее в операторе DECLARE CURSOR, открыт оператором OPEN и позиционирован оператором FETCH.

Статические курсоры

Объявление статического курсора

Назначение

Вид курсора и способ объявления курсора зависят от того, каким методом будет формироваться выборка данных: с помощью select-запроса (статический курсор) или с помощью хранимой процедуры, возвращающей в качестве ответа тип данных «курсор» (динамический курсор).

Синтаксис

```
<объявление статического курсора>::=
EXEC SQL DECLARE <имя курсора>
  [SCROLL | INSENSITIVE ] CURSOR [FOR <предложение>];
<имя курсора>::= <идентификатор> | <строковый литерал>
<предложение>::= <переменная встроенного языка типа «предложение»>
  |<непосредственно предложение SQL>
  | <строковый литерал>
  |<переменная основного языка>
```

Описание

1) Конструкция SCROLL | INSENSITIVE введена для совместимости синтаксиса со стандартом SQL93. В данной версии встроенного SQL не обрабатывается. Если такая конструкция задается, появляется предупреждение «Опция не поддерживается».
2) <Предложение> задает предложение, потенциально возвращающее множественную выборку данных. <Предложение> может быть представлено 3 способами:
   - ссылкой на имя предложения (вариант <имя предложения>), содержащего запрос. В этом случае данное предложение должно быть предварительно подготовлено к выполнению с помощью оператора PREPARE, например:

```
EXEC SQL BEGIN DECLARE SECTION;
char*query="SELECT * FROM test WHERE name LIKE :v1 ORDER BY 1");;
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE ST FROM :query;
EXEC SQL DECLARE CR CURSOR_PCI FOR ST; /* только декларация
 курсора Оператор не влечет за собой создания кода */
```

   - непосредственно (вариант <предложение SQL> или <строковый литерал>), например:

```
EXEC SQL DECLARE CR CURSOR FOR "select name from person"; /*
 создаем курсор из предложения SQL, заданного строковым
литералом */
EXEC SQL DECLARE CR CURSOR_PCI FOR select name from person where
 id>:ident; /* создаем курсор из предложения SQL,
заданного непосредственно и привязываем переменную ident */
```


В этом случае оператор влечет за собой создание кода после претрансляции.

Примечания

1. В двух последних случаях автоматически выполняется оператор PREPARE для неименованного предложения. Явное выполнение оператора PREPARE нужно в том случае, если SQL-запросу необходимо присвоить имя для возможной ссылки на этот запрос в других операторах программы.
2. В качестве SQL-запроса при объявлении курсора можно задать любой запрос (не обязательно возвращающий множественную выборку данных, например, DROP TABLE). В этом случае запрос будет отработан, и при его нормальном выполнении сформируется код завершения «Нет данных».

3) Объявленный таким способом курсор может использоваться только для select-запросов.
4) Со статическим курсором допускается выполнять следующие действия:
   - открыть курсор посредством оператора OPEN;
   - произвести выборку с помощью оператора FETCH;
   - использовать в позиционных SQL-запросах (с конструкцией CURRENT OF …);
   - работать с BLOB-данными посредством операторов {ADD | GET | CLEAR} BLOB;
   - закрыть курсор с помощью оператора CLOSE.

Открытие статического курсора

Назначение

Открытие статического курсора.

Синтаксис

```
<открытие статического курсора>::=
EXEC SQL [AT <имя соединения>]
OPEN <имя курсора>
[EXCLUSIVE | OPTIMISTIC | {SHARED| AUTOCOMMIT}]
[ {<использование переменных основного языка>
| <использование дескриптора>} ]
<использование переменных основного языка>::=
{ USING } < переменная основного языка> [ {, < переменная
 основного языка>} …}
<использование дескриптора>::=
{ USING } SQL DESCRIPTOR <имя дескриптора>
```

Описание

1) <Имя соединения> задает соединение, по которому открывается курсор. Значение параметра может быть задано переменной типа «соединение» (DATABASE) либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, курсор открывается для неименованного соединения. При открытии курсора на

соединении создается дочерний канал, которому присваивается имя курсора (первые 18 символов).

2) Если модуль претранслирован в режиме совместимости со старой версией РСС (ключ -V претранслятора), то помимо собственно открытия дочернего канала оператор OPEN исполняет предложение SQL, иначе исполнение предложения откладывается до первого вызова оператора `FETCH` в данном курсоре.
3) `SHARED` является синонимом `AUTOCOMMIT`.
4) Конструкция `[EXCLUSIVE | OPTIMISTIC | {SHARED| AUTOCOMMIT}]` задает способ обработки транзакций в канале, соответствующем курсору. Если конструкция не задана, по умолчанию используется режим соединения. Не рекомендуется задавать различающиеся режимы работы курсора и соединения.

Примечание
Режим OPTIMISTIC устарел (использовать не рекомендуется).

5) Параметр `USING <список переменных>` используется для привязки в момент открытия курсора других входных переменных (отличающихся от тех, которые были установлены при объявлении предложения SQL).
6) Параметр `USING DESCRIPTOR <имя дескриптора>` используется для открытия динамического курсора. Дескриптор должен быть предварительно объявлен с помощью директивы SQL `DESCRIBE` и инициализирован (см. SET DESCRIPTOR).

Закрытие статического курсора

Назначение

После окончания работы с записями, возвращаемыми курсором, необходимо выполнить операцию закрытия курсора. При закрытии курсора выполняется обработка незавершенной транзакции (если она осталась в курсоре) в соответствии с заданным режимом, высвобождаются системные ресурсы СУБД ЛИНТЕР (занимаемый курсором канал, рабочая область ответа и др.).

Синтаксис

```
EXEC SQL CLOSE :<имя курсора>;
```

Описание

1) Параметр `<имя курсора>` задает курсор, который должен быть закрыт.
2) Оператор закрывает курсор `<имя курсора>`, открытый ранее оператором OPEN или возвращенный хранимой процедурой.

Примечание
В группе связанных курсорных операций `OPEN`, `FETCH`, `CLOSE` должно использоваться либо `<имя курсора>`, либо `<имя курсорной переменной>`.

Динамические курсоры

Объявление динамического курсора

Назначение

Для объявления динамического курсора (курсорной переменной) служит тип данных `CURSOR` встроенного SQL, т.е. в секции объявления переменных встроенного SQL

необходимо объявить переменную типа «курсор» (CURSOR). После объявления курсорная переменная может быть использована в качестве курсора встроенного языка. Область видимости этой переменной такая же, как любой другой переменной основного языка.

Синтаксис

```
<курсорная переменная>::=
   CURSOR <имя курсорной переменной>
```

Описание

1) С динамическим курсором допускается выполнять следующие операции:
   - выделять память для курсора посредством оператора ALLOCATE;
   - открывать курсор путем исполнения хранимой процедуры, возвращающей тип данных «курсор». Явного оператора открытия курсора (типа открытия статического курсора) нет. Динамический курсор всегда открывается неявно путем присвоения ему вызова хранимой процедуры, возвращающей тип данных «курсор»;
   - производить выборку с помощью оператора FETCH;
   - работать с BLOB-данными посредством операторов {ADD | GET | CLEAR} BLOB;
   - завершать работу курсора с помощью оператора CLOSE;
   - освобождать ранее выделенную под курсор память посредством оператора DEALLOCATE.
2) Динамический курсор нельзя применять в динамических SQL-предложениях. Он может использоваться только с операторами ALLOCATE, FETCH, FREE и CLOSE встроенного SQL. Конструкция DECLARE CURSOR не применяется к курсорной переменной.
3) Нельзя применять FETCH к закрытой курсорной переменной.
4) Нельзя применять FETCH к курсорной переменной без выделения ей памяти.
5) Курсорная переменная не может быть загружена в таблицу БД.
6) Курсорная переменная аналогична другим переменным основного языка и имеет область видимости в соответствии с правилами видимости переменных языка C/C ++. Она может передаваться в качестве параметра функций, может быть объявлена внешней функцией в исходном файле. Можно определять функции, которые возвращают курсорные переменные или указатель (ссылку) на нее.

Пример

```
EXEC SQL BEGIN DECLARE SECTION;
   CURSOR proc_desc;
EXEC SQL END DECLARE SECTION;
```

Выделение памяти под динамический курсор

Назначение

Задав имя курсорной переменной, мы лишь уведомили прекомпилятор о существовании ее в программе. Для того чтобы с ней работать, необходимо выделить под нее память. Это осуществляется с помощью оператора встроенного языка ALLOCATE.

Синтаксис

```
EXEC SQL ALLOCATE:<имя курсорной переменной>;
```

Описание

1) <Имя курсорной переменной> должно быть ранее объявлено в секции описаний переменных основного языка.
2) Выделение памяти под курсор не требует обращения к серверу БД.
3) Запрашиваемая под курсор память выделяется на этапе выполнения приложения. Если оператор выделения памяти под курсор содержит ошибку (например, ссылку на необъявленную курсорную переменную), она будет выявлена на этапе прекомпиляции. Выделение памяти под курсоры может породить проблему утечки памяти, т.к. выделенная под курсор память освобождается не при закрытии курсора, а только при явном указании освободить память (см. DEALLOCATE).

Открытие динамического курсора

Открытие динамического курсора происходит неявно через вызов хранимой процедуры, возвращающей тип данных «курсор».

Закрытие динамического курсора

Назначение

После окончания работы с записями, возвращаемыми курсором, необходимо выполнить операцию закрытия курсора. При закрытии курсора выполняется обработка незавершенной транзакции (если она осталась в канале) в соответствии с заданным режимом, высвобождаются системные ресурсы СУБД ЛИНТЕР (занимаемый курсором канал, рабочая область ответа и др.).

Синтаксис

```
<закрытие динамического курсора>::=
EXEC SQL CLOSE :<имя курсорной переменной>;
```

Описание

1) Параметр <имя курсорной переменной> задает курсор, который должен быть закрыт.
2) Оператор закрывает курсор, возвращаемый хранимой процедурой.

Примечание
В группе связанных курсорных операций OPEN, FETCH, CLOSE должно использоваться либо <имя курсора>, либо <имя курсорной переменной>.

3) Курсорная переменная может быть повторно использована. Приложение может сколь угодно раз открывать курсорную переменную, выполнять в нее выборку (FETCH), закрывать курсорную переменную – все это можно осуществлять до отсоединения от ЛИНТЕР-сервера. После повторного соединения с ЛИНТЕР-сервером курсорной переменной должна быть вновь выделена память.

Пример


```
. . .

EXEC SQL BEGIN DECLARE SECTION;
CURSOR Cr;
int Var;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :Cr; /* выделяем память под динамический курсор
 */
EXEC SQL EXECUTE
  BEGIN
  :Cr = stored_proc(); /* исполняем хранимую процедуру, которая
 возвращает курсор */
  END;
END-EXEC;
EXEC SQL FETCH :Cr INTO :Var; /* работаем с выборкой */
. . .
EXEC SQL CLOSE :Cr; /* закрываем динамический курсор. Память не
 освобождается! */
EXEC SQL DEALLOCATE :Cr ;/* только здесь освобождается память,
 занимаемая курсорной переменной */
. . .
```

Освобождение памяти динамического курсора

Назначение

Освобождение памяти неиспользуемой курсорной переменной.

Синтаксис

```
<освобождение памяти>::=
EXEC SQL DEALLOCATE:<имя курсорной переменной>;
```

Примечание
Операторы `ALLOCATE` и `DEALLOCATE` не следят за текущим состоянием курсорной переменной: контроль над повторным выделением и освобождением памяти возлагается на пользовательское приложение.

Выборка записей из курсора

Простая выборка

Назначение

Последовательная выборка записей из курсора (статического или динамического).

Синтаксис

```
<простая выборка>::=
EXEC SQL [FOR :<число выполнений> ]
```

```
   FETCH {<имя курсора> | :<имя курсорной переменной>}
   [<положение курсора>]
   {USING | INTO} {<список переменных>
   | USING DESCRIPTOR <имя дескриптора>};
<положение курсора >::=
NEXT| PRIOR | FIRST | LAST | {[ABSOLUTE | RELATIVE] :<переменная>}
```

Описание

1) Конструкция FOR задает число повторений оператора FETCH. Переменная основного языка, задающая параметр цикла <число выполнений>, должна иметь тип данных short (значение в диапазоне от 1 до 32767). Если переменная другого типа, при претрансляции будет выдана ошибка «Ожидалась целая переменная». Циклическое выполнение предложения предназначено для того случая, когда переменные (объявленные в конструкции INTO ... или USING DESCRIPROR ... данного оператора) являются массивами. Если в список переменных включен массив, а конструкция FOR не указана, то в качестве числа повторений берется размер переменных-массивов, используемых в данном операторе. Если массивы при этом имеют неравные размеры, оператор возвращает ошибку ErrPCI_IncDim.
2) Параметр <имя курсора> задает статический курсор, по которому должна выполняться выборка данных. Курсор должен быть предварительно открыт.
3) Параметр <имя курсорной переменной> задает динамический курсор, для которого предварительно должны быть выполнены следующие операции:
   - объявление в секции описаний переменных основного языка как переменной типа «курсор» (CURSOR);
   - выделение памяти с помощью оператора ALLOCATE;
   - открытие путем исполнения хранимой процедуры.
4) Конструкция <положение курсора> задает порядок выбора очередной записи при выполнении оператора FETCH:
   - NEXT – выдать следующую запись множественной выборки данных;
   - PRIOR – выдать предыдущую запись множественной выборки данных;
   - FIRST – выдать первую запись множественной выборки данных;
   - LAST – выдать последнюю запись множественной выборки данных;
   - ABSOLUTE :<переменная> – выдать указанную запись множественной выборки данных. Номер записи определяется значением аргумента <переменная>, величина которого должна быть целым положительным числом;
   - RELATIVE :<переменная> – выдать следующую запись относительно текущей. Относительный номер записи определяется значением аргумента <переменная>, величина которого задается целым числом (положительным или отрицательным). Если, например, текущий номер выданной записи равен K, то при значении аргумента N будет выдана K+N запись, а при значении аргумента –N K–N запись (в том случае, если записи с такими номерами существуют).
5) Если <положение курсора> не задано, по умолчанию выбирается следующая запись (NEXT).
6) Если задана конструкция FOR, то допустимым <положением курсора> является только NEXT (в противном случае на этапе выполнения курсора фиксируется ошибочная ситуация «Неверный параметр»).

7) Если при выполнении конструкции FOR конец выборки достигнут до исчерпания цикла, возвращается код завершения 0, а реальное количество загруженных записей заносится в псевдопеременную CntPCI_.
8) Параметр USING|INTO <список переменных> должен содержать список переменных, в которые будут загружаться выбираемые значения. Число и тип переменных должны строго соответствовать числу и типу выбираемых значений.
9) Параметр USING DESCRIPTOR <имя дескриптора> используется для привязки динамических параметров. Дескриптор должен быть предварительно объявлен с помощью директивы SQL DESCRIBE.
10) Если модуль претранслирован в режиме совместимости со старой версией РСС (ключ -V претранслятора), то первая выборка осуществляется при открытии курсора, по умолчанию она выполняется при первом FETCH.
11) Оператор FETCH для хранения данных, полученных от СУБД ЛИНТЕР, использует буфер, размер которого равен по умолчанию 4096 байтам. Если размер ответа превышает размер буфера, на этапе выполнения выдается код завершения СУБД ЛИНТЕР 1014 («Заданный пользовательский буфер недостаточен»). Чтобы подобного не произошло, нужно увеличить размер буфера (см. оператор EXEC LINTER OPTION AREASIZE).

Пример

```
. . .
EXEC SQL BEGIN DECLARE SECTION;
CURSOR Cr;
Int Var;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :Cr;
EXEC SQL EXECUTE
  BEGIN
  :Cr = stored_proc();
  END;
END-EXEC;
EXEC SQL FETCH :Cr INTO :Var;
. . .
EXEC SQL CLOSE :Cr;
EXEC SQL DEALLOCATE :Cr;
. . .
```

Циклическая выборка

Назначение

Конструкция используется как альтернатива последовательности операторов OPEN-FETCH-CLOSE.

Синтаксис

```
<циклическая выборка>::=
EXEC SQL [AT <имя соединения>] REPEATED SELECT <предложение SQL>;
EXEC SQL BEGIN;
<операторы основного и встроенного языка>
```

```
[EXEC SQL ENDSELECT;]
<операторы основного и встроенного языка>
EXEC SQL END;
```

Описание

1) <Имя соединения> определяет соединение, по которому открывается курсор. Значение параметра может быть задано переменной типа «соединение» (DATABASE) либо символьной константой. Соединение должно быть объявлено ранее. Если параметр <имя соединения> опущен, курсор открывается для неименованного соединения.
2) <Предложение SQL> должно быть select-запросом, может иметь входные и выходные параметры (задаваемые с помощью конструкции INTO <список переменных основного языка>).
3) EXEC SQL BEGIN – начало тела циклической выборки.
4) EXEC SQL ENDSELECT – принудительный выход из тела цикла по некоторому условию.
5) EXEC SQL END – конец тела цикла.

Пример

```
exec sql repeated select distinct *
into :dbArchiveID, :dbRequestID from AMCmd where ArchiveID
 = :dbArchiveID;
exec sql begin; /* начало тела цикла */
  RetStatus = AddToList(dbArchiveID, dbRequestID); /* некоторые
 действия с полученными данными * /
  if(RetStatus != STATUS_OK /* что-то не так */
    exec sql endselect; /* принудительное завершение цикла */
exec sql end; /* конец тела цикла */
```

Добавление BLOB-данных

Назначение

Добавить порцию BLOB-данных.

Синтаксис

```
<добавить BLOB-данные>::=
EXEC SQL [ AT <имя соединения> ][ <номер BLOB-столбца>]
BLOB ADD FROM:<буфер>
[ WHERE CURRENT OF {<имя курсора> |<имя курсорной переменной>}];
<номер BLOB-столбца>::= {<числовой литерал>| <переменная основного
 языка>}
<буфер>::= <переменная основного языка>
```

Описание

1) <Имя соединения> определяет соединение, по которому открыт курсор. Значение параметра может быть задано переменной типа «соединение» (DATABASE) либо

символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, выборка BLOB-данных выполняется по неименованному соединению.

2) Параметр <номер BLOB-столбца> задает порядковый номер BLOB-столбца в записи, для которой осуществляется добавление порции BLOB-данных. Нумерация BLOB-столбцов начинается с 1. Если параметр не задан, по умолчанию принимается 1.

3) Параметр <буфер> задает имя переменной основного языка типа CHAR, BIT, VARCHAR, VARBIT, которая идентифицирует буфер памяти, содержащий порцию добавляемых BLOB-данных. Размер порции данных (буфера) не должен превышать 4000 байт.

4) Конструкция WHERE CURRENT OF <имя курсора> используется для указания записи, куда производится вставка. Запись должна быть выбрана (OPEN, FETCH).

5) Параметр <имя курсора> задает курсор, по которому осуществляется выборка записей для последующего добавления к ним BLOB-данных. Курсор предварительно должен быть открыт, и над ним необходимо выполнить, по крайней мере, одну операцию FETCH.

6) Параметр <имя курсорной переменной> задает имя переменной типа «курсор» CURSOR. Переменная должна быть предварительно объявлена, под нее необходимо выделить память с помощью оператора ALLOCATE, она должна быть инициализирована путем присвоения ей результата «типа курсор» хранимой процедуры, и над этим курсором должна быть выполнена, по крайней мере, одна операция FETCH.

7) Добавление BLOB-данных возможно только к записям, уже существующим в таблице на момент добавления, операция добавления BLOB-значения всегда выполняется над текущей записью. Поэтому курсор, заданный параметром <имя курсора> или <имя курсорной переменной>, должен быть открыт, и по нему необходимо произвести позиционирование той записи, к которой добавляется порция BLOB-данных. Для записей, ранее занесенных в таблицу, позиционирование выполняется с помощью оператора FETCH или UPDATE; вновь добавляемые записи становятся текущими при операции INSERT. BLOB-значение добавляется в конец существующих в столбце данных.

8) При выполнении операции INSERT BLOB-столбец должен быть пропущен или иметь значение NULL. Он заполняется 24-байтовой строкой системной информации.

9) Размер добавляемой порции равен размеру переменной, представляющей параметр <буфер>, если переменная имеет тип CHAR или BIT.

10) Размер добавляемой порции BLOB-данных равен значению поля len (length для синтаксиса СУБД Ingres) переменной, представляющей параметр <буфер>, если переменная имеет тип VARCHAR, VARBIT.

11) Если параметр WHERE CURRENT OF <имя курсора> не указан, будет произведена попытка выборки из BLOB-столбца, на который ссылается соединение (путем предварительного исполнения предложения оператором EXECUTE или EXECUTE IMMEDIATE SQL, позиционирующего на запись (SELECT, UPDATE, INSERT)).

Выборка BLOB-данных

Назначение

Выбрать порцию BLOB-данных из текущей записи курсора.

Синтаксис

```
<выбрать BLOB-данные>::=
EXEC SQL [ AT <имя соединения> ] [<номер BLOB-столбца>]
  BLOB GET INTO:<буфер>
  [OFFSET :<смещение>]
  [WHERE CURRENT OF {<имя курсора>| <имя курсорной переменной> }];
  <номер BLOB-столбца>::= {<числовой литерал>| <переменная
 основного языка>}
  <буфер>::= <переменная основного языка>
```

Описание

1) <Имя соединения> определяет соединение, по которому открыт курсор. Значение параметра может быть задано переменной типа «соединение» (DATABASE) либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, выборка BLOB-данных выполняется по неименованному соединению.
2) Параметр <номер BLOB-столбца> задает порядковый номер BLOB-столбца в записи, для которой осуществляется добавление порции BLOB-данных. Нумерация BLOB-столбцов начинается с 1. Если параметр не задан, по умолчанию принимается 1.
3) Операция выборки BLOB-данных применима только к текущей записи курсора. Поэтому перед тем, как получить порцию BLOB-значений из заданной записи, необходимо сделать эту запись текущей с помощью оператора FETCH.
4) Параметр <буфер> задает имя переменной основного языка типа CHAR, BIT, VARCHAR, VARBIT, в которую будет помещена порция BLOB-данных.
5) Параметр <смещение> определяет имя переменной основного языка, содержащей положительное целочисленное смещение выбираемой порции BLOB-данных относительно начала BLOB-данных. Смещение первого байта BLOB-значения равно 1. Оно принимается по умолчанию, если параметр <смещение> не задан.
6) Параметр <имя курсора> задает имя курсора. В нем в качестве текущей записи установлена та, из которой должны выбираться BLOB-данные.
7) После выполнения оператора число реально извлеченных байт заносится в глобальную переменную LenPCI_.
8) Размер запрашиваемой порции BLOB-данных равен размеру переменной типа CHAR, BIT или значению поля len (length для СУБД Ingres) для переменной типа VARCHAR, VARBIT.
9) Если параметр WHERE CURRENT OF <имя курсора> не указан, будет произведена попытка выборки из BLOB-данных, на которые ссылается соединение (путем предварительного исполнения предложения оператором EXECUTE или EXECUTE IMMEDIATE SQL, позиционирующего на запись (SELECT, UPDATE, INSERT)).

Удаление BLOB-данных

Назначение

Удалить BLOB-данные из текущей записи курсора (соединения).

Синтаксис

```
<удалить BLOB-данные>::=
```



```
EXEC SQL [ AT <имя соединения> ] [ <номер BLOB-столбца>]
BLOB CLEAR [ WHERE CURRENT OF {<имя курсора>
| <имя курсорной переменной> }];
<номер BLOB-столбца>::= {<числовой литерал>| <переменная основного
 языка>}
```

Описание

1) <Имя соединения> определяет соединение, по которому открыт курсор. Значение параметра может быть задано переменной типа «соединение» (DATABASE) либо символьной константой. В любом случае заданное значение должно быть ранее использовано в операторе CONNECT. Если параметр <имя соединения> опущен, удаление BLOB-данных выполняется по неименованному соединению.
2) Параметр <номер BLOB-столбца> задает порядковый номер BLOB-столбца в записи, где удаляются BLOB-данные. Нумерация BLOB-столбцов начинается с 1. Если параметр не задан, по умолчанию принимается 1.
3) Параметр <имя курсора> (<имя курсорной переменной>) должен задавать имя открытого курсора. В нем в качестве текущей записи устанавливается та, из которой должно быть удалено BLOB-значение.
4) Операция удаления BLOB-значения применима только к текущей записи курсора. Поэтому перед тем, как удалить BLOB-значение из заданной записи, необходимо сделать эту запись текущей с помощью оператора FETCH.
5) Если параметр WHERE CURRENT OF <имя курсора> не указан, будет произведена попытка выборки из BLOB-данных, на которые ссылается соединение (путем предварительного исполнения предложения оператором EXECUTE или EXECUTE IMMEDIATE SQL, позиционирующего на запись (SELECT, UPDATE, INSERT)).

Динамический SQL

Дескрипторы

Встроенный SQL позволяет конструировать и выполнять SQL-запросы, у которых текст, количество и тип переменных неизвестны в момент компиляции программы. Текст таких запросов формируется в процессе выполнения программы и, в зависимости от условий ее выполнения, каждый раз может иметь различный вид. При конструировании динамического запроса необходимо описать число, тип и имена выбираемых столбцов и, соответственно, указать переменные, в которые выбираемые значения должны быть загружены. Такие описания выполняются с помощью дескрипторов – специальных типов данных для связывания переменных в динамических запросах.

Так как в динамическом запросе могут присутствовать две группы неизвестных параметров: входные и выходные (выбираемые столбцы и выходные параметры хранимых процедур), для динамического запроса требуются две переменные, в которых эти параметры будут храниться. Эти переменные называются дескрипторами и имеют одинаковую структуру для входных (BIND-дескриптор) и выходных параметров (SELECT-дескриптор).

Объявление дескриптора

Дескриптор может быть объявлен следующими способами:

1) как идентификатор встроенного языка при первом использовании в исходном тексте модуля. В этом случае прекомпилятор автоматически создает переменную типа «дескриптор» (DESCRIPTOR) с данным именем, например:

```
EXEC SQL DESCRIBE INPUT my_statement INTO SQL DESCRIPTOR 'DSI'; /*
 неявное объявление дескриптора в операторе SQL DESCRIBE.
Перед исполнением этого оператора должен быть вызван оператор
 ALLOCATE DESCRIPTOR, т.к. оператор SQL DESCRIBE
только лишь вводит переменную DSI в пространство имен встроенного
 языка */
EXEC SQL ALLOCATE DESCRIPTOR tabl_desc WITH MAX 12; /* неявное
 задание имени дескриптора при его инициализации */
```

2) как переменная основного языка типа «дескриптор» DESCRIPTOR в секции объявлений переменных основного языка:

```
<объявление дескриптора>::=
DESCRIPTOR <имя дескриптора>;
```

В этом случае дескриптор также должен быть перед использованием инициализирован с помощью оператора ALLOCATE DESCRIPTOR, например:

```
EXEC SQL BEGIN DECLARE SECTION;
 DESCRIPTOR update_desc; /* объявление дескрипторной переменной */
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE DESCRIPTOR :update_desc; /* инициализировали
 дескриптор */
EXEC SQL DESCRIBE INPUT ST INTO SQL DESCRIPTOR :update_desc; /*
 получили описание входных переменных предложения ST */
```


3) как переменные – указатели на тип t_sqlda (см. приложение 1) и использование конструкций SQL DESCRIBE BIND VARIABLES, SQL DESCRIBE SELECT LIST.

Примечание
Такой стиль строго не рекомендуется, т.к. является устаревшим и оставлен только для совместимости со старыми версиями прекомпилятора.

В этом случае объявление должно выполняться вне секции объявлений встроенного SQL. Этот тип описан в sqlda.h. Таким образом, sqlda.h должен быть включен в текст программы с помощью директивы EXEC SQL INCLUDE SQLDA.

Например:

```
EXEC SQL INCLUDE SQLDA;
int main(int argc, char * argv[])
{
 t_sqlda *bind_dp = sqlald(12,67,67);
 t_sqlda *select_dp = sqlald(12,67,67);
…
/* Получение описания входных параметров предложения. Здесь
 переменная bind_dp – переменная */
/* основного языка. Причем при претрансляции не проводится
 проверка на ее видимость */
EXEC SQL DESCRIBE BIND VARIABLES FOR my_statement INTO bind_dp;
/* Получение описания выходных параметров предложения */
EXEC SQL DESCRIBE SELECT LIST FOR my_statement INTO select_dp;
…
}
```

Инициализация дескриптора

Назначение

Выделение необходимой для дескриптора памяти.

Синтаксис

```
<выделение памяти для дескриптора>::=
EXEC SQL ALLOCATE DESCRIPTOR <имя дескриптора>
[ WITH MAX <количество описателей>];
<имя дескриптора>::={ идентификатор встроенного SQL
| <строковый литерал>
| :<переменная основного языка типа «дескриптор»>}
<количество описателей>::= {<числовой литерал> | :<переменная
 основного языка> }
```

Описание

1) <Числовой литерал> – целое положительное число в диапазоне от 1 до 32767.
2) <Переменная основного языка> – переменная типа INTEGER с диапазоном значений от 1 до 32767.

3) Если конструкция WITH MAX опущена, по умолчанию память выделяется по количеству описателей параметров, задаваемых в директиве EXEC LINTER OPTION MAX ENTRIES (см. «Максимальный размер дескриптора»).

Примечание
В ранних версиях встроенного SQL инициализация дескрипторов выполнялась при помощи функции sqlald(), то есть производилась с помощью функций основного языка C/C++, а не встроенного языка. Эта возможность оставлена для совместимости ранее написанных программ с новой версией встроенного SQL, но пользоваться ею в новых проектах строго не рекомендуется.

Пример

(старый синтаксис: строго не рекомендуется.)

```
bind_dp = sqldald(max_vars, max_name, max_ind_name);
select_dp = sqldald(max_vars, max_name, max_ind_name);
```

Функция sqldald() имеет следующий прототип:

```
sqldaid(max_vars, max_name, max_ind_name)
```

где:

max_vars – максимальное число колонок, которое возможно в конструируемом динамическом запросе;

max_name – максимальная длина имени переменной, которая может быть использована в динамическом запросе;

max_ind_name – максимальная длина имени индикаторной переменной, которую дескриптор может описывать.

4) Конструкция <выделение памяти для дескриптора> выделяет оперативную память, необходимую для описания <количества описателей> в дескрипторе <имя дескриптора>, при этом все значения дескриптора являются неопределенными, т.е. никакое начальное значение описателям дескриптора не присваивается.

5) Если память для данного дескриптора была выделена ранее в той же области видимости переменных и не освобождена, фиксируется ошибочная ситуация.

6) Если <количество описателей> меньше 1 или больше MAX ENTRIES, фиксируется ошибочная ситуация.

Привязка динамических параметров

Назначение

Привязка динамических параметров подразумевает получение информации о загружаемых столбцах или динамических параметрах, содержащихся в подготовленном к выполнению предложении SQL, и загрузку ее в дескриптор, т.е. автоматическое формирование дескрипторов для подготовленного предложения.

Синтаксис

Прекомпилятор встроенного SQL распознает и обрабатывает два формата данного предложения:

- формат 1 – синтаксис, используемый в предыдущих версиях прекомпилятора и оставленный для совместимости с разработанными приложениями, не рекомендуется для использования в новых проектах;
- формат 2 – синтаксис данной версии прекомпилятора.

Формат 1 (устаревший):

```
<привязка входных параметров>::=
EXEC SQL DESCRIBE BIND VARIABLES FOR <имя предложения>
INTO <имя дескрипторной переменной типа t_sqlda)>;
```

Формат 2:

```
<привязка входных параметров>::=
EXEC SQL DESCRIBE INPUT <имя предложения>
{ USING | INTO } SQL DESCRIPTOR <имя дескриптора>;
```

Формат 1 (устаревший):

```
<привязка выходных параметров>::=
EXEC SQL DESCRIBE SELECT LIST FOR <имя предложения>
INTO <имя дескрипторной переменной типа t_sqlda >;
```

Формат 2:

```
<привязка выходных параметров>::=
EXEC SQL DESCRIBE [ OUTPUT ] <имя предложения>
{ USING | INTO } SQL DESCRIPTOR <имя дескриптора>;
```

Описание

1) <Имя предложения> – переменная прекомпилятора, которая должна быть ранее объявлена в DECLARE STATEMENT или использована в операторе PREPARE.
2) <Имя дескриптора> – переменная встроенного языка, должна быть уникальной или переменной основного языка типа DESCRIPTOR. В этом случае перед именем ставится двоеточие.
3) Дескриптору <имя дескриптора> необходимо предварительно выделить память (оператор ALLOCATE DESCRIPTOR) для описателей, число которых должно быть не меньше числа динамических параметров в <имени предложения>.
4) Привязка выходных параметров допустима только для select-запросов и вызова хранимых процедур.
5) Так как привязка дескриптора выполняется к уже подготовленному предложению, информация о количестве входных и выходных параметров, их типах, длинах, точности представления данных и др. (кроме значений параметров) известна, поэтому при выполнении оператора все сведения автоматически вносятся в дескриптор.
6) Значения входным параметрам должны присваиваться приложением с помощью оператора SET DESCRIPTOR (или путем явного присвоения значений полям

структуры sqlda, соответствующей данному дескриптору, что строго не рекомендуется).

7) Для select-запросов (для каждого выбираемого значения) в описатель параметра заносятся имя, тип и длина выбираемого значения. Если для выбираемого имени задан псевдоним, в описатель параметра помещается имя псевдонима.

8) Если предложение является конструкцией UNION с несколькими select-запросами, то в качестве имен параметров используются имена из первого select-запроса.

9) Для хранимых процедур для каждого параметра процедуры типа OUT, INOUT в описатель параметра заносятся имя, тип и длина выбираемого значения.

10) Оператор SQL DESCRIBE используется для получения информации о параметрах динамического предложения SQL.

11) Оператор SET DESCRIPTOR применяется в следующих случаях:

- для привязки переменных основного языка путем задания атрибутов TYPE, LENGTH, DATA основной переменной;
- при заполнении описателей параметров в дескрипторе для еще не подготовленного предложения. В этом случае после подготовки предложения необходимость в операторе SQL DESCRIBE отпадает.

Получение информации из дескриптора

Назначение

Получение значений выходных параметров предложения или информации о параметрах предложения SQL после их привязки.

Синтаксис

```
<получить информацию о дескрипторе>::=
EXEC SQL GET DESCRIPTOR <имя дескриптора>
  { :< переменная основного языка> ={COUNT | KEY_TYPE}
  | VALUE <номер описателя параметра>
<описатель параметра> [, <описатель параметра> … ]
<описатель параметра>::=
  :< переменная основного языка> =
  { TYPE
    | LENGTH
    | [RETURNED_LENGTH]
    | PRECISION
    | SCALE
    | NULLABLE
    | INDICATOR
    | DATA
    | NAME }
<номер описателя параметра>::= переменная основного языка
```

Описание

1) COUNT – выдает общее количество описателей динамических параметров в дескрипторе <имя дескриптора>.

2) KEY_TYPE – признак первичного ключа (зарезервировано для будущего использования).

3) <Номер описателя параметра> задает номер описателя динамического параметра в дескрипторе <имя дескриптора>. Значение должно находиться в диапазоне от 1 до <количества описателей> (см. опцию WITH MAX в операторе EXEC SQL ALLOCATE DESCRIPTOR). Если номер меньше 1 или больше <количества описателей>, на этапе выполнения будет выдана ошибка ErrPCI_DescSmall. <Номер описателя параметра> может быть задан целочисленной константой или целочисленной переменной основного языка.

4) О каждом динамическом параметре можно запросить информацию, которая рассмотрена в таблице 8. Поддерживаемые типы данных динамических параметров рассмотрены в таблице 9.

Таблица 8. Атрибуты динамических параметров

Имя атрибута	Тип данных атрибута	Описание
TYPE	int8 int16 int32	Тип данных параметра: а) после выполнения оператора SQL DESCRIBE – тип формального параметра предложения SQL; б) после установки значения посредством SET DESCRIPTOR – тип данных фактического параметра, привязанный к этому описателю. Соответствие типов данных дескриптора и языка C/C++ приведено в таблице 10. Коды типов данных приведены в таблице 11. Символические имена типов данных находятся в заголовочном файле sqlda.h. **Примечание** Строго не рекомендуется использовать числовые коды для задания типов.
LENGTH	int8 int16 int32	Объявленная длина параметра: а) после выполнения оператора SQL DESCRIBE – длина формального параметра предложения SQL; б) после установки значения посредством SET DESCRIPTOR – длина фактического параметра, привязанного к данному описателю.
RETURNED_LENGTH	int8 int16 int32	Фактическая длина параметра: а) после выполнения оператора SQL DESCRIBE совпадает со значением атрибута LENGTH; б) после исполнения запроса с выходными динамическими параметрами содержит

Имя атрибута	Тип данных атрибута	Описание
		фактическую длину полученных данных (при ошибке ErrPCI_TooLong выдается длина со знаком «минус»).
PRECISION	int8 int16 int32	Точность числа типа NUMERIC (в текущей версии не используется).
SCALE	int8 int16 int32	Масштаб числа типа NUMERIC (в текущей версии не используется).
NULLABLE	int8 int16 int32	Признак допустимости NULL-значений: а) 0 – столбец таблицы допускает NULL-значение. б) не 0 – NULL-значения не допускаются (при попытке записать в такое поле NULL-значение на этапе выполнения фиксируется ошибочная ситуация ErrPCI_NullProhibited и значение атрибута RETURNED_LENGTH устанавливается в -1).
DATA		Значение параметра: а) после выполнения оператора SQL DESCRIBE выдает ошибку ErrPCI_NoDest (если не было выполнено привязки параметра); б) после привязки параметра с помощью SET DESCRIPTOR возвращает фактическое значение привязанного параметра; в) после исполнения предложения выходные параметры содержат значения, полученные из БД.
INDICATOR	int8 int16 int32	Фактическое значение индикаторной переменной: а) 0 – значение параметра не NULL; б) не 0 – значение параметра равно NULL. Имеет смысл только в том случае, если значение динамическому параметру было присвоено в результате выполнения select-запроса или оператора Set Descriptor.
NAME	char[n]	Имя параметра. После выполнения оператора SQL DESCRIBE содержит имя формального динамического параметра. Для неименованного параметра возвращается пустая строка.

5) <Имя дескриптора>, указанное в конструкции <получить информацию о дескрипторе>, должно ссылаться на дескриптор, инициализированный ранее (ALLOCATE DESCRIPTOR).

6) Тип данных переменной, обозначенной в параметре DATA, должен соответствовать или приводиться к типу данных и длине описателя, указанного в <номере описателя параметра>.

7) <Номер описателя параметра>, заданный в конструкции <получить информацию о дескрипторе>, должен быть в диапазоне от 1 до MAX ENTRIES, в противном случае фиксируется ошибка. Однако если <номер описателя параметра> больше значения COUNT, фиксируется ситуация «Нет данных».

8) Если некоторый описатель допускает NULL-значение, то в параметре DATA обязательно должна указываться индикаторная переменная.

Таблица 9. Таблица поддерживаемых типов данных

Тип	Имя макроса	Примечание
integer int	PCC_INT_TYP	Входные/выходные переменные
float	PCC_FLT_TYP	Входные/выходные переменные
decimal numeric	PCC_DEC_TYP	Входные/выходные переменные
character char	PCC_CHR_TYP	Входные/выходные переменные
text varchar	PCC_TXT_TYP	Входные/выходные переменные
smallint	PCC_SML_TYP	Входные/выходные переменные
real	PCC_REL_TYP	Входные/выходные переменные
double	PCC_DBL_TYP	Входные/выходные переменные
date	PCC_DAT_TYP	Входные/выходные переменные
byte	PCC_BIT_TYP	Входные/выходные переменные
varbyte	PCC_BVT_TYP	Входные/выходные переменные
bool	PCC_BOO_TYP	Входные/выходные переменные
bigint	PCC_BIGINT_TYP	Входные/выходные переменные

В таблице 10 приведены типы данных, привязываемых дескрипторами, и их соответствие с типами основного языка.

Таблица 10. Соответствие типов данных дескриптора и языка C/C++

Тип данных описателя дескриптора	Соответствие C/C++	Размер, байт
PCC_INT_TYP	int	4
PCC_SML_TYP	short	4
PCC_FLT_TYP	float	4
PCC_REL_TYP	float	4
PCC_DBL_TYP	double	8
PCC_DEC_TYP	decimal	16

Тип данных описателя дескриптора	Соответствие C/C++	Размер, байт
PCC_CHR_TYP	char[n]	n
PCC_TXT_TYP	varchar_pci(n)	n+2
PCC_BIT_TYP	unsigned char[n]	n
PCC_BVT_TYP	varbit_pci(n)	n+2
PCC_DAT_TYP	pccdate	32
PCC_BLB_TYP	char[24]	24

Эти макросы описаны в `sqlda.h`. Их численные значения совпадают с номерами типов SQL, предопределенных стандартом SQL93.

Таблица 11. Коды типов данных, возвращаемые дескриптором

Тип данных СУБД ЛИНТЕР	Возвращаемый код
byte	PCC_BIT_TYP
varbyte	PCC_BVT_TYP
blob	PCC_BLB_TYP
Boolean	PCC_BOO_TYP
char	PCC_CHR_TYP
varchar	PCC_TXT_TYP
date	PCC_DAT_TYP
decimal	PCC_DEC_TYP
double	PCC_DBL_TYP
float	PCC_FLT_TYP
int	PCC_INT_TYP
numeric	PCC_DEC_TYP
real	PCC_REL_TYP
smalint	PCC_SML_TYP

Присвоение значений дескриптору

Назначение

Описание фактических значений параметров динамического запроса и присвоение входным параметрам в дескрипторе требуемых значений для последующего использования в динамических SQL-запросах.

Синтаксис

```
<присвоить значение дескриптору>::=
EXEC SQL SET DESCRIPTOR <имя дескриптора>
 COUNT = :<переменная основного языка> | <литерал>
| VALUE <номер параметра>
 <описатель параметра> [, < описатель параметра> … ]
<описатель параметра>::=
{ TYPE
```



```
| LENGTH
| PRECISION
| SCALE
| NULLABLE
| INDICATOR
| DATA
} = :{< переменная основного языка>| <литерал>} <данные>
```

Описание

1) COUNT – задает общее количество описателей динамических параметров в дескрипторе <имя дескриптора>. Указанное значение не должно превышать количества описателей, для которых выделено место в дескрипторе при выполнении оператора ALLOCATE DESCRIPTOR. Если COUNT превышает <количество описателей> (см. ALLOCATE DESCRIPTOR), на этапе выполнения программы будет фиксироваться ошибочная ситуация (ошибка ErrPCI_InvDescIdx).
2) <Номер параметра> задает номер параметра в дескрипторе <имя дескриптора>. Значение должно находиться в диапазоне от 1 до <количества описателей>, где <количество описателей> – количество объявленных описателей в дескрипторе <имя дескриптора>. <Номер параметра> может быть задан целочисленной константой или целочисленной переменной основного языка. Если <номер параметра> больше значения <количества описателей> или меньше 1, при выполнении возникает ошибка ErrPCI_DescSmall.
3) Каждому динамическому параметру в его описателе можно присвоить следующие атрибуты:
   - TYPE – тип переменной основного языка, привязанной к данному параметру в дескрипторе (см. таблицу 8);

```
EXEC SQL SET DESCRIPTOR DSC VALUE 1 TYPE =PCC_CHR_TYP; /* тип
 задан непосредственно */
type = PCC_CHR_TYP;
EXEC SQL SET DESCRIPTOR DSC VALUE 1 TYPE = :type; /* тип задан
 переменной основного языка */
```

   - LENGTH – длина переменной основного языка, привязанной к данному параметру в дескрипторе. Длину параметра можно установить равной нулю. В этом случае при привязке значения (DATE) длина будет определена автоматически по типу привязываемой переменной и ее фактическому значению на момент привязки (для CHAR, BIT – strlen(); для VARCHAR, VARBIT – значение поля len). Последующая операция GET DESCRIPTOR VALUE LENGTH(RETURNED_LENGTH) дает фактическую длину привязанного параметра.

```
EXEC SQL SET DESCRIPTOR DSC VALUE 1 LENGTH = 32; /* длина задана
 непосредственно */
/* длина задана переменной основного языка и установлена в
 неопределенное значение. */
/* Фактическая длина параметра будет определена в момент привязки
 данных.*/
len = 0;
EXEC SQL SET DESCRIPTOR DSC VALUE 1 LENGHT = :len;
```

- PRECISION – точность числа типа NUMERIC (не используется; зарезервировано для будущего использования);
- SCALE – масштаб числа типа NUMERIC (не используется; зарезервировано для будущего использования);
- NULLABLE – признак допустимости NULL-значений (0 – столбец таблицы допускает NULL-значение, не ноль – NULL-значения не допускаются). Если NULL-значение недопустимо, то при получении его в процессе исполнения предложения SQL возникает ошибка ErrPCI_NullProhibited;
- DATA – задает значение параметру предложения SQL. Перед присвоением значения необходимо инициализировать атрибуты TYPE и LENGTH в соответствии с типом и длиной привязываемой переменной основного языка (непосредственных данных).

  Если атрибут DATA содержит непосредственно данные или переменную типа PCC_DAT_TYP, то они копируются во внутренний буфер описателя. Если DATA содержит переменную основного языка, то в описателе сохраняется ее адрес. Копирование данных в этом случае не производится;
- непосредственно присвоить значение параметру типа VARCHAR, VARBIT нельзя. При выполнении возникнет ошибка ErrPCI_IncTyp.

```
/* данные заданы непосредственно */
EXEC SQL SET DESCRIPTOR DSC VALUE 1 DATA = 'sample_data';
EXEC SQL SET DESCRIPTOR DSC VALUE 1 DATA = :s; /* данные заданы
 переменной основного языка */
```

- INDICATOR – задает значение индикаторной переменной. Для привязки NULL-значения во входном параметре необходимо установить значение -1.

4) Входной дескриптор считается полностью описанным, если заданы атрибуты LENGTH, TYPE, DATA для всех точек входа. Если хотя бы одно из этих значений не установлено в описателе параметра, то фактическим значением параметра станет NULL-значение, и при исполнении будет выдано предупреждение ErrPCI_NotBind.
5) <Имя дескриптора>, указанное в конструкции <присвоить значение дескриптору>, должно ссылаться на дескриптор, инициализированный ранее (ALLOCATE DESCRIPTOR).
6) Тип данных переменной, указанной в атрибуте DATA, должен соответствовать типу данных и длине описателя, указанного в <номере описателя параметра>.

Освобождение памяти дескриптора

Назначение

Освобождение памяти, ранее выделенной под дескриптор.

Синтаксис

```
<освобождение памяти дескриптора>::=
EXEC SQL DEALLOCATE DESCRIPTOR <имя дескриптора>
```

Описание

1) <Имя дескриптора> должно ссылаться на ранее созданный дескриптор.

2) Конструкция <освобождение памяти дескриптора> освобождает ранее выделенную оперативную память для <имя дескриптора> в пределах видимости этой переменной.
3) Контроль повторного освобождения выделенной для дескриптора памяти не производится.

Примечание
В ранних версиях прекомпилятора освобождение памяти, занятой дескриптором, осуществлялось с помощью явного вызова функции `sqlclu()`. В новых проектах использование этой функции строго не рекомендуется.

Пример

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#if defined(VXWORKS)
#include "vxstart.h"
#endif

#ifdef _DOS_
#include <conio.h>
#endif

EXEC LINTER IFDEF SQL;
EXEC SQL INCLUDE SQLCA;
EXEC LINTER ENDIF;

EXEC SQL BEGIN DECLARE SECTION;
  char *user = "SYSTEM";
  char *pswd = "MANAGER8";
  char szSQL[4096] = {0};
  char buf[65535] = {0};
  int i = 0;
  int colCount = 0;
  int type = 0;
  int length = 0;

  int shift = 0;
  char * p = 0;
  DESCRIPTOR  selectOutDesc;
EXEC SQL END DECLARE SECTION;

char *aszTypes[] = {
  "UNKNOWN",
 "CHAR","INT","SMALLINT","BIGINT","DOUBLE","REAL","DATE","DECIMAL",

 "BYTE","BLOB","VARCHAR","VARBYTE","BOOL","NCHAR","NVARCHAR","EXTFILE"
```

```
};

int pcc2index(int pcc)
{
  switch (pcc)
  {
    case 14 : return 9;
    case 15 : return 12;
    case 31 : return 10;
    case 16 : return 13;
    case 1  : return 1;
    case 12 : return 11;
    case 3  : return 8;
    case 9  : return 7;
    case 8  : return 5;
    case 6  : return 6;
    case 7  : return 6;
    case 4  : return 2;
    case 2  : return 8;
    case 5  : return 3;
    default : return 0;
  }
}

int align4(int shift)
{
  return shift % 4 ? shift + 4 - shift % 4 : shift;
}

void printAnswer()
{
  char szMask[200];
  char szLen[20];
  int i = 0;
  float _f = 0;
  int _i = 0;
  L_WORD varcharlen = 0;

  shift = 0;

  for (i = 1; i <= colCount; i++)
  {
    p = &buf[shift];
    EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :type =
 TYPE, :length = LENGTH, :p = DATA;
    switch (type)
```



```
    {
      case 1: /*char*/
        printf("%s ", &(buf[shift]));
      break;
      case 4: /*int*/
        memcpy(&_i, &(buf[shift]), sizeof(_i));
        printf("%d ", _i);
      break;
      case 6: /*dbl*/
        memcpy(&_f, &(buf[shift]), sizeof(_f));
        printf("%g ", _f);
      break;
    }
    shift += align4(length);
  }
  printf("\n");
}

#if defined(VXWORKS)
MainStart(pcc_sample, 1024*32, UninitLinterClient)
#else
int main()
#endif
{
   char c = 0;

   printf("Создание соединения...");
   EXEC SQL WHENEVER SQLERROR GOTO not_conn;
   EXEC SQL CONNECT AUTOCOMMIT :user IDENTIFIED BY :pswd;
   printf("готово.\n");

   printf("Выделение памяти под дескриптор...");
   EXEC SQL ALLOCATE DESCRIPTOR :selectOutDesc;
   printf("готово.\n");

   printf("Введите SELECT-запрос: >");
   gets(szSQL);

   printf("Открытие оператора...", szSQL);
   EXEC SQL WHENEVER SQLERROR GOTO not_st_opened;
   EXEC SQL PREPARE ST_SEL FROM :szSQL;
   printf("готово.\n");

   printf("Заполнение дескриптора...");
   EXEC SQL WHENEVER SQLERROR GOTO not_desc_created;
```

```
  EXEC SQL DESCRIBE OUTPUT ST_SEL INTO SQL
DESCRIPTOR :selectOutDesc;
  printf("готово.\n");

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL GET DESCRIPTOR :selectOutDesc :colCount = COUNT;
  printf("Полей (столбцов) в выборке: = %d\n", colCount);

  for (i = 1; i <= colCount; i++)
  {
    EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :type = TYPE;
    EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :length =
LENGTH;
    printf("Столбец %i. Тип %d, %s Длина %d\n", i, type,
aszTypes[pcc2index(type)], length);
  }

  printf("Привязка буфера...");
  shift = 0;
  for (i = 1; i <= colCount; i++)
  {
    p = &(buf[shift]);
    EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :length =
LENGTH, :type = TYPE;
 if (type == PCC_CHR_TYP)
    length++; /* под тип CHAR нужно резервировать на 1 байт
больше для места под 0-терминатор */
 EXEC SQL SET DESCRIPTOR :selectOutDesc VALUE :i DATA = :p,
LENGTH = :length;
    shift += align4(length);
  }
  printf("готово.\n");

  printf("Открытие курсора...");
  EXEC SQL WHENEVER SQLERROR GOTO not_cur_opened;
  EXEC SQL DECLARE CUR_SEL CURSOR FOR ST_SEL;
  EXEC SQL OPEN CUR_SEL;
  printf("готово.\n");

  EXEC SQL WHENEVER SQLERROR GOTO not_fetched;
  for (i = 1; ;i++)
  {
    memset(buf, 0, sizeof(buf));
    EXEC SQL FETCH CUR_SEL ABSOLUTE :i USING SQL
DESCRIPTOR :selectOutDesc;
    if (ErrPCI_)
```



```
     {
       if (ErrPCI_ == 3000) printf("No more records\n");
       else printf("Error code: %d\n", ErrPCI_);
       break;
     }
     printAnswer();
   }

   printf("Освобождение памяти дескриптора...");
   EXEC SQL DEALLOCATE DESCRIPTOR :selectOutDesc;
/* EXEC SQL DEALLOCATE :CUR_SEL;*/
   printf("готово.\n");

   return 0;

not_conn:
  printf("Не удалось создать соединение\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_selected:
  printf("Не удалось создать выборку\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_st_opened:
  printf("Не удалось открыть оператор\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_desc_created:
  printf("Не удалось создать дескриптор\n");
  printf("Код ошибки: %d\n", ErrPCI_);
not_cur_opened:
  printf("Не удалось открыть курсор\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_fetched:
  printf("Не удалось переместиться по выборке\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
}
```

Работа с хранимыми процедурами

Объявление прототипа процедуры

Назначение

Объявление прототипа хранимой процедуры, которая может быть исполнена с помощью оператора EXECUTE PROCEDURE.

Синтаксис

```
<объявление прототипа>::=
EXEC SQL DECLARE PROCEDURE <прототип>;
<прототип>::=
<имя> (<список параметров>) [RESULT <тип результата>]
<имя>::= [<имя схемы>.]<имя процедуры>
<имя схемы>::= <идентификатор встроенного языка>
<имя процедуры>::= <идентификатор встроенного языка>
<список параметров>::= <параметр>[;<Параметр> ..]
<параметр>::= <модификатор> [<имя параметра>] <тип параметра>
<модификатор>::= {IN | OUT | INOUT}
<имя параметра>::= <идентификатор встроенного языка>
<тип параметра>::=
 { INT
 | SMALLINT
 | REAL
 | NUMERIC
 | CHAR()
 | BYTE()
 | DATE
 | BLOB }
<тип результата>::= {<тип параметра> | CURSOR}
```

Описание

1) Если хранимая процедура создается (модифицируется) в модуле встроенного языка с помощью операторов `{ CREATE | ALTER } PROCEDURE`, то использование оператора `DECLARE PROCEDURE` недопустимо – информация о прототипе берется из текста хранимой процедуры.
2) Типы параметров служат для привязки параметров вызова хранимой процедуры к переменным основного языка.
3) Если параметр `RESULT` пропущен, считается, что хранимая процедура возвращает NULL.
4) В отличие от синтаксиса процедурного блока (см. документ «СУБД ЛИНТЕР. Процедурный язык»), описание курсора, возвращаемого процедурой, не содержит расшифровку его полей.

Создание/модификация процедуры

Назначение

Создание/модификация хранимой процедуры.

Синтаксис

```
<создание хранимой процедуры>::=
EXEC SQL CREATE PROCEDURE <процедурный блок>;
END-EXEC;

<модификация хранимой процедуры>::=
EXEC SQL ALTER PROCEDURE  <процедурный блок>;
END-EXEC;
```

Структура <процедурного блока> подробно рассмотрена в документе «СУБД ЛИНТЕР. Процедурный язык».

Описание

1) Обработка этого оператора прекомпилятором производится по-разному при включенной и выключенной проверке семантики предложений встроенного языка (опция -S прекомпилятора). Если проверка семантики выключена, текст процедуры будет включен в претранслированный текст программы, и во время исполнения процедура будет создана (модифицирована). Если проверка семантики включена, оператор вызовет компилятор хранимых процедур spc, который создаст новую или модифицирует ранее созданную процедуру. Если при создании (модификации) процедуры возникает ошибка, создается файл-листинг с именем, совпадающим с именем процедуры, и расширением .lsr. Если была допущена синтаксическая ошибка в имени хранимой процедуры, создается файл-листинг с именем – порядковым номером процедуры в текущем компилируемом модуле. Этот файл содержит текст процедуры с расшифровкой ошибок.
2) Если проверка семантики включена (-S), должны быть указаны ЛИНТЕР-сервер, пользователь (которому будет принадлежать созданная процедура) и пароль пользователя. Если указаны несуществующий сервер, неверный пользователь или неверный пароль пользователя, то листинг будет содержать расшифровку ошибок работы с ЛИНТЕР-сервером.
3) В результате исполнения оператора прекомпилятор создает описание хранимой процедуры аналогично тому, как это происходит при обработке предложения DECLARE PROCEDURE встроенного языка.
4) Если претрансляция производится при включенной проверке семантики, должно быть указано <имя схемы>, в которой была создана процедура. Т.е. в операторе EXECUTE PROCEDURE нужно явно указать имя схемы, в которой была создана процедура, иначе будет выдана ошибка E_UNKNAME «Неопределенное имя».

Пример

(прекомпилятор вызывается с опцией -U USR1/Pass)

```
EXEC SQL CREATE PROCEDURE EX() RESULT INT
CODE
END;
END-EXEC;
...
```

```
EXEC SQL EXECUTE PROCEDURE USR1.EX();
```

Выполнение процедуры

Назначение

Выполнение хранимой процедуры.

Синтаксис

Оператор выполнения хранимой процедуры имеет два альтернативных стиля вызова: в стиле СУБД ЛИНТЕР и в стиле СУБД Ingres.

Примечание
Если процедура не имеет прототипа (не была объявлена с помощью операторов `DECLARE | CREATE | ALTER PROCEDURE`), считается, что стиль оператора `EXECUTE PROCEDURE` такой же, как у СУБД Ingres.

```
<выполнение процедуры в стиле СУБД ЛИНТЕР >::=
EXEC SQL EXECUTE PROCEDURE
[<результат>=][<имя схемы>.]
<имя процедуры>([<параметр>][,<параметр>][,...]);

<выполнение процедуры в стиле Ingres>::=
EXEC SQL EXECUTE PROCEDURE
 [<имя схемы>.]<имя процедуры>(<формальный параметр>=<фактический
 параметр>,...) [INTO <результат>];

<результат>::= <возвращаемое значение>[.<индикатор>]
<возвращаемое значение>::=
 <переменная основного языка числового типа >
 | <переменная основного языка символьного типа>
 | CURSOR
<имя схемы>::= идентификатор встроенного SQL
 <параметр>::=
  {:<переменная основного языка>[:<индикаторная переменная>]
  | <литерал>
  | <пропуск параметра>}
<пропуск параметра>::= ',,'
```

Описание

1) `<Имя процедуры>` – идентификатор встроенного языка. Задает имя процедуры. Процедура с таким именем и `<имя схемы>` (если оно задано) должна быть предварительно либо объявлена с помощью оператора `DECLARE PROCEDURE`, либо создана (модифицирована) с помощью оператора `CREATE (ALTER) PROCEDURE`.
2) Для выходных параметров тип фактического параметра должен соответствовать типу формального, используемого процедурой.
3) Если процедура может возвращать NULL-значение, то для проверки ее результата или выходного параметра, заданного именованным динамическим параметром, на NULL-значение должна использоваться индикаторная переменная.

4) Если индикаторная переменная не задана, а возвращаемое значение является NULL-значением, фиксируется ошибочная ситуация.
5) Если результат процедуры имеет тип данных DATE, используется формат dd.mm.yyyy:hh:mi:ss.
6) Если результат процедуры типа BLOB, возвращается заголовок длиной 24 байта.
7) Если результат процедуры типа BYTE, возвращается строка байт.
8) Если результат процедуры типа CURSOR, возвращается курсорная переменная, которую можно использовать в операторах FETCH, {ADD | GET | CLEAR} BLOB, CLOSE.
9) Если возвращаемое значение не типа CURSOR и если возвращается NULL-значение, индикаторная переменная имеет нулевое значение, иначе она содержит длину результата (в байтах). Т.е. использование индикаторной переменной отличается от использования в других предложениях встроенного языка. Если возвращаемое значение типа CURSOR, индикаторная переменная содержит количество записей, выбранных для данного курсора, или нулевое значение, если курсор пуст.
10) Если в процессе исполнения хранимой процедуры произошла ошибка, ErrPCI_ содержит код завершения СУБД ЛИНТЕР.
11) Тип переменной основного языка должен соответствовать типу результата, который возвращается процедурой.
12) В режиме совместимости с СУБД Ingres параметры при вызове процедуры должны быть всегда неименованные.

Пример

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#if defined(VXWORKS)
#include "vxstart.h"
#endif

#ifdef _DOS_
#include <conio.h>
#endif

EXEC LINTER IFDEF SQL;
EXEC SQL INCLUDE SQLCA;
EXEC LINTER ENDIF;

EXEC SQL BEGIN DECLARE SECTION;
  char *user = "SYSTEM";
  char *pswd = "MANAGER8";
  char *node = "LISACHEV";
  int m1 = 0, m2 = 0;
  int i1 = 0, i2 = 0;
  int res = 0;

  char ch1[10] = {0};
  char ch2[10] = {0};
EXEC SQL END DECLARE SECTION;
```

```
#if defined(VXWORKS)
MainStart(pcc_sample, 1024*32, UninitLinterClient)
#else
int main()
#endif
{
   char c = 0;

   printf("Создание соединения...");
   EXEC SQL WHENEVER SQLERROR GOTO not_conn;
   EXEC SQL CONNECT AUTOCOMMIT :user IDENTIFIED BY :pswd
 USING :node;
   printf("готово.\n");

   printf("Объявление процедуры...");
   EXEC SQL WHENEVER SQLERROR GOTO not_declared;
   EXEC SQL DECLARE PROCEDURE STRADD(in ch1 char(10); in ch2
 char(10)) result int;
   printf("готово.\n");

   printf("Введите первое число >");
   gets(ch1);
   printf("\n");
   printf("Введите второе число >");
   gets(ch2);
   printf("\n");

   printf("Выполнение процедуры...");
   EXEC SQL EXECUTE PROCEDURE :res = STRADD(:ch1, :ch2);
   printf("Готово. '%s' + '%s' = %d\n", ch1, ch2, res);

   return 0;

not_conn:
  printf("Не удалось создать соединение\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_declared:
  printf("Не удалось объявить процедуру\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
not_executed:
  printf("Не удалось выполнить процедуру\n");
  printf("Код ошибки: %d\n", ErrPCI_);
  return 1;
```



```
}
```

Пример использования хранимых процедур приведен также в приложении 2.

Многомодульные приложения

Реализация встроенного SQL имеет ограничение на видимость имен переменных основного и встроенного языка. Все переменные, за исключением:

- неявно заданной переменной «соединение по умолчанию»;
- переменной `sqlca`;
- псевдопеременных `SQLCODE`, `SQLSTATE`, `ErrPCI`, `CntPCI`, `IsnPCI`, `LenPCI`, `TxtPCI`;

имеют локальную область видимости.

Переменная sqlca по умолчанию экспортируется библиотекой PCL. Кроме того, можно задавать ее область видимости явно путем указания флага `-M` (sqlca принадлежит модулю, претранслированному с флагом `M`, см. в подразделе «Обращение к прекомпилятору»). Только один модуль может быть претранслирован с этим флагом. Остальные претранслируются с флагом `-E` или `-L`:

- `-E` (extern sqlca – модуль использует переменную sqlca, объявленную в другом модуле);
- `-L` (static sqlca – модуль использует свою локальную переменную sqlca);
- `-I` (import sqlca – модуль использует переменную sqlca, импортируемую из PCL). По умолчанию используется этот флаг.

Псевдопеременные `SQLCODE`, `SQLSTATE`, `ErrPCI`, `CntPCI`, `IsnPCI`, `LenPCI`, `TxtPCI` имеют глобальную область видимости.

Для того чтобы использовать встроенный SQL в многомодульных приложениях, необходимо придерживаться следующих правил:

1) использовать соединение по умолчанию. Все остальные соединения локальны;
2) все глобальные переменные основного языка желательно описывать в отдельном модуле основного языка в секции описаний переменных основного языка;
3) модули, в которых используются эти переменные, должны содержать секцию описаний переменных основного языка, где перечислены глобальные переменные с модификатором `extern`;
4) предложения SQL, курсоры и другие переменные встроенного SQL должны использоваться в том же модуле, где они явно или неявно декларированы (например, путем операции `PREPARE`). Эти переменные всегда локальны.

Параллельная обработка запросов (многопоточность)

Средства встроенного SQL позволяют выполнять параллельную обработку нескольких запросов к одной БД или к различным БД. Однако поддержка операционной системой многопоточного режима является всего лишь необходимым, но не достаточным условием распараллеливания обработки запросов в приложении. Приложение, претендующее на многопоточную обработку запросов, должно дополнительно выполнить следующие условия:

- для организации многопоточной обработки использовать предназначенные для этой цели специальные средства встроенного SQL и соблюдать некоторые ограничения, накладываемые на общие структуры данных;
- в процессе исполнения использовать библиотеку, специально разработанную для многопоточных приложений.

Многопоточные приложения имеют несколько потоков, выполняемых в общем адресном пространстве. Потоки являются подпроцессами, которые осуществляются внутри процесса. Они используют общие сегменты кода и данных, но имеют собственные стеки. Глобальные и статические переменные являются общими для всех потоков, поэтому требуется собственный специфичный механизм управления подобными переменными из разных потоков внутри приложения. Также требуется особый механизм синхронизации обработки, чтобы обеспечить целостность данных.

Требования к многопоточному приложению

Реализация параллельной обработки запросов выполняется с помощью механизма контекстов. *Контекст* – это отдельный поток приложения, который может включать в себя:

- ноль или более соединений с одним или несколькими ЛИНТЕР-серверами;
- ноль или более предложений встроенного SQL;
- ноль или более курсоров, созданных на соединении;
- набор функций, возвращающих состояние выполнения запросов данного контекста.

Встроенный SQL предоставляет приложению средства для описания контекста потока и обмена контекстами между потоками.

Например, интерактивное приложение порождает поток П1, выполняет внутри него запрос выборки и возвращает первые 10 записей в приложение, затем поток П1 прекращается. После получения необходимой команды от пользователя приложения порождается другой поток, П2 (или используется уже существующий), и контекст первого потока П1 передается ему, в результате чего поток П2 может продолжить выборку следующих 10 записей того же самого курсора.

Каждый поток должен выполняться в своем контексте. Для организации контекста используются специальные средства встроенного SQL:

- объявление контекстной переменной;
- выделение памяти для контекстной переменной (инициализация контекста);
- привязка контекста к конкретному потоку;

- освобождение памяти, выделенной контекстной переменной (деинициализация контекста).

Требования к контексту многопоточного приложения

Ниже перечислены требования, предъявляемые к контексту потока:

1) переменная sqlca должна объявляться как переменная типа `auto` отдельно для каждого потока. Для этого необходимо задать ее явно в претранслируемом модуле потока. Например:

```
 void __stdcall thread(void)
{
EXEC SQL MODULE M1;
struct sqlca sqlca;
. . .
EXEC SQL END MODULE M1;
}
```

2) переменные основного языка должны объявляться как переменные класса памяти `auto` для каждого потока. Если главные переменные объявляются как глобальные или переменные типа `static`, нужно внимательно следить за их использованием в разных потоках;
3) использовать отдельный модуль встроенного SQL для каждого потока, т.к. в этом случае происходит создание локальных дескрипторов БД, заданных явно, и дескрипторов предложений встроенного языка;
4) использовать явное задание соединения внутри каждого потока;
5) все ссылки к курсору должны быть только внутри того же исходного модуля.

Требования к трансляции и сборке многопоточного приложения

Исходный текст многопоточного приложения должен транслироваться с опцией `-T` прекомпилятора, иначе при попытке использования контекстов будет выдана ошибка «`Отсутствует опция -T в вызове прекомпилятора`». Трансляция с опцией `-T` приводит к подмене предложений встроенного SQL на вызов функций из библиотеки, разработанной для многопоточных приложений (многопоточной библиотеки).

При сборке приложения должна подсоединяться версия многопоточной библиотеки.

Особенности сборки многопоточных приложений:

1) дескрипторы соединений с БД, заданные неявно (без модификатора `AT <имя соединения>`), различны для разных контекстов;
2) если происходит обращение к глобальным дескрипторам предложений встроенного SQL, то эти вызовы необходимо синхронизировать (используя семафоры, критические секции и т.д.), иначе произойдет нарушение внутренних структур библиотеки;
3) особенности интерфейса нижнего уровня (Call-интерфейса) СУБД ЛИНТЕР требуют, чтобы первое обращение к СУБД из текущей сессии было сделано из одного потока. Это связано с особенностями заполнения его внутренних структур. Иначе возможно появление ошибки СУБД ЛИНТЕР «`Ошибка приема сообщения`».

Ниже приведены два схематичных примера, показывающих, как можно использовать контексты в многопоточных приложениях.

Пример использования одного контекста несколькими потоками

```
allocate :ctx
use :ctx
connect
spawning threads...
free :ctx
thread 1,2.. ()
{
USE :ctx
mutex
. . .
unmutex
}
```

Пример использования раздельных контекстов несколькими потоками

```
allocate :ctx1
allocate :ctx2
...
spawning threads...
free :ctx1
free :ctx2
...
thread 1,2.. ()
{
USE :ctx1,2 ..
connect
. . .
}
```

Объявление контекстной переменной

Назначение

Объявление контекстной переменной для использования в операторах, предназначенных для организации многопоточной обработки. Область видимости этой переменной такая же, как любой другой переменной основного языка.

Синтаксис

```
<контекстная переменная>::=
CONTEXT <имя переменной> [, <имя переменной> ….];
```

Пример

```
EXEC SQL BEGIN DECLARE SECTION;
 CONTEXT ctx1,ctx2,ctx3;
EXEC SQL END DECLARE SECTION;
```


Разрешение на создание потока

Назначение

Обеспечение совместимости с Oracle Pro*C.

Синтаксис

```
<разрешить создание потока>::=
EXEC SQL ENABLE THREADS;
```

Описание

Конструкция <разрешить создание потока> используется только для совместимости с прекомпилятором Oracle Pro*C и в прекомпиляторе встроенного SQL СУБД ЛИНТЕР игнорируется, однако, если она включается в текст программы, то должна быть первым исполняемым SQL-оператором в многопоточном приложении (не в потоках) и вызываться перед вызовом директивы создания контекста.

Создание контекста

Назначение

Инициализации контекста.

Синтаксис

```
<создание контекста>::=
EXEC SQL CONTEXT ALLOCATE:<имя контекстной переменной>;
```

Описание

1) <Имя контекстной переменной> задает имя переменной основного языка типа CONTEXT. Переменная должна быть описана в секции описаний переменных основного языка в области видимости оператора.
2) При выполнении оператора для контекстной переменной выделяется память, и заполняются внутренние структуры контекста, на который ссылается данная контекстная переменная.

Установка текущего контекста

Назначение

Привязка контекста к конкретному потоку.

Синтаксис

```
<установка контекста>::=
EXEC SQL CONTEXT USE:<имя контекстной переменной>;
```

Описание

1) <Имя контекстной переменной> задает имя переменной основного языка типа CONTEXT. Переменная должна быть описана в секции описаний

переменных основного языка в области видимости оператора и предварительно инициализирована с помощью оператора CONTEXT ALLOCATE.

2) Оператор служит для уведомления о том, что далее, до конца модуля или до следующего оператора <использование контекста>, все предложения встроенного SQL и все курсоры должны выполняться в данном потоке, во всех вызовах функций состояния будет использоваться переменная <имя контекстной переменной>.

3) Оператор не является обязательным. Если он не был задан, используется контекстная переменная по умолчанию CtxPCI_. Она не нуждается в инициализации. При этом контекст по умолчанию возможно использовать только в одном из нескольких потоков (или в единственном), во всех остальных потоках необходимо явно связывать поток с контекстной переменной с помощью оператора <установка контекста>.

Освобождение контекста

Назначение

Освобождение контекста.

Синтаксис

```
<освобождение контекста>::=
EXEC SQL CONTEXT FREE:<имя контекстной переменной>;
```

Описание

1) <Имя контекстной переменной> должно ссылаться на переменную типа CONTEXT. Переменная должна быть описана в секции описаний переменных основного языка в области видимости оператора и инициализирована исполнением директивы CONTEXT ALLOCATE.

2) При выполнении оператора освобождается память, занимаемая внутренними структурами контекста, на который ссылается данная контекстная переменная.

3) Если будет произведена попытка использования или освобождения контекстной переменной, память под которую не была выделена ранее исполнением оператора CONTEXT ALLOCATE, возникнет исключительная ситуация, связанная с нарушением защиты памяти.

Пример использования контекстов в многопоточных приложениях приведен в приложении 3.

Анализ результатов обработки запросов в потоке

Для анализа результатов обработки запросов в потоке используются те же псевдопеременные, что и для однопоточного приложения. Особенность состоит в том, что для каждого контекста они имеют собственные значения.

Контроль ошибочных ситуаций

Спецификация ошибочных ситуаций и действий по их обработке

Назначение

Управление обработкой ошибочных ситуаций при выполнении встроенных SQL-запросов.

Синтаксис

```
<управление ошибочной ситуацией>::=
EXEC SQL WHENEVER <ситуация> THEN <действие>;
<ситуация>::= {SQLERROR | SQLWARNING | NOT FOUND}
<действие>::= {STOP | CONTINUE | GOTO <метка>
| CALL <процедура-обработчик>}
```

Описание

1) `<Метка>` – метка оператора основного языка, куда необходимо перейти в случае ошибки. Директива `WHENEVER` является декларативной. Она управляет дальнейшей работой программы в случае выявления ошибки при выполнении SQL-запроса. Директива `WHENEVER` относится ко всем выполняемым операторам встроенного SQL, встретившимся после нее в тексте программы до следующей директивы `WHENEVER`.
2) Все ошибочные ситуации можно разделить на 3 категории:
   - `SQLERROR` (ошибка) – СУБД ЛИНТЕР не может выполнить оператор;
   - `SQLWARNING` (предупреждение) – сомнительные преобразования типов данных и т.п.;
   - `NOT FOUND` (данные не найдены) – не найдено ни одной записи, удовлетворяющей заданному условию.
3) Действия при возникновении подобных ситуаций могут быть следующими:
   - `STOP` – завершение выполнения программы;
   - `CONTINUE` – ошибочная ситуация игнорируется, выполнение программы продолжается;
   - `GOTO` – переход на заданную метку;
   - `CALL` – вызов заданной `<процедуры-обработчика>`. `<Процедура-обработчик>` – это функция основного языка, которая не должна иметь аргументов.

Примечание
Если директива `WHENEVER` в программе отсутствует либо объявлена не в начале программы, по умолчанию (до первого объявления `WHENEVER`) действует режим `CONTINUE` для всех возникающих ошибочных ситуаций, т. е. никакая обработка ошибки не производится. В данном случае пользовательская программа может выполнять самостоятельную обработку ошибок, специфическую для каждого конкретного SQL-запроса, используя значение переменных `ErrPCI_` и `CntPCI_`.

Управление прекомпиляцией

Определение и отмена определения макропеременных

Назначение

Определение/отмена определения макропеременной.

Синтаксис

```
<определение макропеременной>::=
EXEC LINTER DEFINE <имя макропеременной>;

<отмена определения макропеременной>::=
EXEC LINTER UNDEF <имя макропеременной>;
```

Описание

1) Имена объявленных макропеременных могут быть использованы в директивах IFDEF/IFNDEF.
2) По умолчанию определена макропеременная SQL.

Директивы условной трансляции

Начало блока условной трансляции

Назначение

Организация условной трансляции исходного текста.

Синтаксис

```
<трансляция при определенной макропеременной>::=
EXEC LINTER IFDEF <имя макропеременной>;

<трансляция при неопределенной макропеременной>::=
EXEC LINTER IFNDEF <имя макропеременной>;
```

Описание

1) Текст за директивой `IFDEF/IFNDEF` включается во входной поток, только если заданная макропеременная определена/не определена.
2) В случае вложенных директив `IFDEF/IFNDEF` для включения текста во входной поток должны выполняться все условия.

Изменение состояния условной трансляции

Назначение

Переключение состояния условной трансляции.

Синтаксис

```
<переключение условной трансляции>::=
EXEC LINTER ELSE;
```

Описание

Если до этой директивы условие трансляции выполнялось, то после нее не выполняется, и наоборот.

Завершение условной трансляции

Назначение

Завершение блока условной трансляции.

Синтаксис

```
<завершение условной трансляции>::=
EXEC LINTER ENDIF;
```

Описание

Директива завершает блок условной трансляции и отменяет условие, заданное в соответствующей директиве `IFDEF/IFNDEF`.

Настройка прекомпилятора

Размер рабочего буфера

Назначение

Установка размера рабочего буфера для приема данных.

Синтаксис

```
<размер рабочего буфера>::=
EXEC LINTER OPTION AREASIZE = <размер буфера>;
<размер буфера>::= целое положительное число
```

Описание

Директива задает размер рабочего буфера для приема данных при обработке предложений SQL.

Если директива не выдана, по умолчанию размер буфера равен 4096 байтам.

Максимальный размер дескриптора

Назначение

Определение максимально допустимого количества описателей в дескрипторе.

Синтаксис

```
<максимальное количество описателей>::=
EXEC LINTER OPTION MAX ENTRIES = <количество описателей>;
<количество описателей>::= целое положительное число
```

Описание

Если директива не выдана, по умолчанию дескриптор создается для 32 описателей.

Класс памяти для коммуникационной области

Назначение

Задание класса памяти для коммуникационной области `sqlca`.

Синтаксис

```
<класс памяти sqlca>::=
EXEC LINTER OPTION MODULE =
{ EXTERNAL | LOCAL | MAIN | IMPORT};
```

Описание

Директива устанавливает класс памяти для коммуникационной области `sqlca`:

- `EXTERNAL` – переменная `sqlca` имеет класс памяти `extern`;

- LOCAL – переменная sqlca имеет класс памяти static;
- MAIN – переменная sqlca описана в данном модуле;
- IMPORT – переменная sqlca импортируется из библиотеки PCL.

Если директива не выдана, по умолчанию устанавливается класс IMPORT.

Включение файла

Назначение

Включение в исходный модуль программы текста другого файла (например, заголовочного файла).

Синтаксис

```
<добавить файл>::=
EXEC SQL INCLUDE <спецификация файла>;
```

Описание

1) <Спецификация файла> должна однозначно определять местонахождение файла на локальном или сетевом диске.
2) Если файл с именем <спецификация файла> отсутствует, будет выдан код завершения «Ошибка открытия файла», и претрансляция аварийно завершится.
3) Директива транслируется в #include <filename>. Исключение – файлы sqlca.h и sqlda.h, включение которых приводит к генерации дополнительного кода помимо директивы препроцессора #include.

Примечания

1. В данной версии прекомпилятора оператор INCLUDE служит только для уведомления прекомпилятора о том, что в тексте модуля встретилось включение файла sqlca.h или sqlda.h (EXEC SQL INCLUDE SQLCA;, EXEC SQL INCLUDE SQLDA;), при этом файлы sqlca.h и sqlda.h не обязательно находятся в указанном каталоге, но при последующей обработке программы препроцессором языка C/C++ они должны быть в указанном месте.
2. В данной версии РСС все файлы, включенные в текст программы оператором INCLUDE, не анализируются претранслятором.

Получение характеристик ЛИНТЕР-сервера

Назначение

Получить характеристики ЛИНТЕР-сервера.

Синтаксис

```
<характеристики ЛИНТЕР-сервера>::=
EXEC LINTER GET SERVER [AT <имя соединения>]
VERSION :<номер версии>, :<релиз версии>, [:<номер сборки>]
<номер версии>::= переменная основного языка
```

```
<релиз версии>::= переменная основного языка
<серийный номер>::= переменная основного языка
```

Описание

1) В переменной <номер версии> типа int8, int16, int32 передается номер версии СУБД ЛИНТЕР, которой соответствует данный прекомпилятор.
2) В переменной <релиз версии> типа int8, int16, int32 передается номер реализации версии СУБД ЛИНТЕР, которой соответствует данный прекомпилятор.
3) В переменной <номер сборки> типа int8, int16, int32 передается номер сборки для лицензионного соглашения на поставку СУБД ЛИНТЕР.

Пример

```
EXEC LINTER GET SERVER VERSION :vers, :release, :build
```


Прекомпилятор встроенного SQL

Прекомпилятор встроенного SQL (далее по тексту PCC) предназначен для прекомпилирования исходных текстов программ, написанных на языке программирования C/C++ с использованием конструкций встроенного SQL СУБД ЛИНТЕР. В приложении 4 приведен фрагмент программы, которая иллюстрирует большую часть возможностей прекомпиляторного интерфейса (PCI).

Условия применения

Выходом PCC является текст программы, содержащий операторы только языка C/C++. Для того, чтобы полученный текст мог быть оттранслирован стандартным C/C++ транслятором, PCC вставляет в текст C/C++ программы директиву подключения include-файла, содержащего объявления процедур и глобальных переменных библиотеки прекомпиляторного интерфейса (`#include "pci.h"`). Чтобы компилятор языка C/C++ нашел указанный include-файл, путь на него должен быть прописан в переменной среды окружения `PATH` или указан компилятору с помощью параметра командной строки.

Для сборки программы на языке C/C++ необходимо включить в файл проекта библиотеку прекомпиляторного интерфейса `pci.a`.

Если задан режим проверки семантики встроенных SQL-предложений (опция -S прекомпилятора), то должна быть активной БД, с которой будет осуществляться соединение в процессе прекомпиляции текста модуля.

Характеристики прекомпилятора

В таблице 12 приведены основные характеристики собственно PCC, в таблице 13 отражены ограничения, накладываемые PCC на конструкции встроенных SQL-запросов.

Входными данными для PCC являются исходные тексты языка программирования C/C++ с элементами встроенного SQL.

Выходными данными PCC является текст программы на языке C/C++, в котором конструкции встроенного SQL заменены вызовами библиотечных процедур и/или блоками текста языка C/C++.

Таблица 12. Основные характеристики PCC СУБД ЛИНТЕР

Характеристика	Значение
Число вложений блоков, не более	40
Число директив class и struct для C++, не более	40
Число глобальных переменных основного языка, не более	25
Число глобальных предложений и курсоров встроенного языка, число локальных предложений и курсоров в модуле, не более	64
Число соединений с БД, не более	10
Число макроопределений, не более	30
Число включаемых файлов, не более	30
Число модулей встроенного языка, не более	40

Характеристика	Значение
Длина строки прекомпилируемой программы (в байтах), не более	255
Число объявлений хранимых процедур, не более	256
Число дескрипторов (не считая переменных основного языка), не более	64
Число точек входа дескриптора, не более	32
Число переменных основного языка, не более	256
Число операторов условной трансляции (IF, ELSE, ENDIF), не более	10
Число параметров для одного предложения, не более	256
Размер приемного буфера по умолчанию (в байтах), не более	4096
Размер предложения встроенного SQL, задаваемого непосредственно (в байтах), не более	4096

Таблица 13. Ограничения на встроенные SQL-запросы

Характеристика	Значение
Длина имени модуля встроенного языка (в байтах), не более	32
Длина имени предложения встроенного языка (в байтах), не более	32
Длина имени локального курсора (в байтах), не более	14
Длина имени глобального курсора (в байтах), не более	18

Обращение к прекомпилятору

Для вызова PCC необходимо в интерпретаторе команд операционной системы подать команду:

```
PCC [-{M|E|L|I}ASDTOJWVC[+] P{<число точек входа>}]
[[-N <имя сервера>]
    -U <имя пользователя>[/<пароль>]]<входной файл> [<выходной
 файл>]
```

Описание

1) `C` – задает основной язык C (этот параметр используется по умолчанию).
2) `C+` – задает основной язык C++.
3) `D` – задает генерацию отладочной информации (вставляет директиву `#line` на каждой строке, содержащей оператор встроенного языка). Генерация отладочной информации позволяет просматривать в отладчике прекомпилируемый, а не результирующий текст программы. По умолчанию она отключена.
4) `J` – режим совместимости с ESQL СУБД Ingres.
5) `W` – получить версию прекомпилятора.
6) `O` – задает Oracle метод обработки кодов завершения. При задании этого ключа в тексте C/C++ программы будут генерироваться коды для проверки кодов завершения через значение поля `sqlca.sqlcode`, при отсутствии ключа проверка кодов завершения будет выполняться через глобальную переменную `ErrPCI_`.
7) `A` – задает режим запрета заполнения коммуникационной области `sqlca` (действителен только при отсутствии опции `-O`).

8) T – разрешает использовать директивы управления многопоточностью (ENABLE THREADS, CONTEXT {ALLOCATE | USE | FREE}).

9) N<имя сервера> – задает имя удаленного ЛИНТЕР-сервера.

10) U<имя пользователя/пароль> – задает имя и пароль, под которыми прекомпилятор будет иметь доступ к БД как пользователь (большие и малые буквы при вводе имени и пароля пользователя различаются).

11) S – задает режим проверки семантики предложений встроенного SQL. Этот ключ должен использоваться, если предполагается создание хранимых процедур непосредственно в процессе прекомпиляции модуля. Иначе эта опция не должна использоваться.

12) M, E, L, I – задают область видимости области описания дескрипторов (переменная sqlda):

- M – главный модуль: переменная sqlda описана и инициализирована в нем;
- E – переменная sqlda внешняя по отношению к модулю;
- L – переменная sqlda, заданная в модуле, локальная;
- I – переменная sqlca импортируется из библиотеки.

13) V – задает режим совместимости со старой версией PCI по исполнению операторов PREPARE, OPEN, FETCH.

Примечания

1. Буквенные коды ключей допустимо задавать большими и малыми буквами.
2. Если указывается одновременно несколько ключей, все они пишутся слитно, и знак дефиса ставится один раз перед всей группой ключей, например: -cdo. Исключение составляют ключи -n и -u, которые должны вводиться отдельно.
3. Если используется ключ -s, обязательно должны задаваться ключи -n (только для удаленного сервера, не объявленного сервером по умолчанию), -u.
4. Параметр <входной файл> задает спецификацию файла, содержащего исходный текст прекомпилируемой программы. Если расширение имени файла не задано, по умолчанию используется .pc.
5. Параметр <выходной файл> задает спецификацию файла, в который должен быть помещен результат прекомпиляции (исходный текст программы на языке C). Если спецификация выходного файла не задана, то он создается в текущем каталоге, а его имя совпадает с именем входного файла. Если расширение имени выходного файла не задано, оно берется равным .c, если основным языком является C, или .cpp, если основным языком является C++.

Примеры

1)

```
pcc -c sample.pc
(выходной файл sample.c)
```

2)

```
pcc -c+ sample.pc
(выходной файл sample.cpp)
```

3)

```
pcc -dc+ sample.pc
(выходной файл sample.cpp с отладочной информацией)
```

4)

```
pcc sample.pc sample1.cpp
(входной файл sample.pc, выходной файл sample1.cpp)
```

5)

```
pcc -c+s –nLinter -uSYSTEM/MANAGER8 sample.pc
(выходной файл sample.cpp; пользователь SYSTEM с паролем MANAGER8
 подключается к ЛИНТЕР-серверу)
```

Компилятор хранимых процедур

Компилятор хранимых процедур (SPC) входит в состав прекомпилятора PCC, но может использоваться отдельно. Подробнее см. приложение 5.

Коды завершения

Коды завершения этапа прекомпиляции

Коды завершения этапа прекомпиляции представлены в таблице 14.

Таблица 14. Коды завершения этапа прекомпиляции

Код завершения	Описание	Примечание
F_INTERN	Внутренняя ошибка прекомпилятора.	Обратитесь к разработчикам РСС.
E_MANYVAR	Превышен лимит числа переменных в программе.	Общее число переменных, описанных во всех секциях описаний, больше 100.
E_MANYLEV	Превышен лимит уровня вложения блоков.	Число вложений блоков в C/C++ программе больше 40.
E_MANYSTAT	Превышен лимит числа объявлений предложений.	Число объявлений предложений больше 64.
E_MANYCUR	Превышен лимит числа объявлений курсоров.	Число объявлений курсоров больше 64.
E_MANYPROC	Превышен лимит числа объявлений хранимых процедур.	Число объявлений хранимых процедур (директив CREATE PROCEDURE и DECLARE PROCEDURE) больше 256.
E_MANYDB	Превышен лимит числа объявлений БД.	Число объявлений БД больше 10.
E_MANYVARST	Превышен лимит числа переменных в одном предложении.	Число используемых переменных в одном предложении больше 256.
E_MANYINC	Превышен лимит уровня вложенности файлов.	Число вложений файлов (директив EXEC SQL INCLUDE) больше 30.
E_MANYMAC	Слишком много макропеременных.	Число использованных макропеременных больше 30.
E_MANYMOD	Превышен лимит числа объявлений модулей встроенного языка.	Число объявлений модулей встроенного языка больше 40.
E_MANYIF	Слишком большая вложенность условной трансляции.	Вложенность директив условной трансляции превышает 10.
E_MANYDESCR	Превышен лимит числа объявлений дескрипторов встроенного языка.	Число объявлений дескрипторов встроенного языка больше 64.
E_MANYCLS	Превышен лимит числа объявлений классов (C++).	Число объявлений классов основного языка (C++) больше 40.
E_MANYCLSVAR	Превышен лимит числа объявлений переменных основного языка, принадлежащих классам.	Число объявлений переменных основного языка, принадлежащих классам, больше 256.

Код завершения	Описание	Примечание
E_STRTOOLONG	Слишком длинная строка.	Строка, задающая предложение SQL, превышает 256 символов.
E_NOMEM	Превышен размер доступной памяти.	При работе прекомпилятора невозможно выделить память.
E_HASHFULL	Превышен размер хэш-таблицы.	При работе прекомпилятора переполнилась хеш-таблица имен.
E_BUFFULL	Превышен размер буфера.	При работе прекомпилятора переполнился буфер строки (4096 байт).
E_THEAPFULL	Превышен размер количества лексем.	Число лексем в предложении SQL больше 256.
E_TBUFFULL	Превышен размер буфера лексем.	Общий размер предложения SQL превышает 2096 байт.
E_UNEXPEOF	Неожиданный конец файла.	Конец файла в середине директивы прекомпилятора.
E_COMNOTEND	Конец файла в середине комментария.	Конец файла в середине комментария.
E_CLOSLEV	Закрывающая скобка без предшествующей открывающей скобки.	Несоответствие фигурных скобок (операторные скобки основного языка).
E_INVCHAR	Непечатный символ.	В директиве прекомпилятора встретился недопустимый символ.
E_UNKNAME	Неопределенное имя.	В качестве переменной прекомпилятора или основного языка использовано неопределенное имя.
E_DUPNAME	Повторное определение имени.	В качестве переменной прекомпилятора или основного языка определяется имя, уже определенное ранее.
E_CANTIND	Недопустима индикаторная переменная.	В данном контексте использование индикаторной переменной недопустимо.
E_CANTARR	Недопустим массив.	В данном контексте использование массива недопустимо.
E_CANTPTR	Недопустим указатель.	В данном контексте использование указателя недопустимо (не *CHAR, *BIT).
E_MANYDIM	Размерность массива больше 1.	В качестве переменных основного языка нельзя использовать многомерные массивы.
E_UNSFLOAT	Плавающие типы не могут быть unsigned.	Вещественные числа не могут быть объявлены как беззнаковые.
E_NODECLSEC	Ошибка в секции объявлений.	Синтаксическая ошибка.
E_EMPTARR	Объявлен массив из 0 элементов.	Пустой массив недопустим. Номера элементов должны начинаться с 1.

Код завершения	Описание	Примечание
E_INVTYP	Неверный тип параметра.	Типы формального и фактического параметров вызова процедуры не совпадают.
E_INVPARCNT	Неверное число параметров вызова процедуры.	Несовпадение числа параметров при объявлении процедуры и при ее вызове.
E_DIFSIZE	Разные размеры массивов главных и индикаторных переменных.	Недопустимо использование индикаторной переменной с размером, отличным от размера главной переменной.
E_TWOCURS	Нельзя объявить два курсора для одного предложения.	Для одного и того же предложения нельзя объявить два разных курсора (ограничение действует в текущей версии).
E_NODECLSEC	Нет секции объявлений на верхнем уровне.	В прекомпилируемом модуле не выделено место под секцию деклараций внутренних переменных (внутренняя ошибка прекомпилятора).
E_NOTAREA	Нет коммуникационной области.	Отсутствует директива INCLUDE SQLCA.
E_UNEXPEND	END без предшествующего BEGIN.	В тексте прекомпилируемой программы встретилась директива END DECLARE SECTION, а директивы BEGIN DECLARE SECTION не было.
E_TVAREXP	Ожидалась текстовая переменная.	В качестве имени пользователя и пароля, а также буфера для BLOB, допустимы только переменные типов CHAR и VARCHAR.
E_SINTEXP	Ожидалась целая переменная.	В качестве индикаторной переменной, счетчика выполнения (конструкция FOR) и смещения в BLOB-операциях должна быть использована переменная целого типа, причем в первых двух случаях она должна быть двухбайтовой.
E_CURSEXP	Ожидалась переменная типа CURSOR.	Использование курсорной переменной без ее объявления.
E_CTXEXP	Ожидалась переменная типа CONTEXT.	Использование контекстной переменной без ее объявления.
E_DSCEXP	Ожидалась переменная типа DESCRIPOR.	Использование дескрипторной переменной без ее объявления.
E_NOTOPER	Ожидался оператор встроенного языка.	После префикса EXEC SQL встретилась нераспознанная конструкция.
E_CONDEXP	Ожидалось условие проверки.	После WHENEVER должно встретиться либо NOT FOUND,

Код завершения	Описание	Примечание
		либо SQLERROR, либо SQLWARNING.
E_ACTNEXP	Ожидалось действие по условию.	После THEN должно встретиться либо STOP, либо CONTINUE, либо GOTO.
E_TOBJEXP	Ожидался тип объявляемого объекта.	В директиве DECLARE должен быть указан тип объекта: либо STATEMENT, либо CURSOR_PCI, либо DATABASE.
E_LANGEXP	Ожидалось имя встроенного языка (SQL) или СУБД ЛИНТЕР.	После слова EXEC должно встретиться слово SQL, либо LINTER.
E_OPTNEXP	Ожидалась опция СУБД ЛИНТЕР.	После EXEC LINTER должно встретиться AREASIZE, OPTION, GET.
E_BISEEXP	Ожидались ключевые слова, BIND или SELECT.	После SQL DESCRIBE должно встретиться либо BIND, либо SELECT. Ожидалось описание типа параметра. Ожидалась декларация типа параметра при описании хранимой процедуры. Ожидался модификатор параметра. Ожидались слова IN, OUT или INOUT перед именем параметра при описании хранимой процедуры.
E_SEMIEXP	Ожидался конец директивы.	Директиву прекомпилятора должен завершать символ ';'.
E_UNKDECL	Нераспознанный описатель.	Нераспознанная конструкция.
E_STRNOTEND	Нет закрывающей кавычки.	При чтении строки в кавычках не найдена закрывающая кавычка.
E_BOLEXP	Префикс директивы не в начале предложения.	Нельзя смешивать в одной строке директиву прекомпилятора и предложение исходного языка.
E_EOLEXP	Ожидался конец строки.	Нельзя смешивать в одной строке директиву прекомпилятора и предложение основного языка.
E_UNEXPELSE	ELSE без IFDEF или IFNDEF.	Неверная последовательность директив условной трансляции.
E_UNEXPEIF	Неверная последовательность директив условной трансляции.	Не закрыт блок условной трансляции.
E_NOENDIF	Отсутствие директивы ENDIF после IFDEF.	

Код завершения	Описание	Примечание
E_BLOPEXP	Ожидалась операция с типом BLOB.	После фразы BLOB должно встретиться либо ADD, либо CLEAR.
E_ENCLREPSEL	Вложенные директивы REPEATED SELECT недопустимы.	
E_TOKNEXP	Ожидалась лексема <имя лексемы>.	При обработке текста программы ожидалась указанная лексема, но встретилось нечто иное.
F_OPEN	Ошибка открытия файла <имя файла>.	Ошибка при открытии прекомпилируемого файла.
F_CLOSE	Ошибка создания файла <имя файла>.	Ошибка при создании файла результата прекомпиляции.
F_POSIT	Ошибка позиционирования в выходном файле.	Ошибка при записи в файл результата прекомпиляции.
E_IDNEXP	Ожидался идентификатор.	Ожидался идентификатор, а встретилось нечто иное.
E_ENCLMODUL	Вложенные модули недопустимы.	Данная версия препроцессора не допускает использования вложенных модулей.
E_INVLEV	Неверный уровень вложенности предложений MODULE <имя модуля> и END MODULE <имя модуля>.	Директивы MODULE и END MODULE должны находиться на одинаковом уровне вложенности блоков основного языка.
E_EXPMODULBEG	Отсутствует директива MODULE <имя модуля>.	В блоке основного языка встретилась директива END MODULE <имя модуля> без директивы MODULE<имя модуля>.
E_EXPMODULEND	Отсутствует директива END MODULE <имя модуля>.	Блок основного языка, содержащий директиву MODULE <имя модуля>, завершился без директивы END MODULE <имя модуля>.
E_NOTHREAD	Отсутствует опция -T в вызове прекомпилятора.	Попытка использования директив ENABLE THREADS, CONTEXT {ALLOCATE | USE | FREE} без опции -T (разрешение многопоточности) в вызове прекомпилятора.
E_TOKNEXP	Ожидалась неизвестная лексема.	Внутренняя ошибка прекомпилятора. Обратитесь к разработчикам.
E_SETNOIMP	Опция в SET DESCRIPTOR распознается синтаксическим	Используется старая версия прекомпилятора.

Код завершения	Описание	Примечание
	анализатором, но не обрабатывается.	
E_GETNOIMP	Опция в GET DESCRIPTOR распознается синтаксическим анализатором, но не обрабатывается.	Используется старая версия прекомпилятора.
E_INQUIRENOIMP	Опция в INQUIRE_SQL распознается синтаксическим анализатором, но не обрабатывается.	В данной версии прекомпилятора эта опция не обрабатывается.
E_OPTNOTIMPL	Опция распознается синтаксическим анализатором, но не обрабатывается.	В данной версии прекомпилятора эта опция не обрабатывается.

Коды завершения этапа выполнения

В таблице 15 приведены коды завершения конструкций встроенного SQL, возвращаемые PCI – претранслятором встроенного SQL (для языка C).

Коды завершения, возвращаемые СУБД ЛИНТЕР на этапе выполнения, приведены в документе «СУБД ЛИНТЕР. Справочник кодов завершения».

Таблица 15. Коды завершения этапа выполнения

Идентификатор кода	Числовое значение	Причина	Комментарий
OK_	0	Нормальное завершение.	
ErrPCI_NotFound	3000	Конец выборки.	Не найдено (не удалено, не заменено, не добавлено) ни одной записи.
ErrPCI_LimRes	3001	Превышение ресурсов (пределов) реализации.	Превышен один из пределов для данной реализации: не более 256 переменных в одном операторе; не более 4096 символов в тексте оператора.
ErrPCI_IncTyp	3002	Несовместимость типов.	Попытка выбрать значение поля записи в переменную, тип которой несовместим с типом выбираемого значения.
ErrPCI_InvVarCnt	3003	Неверное число переменных.	Конструкция USING содержит иное число

Идентификатор кода	Числовое значение	Причина	Комментарий
			входных переменных, чем было указано при объявлении курсора, либо конструкция INTO содержит иное число выходных переменных, чем выбирает оператор SELECT.
ErrPCI_NoMem	3004	Не хватает памяти.	Нехватка памяти при резервировании рабочих структур.
ErrPCI_InvPar	3005	Синтаксическая ошибка в запросе.	При анализе текста SQL-предложения обнаружена синтаксическая ошибка.
ErrPCI_ManyKor	3006	Выбрано слишком много записей.	Попытка выполнить оператор SELECT, который выбирает больше записей, чем размерность массива, указанного в качестве приемника записей: приемник данных – скаляр, а выбрано более 1 записи.
ErrPCI_IncDim	3007	Несоответствие размеров массивов.	В качестве переменных, участвующих в одном операторе, указаны массивы разных размерностей, либо переменные-массивы и входные параметры имеют неодинаковые длины, либо переменные-массивы и входные параметры имеют длину меньшую, чем указано повторов в опции FOR.
ErrPCI_InvFetchPar	3008	Неверные параметры FETCH.	При выборе одновременно нескольких записей указан модификатор команды FETCH, отличный от NEXT.
ErrPCI_ChanAlrOpen	3010	Канал уже открыт.	Попытка повторной подачи оператора CONNECT для той же БД.

Идентификатор кода	Числовое значение	Причина	Комментарий
ErrPCI_ChanNotOpen	3011	Канал не был открыт.	Попытка работы с каналом до установления логической связи с помощью команды CONNECT либо попытка закрыть неоткрытый канал.
ErrPCI_StatNotPrep	3012	Предложение не подготовлено – не было PREPARE.	Попытка выполнить предложение, которое не подготовлено к выполнению командой PREPARE.
ErrPCI_CursNotOpen	3013	Курсор не открыт.	Попытка работы с курсором, который еще не был открыт.
ErrPCI_CursAlrOpen	3014	Курсор уже был открыт.	Попытка повторного открытия курсора.
ErrPCI_NotDesc	3015	Предложение не подготовлено – не было DESCRIBE.	Попытка выполнить оператор OPEN или FETCH с конструкцией USING DESCRIPTOR для дескриптора, не заполненного командой SQL DESCRIBE.
ErrPCI_DescSmall	3016	В дескрипторе слишком мало элементов.	Зарезервированное число элементов в дескрипторе недостаточно для занесения информации обо всех переменных.
ErrPCI_CursNotFetch	3017	Не был выполнен FETCH – выборка BLOB невозможна.	При работе с BLOB-значением запись, содержащую это значение, необходимо предварительно выбрать с помощью оператора FETCH.
ErrPCI_InvDescIdx	3019	Неверный идентификатор дескриптора.	Неверный номер точки входа дескриптора (<1).
ErrPCI_DescNotAll	3020	Дескриптор не инициализирован.	Не все описатели дескриптора инициализированы.
ErrPCI_NullNoInd	3021	Нет индикатора NULL-значений.	Возвращено NULL-значение, а индикаторной переменной нет.

Идентификатор кода	Числовое значение	Причина	Комментарий
ErrPCI_NoDest	3022	Не привязан выходной параметр.	В динамическом SQL-предложении к выходному параметру не привязан описатель дескриптора.
ErrPCI_InvVal	3023	Неверное значение параметра.	Значение не соответствует типу данных параметра.
ErrPCI_NotBind	3024	Не привязан входной параметр.	В динамическом SQL-предложении к входному параметру не привязан описатель дескриптора.
ErrPCI_NoContext	3025	Контекст не инициализирован.	Не выполнен оператор инициализации контекста в многопоточной программе.
ErrPCI_WrongPrototype	3026	Прототип процедуры не соответствует ее вызову.	
ErrPCI_TooLong	3027		Слишком длинная строка во входном параметре.
ErrPCI_NullProhibited	3028		Поле не может быть NULL.
ErrPCI_StatNotSelect	3029		Попытка создать курсор для не select-предложения.
ErrPCI_InvIVarCnt	3030		При привязке параметров не совпало число формальных и фактических входных параметров.
ErrPCI_InvOVarCnt	3031	Несовпадение количества параметров.	При привязке параметров не совпало число формальных и фактических выходных параметров.
ErrPCI_NoncompVer	3032	Несовместимые версии сервера и библиотеки.	
ErrPCI_Internal	3100	Неверный указатель структуры.	Внутренняя ошибка в библиотеке: испорчено содержимое внутренних структур данных.

Стандартные переменные состояния

Для проверки кода завершения на этапе выполнения используются переменные состояния SQLCODE и SQLSTATE.

SQLCODE – переменная типа LONG. Допустимые значения:

- 0 – нормальное завершение;
- 100 – нет данных;
- < 0 – код завершения обработки SQL-запроса.

SQLSTATE – переменная типа CHAR[6], содержащая пятисимвольную строку стандартного кода завершения в формате СУБД ORACLE (см. таблицу 16).

Таблица 16. Коды завершения при использовании переменной SQLCODE

Код	Комментарий
00000	Нормальное завершение
22012	Деление на ноль
22003	Значение вышло из диапазона
22018	Ошибка явного преобразования типов
02000	Нет данных
22020	Превышен один из пределов реализации встроенного SQL
21000	Выбрано слишком много записей
08006	Канал не открыт
01005	Дескриптор имеет слишком мало точек входа
07008	Число точек входа дескриптора меньше числа фактических параметров
07009	Неверный номер точки входа дескриптора
22002	NULL-значение. Нет индикаторной переменной
30000	Ошибка в тексте предложения SQL
07000	Дескриптор не описан
22024	Слишком большая строка
33000	Дескриптор не инициализирован
07001	Неверное число входных переменных
07002	Неверное число выходных переменных

Приложение 1
Структура дескриптора t_sqlda

```
struct t_sqlda
{
    int N;            /* Число описателей динамических параметров
 */
    void(*DF)(void);  /* Зарезервировано */
    char **V;         /* Массив указателей на адреса главных
 переменных */
    int *L;           /* Указатель на массив, в котором хранятся
 длины главных переменных */
    short *T;         /* Массив указателей на типы главных
 переменных */
    int *D;           /* Массив указателей на размерности массивов
 главных переменных */
    short *P;         /* Значение точности для типов данных
 NUMERIC */
    short *SC;        /* Значение масштаба для типов данных
 NUMERIC */
    short *RL;        /* Массив длин формальных параметров */
    short *BL;        /* Массив реальных длин параметров */
    char **I;         /* Массив указателей на адреса индикаторных
 переменных */
    int F;            /* Количество переменных, реально найденных
 оператором SQL DESCRIBE */
    char **S;         /* Массив указателей на адреса имен главных
 переменных */
    short *M;         /* Указатель на максимальную длину имени */
    short *C;         /* Указатель на длину имени текущей
 переменной */
    char **X;         /* Массив указателей на адреса имен
 индикаторных переменных */
    short *Y;         /* Максимальная длина имени индикаторной
 переменной */
    short *Z;         /* Указатель на текущую длину имени
 индикаторной переменной */
    short *MN;        /* Указатель флага состояния */
    int NL;           /* Максимальное число описателей параметра в
 данном дескрипторе */
    int FR;           /* Дескриптор был использован оператором SQL
 DESCRIBE */
    int UD;
};
```

Описание полей дескриптора приведено в таблице П1.1.

Таблица П1.1. Описание полей дескриптора

Поле	Описание поля
N	Суммарное максимальное число входных или выходных описателей параметров в данном дескрипторе. Это значение устанавливается при выделении памяти для дескриптора и равно max_vars.
**V	Указатель на массив адресов буферов данных. Во время инициализации обнуляются элементы V[0] - V[N-1]. Для дескриптора SELECT LIST память под буферы должна быть выделена перед выполнением предложения EXEC SQL FETCH ... USING DESCRIPTOR…. Полученные данные сохраняются в V[i]. Для дескриптора BIND VARIABLES инициализация буферов должна быть проведена перед выполнением предложения EXEC SQL OPEN ... USING DESCRIPTOR…. Поиск входных переменных ведется в V[i].
*L	Указатель на массив, в котором содержатся длины фактических входных или выходных параметров, привязанных к формальным параметрам. Для дескриптора SELECT LIST значение этого поля определяется в момент получения информации о структуре ответа при выполнении предложения EXEC SQL DESCRIBE. Для дескриптора BIND значение этого поля должно быть определено перед открытием курсора командой EXEC SQL OPEN . . . USING DESCRIPTOR . . ...Это можно сделать с помощью присвоения адреса главной переменной полю V[i] дескриптора, например: bind_dp->V[i] = (char *)&Host_Var; здесь Host_Var – главная переменная, описанная в секции объявлений переменных основного языка. Таким образом, привязка главных переменных к дескрипторам осуществляется с помощью функций основного языка C/C++, а не встроенного языка.
*T	Массив кодов типов данных формальных входных или выходных параметров. Типы данных – внутренние типы библиотеки PCI (см. таблицу 9).
*D	Массив указателей на размерности массивов главных переменных.
*P	Массив значений точности для параметров типа NUMERIC.
*SC	Массив значений масштаба для параметров типа NUMERIC.
*RL	Содержит длину формального параметра (для входного параметра).
*BL	Для внутреннего использования. Содержит реальную длину привязываемого параметра (после исключительной ситуации Err_TooLong).
**I	Массив адресов главных переменных, привязанных к индикатору.
F	Число параметров, обнаруженных во время грамматического разбора предложения в операторе SQL DESCRIBE. Если F меньше нуля, это означает, что размер дескриптора слишком мал для описания того количества переменных, которое встретилось при грамматическом разборе предложения для BIND VARIABLE дескриптора или при получении структуры ответа для SELECT LIST дескриптора.
**S	Массив указателей имен главных переменных, привязанных к параметрам.
*M	Указатель максимальной длины имени параметра. Данный параметр устанавливается при инициализации дескриптора и равен max_name.
*C	Массив, в котором хранятся длины имен входных (после предложения EXEC SQL DESCRIBE BIND VARIABLES . . .) или выходных параметров

Поле	Описание поля
	запроса (после выполнения предложения EXEC SQL OPEN . . . USING DESCRIPTOR . . .).
**X	Массив указателей имен индикаторных переменных.
*Y	Указатель максимальной длины имени индикаторных переменных. Данный параметр устанавливается при инициализации дескриптора и равен max_ind_name.
*Z	Массив указателей текущих длин имен индикаторных переменных.
*MN	Флаги состояния: • 0x1 – поле V[i] заполнено непосредственными данными (без привязки к переменной основного языка); • 0x2 – поле I[i] заполнено непосредственными данными (без привязки к переменной основного языка); • 0x4 – фактический параметр номер i привязан к формальному параметру; • 0x8 – параметр номер i не может принимать NULL-значение.
NL	Максимальное число описателей параметра в данном дескрипторе по умолчанию MAX ENTRIES. Переменная NL – лимит количества переменных, описываемых дескриптором. Поле заполняется при инициализации дескриптора и не должно модифицироваться пользователем.
FR	Состояние описателя параметра: • 1 – описатель описан операторами SQL DESCRIBE или SET DESCRIPTOR; • 0 – описатель не описан.
UD	Способ задания описателя параметра: • 1 – описатель параметра задан пользователем; • 0 – описатель параметра задан исполняющей системой (библиотекой прекомпилятора).

Приложение 2
Пример использования хранимых процедур

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
/* Создается процедура RETCUR(): прекомпилятор должен быть вызван
 таким образом:*/
/* PCC -S -U SYSTEM/PASSWORD ... */
EXEC SQL CREATE PROCEDURE RETCUR(IN AgeMin INT; IN AgeMax
 INT;INOUT name CHAR(32); OUT cnt INT) RESULT CURSOR_PCI(
name char(20),
city char(15)
)
declare
var c typeof(result);
exception notab for 2202;
code
 cnt := 16383;
 name := "16383";
 open c for direct "select name,city from person where
 Age>="+itoa(AgeMin)+"and Age<="+itoa(AgeMax)+";";
 return c;
exceptions
 when notab then
 print("no such table!\n");
END;
END-EXEC;
EXEC SQL BEGIN DECLARE SECTION;
CURSOR_PCI CR;
char *User;
int MinAge, MaxAge, CNT;
char NAME[32], CITY[32];
long KOLKOR, isNULL;
int RETINT;
short RETSMI;
float RETREL;
double RETNUM;
char RETCHR[32];
char RETBYT[32];
char RETDAT[32];
EXEC SQL END DECLARE SECTION;
/* Описание процедуры SYSTEM.RETCUR. Результат процедуры – курсор.
 */
/* Расшифровки его полей нет, в отличие от случая создания
 процедуры. */
```

```
EXEC SQL DECLARE PROCEDURE
 SYSTEM.RETCUR(IN INT;IN INT;INOUT CHAR(32);OUT INT)RESULT
 CURSOR_PCI;
/* Описания процедур, возвращающих значения различных типов */
/* Процедуры должы быть уже созданы к моменту запуска программы */
EXEC SQL DECLARE PROCEDURE SYSTEM.RETCHR()RESULT char(32);
EXEC SQL DECLARE PROCEDURE SYSTEM.RETINT()RESULT int;
EXEC SQL DECLARE PROCEDURE SYSTEM.RETSMI()RESULT SMALLINT;
EXEC SQL DECLARE PROCEDURE SYSTEM.RETREL()RESULT real;
EXEC SQL DECLARE PROCEDURE SYSTEM.RETDAT()RESULT date;
EXEC SQL DECLARE PROCEDURE SYSTEM.RETNUM()RESULT real;
EXEC SQL DECLARE PROCEDURE SYSTEM.RETBYT()RESULT byte(32);
EXEC SQL DECLARE PROCEDURE SYSTEM.RETNUL()RESULT byte(32);
void main()
{
MinAge=20; MaxAge=30; CNT=0;
strcpy (NAME, "33");
EXEC SQL WHENEVER SQLERROR GOTO err_open;
User = "SYSTEM/MANAGER8";
EXEC SQL CONNECT :User;
/* Создаем динамический курсор: он получит значение в результате
 работы процедуры SYSTEM.RETCUR() */
EXEC SQL ALLOCATE :CR;
/* Исполнение процедур */
EXEC SQL EXECUTE PROCEDURE :RETCHR = system.retchr();
printf("\nThis is the string: %s",RETCHR);
EXEC SQL EXECUTE PROCEDURE :RETINT = system.retint();
printf("\nThis is the Int : %ld",RETINT);
EXEC SQL EXECUTE PROCEDURE :RETSMI = system.retsmi();
printf("\nThis is the Small : %d", RETSMI);
EXEC SQL EXECUTE PROCEDURE :RETREL = system.retrel();
printf("\nThis is the Real : %f", RETREL);
EXEC SQL EXECUTE PROCEDURE :RETDAT = system.retdat();
printf("\nThis is the Date : %s", RETDAT);
EXEC SQL EXECUTE PROCEDURE :RETNUM = system.retnum();
printf("\nThis is the Num : %f", RETNUM);
EXEC SQL EXECUTE PROCEDURE :RETBYT = system.retbyt();
printf("\nThis is the Byte(): %s", RETBYT);
EXEC SQL EXECUTE PROCEDURE :RETBYT:isNULL = system.retnul();
if (isNULL == 0)
 printf("\nThis is the NULL");
EXEC SQL EXECUTE PROCEDURE SYSTEM.RETINT();
printf("\nThis procedure nothing return");
EXEC SQL EXECUTE PROCEDURE
 :CR = system.retcur(:MinAge, :MaxAge, :NAME, :CNT);
printf("\nThis is the NAME : %s", NAME);
```

```
printf("\nThis is the CNT : %ld", CNT);
printf("\nKOLKOR : %ld", KOLKOR);
/* FETCH по курсору, возвращенному процедурой SYSTEM.RETCUR() */
for(;ErrPCI_==0;)
{
EXEC SQL FETCH :CR INTO :NAME, :CITY;
if (!ErrPCI_)
 printf("\nNAME is : %s ; CITY is : %s", NAME, CITY);
}
/* Закрытие динамического курсора */
EXEC SQL CLOSE :CR;
/* Освобождение памяти, занимаемой динамическим курсором */
EXEC SQL DEALLOCATE :CR;
EXEC SQL COMMIT RELEASE;
err_open:
printf("\n\nExit code: %ld", ErrPCI_);
exit(1);
}
```

Приложение 3
Пример многопоточной обработки

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef WIN32
#include <process.h>
#else
#ifdef UNIX
# ifdef LINUX
#  include <pthread/mit/pthread.h>
# endif
#endif
#endif
#ifdef __WATCOMC__
#include <dos.h>
#endif
#ifdef WIN32
void __stdcall th1(void*);
void __stdcall th2(void*);
void __stdcall th3(void*);
#else
#ifdef UNIX
# ifdef LINUX
  void* th1(void*);
  void* th2(void*);
  void* th3(void*);
# endif
#endif
#endif
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
CONTEXT ctx1, ctx2, ctx3;
VARCHAR uid[20], pwd[20];
char name[20];
char city[15];
EXEC SQL END DECLARE SECTION;
#ifdef WIN32
unsigned long id1, id2, id3;
#else
#ifdef UNIX
# ifdef LINUX
  thread_t id1, id2, id3;
# endif
```

```
#endif
#endif
int ret;
int Wth1, Wth2, Wth3;
void main()
{
strcpy(uid.arr,"SYSTEM");
uid.len=strlen(uid.arr);
strcpy(pwd.arr,"MANAGER8");
pwd.len=strlen(pwd.arr);
EXEC SQL WHENEVER SQLERROR GOTO sqlerr;
EXEC SQL ENABLE THREADS;
EXEC SQL CONTEXT ALLOCATE :ctx1;
EXEC SQL CONTEXT ALLOCATE :ctx2;
EXEC SQL CONTEXT ALLOCATE :ctx3;
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
printf ("Thr0 : PCIA_Connect()\n");
Wth1 = Wth2 = Wth3 = 1;
#if defined WIN32
  _beginthreadex(0, 16384,th1, 0, 0, &id1);
  _beginthreadex(0, 16384,th2, 0, 0, &id2);
  _beginthreadex(0, 16384,th3, 0, 0, &id3);
#else
#if defined UNIX
# if defined LINUX
  pthread_create(&id1,NULL,th1,NULL);
  pthread_create(&id2,NULL,th2,NULL);
  pthread_create(&id3,NULL,th3,NULL);
# endif
#endif
#endif
EXEC SQL PREPARE ST FROM select name, city from person where Age
 <=30;
printf ("Thr0 : PCIA_Prepare()\n");
EXEC SQL DECLARE CR CURSOR_PCI FOR ST;
EXEC SQL OPEN CR;
printf ("Thr0 : PCIA_Open()\n");
for(;;)
{
EXEC SQL WHENEVER NOT FOUND GOTO not_found;
EXEC SQL FETCH CR INTO :name, :city;
printf ("Thr0 : PCIA_Fetch()\n");
printf("\nThr0 Name is : %s; City is : %s;", name,
    city);
}
not_found:
```



```
EXEC SQL CLOSE CR;
EXEC SQL COMMIT RELEASE;
printf ("Thr0 : PCIA_CommRoll()\n");
sqlerr:
printf ("Thr0 Error: %ld\n",ErrPCI(CtxPCI_));
while (Wth1 || Wth2 || Wth3)sleep(0);
EXEC SQL CONTEXT FREE :ctx1;
EXEC SQL CONTEXT FREE :ctx2;
EXEC SQL CONTEXT FREE :ctx3;
}
/*****************************************************************/
/************************** THREAD 1 **************************/
/*****************************************************************/
#ifdef WIN32
void __stdcall th1(void*p)
#else
#ifdef UNIX
void * th1(void*p)
#endif
#endif
{
struct sqlca sqlca;
EXEC SQL MODULE M1;
EXEC SQL CONTEXT USE :ctx1;
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
printf ("Thr1 : PCIA_Connect()\n");
EXEC SQL PREPARE ST FROM select name, city from person where Age
 <=30;
EXEC SQL DECLARE CR CURSOR_PCI FOR ST;
EXEC SQL OPEN CR;
printf ("Thr1 : PCIA_Open()\n");
for(;;)
{
 EXEC SQL WHENEVER NOT FOUND GOTO not_found;
 EXEC SQL FETCH CR INTO :name, :city;
 printf("\nThr1: Name is : %s; City is : %s;", name,
     city);
 printf ("\nThr1 : PCIA_Fetch()");
}
not_found:
EXEC SQL CLOSE CR;
printf ("Thr1 : PCIA_Close()\n");
EXEC SQL COMMIT RELEASE;
printf ("Thr1 : PCIA_CommRoll()\n");
sqlerr:
printf ("Thr1 Error: %ld\n",ErrPCI(ctx1));
```

```
EXEC SQL END MODULE M1;
Wth1=0;
#ifdef WIN32
  _endthreadex(0);
#else
#ifdef UNIX
  pthread_exit(&Wth1); return Wth1;
#endif
#endif
}
/*****************************************************************/
/***************************** THREAD 2 **************************/
/*****************************************************************/
#ifdef WIN32
void __stdcall th2(void*p)
#else
#ifdef UNIX
void * th2(void*p)
#endif
#endif
{
struct sqlca sqlca;
EXEC SQL MODULE M2;
EXEC SQL CONTEXT USE :ctx2;
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
printf ("Thr2 : PCIA_Connect()\n");
EXEC SQL PREPARE ST FROM select name, city from person where Age
 <=30;
EXEC SQL DECLARE CR CURSOR_PCI FOR ST;
EXEC SQL OPEN CR;
printf ("Thr2 : PCIA_Open()\n");
for(;;)
{
 EXEC SQL WHENEVER NOT FOUND GOTO not_found;
 EXEC SQL FETCH CR INTO :name, :city;
 printf("\nThr2: Name is : %s; City is : %s;", name,
     city);
 printf ("\nThr2 : PCIA_Fetch()");
}
not_found:
EXEC SQL CLOSE CR;
printf ("Thr2 : PCIA_Close()\n");
EXEC SQL COMMIT RELEASE;
printf ("Thr2 : PCIA_CommRoll()\n");
sqlerr:
printf ("Thr2 Error: %ld\n",ErrPCI(ctx2));
```



```
EXEC SQL END MODULE M2;
Wth2=0;
#ifdef WIN32
  _endthreadex(0);
#else
#ifdef UNIX
  pthread_exit(&Wth2); return &Wth2;
#endif
#endif
}
/*****************************************************************/
/*************************** THREAD 3 ****************************/
/*****************************************************************/
#ifdef WIN32
void __stdcall th3(void*p)
#else
#ifdef UNIX
void * th3(void*p)
#endif
#endif
{
struct sqlca sqlca;
EXEC SQL MODULE M3;
EXEC SQL CONTEXT USE :ctx3;
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
printf ("Thr3 : PCIA_Connect()\n");
EXEC SQL PREPARE ST FROM select name, city from person where Age
 <=30;
EXEC SQL DECLARE CR CURSOR_PCI FOR ST;
EXEC SQL OPEN CR;
printf ("Thr3 : PCIA_Open()\n");
for(;;)
{
 EXEC SQL WHENEVER NOT FOUND GOTO not_found;
 EXEC SQL FETCH CR INTO :name, :city;
 printf("\nThr3: Name is : %s; City is : %s;", name,
     city);
 printf ("\nThr3 : PCIA_Fetch()");
}
not_found:
EXEC SQL CLOSE CR;
printf ("Thr3 : PCIA_Close()\n");
EXEC SQL COMMIT RELEASE;
printf ("Thr3 : PCIA_CommRoll()\n");
sqlerr:
printf ("Thr3 Error: %ld\n",ErrPCI(ctx3));
```

```
EXEC SQL END MODULE M3;
Wth3=0;
#ifdef WIN32
  _endthreadex(0);
#else
#ifdef UNIX
  pthread_exit(&Wth3); return Wth3;
#endif
#endif
}
/*****************************************************************/
/************************ END OF SOURCE**************************/
/*****************************************************************/
```

Приложение 4
Пример PCI-программы

```
/* Пример программы для СУБД ЛИНТЕР, написанной с помощью
 прекомпиляторного интерфейса */
#include <stdio.h>
#include <string.h>
/* Определение коммуникационной области – обязательно для
 встроенного  языка SQL */
EXEC LINTER IFDEF SQL;
EXEC SQL INCLUDE SQLCA;
EXEC LINTER ENDIF;
/* Секция объявлений прекомпилятора */
EXEC SQL BEGIN DECLARE SECTION;
 char cName [21];
 char cFirstNam [16];
 char cCity [16];
 char cPhone [9 ];
 char *pName, *Query, *User;
EXEC SQL END DECLARE SECTION;
void main()
{
/* 1. Установить связь с СУБД ЛИНТЕР */
EXEC SQL WHENEVER SQLERROR GOTO err_open;
User = "SYSTEM/MANAGER8";
EXEC SQL CONNECT :User;
/* 2. Подготовить предложение к выполнению */
Query = "SELECT NAME,FIRSTNAM,CITY,PHONE FROM PERSON WHERE
 NAME=:v1;";
  pName = "CLINTON";
  EXEC SQL PREPARE ST FROM :Query;
EXEC SQL DECLARE CR CURSOR_PCI FOR ST;
/* 3. Открыть курсор */
  EXEC SQL WHENEVER SQLERROR GOTO err_read;
  EXEC SQL WHENEVER NOT FOUND GOTO not_found;
  EXEC SQL OPEN CR USING :pName;
/* 4. Выбирать, пока есть записи */
  EXEC SQL WHENEVER NOT FOUND GOTO no_more;
  for (;;) {
 EXEC SQL FETCH CR INTO :cName, :cFirstNam, :cCity,
 :cPhone;
 printf(" %20s %15s %15s %8s\n", cName, cFirstNam,
     cCity, cPhone);
 }
not_found: printf(" Нет таких записей\n");
```

```
no_more: /* 5. Закрыть курсор */
  EXEC SQL CLOSE CR;
exit: /* 6. Закрыть связь с СУБД ЛИНТЕР */
  EXEC SQL WHENEVER SQLERROR GOTO err_clos;
  EXEC SQL COMMIT RELEASE;
  return;
err_open: printf(" Ошибка открытия канала : %d\n",
         ErrPCI_); return;
err_clos: printf(" Ошибка закрытия канала : %d\n",
         ErrPCI_); return;
err_read: printf(" Ошибка обращения к ядру: %d\n",
         ErrPCI_); goto exit;
}
```


Приложение 5
Вызов компилятора хранимых процедур

Для вызова SPC процедур необходимо в интерпретаторе команд операционной системы подать команду (регистр не важен):

```
SPC [-H][-V][-Q][-F] [-R] [-N <имя сервера>]
 [-c <кодовая страница>] -U <имя пользователя/ пароль>
 <имя исходного файла[.prc]>[-L <листинг>[.lsr]]
```

Опции командной строки

1) `-H` – получение справочной информации по использованию SPC.
2) `-V` – получение информации о версии транслятора хранимых процедур.
3) `-Q` – режим безэховой (без отображения на устройстве вывода) работы.
4) `-N <имя сервера>` – имя ЛИНТЕР-сервера.
5) `-U <имя пользователя/пароль>` – имя и пароль пользователя.
6) `-F` – создавать полный листинг ошибок (т.е. листинг исходного текста хранимой процедуры с выявленными ошибками).
7) `<Имя исходного файла[.prc]>` – спецификация файла с текстом процедуры. По умолчанию используется расширение файла `.prc`.
8) `-L <листинг>[.lsr]` – спецификация файла, в который будут помещены результаты трансляции.
9) `-R`, читать входной файл как есть, без принудительного добавления префикса `CREATE/ALTER`.
10) `-C <кодовая страница>` – спецификация кодировки, в которой должен содержаться текст хранимой процедуры (триггера).

Примеры

1)

```
spc -F -U ivanov/parol source.spc
(подробный отчет о трансляции процедуры в файле source.lsr)
```

2)

```
spc -Q -U SYSTEM/PASSOWRD src -L list.lst
(трансляция хранимой процедуры без отображения на терминале файла
 src.spc с коротким листингом в файле list.lst)
```

Указатель операторов