

Database  
Management  
System

# LINTER<sup>®</sup>

Version 5.9

## Embedded SQL

Relational Expert Systems

---



# Table of Contents

<b>Introduction</b> .....	<b>4</b>
Using Linter’s Embedded SQL Interface .....	4
Running the Embedded SQL Precompiler .....	4
Building Executables Using the Embedded SQL Library .....	5
<b>Linter’s Embedded SQL Syntax</b> .....	<b>6</b>
Host Language Variables.....	6
Declaration of Variables .....	6
Inheriting Variables (C++ only) .....	7
Using Variables in Embedded SQL Statements .....	7
Type Conversion .....	7
Embedded SQL variables .....	8
Connections .....	8
Statements .....	8
Cursors .....	8
Descriptors .....	9
Connecting to and Disconnecting from LINTER.....	10
Connecting to Linter .....	10
Disconnecting from Linter and Transactions.....	10
Preparing SQL statements for execution.....	11
<b>Executable Embedded SQL operators</b> .....	<b>12</b>
Executing Embedded SQL Statements.....	12
Static Execution Operator .....	12
EXECUTE IMMEDIATE Operator .....	13
EXECUTE Operator.....	13
Working with Cursors .....	13
Opening a Cursor.....	13
Fetching Records Through a Cursor .....	14
Closing a Cursor.....	15
Dynamic Cursors.....	15
Embedded SQL Global Variables.....	15
<b>Exception Processing</b> .....	<b>16</b>
Working with BLOB Fields .....	16
Insert a BLOB Value: ADD .....	17
Deleting a BLOB Value: CLEAR.....	17
Selecting a BLOB Value: GET .....	17
Context Use and Multithreading.....	17
Allow Thread Creation.....	19
Contexts .....	19
Context Creation .....	19
Context Use.....	19
Context Free.....	20

---

Communication area .....	20
Working with Descriptors.....	20
<b>Using Stored Procedures .....</b>	<b>24</b>
Stored Procedures Creation and Modification .....	24
Stored Procedure Prototype Declaration .....	24
Stored Procedures Execution.....	25
<b>Precompilation Control .....</b>	<b>26</b>
Declaring Macro Names.....	26
Conditional Compilation Statements.....	26
<b>Embedded SQL Program Sample .....</b>	<b>27</b>
<b>Precompiler Messages.....</b>	<b>28</b>
<b>Completion Codes .....</b>	<b>32</b>

## Introduction

Embedding SQL into another programming language means incorporating SQL statements within the source code written in that other programming language, called the host language. In this document, C will be the example host language into which SQL statements are embedded. In the remainder of this document a reference to C will include C++ unless otherwise specified.

The three steps in translating embedded SQL programs into executable code are

1. Linter's embedded SQL precompiler, pcc, processes the source code, extracts the SQL statements, converts those statements into C function calls, and replaces the original embedded SQL statements with these calls in a result file that contains pure C code without the SQL statements;
2. The result C file is then sent to the C compiler to be converted into an object file; and
3. The resulting object file is sent to linker to be linked with other object files and Linter's embedded SQL library, pci, to obtain an executable code.

## Using Linter's Embedded SQL Interface

### Running the Embedded SQL Precompiler

To run the precompiler you have to submit the following command to the command interpreter of your operating system

```
pcc [flags] source_file [target_file]
```

source file specifies a source embedded SQL file name. If an extension to the source file name is omitted, a default .pc extension will be added.

target file specifies a target C file name. If this parameter is omitted, the target file will have the same name as the source file with the default extension. The default target file name is .c if C is the host language. The default extension is .cpp if C++ is the host language.

Each precompiler flag (or flag sequence) must be preceded with a minus sign (-). The following flags are available:

<u>Flag</u>	<u>Function</u>
C	specifies C as the host language. This is the default.
C+	specifies C++ as the host language.
D	enables generation of debugging information. The debugger walks through the source program code instead of the precompiled code, generating debugging information (#line directives, etc.). By default, debugging information is not generated.
O	specifies the Oracle completion codes checking mode. If this flag is present, the result program code will contain checks of sqlca.sqlcode values, otherwise it will contain checks of ErrPCI_ values.
a	disables sqlca communication area filling, valid only if the o flag is not set;
t	enables thread control statements usage (ENABLE THREADS, CONTEXT { ALLOCATE   USE   FREE } );

<u>Flag</u>	<u>Function</u>
n<node>	specifies remote Linter server node name;
u <username/ password>	specifies username and password for connecting to a database (case-sensitive);
s	specifies the SQL statement semantic checking mode. This flag can be used if stored procedures are created in an embedded SQL program.

If a flag sequence is used after a minus character, flag letters cannot be separated by any delimiters, e.g., -cdo

Flag u must be used and flag n can be used if flag s is set.

### Examples

<code>pcc -c sample.pc</code>	The target file will be sample.c.
<code>pcc -c+ sample.pc</code>	The target file will be sample.cpp.
<code>pcc -dc+ sample.pc</code>	The target file will be sample.cpp and will contain debugging information.
<code>pcc sample.pc sample1.cpp</code>	The target file will be sample1.cpp.
<code>pcc -c+s -nLinter\ -uSYSTEM/MANAGER \ sample.pc</code>	The target file is sample.cpp; the user is SYSTEM with password MANAGER; will connect to remote node Linter server named Linter.

## Building Executables Using the Embedded SQL Library

The Linter embedded SQL precompiler inserts the `#include` directives into the target C code, e.g.

```
#include "pci.h"
#include "sqlca.h"
```

These include files contain declarations of structures, procedures, and global variables used by Linter's embedded SQL library. The path to the files `pci.h` and `sqlca.h` must be passed to the compiler through environment variable or compiler flag. Usually these files are placed in the `intlib` subdirectory of the Linter main directory.

To build an executable program, you have to include the embedded SQL library in your project. This library file has the name `pci` and a system-dependent file extension `pci.lib` in MS DOS, `pci.a` in UNIX, `pci.o1b` in VAX/VMS. Usually the `pci` library is placed in the `intlib` subdirectory of the Linter main directory.

The Linter distribution for MS DOS includes three versions of the embedded SQL library:

<code>pci-lib</code>	For Borland C compilers in the large memory model.
<code>pci-_wp.lib</code>	For Watcom C compilers in the flat memory model.
<code>pci-_m.lib</code>	For Microsoft C compilers in the large memory model.

## Linter's Embedded SQL Syntax

Embedded SQL statements have a free format, i.e., SQL statement can consist of several source lines and include C style comments. However it is illegal to mix an embedded SQL statement with a host language statement on the same source line. Each embedded SQL statement must be prefixed with keywords EXEC SQL (EXEC LINTER for conditional precompilation statements) and terminated with a semicolon (;).

All embedded SQL keywords in Linter are case insensitive.

All embedded SQL statements can be separated into two groups: declarative statements (directives) and executable statements (operators). Declarative statements do not generate host language code; they can appear in any part of the source program. Executable statements (operators) generate host language function calls; they should appear only in an executable part of the host language source program.

## Host Language Variables

Each variable to be used in an SQL statement must be declared, prior to its use, in an SQL declaration section bounded by BEGIN DECLARE SECTION and END DECLARE SECTION. Variables declared within this SQL declaration section do not need to be independently declared elsewhere in the sounding host language program.

## Declaration of Variables

An SQL declaration section has the following format:

```
EXEC SQL BEGIN DECLARE SECTION;  
    declaration of variables  
EXEC SQL END DECLARE SECTION;
```

Variable declarations are made according to host language syntax rules, but have the following restrictions

Any variable must be of one of the following types:

- integer (int, short, long, unsigned, etc.)
- real (float, double, etc.)
- character (char, char[N] etc.)
- string(VARCHAR[N]) macro type
- CURSOR\_PCI macro type (for working with dynamic cursors)
- pointer to CHAR variable

Use of structures, unions, or user-defined types is disabled.

If a variable is an array (or a pointer), the array must be one-dimensional and its dimension must be specified by a decimal constant.

Declaration of register variables is disabled.

Assignment of initial values to variables is also disabled.

There can be several SQL declaration sections in one source module, but each section must be included in a separate host language block.

The visibility scope of host language variables in embedded SQL statements is determined by the same rules that apply to host language statements.

## Inheriting Variables (C++ only)

The Heir modifier is used for C++ variable declaration. If a variable in a host language variables declaration section is declared with the Heir modifier, the precompiler assumes that that variable declaration is inherited from the parent class. In the precompiled program text, this variable declaration will be present as a comment. Therefore, a declaration of a variable with an Heir modifier must begin on a new line and this line must contain only Heir variable declarations. The precompiler doesn't verify that the inherited variable exists in the parent class.

So, the following text:

```
EXEC SQL BEGIN DECLARE SECTION;
    char*p; Heir char*q;
EXEC SQL END DECLARE SECTION;
```

will be precompiled to:

```
/*
EXEC SQL BEGIN DECLARE SECTION;
*/
    /* char*p; Heir char*q; */
/*
EXEC SQL END DECLARE SECTION;
*/
```

## Using Variables in Embedded SQL Statements

Each host language variable used in any embedded SQL statement must be declared in some SQL declaration section. When a host language variable is to be included in an SQL statement, a colon precedes it (:) with no intervening space. Multiple declarations may be made on the same line as the following construction shows

```
:main_variable[:indicator_variable]
```

Indicator variables must be a two-byte integer type. If a main variable is an array, the corresponding indicator variable should be an array of the same dimension. Indicator variables are used to send and receive NULL values or for receiving information about truncation of values.

## Type Conversion

The Linter embedded SQL library automatically performs conversions between database value types and host language variable types in the most usual cases, such as:

- any numeric type to any numeric type;
- character string to any database type;
- database character field to any host variable type.

If conversion cannot be performed, the exception code, incompatible types, is returned.

## Embedded SQL variables

Embedded SQL deals with such objects as modules, connections, statements, cursors, and descriptors. Embedded SQL variables are names of these objects. Embedded SQL variables can be local or global.

All variables declared with DECLARE or PREPARE operators outside of any modules are global. All variables declared within an embedded SQL module are local to that module.

All global embedded SQL variables in one source module must have names unique within that module.

Embedded SQL variable names cannot coincide with embedded SQL or C keywords, or with C precompiler macro names.

## Connections

A connection may be declared by means of a DECLARE DATABASE directive

```
EXEC SQL DECLARE <Connection_name> DATABASE ;
```

<Connection\_name> is an embedded SQL variable. It can be implicitly declared when being used in the CONNECT operator. After such implicit declaration, <Connection\_name> can be used with a CONNECT operator and in subsequent executable statements. Connection name refers to a connection to a Linter SQL server.

One unnamed connection to the server may also be made by the CONNECT operator without an AT Connectionname clause.

## Statements

A statement may be declared by means of a DECLARE STATEMENT directive

```
EXEC SQL DECLARE <Statement_name> STATEMENT;
```

<Statement\_name> is an embedded SQL variable. It may be implicitly declared when used with the PREPARE operator. After such implicit declaration, <Statement\_name> may then be used in the PREPARE, EXECUTE, and DECLARE CURSOR operators to provide execution of the prepared SQL statement.

The precompiler also creates an unnamed statement when processing the following embedded SQL directive

```
EXEC SQL DECLARE <Cursor_name>CURSOR FOR <Statement_text>;
```

## Cursors

A cursor may be declared by means of a DECLARE CURSOR directive

```
EXEC SQL DECLARE <Cursor_name>  
CURSOR FOR {<Statement_name> | <Statement_text>;
```

<Cursor\_name> and <Statement\_name> are embedded SQL variables.

<Statement\_name> must have been declared previously with a DECLARE STATEMENT directive or previously used in a PREPARE operator.

<Statement\_text> (which contains SQL text) can be represented by:

- Plain text;
- A host language string constant which contains SQL text; or
- A variable construction, where variable must be a char or VARCHAR variable of the host language.

The following statement:

```
DECLARE CURSOR FOR <Statement_text>
```

is executable, because it implicitly executes a PREPARE operator or an unnamed statement.

The declared <Cursor\_name> can then be used:

- 1) for fetching records from a cursor (in OPEN, FETCH, and CLOSE operators) or
- 2) for updating or deleting selected records (in positional UPDATE and DELETE operators in the SQL construction WHERE CURRENT OF <Cursor\_name>).

## Descriptors

Descriptors are declared by the DESCRIBE statement, which has the following formats:

```
EXEC SQL DESCRIBE BIND VARIABLES FOR <Statement_name>
INTO <Descriptor_name>;
```

```
EXEC SQL DESCRIBE SELECT LIST FOR <Statement_name>
INTO <Descriptor_name>;
```

<Statement\_name> is a precompiler variable that must have been declared previously in DECLARE STATEMENT operator or previously used in a PREPARE operator.

<Descriptor\_name> is a precompiler variable that must be unique.

BIND VARIABLES provide descriptor initialization for storing information about input and output variables used in SQL queries.

SELECT LIST holds descriptor initialization for storing the parameters of a SELECT statement list.

Example of descriptors declaration

```
...
void f()
{
EXEC SQL PREPARE mystatement FROM :my_string;
EXEC SQL DECLARE <emp_cursor> FOR select name, firstnam FROM person WHERE
phone=:Phone;
EXEC SQL DESCRIBE BIND VARIABLES FOR mystatement INTO select_descriptor;
EXEC SQL OPEN empcursor USING DESCRIPTOR bind_descriptor;
EXEC SQL DESCRIBE SELECT LIST FOR mystatement INTO select_descriptor;
EXEC SQL FETCH empcursor USING DESCRIPTOR select_descriptor;
}
```

## Connecting to and Disconnecting from LINTER

### Connecting to Linter

You can connect to Linter via the CONNECT operator:

```
EXEC SQL CONNECT [ EXCLUSIVE | AUTOCOMMIT | OPTIMISTIC ]
[ :<Username>[ IDENTIFIED BY <Password> ] ]
[ AT <Connection_name>] [ USING :<Server_name>];
```

The EXCLUSIVE mode is used by default.

<Username>, <Password> and <Server\_name> are host language variables. They must be previously declared in an embedded SQL declaration section and should be of char or varchar type. <Connection\_name> is an embedded SQL variable which may be declared earlier by a DECLARE DATABASE directive, but such a declaration is not necessary.

Execution of a CONNECT operator generates submission of the Linter Call interface OPEN command. The Server\_name, if present is written into the Node field of the Linter control block. If either <Username> or <Password> are given, they are united in one string with the slash character (/) as a delimiter. This string is then transferred as the second argument when opening a channel to Linter.

If a CONNECT operator is executed for a connection which was already opened by another CONNECT operator without its subsequent closing COMMIT RELEASE or ROLLBACK RELEASE operator an exception code, channel is already opened is returned. If the CONNECT operation successfully opens a Linter channel, the completion code 0 (successful completion) is returned. Otherwise, an error code generated by the Linter kernel is returned.

### Disconnecting from Linter and Transactions

You can save the last transaction results by means of COMMIT operator:

```
EXEC SQL [AT <Connection_name>] COMMIT [WORK] [RELEASE];
```

You can cancel the last transaction results by means of a ROLLBACK operator:

```
EXEC SQL [AT <Connection_name>] ROLLBACK [WORK] [RELEASE];
```

<Connection\_name> is an embedded SQL variable. It must be declared earlier by a DECLARE DATABASE directive or used in a CONNECT operator. The WORK keyword is optional and does not change the action performed by this operator. The RELEASE keyword specifies that after saving or canceling results of the last transaction connection to the DBMS (default or specified by Connection\_name) will be closed.

Executing the COMMIT operator generates submission of the COMT command through all channels belonging to the specified connection. Executing the ROLLBACK operator submits RBAC commands through all channels belonged to the specified connection. After either of these submissions, if the RELEASE keyword is specified, he CLOS command is submitted through all channels belonging to the specified connection.

If the connection named by the <Connection\_name> argument was not created by the CONNECT operator or the connection was unnamed, an exception code, channel is not opened, is returned. If all Linter Call interface commands are completed successfully, a successful

completion code, 0, is returned. Otherwise, a Linter error code will be returned by the first operation completed with an error. However all subsequent operations will also be executed.

## Preparing SQL statements for execution

To prepare an SQL statement text for subsequent execution, you can use a PREPARE operator:

```
EXEC SQL PREPARE <Statement_name> FROM  
{ <Statement_text> | <String_constant> | <Host_variable>};  
  
<Statement_text> ::= { :<String_variable> | <SQL_statement> }
```

<Statement\_name> is an embedded SQL variable, which may be declared earlier by a DECLARE STATEMENT directive, but such declaration is not necessary.

<String\_constant> is a host language string constant.

<String\_variable> is a host language variable, which must be declared earlier in an embedded SQL declaration section and must be of char or varchar type.

<SQL\_statement> may be any SQL language statement, including a SELECT operator.

If a SQL\_statement is written directly into a PREPARE operator, it may contain input and output variables. If a host language constant or variable specifies a statement the statement text is unknown during precompilation time. The dynamically created statement text can contain the constructs Name, :Number or ? for unnamed parameters. At execution time, these constructs must be replaced by values of variables specified in INTO and USING operations.

Executing a PREPARE operator creates a reference between a given text and Statement\_name. The prepared statement then can be executed with an EXECUTE operator. If a cursor is declared for this statement, it can also be executed by means of the OPEN, FETCH and CLOSE operators.

It is possible to specify input and output variables in USING and INTO clauses.

If a PREPARE operator is used for a statement concerned with a cursor previously opened with an OPEN operator and the cursor was not closed with the CLOSE operator an exception code, cursor is already opened is returned.

# Executable Embedded SQL operators

## Executing Embedded SQL Statements

The following operators are used for statement execution in embedded SQL:

- Static execution operator;
- EXECUTE IMMEDIATE;
- and EXECUTE.

All of these operators may contain the constructions AT <Connection\_name> and FOR: <Execution\_count>.

The first construction specifies the name of the connection that will be used for statement execution. The default connection will be used if no <Connection\_name> is specified.

The second construction specifies the operator's execution count. This is useful when some of the operator parameters are arrays. If FOR construction is not specified the minimal size of array variables becomes the execution count.

The executed statement may be any SQL statement, including the SELECT statement. It may contain input and output variables of the host language. Before statement execution input variables are replaced by their values. After statement execution, output values are written to output variables.

If some input variable is an array, the value of the nth element of the array replaces the variable name at the time of the nth execution. If an output variable is an array variable, the value of a selected record field is written into the nth element of the array at the time of the nth execution.

You can have both scalar and array input variables in a single operator. You cannot mix scalar and array output variables.

A scalar input variable specifies the same value at each execution. An array input variable specifies the next element at each execution.

Two alternatives of execution are possible for SELECT statements:

- 1) If all input variables are scalar and all output variables are arrays, the SQL statement is executed once and all records selected by it are written into output arrays;
- 2) If one or more of the input variables is an array, all output variables must also be arrays. A new SQL statement must be executed for each element of the input array and each iteration must return no more than one record.

If, during execution of the given statement by Linter, an error occurs, an error code is returned. After execution of the DML statements SELECT, INSERT, DELETE, and UPDATE, the total count of records processed by the operator is written into the CntPCI variable. If this count is equal to 0, an exception code, not found is returned.

## Static Execution Operator

The static execution operator has the following format:

```
EXEC SQL [AT<Connection_name>]
[ FOR :<Execution_count>] <SQLstatement>;
```

Static execution SQL statement text is known at precompilation time. It can contain host language variables.

## EXECUTE IMMEDIATE Operator

The EXECUTE IMMEDIATE operator has the following format:

```
EXEC SQL [AT <Connection_name>] [FOR :<Execution_count>]
EXECUTE IMMEDIATE <SQL_statement>;
```

The SQL statement contained in the EXECUTE IMMEDIATE operator may be written directly, or as a string constant, or as a string variable preceded by a colon (). In the last two cases, statement text may not contain host language variables because they are unknown at precompilation time.

## EXECUTE Operator

An EXECUTE operator has the following format:

```
EXEC SQL [AT <Connection name>] [FOR :<Execution_count>]
EXECUTE <Statement_name> [USING <Variables_List>;]
```

Statement\_name is an embedded SQL variable. The EXECUTE operator performs dynamic statement execution, i.e., the SQL statement to be executed is unknown at precompilation time. The dynamically created statement text contains the constructions Name or :Number. These constructions are replaced at execution time by the values of variables specified in the USING construction.

If the Statement\_name specifies a statement concerned with a cursor opened earlier with the OPEN operator and the cursor has not been closed with the CLOSE operator an exception code, cursor is already opened is returned. If the Statement\_name specifies a statement not yet prepared with the PREPARE operator an exception code, statement is not prepared, is returned.

## Working with Cursors

### Opening a Cursor

The OPEN operator is used to open a cursor:

```
EXEC SQL [AT <Connection_name>] OPEN
[EXCLUSIVE | AUTOCOMMIT | OPTIMISTIC ]
<Cursor_name> [USING <List_of_variables>
| USING DESCRIPTOR <Descriptor_name>;]
```

The EXCLUSIVE mode is set by default.

<Cursor\_name> is an embedded SQL variable which must be declared earlier in a DECLARE CURSOR statement.

Execution of the OPEN operator creates of a new channel to Linter via the Linter Call interface command, OPEN. This channel is created within some connection; either the default or as specified

by the AT Connection\_name construction. After the channel is created, the SQL statement using the cursor is executed. SELECT statement execution here means creating a set of matching records, not fetching them into C variables.

If <Cursor\_name> specifies a cursor already opened by another OPEN operator and was not closed with a CLOSE operator, an exception code, cursor is already opened is returned. If the statement for which this cursor is declared had not yet been prepared with the PREPARE operator an exception code, statement is not prepared, is returned. If the SELECT statement returns no matching records, an exception code, not found is returned.

The USING <List\_of\_variables> clause is used to open a cursor with variable parameters.

The USING DESCRIPTOR <Descriptor\_name> clause is used to open dynamic cursors. The descriptor referenced by <Descriptor\_name> must have been declared previously by a DESCRIBE directive.

## Fetching Records Through a Cursor

The FETCH operator is used to select consecutive records satisfying the same condition:

```
EXEC SQL [FOR :<Execution_count> ]
FETCH { <Cursor_name> | :<Cursor_variable> } [<Orientation> ]
[ INTO <Variables_list>
| USING DESCRIPTOR Descriptor_name];

<Orientation> ::=
    NEXT | PRIOR | FIRST | LAST | { [ ABSOLUTE | RELATIVE ] :<Variable>}
```

<Cursor\_name> is an embedded SQL variable. <Cursor\_variable> is the name of a host language variable which has been declared in a host language variables declaration section as a CURSOR type variable and has been allocated by the ALLOCATE operator. If <Orientation> is omitted the following, NEXT, record is fetched by default.

The ABSOLUTE | RELATIVE Variable option selects the record:

- for ABSOLUTE                    where the number in the result set matches the value of Variable.
- for RELATIVE                    where the relative offset from the currently selected record matches the value of Variable.

Execution of the FETCH operator submits one of the Linter Call interface commands to set a position in a cursor and to fetch a record, if needed, using GETN GETP, GETF, GETL, or GETS. When fetched, the values of fields of a selected record are written into output variables.

If the <Execution\_count> value is greater than 1 (this is permitted only if Orientation is NEXT), the described operation will be repeated.

If <Cursor\_name> specifies a cursor that was not opened with an OPEN operator, an exception code, cursor is not opened, is returned. If no records are selected, an exception code, not found, is returned.

The USING <Variables\_list> clause contains a list of variables that are to be loaded with selected values. Variable amounts and types must be equal to the amounts and types of the selected values.

The USING DESCRIPTOR <Descriptor\_name> clause is used for dynamic cursors. The descriptor <Descriptor\_name> must have been declared previously by the DESCRIBE directive.

## Closing a Cursor

The CLOSE operator is used to close a cursor:

```
EXEC SQL CLOSE { <Cursor_name> | <Cursor_variable_name>;
```

<Cursor\_name> is an embedded SQL variable. <Cursor\_variable\_name> is a host language variable of type CURSOR.

Execution of the CLOSE operator closes a channel to Linter.

If <Cursor\_name> specifies a cursor that was not opened by the OPEN operator, an exception code, cursor is not opened, is returned.

## Dynamic Cursors

The CURSOR data type of C/C++ is used to describe a dynamic cursor. A variable of type CURSOR must be declared in a host language variables declaration section.

After declaration the variable can be used as a cursor of the embedded language. The visibility scope of the variable is the same as if it were a host language variable. Having declared the cursor variable name, we only notify the compiler of the existence of the variable in the program.

Required memory for the cursor is provided by the embedded language operator ALLOCATE, which has the following format:

```
EXEC SQL ALLOCATE <Cursor_variable_name>;
```

The <Cursor\_variable\_name> must be declared earlier in a host language variables declaration section. It is then possible to perform the following dynamic cursor operations:

- 1) assign a value to a cursor by means of a stored procedure call;
- 2) make a fetch using the FETCH operator;
- 3) work with BLOB data using the {ADD | GET | CLEAR} BLOB operators;
- 4) and close the cursor with the CLOSE operator.

When the cursor variable is no longer needed, the memory occupied by it may be released with the DEALLOCATE operator, which has the following format:

```
EXEC SQL DEALLOCATE <Cursor_variable_name>;
```



ALLOCATE and DEALLOCATE operators don't check the current state of the cursor variable. Repeated allocation or deallocation of unoccupied memory areas must be handled by the programmer.

## Embedded SQL Global Variables

Linter's embedded SQL library contains several global variables, such as ErrPCI\_ (Linter Call completion code) and CntPCI\_ (processed records count). The Linter Call completion code variable is filled in after each executed SQL operator. The processed records count is filled in after each executed DML operator (SELECT, INSERT, DELETE, and UPDATE). All other embedded SQL operators set Linter Call completion code to 0 (successful completion) and do not change the processed records count.

## Exception Processing

A WHENEVER statement is used for exception processing. It has the following format:

```
EXEC SQL WHENEVER <Condition> THEN <Action>;
```

```
<Condition> ::= { SQLERROR | SQLWARNING | NOT FOUND }
```

```
<Action> ::= { STOP | CONTINUE | GOTO <Label> | CALL <Procedure> }
```

<Label> is a host language statement label.

<Procedure> is a host language procedure name. This procedure cannot have any arguments.

A WHENEVER statement is declarative. It specifies exception processing for all executable embedded SQL statements until the next WHENEVER statement is encountered. Before the first WHENEVER directive or when there is no WHENEVER directive in the source file, CONTINUE is performed by default for all exceptions.

All exceptions arising during execution of embedded SQL library functions are divided into 3 categories:

- 1) ERRORS, if an SQL operator cannot be executed properly;
- 2) WARNINGS, if an SQL operator was executed, but some doubtful type conversions were performed, etc.;
- 3) NOT FOUND situations, if there is no record satisfying the specified search condition.

An action in any of these situations maybe one of the following:

- 1) STOP – terminate program execution;
- 2) CONTINUE – explicit directive to ignore exception and default action if a WHENEVER directive is not declared;
- 3) GOTO – transfer control to a given host language label; or
- 4) CALL – call a given host language procedure.

After processing each executable operator, the precompiler inserts code that permits checking the values of the global variables ErrPCI\_ and CntPCI\_.

CONTINUE is a default state that causes program execution to continue in spite of an error if a WHENEVER directive has not been declared prior to the error's being encountered.

## Working with BLOB Fields

There are three statements for processing BLOB data type fields in Linter embedded SQL:

- 1) ADD – insert values;
- 2) CLEAR – delete values;
- 3) GET – select values.

## Insert a BLOB Value: ADD

The statement format for inserting a BLOB value is

```
EXEC SQL BLOB ADD FROM :<Buffer>  
[ WHERE CURRENT OF <Cursor_name> ] ;
```

<Buffer> is a host language variable. <Cursor\_name> is an embedded SQL variable that must be declared earlier in a DECLARE CURSOR statement.

Before execution of an ADD statement, the record that is to receive the specified BLOB value must be inserted into table with INSERT or selected by a cursor with FETCH. In the second case, <Cursor\_name> must be specified.

## Deleting a BLOB Value: CLEAR

The statement used to delete a BLOB value has the following format:

```
EXEC SQL BLOB CLEAR WHERE CURRENT OF <Cursor_name>;
```

<Cursor\_name> is an embedded SQL variable that must be declared previously in a DECLARE CURSOR statement.

Before execution of a CLEAR statement the record, which contains a BLOB value that is to be deleted must be selected from a cursor with FETCH.

## Selecting a BLOB Value: GET

The BLOB value selecting statement has the following format:

```
EXEC SQL BLOB GET INTO :<Buffer> [OFFSET: <Displacement>]  
WHERE CURRENT OF <Cursor_name>;
```

<Buffer> is a host language variable that must be of type char or VARCHAR.

<Displacement> is a host language variable, which should be integer.

<Cursor\_name> is an embedded SQL variable that must have been declared previously in a DECLARE CURSOR statement.

Before executing a GET statement, the record that contains a selected BLOB value must be selected by a cursor with FETCH.

If <Displacement> is omitted, its default value is 1 (no displacement).

After execution of this statement, the number of bytes actually selected is written into the global variable LenPCI\_.

## Context Use and Multithreading

Embedded SQL permits designing programs that use several threads. Multithreading is possible on the Win32, UNIX, and VAX/VMS operating systems. Implementing multiple threads requires:

- 1) Program text must be translated with the precompiler's -T option. Otherwise the error, No -T option in precompiler call, will be displayed. Precompiling with the -T option replaces embedded language statements with multithreaded applications library function calls;

- 2) Every thread must be executed in its own context (see below), defined by a context variable. This variable must have been initialized by the CONTEXT operator, must not have been deallocated by the CONTEXT FREE operator and must have been selected by the CONTEXT USE operator. The context variable is a host language variable of type CONTEXT, declared in a host language variables declaration section.

If the context is not explicitly defined (there was no CONTEXT USE directive), the context default variable CtxPCI\_, which doesn't require initialization is used.

Variable sqllda must be declared as auto type variable for every thread. That can be done by its explicit declaration in the precompiled host language module.

For example:

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

Host language variables must be declared as AUTO type variables or every thread. If global variables or static type variables are declared, it is necessary to watch carefully for their use in different threads.

Use of host language modules for each thread is desirable because explicitly declared local descriptions of databases and embedded language statement descriptors are used extensively.

Implicitly declared database descriptors (without the modifier AT Database\_name) are different in different contexts.

If embedded language statement global descriptors are accessed, he calls must be synchronized (by means of semaphores, critical sections, etc.). Otherwise, library internal structures can be damaged.

The two following examples illustrate the use of contexts in multithreading applications.

Example using the same context with different threads:

```
al locate :ctx
connect
spawning threads...
free:ctx
...
thread 1,2...( )
{
USE:ctx
mutex
...
}
```

Example using different contexts with different threads

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

## Allow Thread Creation

The statement that allows thread creation is used for compatibility with Oracle Pro\*C and has no function in Linter's embedded SQL. Its format:

```
EXEC SQL ENABLE THREADS;
```

## Contexts

Contexts are used to support of multithreading applications using embedded SQL.

Context includes:

Zero or more connections to one or more servers;

Zero or more cursors opened through each such connection; and

The following variables containing embedded language query execution status.

<u>Variable</u>	<u>Content</u>
ErrPCI_	completion code
CntPCI_	count of processed rows
IsnPCI_ or RowidPCI	ROWID of the last record fetched from the table
LenPCI_	length of received BLOB portion
DdbPCI_	default connection
Rows_PCI	count of rows
TxtPCI	text of processed embedded language query

## Context Creation

This directive is used for context initialization - memory allocation and internal structures for filling context. The directive format:

```
EXEC SQL CONTEXT ALLOCATE :<Context_variable_name>;
```

<Context\_variable\_name> is a host language variable of type CONTEXT\_PCI. The variable must be declared in a host language variables declaration section that is contained within the operator's visibility scope.

## Context Use

This directive specifies that Context\_variable\_name will be used in all future PCL functions calls until the end of the module or the next CONTEXT USE directive. The directive format:

```
EXEC SQL CONTEXT USE :<Context_variable_name>;
```

<Context\_variable\_name> is a host language variable of type CONTEXTPCI. The variable must be declared in a host language variables declaration section contained within the operator's visibility scope and initialized with the CONTEXT ALLOCATE directive.

## Context Free

This directive is used to release, deallocate, memory occupied by internal structures of the context. The directive format:

```
EXEC SQL CONTEXT FREE :<Context_variable_name>;
```

<Context\_variable\_name> is a host language variable of type CONTEXTPCI. The variable must have been declared in a host language variables declaration section contained within the operator's visibility scope and initialized by the CONTEXT ALLOCATE directive.

An attempt to use or release context variable memory that has not been previously allocated with the CONTEXT ALLOCATE directive will cause a memory protection violation error.

Section "Context Use and Multithreading" contains an example of context use in multithreading applications.

## Communication area

Interaction between a user program and Linter during program execution is handled in a special data exchange communication area.

The statement that creates the area:

```
EXEC SQL include sqlca;
```

Must be present in the text of the embedded SQL program. This statement is a declarative statement. It must be placed in a declarative operators area before executable embedded SQL statements; this is usually in an include files declaration section. The communication area is of static type, in host language terms. So, if the target program links several C modules with embedded SQL operators, each C module must contain its own declaration of communication area.

## Working with Descriptors

Embedded SQL permits construction and execution of SQL queries for which query text quantity, and type of query variable are unknown at program compilation time. Such queries are formed during program execution. Most program executions will present different views because the queries will vary and data will change over time.

When constructing such dynamic queries, we need to describe the quantity, types and names of selected columns. We also need to define pointers to variables in which selected values are to be loaded. These descriptions are stored in descriptors. Descriptors are data structures used to bind variables in dynamic queries.

There are two types of unknown parameters that require specifying two types of descriptors in dynamic queries:

<u>Type of Descriptor</u>	<u>Description</u>
BIND	input variables the values of which are to be substituted in query text.
SELECT	variables into which selected values are to be loaded.

For example

```
t_sqlda * bind_dp;
t_sqlda * select_dp;
```

After declaring the descriptors, it is necessary to allocate memory or them by means of the `sqlald()` function, e.g.

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

The function `sqlald()` has the following prototype:

```
sqlald(max_vars,max_name,max_ind_name),
```

where

<code>max_vars</code>	the maximum number of columns that can be used in the dynamic query;
<code>max_name</code>	the max length of a variable name that can be used in the dynamic query;
<code>max_ind_name</code>	the max length of the indicated variable name that can be described by the descriptor. In addition, the <code>sqlald()</code> function initializes the descriptor's internal buffers.

In addition, the `sqlald()` function initializes the descriptor's internal buffers.

The `DESCRIBE BIND VARIABLES` and `DESCRIBE SELECT LIST` statements are used to fill the `t_sqlda` structure with information about input and output variables, respectively.

The structure of `t_sqlda` is

```
struct t_sqlda {
int N;          /* Max count of variables */
char **V;      /* Pointer to array of host language variables addresses */
int *L;        /* Pointer to array of host language variable length pointers */
Short *T;      /* Pointer to array of host language variable types pointers */
int *D;        /* Pointer to array of host language variable array dimensions */
char **I;      /* Pointer to array of host language indicating variables addresses
pointers */
int F;         /* Count of variables really found by DESCRIBE operator */
char **S;      /* Pointer to array of host language variable name addresses */
Short *M;      /* Max name length pointer */
Short *C;      /* Current variable name length pointer */
char **X;      /* Pointer to array of host language indicating variable name
addresses */
Short *Y;      /* Max indicating variable name length pointer */
Short *Z;      /* Indicating variable current name length pointer */
int NL;        /* Reserved field */
int FR;        /* The descriptor has been used by DESCRIBE operator */
};
typedef struct t_sqlda t_sqlda;
```

Description of `t_sqlda` structure fields:

- N** max number of input and output variables, described by the descriptor. The parameter is set during descriptor initialization and is equal to `max_vars`.
- V** pointer to data buffer addresses array. Elements `V[0]` to `V[N-1]` are cleared during initialization. Buffer memory for `SELECT LIST` must be allocated before processing of statement `EXEC SQL FETCH ... USING DESCRIPTOR ...` Received data are stored in `V[i]`. Buffer initialization for `BIND VARIABLES` descriptor must be performed before processing the statement `EXEC SQL OPEN ... USING DESCRIPTOR ...` Input data are

taken from V[i].

**L** pointer to an array of lengths of input and output variables stored in V[i] buffers. The value of the variable for the SELECT LIST descriptor is determined after receiving information about the structure of the query answer during processing of the EXEC SQL DESCRIBE ... statement (GETA instruction of Call interface). The value of the variable for a BIND descriptor must be determined before opening a cursor with the EXEC SQL OPEN ... USING DESCRIPTOR ... statement. This can be done, for example, by means of PCI\_NewVar( ) function call:

```

PCI_NewVar((char*)&LB,4,1,0,(int*)0);
PCI_NewVar((char*)&RB,4,1,0,(int*)0);
EXEC SQL OPEN CR USING DESCRIPTOR DSI;
    
```

**T** pointer to input or output variable data types array. Data types here are internal types from the PCI library. Supported types:

<u>Data Type</u>	<u>Type Name</u>	<u>Type Code</u>	<u>Comment</u>
int	PCC_INTTYP	1	I/O
float	PCC_FLT_TYP	2	I/O
decimal/numeric	PCC_DEC_TYP	3	I/O
char	PCC_CHR_TYP	4	I/O
varchar	PCC_TXT_TYP	5	I/O
char	PCC_CHR_TYP	6	Input only
varchar	PCC_TXT_TYP	7	Input only

**D** pointer to array of host language variable array dimensions.

**I** an array of addresses of data buffers which store indicating variables.

**F** contains the number of input variables found while parsing a query in a DESCRIBE operator. If F is negative, that descriptor dimension is too small to describe the number of variables encountered during statement parsing for BIND VARIABLE descriptor or when receiving the answer structure from a SELECT LIST descriptor.

**S** pointer to an array of input and output variable names in the order in which they appear during an SQL query.

**M** max variable name length. The parameter is set during descriptor initialization and is equal to max\_name.

**C** pointer to an array which contains name lengths of a query input (after processing of EXEC SQL DESCRIBE BIND VARIABLES ... statement) or output (after processing EXEC SQL OPEN ... USING DESCRIPTOR ... statement) variables.

**X** pointer to indicating variable names array.

**Y** max length of indicating variable name. The parameter is set during descriptor initialization and is equal to max\_ind\_name.

- Z** pointer to indicating variable name lengths array.
- NL** quantity limit for variables described by descriptor.
- FR** flag: descriptor fields were filled by a DESCRIBE statement.

All these variables except NL and FR coincide with the SQLDA structure variables of the Pro\*C Oracle precompiler interface. NL and FR are not supported by Pro\*C.

# Using Stored Procedures


## Stored Procedures Creation and Modification

The operator {CREATE|ALTER}PROCEDURE is used to create or modify stored procedures. It has the following format:

```
EXEC SQL {CREATE|ALTER}PROCEDURE <Procedure_block>;
END-EXEC;
```

The <Procedure\_block> structure is described in detail in document “Stored Procedure Functions”.

END-EXEC; must be present at the end of the procedure block description.

 The statement is processed by the precompiler only when the embedded language statement semantic check is turned on with the precompiler's -S option. If semantic checking is turned off, the compiler will return the error, Semantic check must be turned on.

If semantic checking is turned on, the operator calls the stored procedure compiler spc, which creates the procedure or if already created, modifies it. If an error occurs during procedure creation or modification, a log file will be created with the name of the procedure and an .lsp extension. If there is an error in the stored procedure's name, a log file is created with a name that is the number of the procedure in the current compilation module. The file contains procedure text with error descriptions.

If semantic checking is turned on (-S), the Linter server name, user name (the owner of the created procedure), and user password must be specified. If a non-existent server, wrong user name, or wrong password are specified, the listing will contain Linter's error descriptions.

The compiler creates a stored procedure description with the operator {CREATE|ALTER}PROCEDURE. On execution, the operator is processed in the same manner as is the DECLARE PROCEDURE statement except for the procedure owner's name specification. That is, the name of the procedure's creator must be specified in the EXECUTE PROCEDURE operator, otherwise an error, Undefined name, will be reported.

For example:

```
(precompiler is called with -U USR1/Pass option)
EXEC SQL CREATE PROCEDURE EX() RESULT INT
CODE
END;
END-EXEC;
...
EXEC SQL EXECUTE PROCEDURE USR1.EX();
```

## Stored Procedure Prototype Declaration

The DECLARE operator is used for stored procedure prototype declaration. Its format:

```
EXEC SQL DECLARE <Stored_procedure_header>;
<Stored_procedure_header> ::=
    PROCEDURE <Name> (<Parameter_list>) [ RESULT<Result_type>]
<Name> ::= [<Procedure_owner>]<Procedure_name>
<Procedure_owner> ::= <literal>
<Procedure_name> ::= <literal>
<Parameter_list> ::= <Parameter>, <Parameter> ...
<Parameter> ::= <Modifier> [<Parameter_name>]<Parameter_type>
```

```

<Modifier> ::= { IN | OUT | INOUT }
<Parameter_name> ::= <literal>
<Parameter_type> ::=
    { INT | SMALLINT | REAL | NUMERIC
      | CHAR( ) | BYTE( ) | DATE | BLOB }
<Result_type> ::= { <Parameter_type | CURSOR_PCI }

```

Parameter type serves to bind stored procedure call parameters to host language variables.

If the RESULT parameter is omitted, the stored procedure returns is treated as having returned a NULL.

Unlike the procedure block syntax (see document “Stored Procedure Functions”) the returned cursor description of this procedure does not decode it's fields.

## Stored Procedures Execution

The EXECUTE operator executes the stored procedure. Its format:

```

EXEC SQL EXECUTE PROCEDURE [:<Result_name>
[:<Result_indicating_name>] = ] <Name> (<Actual_parameter_list>);
<Result_name> ::=
    name of host language variable whose type issupported by embedded SQL
<Result_indicating_name> ::=
    name of host language variable of LONG type
<Actual_parameter_list> ::= <Actual_parameter>, <Actual_parameter>
<Actual_parameter> ::=
    { <:Host_language_variable_name>
      [:<Indicating_variable name>] | <Constant> | <NULL> }
<:Host_language_variable_name> ::=
    name of host language variable,declared in host language variable
    declaration section
<Constant> ::= <literal>
<NULL> ::= ',,'(commas indicate skipped parameter)

```

### Notes:

1. If the procedure result is of DATE type, the format of the string returned will be: dd.mm.yyyy:hh:mi:ss.
2. If the procedure result is of BLOB type, a 14 byte header is returned.
3. If the procedure result is of BYTE type, a byte string (unsigned char) is returned.
4. If the procedure result is of CURSOR\_PCI type, a cursor variable is returned. This variable can be used in FETCH, {ADD|GET|CLEAR} BLOB, and CLOSE operators. Also, the indicating variable contains a count of rows selected from the cursor or a zero value if the cursor is empty.
5. If the procedure result is not of CURSOR\_PCI type, a NULL value return indicates a zero value; otherwise, the stream of bytes is the value returned.
6. If an error occurs during stored procedure execution, the ErrPCI\_ variable will contain the Linter error code.

## Precompilation Control

The following statements are used to control the precompilation process DEFINE, UNDEF, IFDEF, IFNDEF, ELSE, ENDIF.

### Declaring Macro Names

The DEFINE and UNDEF statements are used to declare and release macro names:

```
EXEC LINTER DEFINE <name>;  
EXEC LINTER UNDEF <name>;
```

Macro names maybe used in IFDEF and IFNDEF statements. Only one macro name, SQL, is declared by default.

### Conditional Compilation Statements

An IFDEF or IFNDEF statement begins a conditional compilation block:

```
EXEC LINTER IFDEF <name>;  
EXEC LINTER IFNDEF <name>;
```

The text following the IFDEF or IFNDEF statement will be processed if the specified name is declared, or not declared for IFNDEF, as a macro name.

In case of recursively included IFDEF/IFNDEF statements, the input text will be processed only if all conditions are satisfied.

An ELSE statement switches the state of conditional compilation:

```
EXEC LINTER ELSE;
```

If the text before this statement was processed / not processed, hen the text after this statement will not be processed / will be processed.

```
EXEC LINTER ENDIF;
```

This statement finishes a block of conditional compilation which begins with the IFDEF or IFNDEF statement.

## Embedded SQL Program Sample

This sample program fetches records from the Linter demo database.

```
#include <stdio.h> #include <string.h>
/* Communication area declaration - cannot be omitted! */
EXEC SQL INCLUDE SQLCA;
/* SQL declaration 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. Open connection to Linter */
EXEC SQL WHENEVER SQLERROR GOTO Erropen;
User = "SYSTEM/MANAGER";
EXEC SQL CONNECT :User;
/* 2. Prepare statement for execution */
Query ="SELECT NAME, FIRSTNAM, CITY, PHONE FROM
PERSON WHERE NAME=:Name;"; pName = "CLINTON";
EXEC SQL PREPARE ST FROM :Query;
EXEC SQL DECLARE CR CURSOR FOR ST;
/* 3. Open cursor */
EXEC SQL WHENEVER SQLERROR GOTO Err_read;
EXEC SQL WHENEVER NOT FOUND GOTO Not_found;
EXEC SQL OPEN CR USING :pName;
/* 4. Fetch records until no more records */
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 ("No such records\n");
No_more:
/* 5. Close cursor */
EXEC SQL CLOSE CR;
Exit:
/* 6. Close connection to LINTER */
EXEC SQL WHENEVER SQLERROR GOTO Err_clos;
EXEC SQL COMMIT WORK RELEASE;
return;
Err_open:
printf ("Channel opening error: %d\n", ErrPCI_); return;
Err_clos:
printf ("Channel closing error: %d\n", ErrPCI_);
return;
Err_read:
printf ("LINTER error: %d\n ", ErrPCI_);
goto Exit;
```

## Precompiler Messages

<u>Message</u>	<u>Meaning or Solution</u>
Precompiler internal error	Precompiler internal error; contact Linter developers.
Too many host language variables	Total number of variables that declared in all SQL declaration sections of one source module should not exceed 256 for this version of Linter.
Too many included blocks	Total number of recursively included C blocks should not exceed 40 for this version of Linter.
Too many statement declarations	Total number of statement declarations in one declarations source module should not exceed 30 for this version of Linter.
Too many cursor declarations	Total number of cursor declarations in one source module should not exceed 30 for this version of Linter.
Too many stored procedure declarations	Total number of stored procedure declarations (made by means of CREATE PROCEDURE or DECLARE PROCEDURE) should not exceed 64 for this version of Linter.
Too many database declarations	Total number of DB connection declarations in one source module cannot exceed 10 in this version of Linter.
Too many host language variables in one statement	Total number of host variables used in one statement should not exceed 256 for this version of Linter.
Too many included files	Total number of recursively included files (by means of EXEC SQL INCLUDE statement) should not exceed 20 for this version of Linter.
Too many macro-variables	Total number of macro variables should not exceed 20 for this version of Linter.
Too many module declarations	Total number of declared host language modules should not exceed 40 for this version of Linter
Too many including conditional translation directives	Total number of recursively included conditional compilation directives should not exceed 10 for this version of Linter.
String is too long	SQL language statement string should not exceed 4096 chars for this version of Linter.
No memory available	Cannot allocate memory needed for precompiler running.
No hash table entry available	Precompiler's names hash table overflow.
Buffer size exceeded	String buffer overflow.
Unexpected end of file	End of file encountered in the middle of an embedded SQL statement.
End of file within comment	End of file encountered in the middle of comment.
Closing level without opening it	Closing brace (}) encountered without corresponding opening brace ({}).
Unknown character	Illegal character encountered in a precompiler statement.

<u>Message</u>	<u>Meaning or Solution</u>
Undefined name	An undefined name is used as a host language variable or an embedded SQL variable.
Duplicate defined name	A name already declared earlier is declared as a host language variable or an embedded SQL variable.
Indicating variable is illegal here	Using an indicating variable is disabled here.
Array is illegal here	Using an array variable is disabled here.
Too many array dimensions	Multi-dimensional arrays cannot be used as host language variables in Linter embedded SQL.
Floating numbers cannot be unsigned	An error in SQL declaration section.
Invalid array of zero elements	An empty array is disabled.
Element numbers should begin from 1	Reserved for Pascal host language (in the future versions).
Invalid type	Parameter types in procedure description and procedure call are not compatible.
Wrong parameter count	Parameter counts in procedure description and procedure call is different.
Arrays of main and indicating variables have different sizes	Using indicating variable with an array size different from the array size of the main variable is disabled.
You cannot declare two cursors for the same statement	In this version of Linter it is impossible to declare two different cursors for the same statement (this restriction will be cancelled in the future versions).
No declaration section on highest level	A source module doesn't include SQL declaration section, which is not contained in any host language block.
No communication area	There is no INCLUDE SQLCA statement in a source module.
END without BEGIN	END DECLARE SECTION statement encountered in the source module without previous BEGIN DECLARE SECTION statement.
Text variable expected	Only char and VARCHAR variables may be used as username/password buffers as well as BLOB buffers.
Integer2 variable expected	Only an integer variable should be used as an indicating variable, as an execution counter (in FOR clause) or as an offset in BLOB operations. In the first two cases this variable should be two-byte in length.
Embedded language operator expected	Unrecognizable syntax encountered after EXEC SQL keywords.
Check condition expected	After WHENEVER keyword NOT FOUND, SQLERROR or SQLWARNING should follow.
Action expected	After THEN keyword STOP, CONTINUE or GOTO should follow.
Declared object type expected	DECLARE statement should contain declared object type: STATEMENT, CURSOR or DATABASE.

<u>Message</u>	<u>Meaning or Solution</u>
Embedded language or DBMS name expected	SQL or Linter should follow after EXEC.
DBMS option expected	AREASIZE should follow EXEC LINTER.
BIND or SELECT keyword expected	BIND or SELECT should follow after DESCRIBE.
CURSOR_PCI variable expected	CURSOR_PCI type variable is used but not declared.
CONTEXT_PCI variable expected	CONTEXT_PCI type variable is used but not declared.
Parameter type expected	Parameter type declaration expected in a stored procedure declaration.
Parameter modifier expected	Parameter modifier (IN, OUT or INOUT) expected in a stored procedure declaration.
Semicolon (directive terminator) expected	Embedded SQL statement should be finished with the semicolon (;).
Unrecognized declaration	Unknown type of host variable in an SQL declaration section.
No closing quote	Closing quote (" or ') was not found when reading a string within quotes.
Directive prefix not on line beginning	Embedded SQL statement and host language statement cannot be mixed in one source line.
End of line expected	Embedded SQL statement and C statement cannot be mixed in one source line.
ELSE without IFDEF or IFNDEF	An incorrect sequence of the conditional compilation statements.
ENDIF without IFDEF or IFNDEF	An incorrect sequence of the conditional compilation statements.
EOF within conditional translation	There is no ENDIF statement after IFDEF.
BLOB type operation expected	ADD, GET or CLEAR should follow BLOB
Token <Token Name> was expected	The specified token was expected in the source program, but something other appeared.
Error opening file <File Name>	Precompiler cannot open a source file.
Error creating file <File Name>	Precompiler cannot create a target file.
Error rewriting output file	Precompiler cannot write precompilation result into a target file.
Nested modules not allowed	Nested modules not allowed for this version of Linter.
Invalid level for module declaration	MODULE and END MODULE directives must be on the same block include level of the host language.
No MODULE statement	END MODULE directive encountered in a host language block without corresponding MODULE directive.
No END MODULE statement	Host language block contains MODULE directive without corresponding END MODULE directive.

<u>Message</u>	<u>Meaning or Solution</u>
-T precompiler key expected	Precompiler must be called with -T option if statements ENABLE THREADS, CONTEXT { ALLOCATE   USE   FREE } are used.
ENABLE THREADS directive expected	ENABLE THREADS statement must precede CONTEXT { ALLOCATE   USE   FREE } statement.
Unknown token expected	Precompiler internal error; contact Linter developers.
Unknown error	Precompiler internal error; contact Linter developers.
Nested REPEATED SELECT not allowed	For Ingres-compatible style: REPEATED SELECT block cannot contain another REPEATED SELECT call.
SET statement not implemented	SET statement not implemented (except SET DESCRIPTOR).
INQUIRE_SQL statement not implemented	For Ingres-compatible style: the specified option of INQUIRE_SQL statement is not implemented.
Option not implemented	Option must be supported according to SQL standard, but not implemented in this version of Linter.
Lintor Error.CodErr = %ld, SysErr = %ld	Lintor DBMS error (received when connecting to Linter using -S option).

## Completion Codes

<u>Code</u>	<u>Message</u>	<u>Meaning or Solution</u>
0		Normal successful completion.
3000	Not found	No records found (or no records removed, or replaced, or added).
3001	Implementation limit exceeded	One of the following implementation limits is exceeded one SQL statement cannot contain more than 256 host language variables; one SQL statement text cannot be longer than 4096 characters.
3002	Incompatible types	An attempt to fetch a value of a record field into a variable, which type is incompatible with the type of selected value.
3003	Invalid variable count	The USING clause specifies the amount of source variables different from the amount of variables specified in the cursor declaration, or the INTO clause specifies the amount of target variables different from the amount of values selected by the SELECT statement.
3004	No enough memory	Memory allocation error.
3005	Error in statement text	SQL statement text is invalid.
3006	Too many records found	An attempt to execute the SELECT statement which selects more records than target variables can receive.
3007	Incompatible array sizes	Arrays of different dimensions are used as host language variables in the same SQL statement.
3008	Invalid FETCH parameters	When fetching several records in one time, you can use FETCH NEXT only.
3010	Channel is already opened	An attempt to open connection (DATABASE) which is already opened (by means of a CONNECT statement).
3011	Channel is not opened	An attempt to use connection (DATABASE) before connecting to Linter (by means of a CONNECT statement).
3012	Statement is not prepared	An attempt to execute a statement which is not prepared for execution by a PREPARE operator.
3013	Cursor is not opened	An attempt to use a cursor which is not opened yet.
3014	Cursor is already opened	An attempt to open a cursor which is already opened.
3015	DESCRIBE has not performed	If you want to execute OPEN or FETCH statement with a USING DESCRIPTOR clause, you must previously fill in this descriptor by a DESCRIBE statement.
3016	Too few elements in descriptor	Descriptor size is insufficient to contain information about the specified variables.
3017	Row not fetched	To have access to a BLOB value, you must previously FETCH or INSERT a record containing this BLOB value.
3100	An internal error	Internal structures' contents is damaged. Internal library error; contact Linter developers.