Pro*PL/I

Supplement to the Oracle Precompilers Guide

Release 1.8

ORACLE®

Pro*PL/I Supplement to the Oracle Precompilers Guide, Release 1.8

Part No. A87540-01

Contents

1 Writing a Pro*PL/I Program

2 Error Handling and Diagnostics

3 Running the Pro*PL/I Precompiler

4 Sample Programs

5 Implementing Dynamic SQL Method 4

A Differences from Previous Release

B Operating System Dependencies

Index

Send Us Your Comments

Pro*PL/I Supplement to the Oracle Precompilers Guide, Release 1.8

Part No. A87540-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev@us.oracle.com
- FAX: (650) 506-7228   Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op12
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

x

Preface

Devoted exclusively to the Pro*PL/I Precompiler, this manual supplements the language-independent *Programmer's Guide to the Oracle Precompilers.* An understanding of the material in the *Programmer's Guide* is assumed.

This companion book shows you how to write PL/I programs that use the powerful database language SQL to access and manipulate Oracle data. It provides easy-to-follow examples, instructions, and programming tips, as well as several full-length programs to better your understanding and demonstrate the usefulness of embedded SQL.

Who Should Read This Manual?

Developers who are writing new PL/I applications or converting existing PL/I applications to run in the Oracle environment will benefit from reading this manual. Though written especially for programmers, it will also be of value to systems analysts, project managers, and others interested in embedded SQL applications.

To use this manual effectively, you need a working knowledge of the following:

- applications programming in PL/I

- the SQL database language

- Oracle database concepts and terminology

- the concepts, terminology, methods, requirements, and options detailed in the *Programmer's Guide to the Oracle Precompilers*

How This Manual Is Organized

This manual contains five chapters and one appendix. A brief summary of what you will find in each chapter and appendix follows:

Chapter 1: Writing a Pro*PL/I Program

This chapter provides the basic information you need to write a Pro*PL/I program. You learn programming guidelines, coding conventions, language-specific features and restrictions, how to equivalence datatypes, and how to connect to Oracle.

Chapter 2: Error Handling and Diagnostics

This chapter discusses error reporting and recovery. It shows you how to use the SQLCA and the WHENEVER statement to detect errors and status changes. It also shows you how to use the ORACA to diagnose problems.

Chapter 3: Running the Pro*PL/I Precompiler

This chapter details the requirements for running the Pro*PL/I Precompiler. You learn how to issue the precompiler command, how to specify useful precompiler options, and when to do separate and conditional precompilations.

Chapter 4: Sample Programs

This chapter provides several embedded SQL programs to guide you in writing your own. These well-commented programs illustrate the key concepts and features of Pro*PL/I programming.

Chapter 5: Implementing Dynamic SQL Method 4

This chapter shows you how to implement dynamic SQL Method 4, an advanced programming technique that lets you write highly flexible applications. Numerous examples, including a full-length sample program, are used to illustrate the method.

Appendix A: Differences from Previous Release

This appendix lists differences between Pro*PL/I precompiler release 1.5 and 1.6.

Appendix B: Operating System Dependencies

Some details of Pro*PL/I programming vary from system to system. So, occasionally, you are referred to other manuals. For convenience, this appendix collects all external references to system-specific information.

Conventions Used in This Manual

Important terms being defined for the first time are italicized. In discussions, uppercase is used for database objects and SQL keywords, and boldface uppercase is used for the names of PL/I functions and procedures. All PL/I code is in uppercase.

Notation

The following notational conventions are used in code examples:

> < >

Angle brackets enclose the name of a syntactic element.

> [ ]

Brackets indicate that the enclosed items are optional.

> { }

Braces indicate that one, and only one, of the enclosed items is required.

> |

A vertical bar is used to separate options within brackets or braces.

> ...

An ellipsis indicates that the preceding argument or parameter can be repeated, or that statements or clauses irrelevant to the discussion were left out.

ANSI Compliance

The Version 1.6 Pro*PL/I Precompiler complies completely with the following standards:

- American National Standard Database Language SQL, with Integrity Enhancement Option (ANSI X3.135-1989)

- International Standards Organization Database Language SQL (ISO 9075-1989)

- American National Standard Database Language Embedded SQL (ANSI X3.168-1989)

Compliance with these standards has been tested and validated using the appropriate National Institute of Standards and Technology (NIST) test suites.

ANSI compliance is governed by the MODE option. For more information about ANSI compliance and the MODE option, see the *Programmer's Guide to the Oracle Precompilers.*

Related Publications

The following publications are recommended for use with this manual:

- *Oracle7 Server Application Developer's Guide*

- *Oracle7 Server Messages and Codes Manual*

- *Oracle7 Server SQL Language Reference Manual*

- *Programmer's Guide to the Oracle Call Interfaces*

- *Programmer's Guide to the Oracle Precompilers*

- *PL/SQL Release 2.1 and Oracle Precompilers Release 1.6 Addendum*

- *SQL*Forms Designer's Reference*

- *SQL*Plus User's Guide and Reference Manual*

- *Trusted Oracle7 Server Administrator's Guide*

- *Understanding SQL*Net User's Guide*

- *Oracle Network Manager Administrator's Guide*

Occasionally, this manual refers you to other Oracle manuals for system-specific information. Typically, these manuals are called installation or user's guides, but their exact names vary by operating system and platform. For convenience, Appendix A collects all external references to system-specific information.

1

Writing a Pro*PL/I Program

This chapter provides the basic information you need to develop a Pro*PL/I application. You learn the following:

- programming guidelines

- coding conventions

- language-specific features and restrictions

- how to declare and reference host variables, indicator variables, host arrays, and variable-length strings

- how to equivalence datatypes

- how to connect to Oracle

Programming Guidelines

The following sections (arranged alphabetically for quick reference) deal with coding conventions, embedded SQL syntax, and other features that are specific to host PL/I programs.

Comments

You can place PL/I comments (/* ... */) in a SQL statement wherever blanks can be placed (except between the keywords EXEC SQL and within quoted literals). You can also place ANSI SQL-style comments (-- ...) in SQL statements at the end of a line if the statement continues over multiple lines. However, you cannot place an ANSI SQL-style comment at the end of the last line, following the semicolon that terminates the SQL statement. The following example shows both comment styles:

```
EXEC SQL SELECT ENAME, SAL    /* name and salary */
    INTO :EMP_NAME, :SALARY  -- output host variables
    FROM EMP
    WHERE DEPTNO = :DEPT_NUMBER;
```

You cannot nest comments.

Continuation Lines

You can continue SQL statements from one line to the next, even in the middle of a quoted string literal. For example, the following statement inserts the string value 'PURCHASING' into the DNAME column:

```
EXEC SQL INSERT INTO dept (deptno, dname) VALUES (50, 'PURCHASING');
```

Embedded SQL Syntax

When using SQL commands in your host program, you precede the SQL command with the EXEC SQL clause. Embedded SQL syntax is described in the *Oracle7 Server SQL Language Reference Manual*. The precompiler translates all EXEC SQL statements into calls to the runtime library SQLLIB.

Host Variable Names

Host variable names must consist only of letters and digits, and must begin with a letter. They can be any length, but only the first 31 characters are significant. The PL/I compiler normally converts variable names to upper case. Check your compiler documentation for the rules for forming PL/I identifiers on your system.

MAXLITERAL Default Value

The MAXLITERAL precompiler option lets you specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. The MAXLITERAL default value is 256 for Pro*PL/I. But, you might have to specify a lower value if your PL/I compiler cannot handle string literals of that length.

Nulls

In SQL, a NULL column "value" is simply one that is missing, unknown, or inapplicable; it equates neither to zero nor to a blank. Therefore, use either the NVL function, the IS [NOT] NULL operator, or indicator variables when selecting or testing for nulls, and use indicator variables to insert nulls.

In PL/I, the built-in function NULL() simply returns a null pointer value; it is not related to the SQL NULL value in any way.

Operators, Logical

The logical operators are different in SQL and PL/I, as the table below shows.

SQL Operators	PL/I Operators
NOT	^ (prefix)
AND	&
OR	| &: |: ^ (infix)

PL/I logical operators are *not* allowed in SQL statements.

Operators, Relational

The relational operators are similar in SQL and PL/I, as the following table shows:

SQL Operators	PL/I Operators
=	=
<>, !=, ^=	^=
>	>
<	<
>=	>=
<=	<= ^< ^>

PL/I Versions

The Pro*PL/I Precompiler supports the standard implementation of PL/I for your operating system. See your Oracle installation or user's guide for more information.

Preprocessor

The Pro*PL/I Precompiler does not support PL/I preprocessor directives. Do not use preprocessor directives, even %INCLUDE, within EXEC SQL blocks. You can, of course, use whatever PL/I preprocessor facilities your compiler makes available in pure PL/I code. Code included by a PL/I preprocessor is not precompiled.

Quotation Marks and Apostrophes

In SQL, quotation marks are used to delimit identifiers containing special or lowercase characters, as in

```
EXEC SQL DELETE FROM "Emp2" WHERE DEPTNO = 30;
```

Both SQL and PL/I use apostrophes to delimit strings, as in the PL/I statements

```
DCL NAME CHAR(20) VARYING;
...
NAME = 'Pro*PL/I';
PUT SKIP LIST (NAME);
```
or the SQL statement

```
EXEC SQL SELECT ENAME, SAL FROM EMP WHERE JOB = 'MANAGER';
```

Scope of Variables

Host variables used in embedded SQL statements follow the normal scoping rules of PL/I. Any host variable that you want to use in an embedded SQL statement must also be accessible to PL/I statements in the same block.

SQL Statement Terminator

End all embedded SQL statements with a semicolon, as shown in the following examples:

```
EXEC SQL DELETE FROM EMP WHERE DEPTNO = 20;

EXEC SQL SELECT ENAME, EMPNO, SAL, COMM
    INTO :EMP_NAME, :EMP_NUMBER, :SALARY, :COMMISSION
    FROM EMP
    WHERE JOB LIKE 'SALES%'
    AND COMM IS NOT NULL;
```

Statement Labels

You can associate standard PL/I statement labels (for example, LABEL_NAME:) with SQL statements. The Pro*PL/I Precompiler can handle labels up to 31 characters long.

Required Declarations and SQL Statements

This section describes the variable declarations and SQL statements that must be present in every Pro*PL/I source file.

The Declare Section

You must declare all variables that will be used in embedded SQL statements inside a SQL Declare Section, and the Declare Section must physically precede the embedded SQL statements that use the declared host variables. This section can be placed outside a procedure block or within a procedure block or a begin block. Each block that contains a SQL statement must have a Declare Section in scope, even if the statement does not contain host variables. In this case, the Declare Section is empty. A common solution is to place a Declare Section near the start of the main procedure block. The following example shows a SQL Declare Section in which two host variables are declared.

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
        DCL EMP_NAME CHAR(20) VARYING,
            SALARY   FLOAT(6);
EXEC SQL END DECLARE SECTION;
```

The only statements that are allowed in a Declare Section are

■   host and indicator variable declarations

■   EXEC SQL INCLUDE statements

■   EXEC SQL VAR statements

■   comments

A Pro*PL/I source file can have multiple Declare Sections.

You must declare PL/I variables that are to be used as host variables inside a SQL Declare Section. You should *not* declare PL/I variables that are not to be used as host variables in a SQL Declare Section. Always use the standard PL/I declaration syntax to declare variables.

Using the INCLUDE Statement

The INCLUDE statement lets you copy files into your host program, as the following example shows:

```
/* copy in the SQL Communications Area (SQLCA) */
EXEC SQL INCLUDE SQLCA;
...
/* copy in the Oracle Communications Area (ORACA) */
EXEC SQL INCLUDE ORACA;
```

You can INCLUDE any file. When you precompile your Pro*PL/I program, each EXEC SQL INCLUDE statement is replaced by a copy of the file named in the statement.

If your system uses file extensions but you do not specify one, the Pro*PL/I Precompiler assumes the default extension for source files (usually PLI). The default extension is system dependent. Check the Oracle installation or user's guide for your system.

You can set a directory path for INCLUDEd files by specifying the precompiler option

```
INCLUDE=PATH
```

where *path* defaults to the current directory. The precompiler searches first in the current directory, then in the directory specified by INCLUDE, and finally in a directory for standard INCLUDE files. So, you need not specify a directory path for standard files such as the SQLCA and ORACA. However, you must use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory.

You can specify more than one path on the command line, as follows:

```
INCLUDE=PATH1 INCLUDE=PATH2 ...
```

The precompiler searches first in the current directory, then in the directory named by PATH1, then in the directory named by PATH2, and finally in the directory for standard INCLUDE files.

The syntax for specifying a directory path is system specific. Check the Oracle installation or user's guide for your system.

Caution

Remember, the precompiler searches for a file in the current directory first—even if you specify a directory path. So, if the file you want to INCLUDE resides in another directory, make sure no file with the same name resides in the current directory.

> **Note:** Don't confuse the SQL command INCLUDE with the PL/I directive %INCLUDE. You can use %INCLUDE to copy in the SQLCA, ORACA, or SQLDA because they contain no embedded SQL. But, for a file that contains embedded SQL, you must use the EXEC SQL INCLUDE statement to ensure that the file's contents are examined and processed by the precompiler. For improved readability, it is recommended that you use the SQL INCLUDE command to include all files that pertain to embedded SQL, even those that contain only PL/I statements.

Event and Error Handling

Release 1.6 provides forward and backward compatibility with regard to checking the outcome of executing SQL statements. The SQLCA data structure and SQLCODE status variable can be used in the same manner as in previous releases. The SQLSTATE status variable is introduced in release 1.6. There are restrictions on using SQLCA, SQLCODE, and SQLSTATE depending on how you set the MODE and DBMS options. For more information, see Chapter 2, "Error Handling and Diagnostics".

Host Variables

Host variables are the key to communication between your host program and Oracle. Typically, a host program inputs data to Oracle, and Oracle outputs data to the program. Oracle stores input data in database columns, and stores output data in program host variables.

Declaring Host Variables

You declare a host variable in the SQL Declare Section according to the rules of PL/I, specifying a PL/I datatype supported by Oracle. The PL/I datatype must be compatible with that of the host variable's source or target database column.

The following table describes the PL/I datatypes you can specify in the Declare Section:

PL/I Datatype	Description
CHARACTER (N)	string of length N
CHARACTER (N) VARYING	string of maximum length N
FIXED BINARY (15)	short signed integer
FIXED BINARY (31)	signed integer
FIXED DECIMAL (N)	decimal number of N digits
FIXED DECIMAL (P,S)	decimal with precision and scale
FLOAT BINARY (N)	floating point number
FLOAT DECIMAL (N)	float of maximum length N

You can also declare one-dimensional arrays of each of these types.

Note: Oracle supports *only* the PL/I datatypes in the table above.

Structures

You can declare structures in the SQL Declare Section, and can use individual structure elements as host variables. The elements of the structure must be of datatypes allowed in a Declare Section. You cannot reference the whole structure as a host variable. This means, for example, that if you have a structure containing three elements, you cannot SELECT three columns into the elements by referencing the top level of the structure. You must reference each element by name as a host variable.

You cannot use the LIKE attribute with structure declarations in the SQL Declare Section.

An Example

In the following example, you declare five host variables for use later in the Pro*PL/I program:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL USERNAME   CHAR(10) VARYING INIT('SCOTT'),
        PASSWORD   CHAR(10) VARYING INIT('TIGER'),
        EMP_NUMBER FIXED BINARY(31),
        SALARY     FIXED DECIMAL(7,2),
        DEPTNO     FIXED DECIMAL(2) INIT(10);
EXEC SQL END DECLARE SECTION;
```

You can use attribute factoring within a SQL Declare Section, as in

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL (VAR1, VAR2, VAR3) FIXED DECIMAL;
EXEC SQL END DECLARE SECTION;
```

which is equivalent to

```
EXEC SQL BEGIN DECLARE SECTION;
   DCL VAR1 FIXED DECIMAL,
       VAR2 FIXED DECIMAL,
       VAR3 FIXED DECIMAL;
EXEC SQL END DECLARE SECTION;
```

Special Requirements

You must explicitly declare host variables in the Declare Section of the procedure block that uses them in SQL statements. Thus, variables passed to a subroutine or function and used in SQL statements within the routine must be declared in the Declare Section. An example follows:

```
PROCEDURE TEST: OPTIONS(MAIN)

    DCL  EMPNO     FIXED BIN(31),
         EMP_NAME  CHAR(10) VARYING,
         SALARY    FLOAT(6);
..
.   EMPNO = 7499;
    CALL GET_SALARY(EMPNO, EMP_NAME, SALARY);
..
```

```
            .
GET_SALARY: PROCEDURE(EMPNO, NAME, SALARY);
EXEC SQL BEGIN DECLARE SECTION;
    DCL  EMPNO    FIXED BIN(31),
         NAME     CHAR(10) VARYING,
         SALARY   FLOAT(6);
EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT ENAME, SAL
               INTO :EMP_NAME, :SALARY
               FROM EMP
               WHERE EMPNO = :EMPNO;
END GET_SALARY;
```

Referencing Host Variables

A host variable must be prefixed with a colon (:) in SQL statements, but must not be prefixed with a colon in PL/I statements, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL (EMP_NUMBER, SAL) FIXED DECIMAL(7,2);
EXEC SQL END DECLARE SECTION;

PUT SKIP LIST ('Enter employee number: ');
GET EDIT (EMP_NUMBER) (F(4));

EXEC SQL SELECT SAL
    INTO :SAL
    FROM EMP
    WHERE EMPNO = :EMP_NUMBER;
```

Though it might be confusing, you can give a host variable the same name as an Oracle table or column, as the previous example showed (SAL).

Restrictions

A host variable cannot substitute for a column, table, or other Oracle object in a SQL statement, and must not be an Oracle reserved word.

Table 1–1   Compatible Internal Datatypes

Internal Type	PL/I Type	Description
CHAR(X)[1] VARCHAR2(Y)[1]	CHARACTER (N)	string
	CHARACTER (N) VARYING	variable-length string
	FIXED BINARY (15)	small signed integer
	FIXED BINARY (31)	signed integer
	FIXED DECIMAL (p,s)	fixed-point number
	FIXED DECIMAL (N)	fixed-point number
	FLOAT	floating-point number
	FLOAT DECIMAL (N)	floating-point number
NUMBER NUMBER (P,S)[2]	FIXED BINARY (15)	small signed integer
	FIXED BINARY (31)	signed integer
	FIXED DECIMAL (p,s)	fixed-point number
	FIXED DECIMAL (N)	fixed-point number
	FLOAT	floating-point number
	FLOAT DECIMAL (N)	floating-point number
	CHARACTER (N)	string[3]
	CHARACTER (N) VARYING	variable-length string
DATE[4] LONG RAW(X)[1] LONG RAW ROWID[5] MLSLABEL[6]	CHARACTER (N)	string
	CHARACTER (N) VARYING	variable-length string

Notes:
1. X ranges from 1 to 255. Y ranges from 1 to 2000. 1 is the default value.
2. P ranges from 2 to 38. S ranges from -84 to 127.
3. Strings can be converted to NUMBERs only if they contain convertible characters ('0' to '9', '.', '+', '-', 'E', 'e'). Note that the NLS settings in effect on your system might change the decimal point from '.' to ','.
4. When converted as a string type, the default size of a DATE depends on the NLS settings in effect on your system. When converted as a binary type, the size is 7 bytes.
5. When converted as a string type, a ROWID requires between 18 and 256 bytes. When converted as a binary value, the length is system dependent.
6. Trusted Oracle7 only.

IndicatorVariables

You use indicator variables to provide information to Oracle about the status of a host variable, or to monitor the status of data that is returned from Oracle. An indicator variable is always associated with a host variable.

Declaring Indicator Variables

An indicator variable must be explicitly declared in the SQL Declare Section as a 2-byte signed integer (FIXED BINARY(15)) and must not be an Oracle reserved word. In the following example, you declare
two indicator variables (the names SAL_IND and COMM_IND are arbitrary):

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL EMP_NAME CHAR(10) VARYING,
        (SALARY, COMMISSION) FIXED DECIMAL(7,2),
        /* indicator variables */
        (SAL_IND, COMM_IND) FIXED BINARY(15);
EXEC SQL END DECLARE SECTION;
```

Referencing Indicator Variables

You can use indicator variables in the VALUES, INTO, and SET clauses. In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable. In PL/I statements, an indicator variable must *neither* be prefixed with a colon *nor* appended to its associated host variable. An example follows:

```
EXEC SQL SELECT sal INTO :SALARY :SAL_IND FROM emp
    WHERE empno = :EMP_NUMBER;

IF SAL_IND = -1 THEN
    PUT SKIP LIST('Salary is null.');
```

To improve readability, you can precede any indicator variable with the optional keyword INDICATOR. You must still prefix the indicator variable with a colon. The correct syntax is

```
:HOST_VARIABLE INDICATOR :INDICATOR_VARIABLE
```

which is equivalent to

```
:HOST_VARIABLE :INDICATOR_VARIABLE
```

You can use both forms of expression in your host program.

Restriction

Indicator variables cannot be used in the WHERE clause to search for nulls. For example, the following DELETE statement triggers an Oracle error at run time:

```
/* Set indicator variable. */
COMM_IND = -1;
EXEC SQL DELETE FROM emp WHERE comm = :COMMISSION :COMM_IND;
```

The correct syntax follows:

```
EXEC SQL DELETE FROM emp WHERE comm IS NULL;
```

Oracle Restrictions

When DBMS=V6, Oracle does not issue an error if you SELECT or FETCH a null into a host variable that is not associated with an indicator variable. However, when DBMS=V7, if you SELECT or FETCH a null into a host variable that has no indicator, Oracle issues the following error message:

```
ORA-01405: fetched column value is NULL
```

ANSI Requirements

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host variable that is not associated with an indicator variable, Oracle issues the following error message:

```
ORA-01406: fetched column value was truncated
```

However, when MODE={ANSI | ANSI14 | ANSI13}, no error is generated.

Host Arrays

Host arrays can boost performance by letting you manipulate an entire collection of data items with a single SQL statement. With few exceptions, you can use host arrays wherever scalar host variables are allowed. And, you can associate an indicator array with any host array.

Declaring Host Arrays

You declare and dimension host arrays in the Declare Section. In the following example, you declare three host arrays and dimension them with 50 elements:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL EMP_NAME(50) CHAR(10),
```

```
          (EMP_NUMBER(50), SALARY(50)) FIXED DECIMAL(7,2);
EXEC SQL END DECLARE SECTION;
```

Restrictions

You cannot specify a lower dimension bound for host arrays. For example, the following declarations are *invalid*:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL EMP_NAME(26:50) CHAR(10),
        (EMP_NUMBER(26:50), SALARY(26:50)) FIXED DECIMAL(7,2);
EXEC SQL END DECLARE SECTION;
```

Multidimensional host arrays are *not* allowed. Thus, the two-dimensional host array declared in the following example is *invalid*:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL HI_LO_SCORES(25,25) FIXED BIN(31);   /* invalid */
EXEC SQL END DECLARE SECTION;
```

Referencing Host Arrays

If you use multiple host arrays in a single SQL statement, their sizes should be the same. This is not a requirement, however, because the Pro*PL/I Precompiler always uses the *smallest* array size for the SQL operation.

```
DO J = 1 TO 10;
    EXEC SQL INSERT INTO EMP (EMPNO, SAL)
        VALUES (:EMP_NUMBER(J), :SALARY(J));   /* invalid */
END;
```

You do not need to process host arrays in a loop. Simply use the unsubscripted array names in your SQL statement. Oracle treats a SQL statement containing host arrays of dimension *n* like the same statement executed *n* times with *n* different scalar variables. For more information about using host arrays, see Chapter **8** of the *Programmer's Guide to the Oracle Precompilers*.

Using Indicator Arrays

You can use indicator arrays to assign nulls to input host arrays, and to detect nulls or truncated values in output host arrays. The following example shows how to INSERT using indicator arrays:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL  EMP_NUMBER(50)     FIXED BIN(31),
```

```
         DEPT_NUMBER(50)    FIXED BIN(31),
         COMMISSION(50)     REAL,
         COMM_IND(50)       FIXED BIN(15);
EXEC SQL END DECLARE SECTION;

/* Populate the host and indicator arrays. To insert a
   null, assign a -1 to the appropriate element in the
   indicator array. */

EXEC SQL INSERT INTO emp (empno, deptno, comm)
    VALUES (:EMP_NUMBER, :DEPT_NUMBER, :COMMISSION :COMM_IND);
```

Oracle Restrictions

Mixing scalar host variables with host arrays in the VALUES, SET, INTO, or WHERE clause is *not* allowed. If any of the host variables is an array, all must be arrays.

Also, you cannot use host arrays with the CURRENT OF clause in an UPDATE or DELETE statement.

When DBMS=V6, no error is generated if you SELECT or FETCH nulls into a host array that is not associated with an indicator array. So, when doing array SELECTs and FETCHes, always use indicator arrays. That way, you can test for nulls in the associated output host array.

When DBMS=V7, if you SELECT or FETCH a null into a host variable that is not associated with an indicator variable, Oracle stops processing, sets *sqlerrd[3]* to the number of rows processed, and issues the following error message:

```
ORA-01405: fetched column values is NULL
```

ANSI Restrictions and Requirements

When MODE={ANSI | ANSI13 | ORACLE}, array SELECTs and FETCHes are allowed. You can flag the use of arrays by specifying the FIPS precompiler option.

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host array that is not associated with an indicator array, Oracle stops processing, sets *sqlerrd[3]* to the number of rows processed, and issues the following error message:

```
ORA-01406: fetched column value was truncated
```

When MODE=ANSI13, Oracle stops processing and sets *sqlerrd[3]* to the number of rows processed but no error is generated.

When MODE=ANSI, Oracle does not consider truncation an error.

Pointers as Host Variables

You cannot use PL/I BASED variables in SQL statements. Also, PL/I pointers cannot be directly referenced in SQL statements. This restriction includes reference to structure elements using pointers. That is, you cannot declare a BASED structure in a Declare Section, allocate the structure, and then refer to the elements with respect to the pointer in a SQL statement.

The following code is accepted by the precompiler, but does not execute correctly (an Oracle error message is issued at runtime):

```
EXEC SQL BEGIN DECLARE SECTION;
DCL 1 EMP_STRUCT BASED,
     2 EMP_NAME CHAR(20),
     2 EMP_SAL FIXED DECIMAL(7,2);
DCL  EMP_PTR POINTER;
EXEC SQL END DECLARE SECTION;
...
ALLOCATE EMP_STRUCT SET(EMP_PTR);
PUT SKIP LIST ('Enter employee name: ');
GET LIST (EMP_PTR->EMP_NAME);
EXEC SQL INSERT INTO EMP (ENAME, EMPNO, DEPTNO)
    VALUES (:EMP_PTR->EMP_NAME, 8000, 20):
```

You can, of course, use pointers in pure PL/I code.

CHARACTER VARYING

The Oracle character datatypes can be directly converted to the PL/I CHARACTER VARYING datatype. You declare CHARACTER VARYING as a host variable in the normal PL/I style, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   ..
.   DCL EMP_NAME CHARACTER (20) VARYING,
   ...
EXEC SQL END DECLARE SECTION;
```

VARYINGReturning Nulls to a CHARACTER (N)

Oracle automatically sets the length of a CHARACTER (N) VARYING output host variable. If you SELECT or FETCH a null into a CHARACTER (N) VARYING variable, Oracle sets the length to zero. The variable itself remains unchanged.

Handling Character Data

This section explains how the Pro*PL/I Precompiler handles character host variables. There are two host variable character types:

- CHARACTER (N) variables

- CHARACTER (N) VARYING variables

Effects of the MODE Option

The MODE option, which you can specify on the command line, determines how the Pro*PL/I Precompiler treats data in character arrays and strings. The MODE option allows your program to take advantage of ANSI fixed-length strings, or to maintain compatibility with previous versions of Oracle and Pro*PL/I.

- MODE=Oracle

- MODE=ANSI

  Note: The MODE option does not affect the way Pro*PL/I handles CHARACTER (N) VARYING host variables.

These choices are referred to in this section as Oracle mode and ANSI mode, respectively. Oracle is the default mode, and is in effect when the MODE option is not specified on the command line. When discussing character handling, MODE={ANSI13 | ANSI14} is effectively equivalent to Oracle mode.

The MODE option affects the way character data is treated on input from your host variables to the Oracle table.

On Input

When the mode is Oracle, the program interface strips trailing blanks up to the first non-blank character. After stripping the blanks, the value is sent to the database. If you are inserting into a fixed-length CHAR column, trailing blanks are then re-appended to the data by Oracle, up to the length of the database column. If you are inserting into a variable-length VARCHAR2 column, Oracle never appends blanks.

When the mode is ANSI, trailing blanks in the CHARACTER host variable are never stripped.

Be sure that the input host variable does not contain characters other than blanks after the data. For example, null characters are not stripped, and are inserted into the database. The easiest way to insure this in PL/I is to always use CHARACTER(N) host variables for character data. When values are read into these variables, or assigned to them, PL/I appends blanks to the data, up to the length of the variable. The following example illustrates this:

```
/* Declare host variables  */
EXEC SQL BEGIN DECLARE SECTION;
    DCL  EMP_NAME        CHAR(10),
         JOB_NAME        CHAR(8)    /* Note size */
EXEC SQL END DECLARE SECTION;

PUT SKIP LIST('Enter employee name: ');
/*  Assume the user enters 'MILLER'  */
GET EDIT(EMP_NAME) (A(10));
JOB_NAME = 'SALES';

EXEC SQL INSERT INTO emp (empno, ename, deptno, job)
         VALUES (1234, :EMP_NAME, 20, :JOB_NAME)
```

If you precompile this example in Oracle mode, the values 'MILLER' and 'SALES' are sent to the database, since the program interface strips the trailing blanks that PL/I appends to the CHARACTER host variables. If you precompile this example in ANSI mode, the trailing blanks that PL/I appends are not stripped, and the values 'MILLER   ' (four trailing blanks) and 'SALES   ' (three trailing blanks) are sent to the database.

In ANSI mode, if the JOB column in the EMP table is defined as CHAR(10), the resulting value in the JOB column is 'SALES   ' (five trailing blanks). If the JOB column is defined as VARCHAR2(10), the resulting value in the column is 'SALES   ' (three trailing blanks, since the host variable is a CHARACTER (8). This might not be what you want, so be careful when inserting data into VARCHAR2 columns using programs that were precompiled in ANSI mode.

On Output

The MODE option does not affect the way that character data are treated on output. When you use a CHARACTER (N) variable as an output host variable, the program interface always blank-pads it. In our example, when your program fetches the string 'MILLER' from the database, EMP_NAME contains the value 'MILLER   '

(with 4 blanks). This character string can be used without change as input to another SQL statement.

The Oracle Datatypes

Oracle recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify the formats used by Oracle to store column values in database tables, as well as the formats used to represent pseudocolumn values. External datatypes specify the formats used to store values in input and output host variables. For descriptions of the Oracle datatypes, see Chapter 3 of the *Programmer's Guide to the Oracle Precompilers*.

Internal Datatypes

For values stored in database columns, Oracle uses the following internal datatypes, which were chosen for their efficiency:

Name	Code	Description
VARCHAR2	1	2000-byte, variable-length character string
NUMBER	2	fixed or floating point number
LONG	8	2^{31}-1 byte, variable-length character string
ROWID	11	operating-system dependent
DATE	12	7-byte, fixed-length date/time value
RAW	23	255-byte, variable-length binary data
LONG RAW	24	$2^{3}1$-1 byte, variable-length binary data
CHAR	96	255-byte, fixed-length character string
MLSLABEL	106	variable-length binary data, 2-5 bytes

These internal datatypes can be quite different from PL/I datatypes. For example, PL/I has no equivalent to the NUMBER datatype, which was specially designed for portability.

SQL Pseudocolumns and Functions

SQL recognizes the following pseudocolumns and parameterless functions, which return specific data items:

Pseudocolumn	Corresponding Internal Datatype	Code
NEXTVAL	NUMBER	2
CURRVAL	NUMBER	2
LEVEL	NUMBER	2
ROWNUM	NUMBER	2
ROWID	ROWID	11
ROWLABEL	MLSLABEL	106

Function	Corresponding Internal Datatype	Code
USER	VARCHAR2	1
UID	NUMBER	2
SYSDATE	DATE	12

Pseudocolumns are not actual columns in a table, but, like columns, their values must be SELECTed from a table.

You can reference SQL pseudocolumns and functions in SELECT, INSERT, UPDATE and DELETE statements.

External Datatypes

As the table below shows, the external datatypes include all the internal datatypes plus several datatypes found in popular host languages. For example, the INTEGER external datatype refers to a PL/I FIXED BINARY(31).

Name	Code	Description
VARCHAR2	1	variable-length character string
NUMBER	2	number
INTEGER	3	signed integer
FLOAT	4	floating point number
STRING	5	variable-length null-terminated character string
VARNUM	6	variable-length number
DECIMAL	7	COBOL or PL/I packed decimal
LONG	8	fixed-length character string
VARCHAR	9	variable-length character string
ROWID	11	binary value
DATE	12	fixed-length date/time value
VARRAW	15	variable-length binary data
RAW	23	fixed-length binary data
LONG RAW	24	fixed-length binary data
UNSIGNED	68	unsigned integer
DISPLAY	91	COBOL numeric character string
LONG VARCHAR	94	variable-length character string
LONG VARRAW	95	variable-length binary data
CHAR	1	variable-length character string, if DBMS=V6
	96	fixed-length character string, if DBMS=V7
CHARZ	97	fixed-length null-terminated character string
MLSLABEL	106	variable-length binary data

Datatype Conversion

At precompile time, an external datatype is assigned to each host variable in the Declare Section. For example, the precompiler assigns the Oracle FLOAT external datatype to host variables of type FLOAT DECIMAL.

At run time, the datatype code of every host variable used in a SQL statement is passed to Oracle. Oracle uses the codes to convert between internal and external datatypes.

Before assigning a SELECTed column (or pseudocolumn) value to an output host variable, Oracle must convert the internal datatype of the source column to the datatype of the host variable. Likewise, before assigning or comparing the value of an input host variable to a column, Oracle must convert the external datatype of the host variable to the internal datatype of the target column.

Conversions between internal and external datatypes follow the usual data conversion rules. For example, you can convert a CHAR value of '1234' to a FIXED BINARY(15) value. But, you cannot convert a CHAR value of '65543' (number too large) or '10F' (number not decimal) to a FIXED BINARY(15) value. Likewise, you cannot convert a CHARACTER(N) VARYING value that contains alphabetic characters to a NUMBER value.

For more information about datatype conversion, see Chapter 3 of the *Programmer's Guide to the Oracle Precompilers.*

Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle interprets input data, and the way Oracle formats output data. On a variable-by-variable basis, you can equivalence supported PL/I datatypes to Oracle external datatypes.

Host Variable Equivalencing

By default, the Pro*PL/I Precompiler assigns a specific external datatype to every host variable. The following table shows the default assignments:

Host Datatype	External Datatype	Code
CHARACTER (N)	VARCHAR2	1 when MODE!=ANSI
FIXED BINARY (15)	INTEGER	3 when MODE=ANSI
FIXED BINARY (31)	INTEGER	3
FLOAT BINARY (N)	FLOAT	4
FLOAT DECIMAL (P,S)	FLOAT	4
FIXED DECIMAL (N)	DECIMAL	7
FIXED DECIMAL (P,S)	DECIMAL	7
CHARACTER (N) VARYING	VARCHAR	9

Using the VAR statement, you can override the default assignments by equivalencing host variables to Oracle external datatypes in the Declare Section. The syntax you use is

```
EXEC SQL VAR host_variable
    IS type_name [ ( {length | precision,scale} ) ];
```

where:

host_variable	Is an input or output host variable (or host array) declared earlier in the Declare Section.
type_name	Is the name of a valid external datatype.
length	Is an integer literal specifying a valid length in bytes.
precision, scale	Specify those quantities where required by the type.

Host variable equivalencing is useful in several ways. For example, you can use it when you want Oracle to store but not interpret data. Suppose you want to store a host array of 4-byte integers in a RAW database column. Simply equivalence the host array to the RAW external datatype, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL INT_ARRAY(50) FIXED BINARY(31);
...
/*  Reset default external datatype to RAW */
    EXEC SQL VAR INT_ARRAY IS RAW(200);
```

```
EXEC SQL END DECLARE SECTION;
```

With host arrays, the length you specify must match exactly the buffer size required to hold the array. So, you specify a length of 200, which is the buffer size required to hold 50 4-byte integers.

The following external datatypes cannot be used in the VAR command in Pro*PL/I:

- NUMBER (use VARNUM instead)
- UNSIGNED (not supported in Pro*Pl/I)
- DISPLAY (COBOL only)
- CHARZ (C only)

Embedding PL/SQL

The Pro*PL/I Precompiler treats a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in a host program that you can place a SQL statement.

To embed a PL/SQL block in your host program, you simply declare the variables to be shared with PL/SQL, and bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END-EXEC.

Using Host Variables with PL/SQL

Inside a PL/SQL block, host variables are treated as global to the entire block, and can be used anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.

Using Indicator Variables with PL/SQL

In a PL/SQL block, you cannot refer to an indicator variable by itself; it must be appended to its associated host variable. And, if you refer to a host variable with its indicator variable, you must always refer to it that way in the same block.

Handling Nulls

When entering a block, if an indicator variable has a value of -1, PL/SQL automatically assigns a NULL value to the host variable. When exiting the block, if a host variable has a NULL value, PL/SQL automatically assigns a value of -1 to the indicator variable.

Handling Truncated Values

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string.

SQLCHECK

You must use the SQLCHECK=SEMANTICS option when precompiling a program with an embedded PL/SQL block. You may also want to use the USERID option. See the *Programmer's Guide to the Oracle Precompilers* for more information.

Connecting to Oracle

Your host program must log in to Oracle before it can manipulate data. To log in, use the SQL connect statement

```
EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
```

where "USERNAME" and "PASSWORD" are PL/I character strings containing the user ID and the Oracle password. Or, you can use the SQL statement

```
EXEC SQL CONNECT :USER_PASSWORD;
```

where "USER_PASSWORD" is a PL/I character variable containing the user ID, a slash (/), and the password. The following examples show both ways of connecting to Oracle:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL USER_NAME CHAR(6) INIT('SCOTT'),
        PASSWORD  CHAR(6) INIT('TIGER');
EXEC SQL END DECLARE SECTION;
...
EXEC SQL CONNECT :USER_NAME IDENTIFIED BY :PASSWORD;
DCL USER_PWD CHAR(14);
...
USER_PWD = 'SYSTEM/MANAGER';
EXEC SQL CONNECT :USER_PWD;
```

Automatic Logins

You can automatically log in to Oracle with the user ID

prefixusername

where *username* is the current operating system user or task name, *prefixusername* is a valid Oracle username, and *prefix* is a value determined by the Oracle initialization parameter OS_AUTHENT_PREFIX. For backward compatibility, *prefix* defaults to OPS$. For more information about operating system authentication, see the *Oracle7 Server Administrator's Guide.*

```
EXEC SQL BEGIN DECLARE SECTION;
   ..
.   DCL OracleID CHAR(1) INIT('/');
EXEC SQL END DECLARE SECTION;
...
EXEC SQL CONNECT :OracleID;
```

This automatically connects you as user *prefixusername*. For example, if your operating system username is RHILL, and OPS$RHILL is a valid Oracle username, connecting with '/' automatically logs you on to Oracle as user OPS$RHILL.

Concurrent Logins

Your application can use SQL*Net to concurrently access any combination of remote and local databases, or make multiple connections to the same database. In the following example, you connect to two non default databases concurrently:

```
EXEC SQL BEGIN DECLARE SECTION;
    DCL USR    CHAR(5),
        PWD    CHAR(5),
        DBSTR1 CHAR(11),
        DBSTR2 CHAR(11);
EXEC SQL END DECLARE SECTION;

USR = 'SCOTT';
PWD = 'TIGER';
DBSTR1 = 'D:NODE1-Database1';
DBSTR2 = 'D:NODE1-Database2';

/* Give each database connection a unique name. */
EXEC SQL DECLARE DBNAM1 DATABASE;
EXEC SQL DECLARE DBNAM2 DATABASE;

/* Connect to the two non-default databases. */
EXEC SQL CONNECT :USR IDENTIFIED BY :PWD
    AT DBNAM1 USING :DBSTR1;
EXEC SQL CONNECT :USR IDENTIFIED BY :PWD
    AT DBNAM2 USING :DBSTR2;
```

DBNAM1 and DBNAM2 name the non default connections; they are identifiers used by the precompiler, *not* host variables.

2

Error Handling and Diagnostics

This chapter discusses error reporting and recovery. You learn how to handle errors and status changes using SQLSTATE, the SQLCA, SQLCODE and the WHENEVER statement. You also learn how to diagnose problems using the ORACA.

SQLSTATE, the SQLCA, and SQLCODE

Release 1.6 provides forward and backward compatibility with regard to checking the outcome of executing SQL statements. The SQLCA data structure containing status information and SQLCODE status variable can be used in the same manner as in previous releases. The SQLSTATE status variable is introduced in release 1.6.

Declaring SQLSTATE

When MODE=ANSI, you must declare SQLSTATE or SQLCODE. Declaring the SQLCA is optional. When MODE=ORACLE, not declaring the SQLCA causes compile time warnings and runtime errors.

Unlike SQLCODE, which stores signed integers and can be declared outside the Declare Section, SQLSTATE stores 5-character strings and must be declared inside the Declare Section. You declare SQLSTATE as:

```
DCL SQLSTATE CHAR(5);
```

Note: SQLSTATE must be declared with *exactly* 5 characters.

SQLSTATE Values

SQLSTATE status codes consist of a 2-character *class code* followed by a 3-character *subclass code*. Aside from class code 00 ("successful completion"), the class code denotes a category of exceptions. And, aside from subclass code 000 ("not applicable"), the subclass code denotes a specific exception within that category. For example, the SQLSTATE value '22012' consists of class code 22 ("data exception") and subclass code 012 ("division by zero").

Each of the five characters in a SQLSTATE value is a digit (0..9) or an uppercase Latin letter (A..Z). Class codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined conditions (those defined in SQL92). All other class codes are reserved for implementation-defined conditions. Within predefined classes, subclass codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined subconditions. All other subclass codes are reserved for implementation-defined subconditions. Figure 2–1 shows the coding scheme.

Figure 2–1   SQLSTATE Coding Scheme
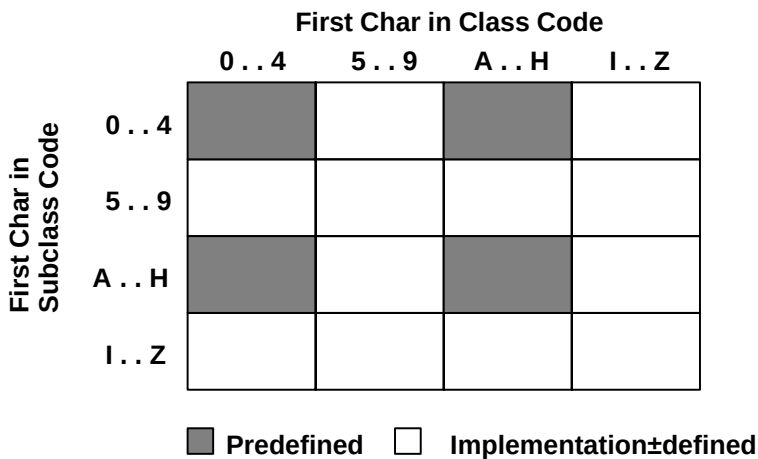
Table 2–1 shows the classes predefined by SQL92.

Table 2–1   Predefined Classes

Class	Condition
00	success completion
01	warning
02	no data
07	dynamic SQL error
08	connection exception
0A	feature not supported
21	cardinality violation
22	data exception
23	integrity constraint violation
24	invalid cursor state
25	invalid transaction state
26	invalid SQL statement name
27	triggered data change violation

Table 2–1   Predefined Classes

Class	Condition
28	invalid authorization specification
2A	direct SQL syntax error or access rule violation
2B	dependent privilege descriptors still exist
2C	invalid character set name
2D	invalid transaction termination
2E	invalid connection name
33	invalid SQL descriptor name
34	invalid cursor name
35	invalid condition number
37	dynamic SQL syntax error or access rule violation
3C	ambiguous cursor name
3D	invalid catalog name
3F	invalid schema name
40	transaction rollback
42	syntax error or access rule violation
44	with check option violation
HZ	remote database access

Note: The class code HZ is reserved for conditions defined in International Standard ISO/IEC DIS 9579-2, *Remote Database Access*.

Table 2–2 shows how SQLSTATE status codes and conditions are mapped to Oracle errors. Status codes in the range 60000 .. 99999 are implementation-defined.

Table 2–2   SQLSTATE Status Codes

Code	Condition	OracleError(s)
00000	successful completion	ORA-00000
01000	warning	
01001	cursor operation conflict	

Table 2–2 SQLSTATE Status Codes

Code	Condition	OracleError(s)
01002	disconnect error	
01003	null value eliminated in set function	
01004	string data-right truncation	
01005	insufficient item descriptor areas	
01006	privilege not revoked	
01007	privilege not granted	
01008	implicit zero-bit padding	
01009	search condition too long for info schema	
0100A	query expression too long for info schema	
02000	no data	ORA-01095 ORA-01403
07000	dynamic SQL error	
07001	using clause does not match parameter specs	
07002	using clause does not match target specs	
07003	cursor specification cannot be executed	
07004	using clause required for dynamic parameters	
07005	prepared statement not a cursor specification	
07006	restricted datatype attribute violation	
07007	using clause required for result components invalid descriptor count	
07008	invalid descriptor count	SQL-02126

Table 2–2   SQLSTATE Status Codes

Code	Condition	OracleError(s)
07009	invalid descriptor index	
08000	connection exception	
08001	SQL-client unable to establish SQL-connection	
08002	connection name is use	
08003	connection does not exist	SQL-02121
08004	SQL-server rejected SQL-connection	
08006	connection failure	
08007	transaction resolution unknown	
0A000	feature not supported	ORA-03000..03099
0A001	multiple server transactions	
21000	cardinality violation	ORA-01427 SQL-02112
22000	data exception	
22001	string data - right truncation	ORA-01406
22002	null value-no indicator parameter	SQL-02124
22003	numeric value out of range	ORA-01426
22005	error in assignment	
22007	invalid datetime format	
22008	datetime field overflow	ORA-01800..01899
22009	invalid time zone displacement value	
22011	substring error	
22012	division by zero	ORA-01476
22015	interval field overflow	

Table 2–2   SQLSTATE Status Codes

Code	Condition	OracleError(s)
22018	invalid character value for cast	
22019	invalid escape character	ORA-00911
22021	character not in repertoire	
22022	indicator overflow	ORA-01411
22023	invalid parameter value	ORA-01025 ORA-04000..04019
22024	unterminated C string	ORA-01479 ORA-01480
22025	invalid escape sequence	ORA-01424 ORA-01425
22026	string data-length mismatch	ORA-01401
22027	trim error	
23000	integrity constraint violation	ORA-02290..02299
24000	invalid cursor state	ORA-001002 ORA-001003 SQL-02114 SQL-02117
25000	invalid transaction state	SQL-02118
26000	invalid SQL statement name	
27000	triggered data change violation	
28000	invalid authorization specification	
2A000	direct SQL syntax error or access rule violation	
2B000	dependent privilege descriptors still exist	
2C000	invalid character set name	

Table 2–2   SQLSTATE Status Codes

Code	Condition	OracleError(s)
2D000	invalid transaction termination	
2E000	invalid connection name	
33000	invalid SQL descriptor name	
34000	invalid cursor name	
35000	invalid condition number	
37000	dynamic SQL syntax error or access rule violation	
3C000	ambiguous cursor name	
3D000	invalid catalog name	
3F000	invalid schema name	
40000	transaction rollback	ORA-02091 ORA-02092
40001	serialization failure	
40002	integrity constraint violation	
40003	statement completion unknown	
42000	syntax error or access rule violation	ORA-00022 ORA-00251 ORA-00900..00999 ORA-01031 ORA-01490..01493 ORA-01700..01799 ORA-01900..02099 ORA-02140..02289 ORA-02420..02424 ORA-02450..02499 ORA-03276..03299 ORA-04040..04059 ORA-04070..04099
44000	with check option violation	ORA-01402

Table 2–2   SQLSTATE Status Codes

Code	Condition	OracleError(s)
60000	system error	ORA-00370..00429 ORA-00600..00899 ORA-06430..06449 ORA-07200..07999 ORA-09700..09999
61000	multi-threaded server and detached process errors	ORA-00018..00035 ORA-00050..00068 ORA-02376..02399 ORA-04020..04039
62000	multi-threaded server and detached process errors	ORA-00100..00120 ORA-00440..00569
63000	Oracle*XA and two-task interface errors	ORA-00150..00159 ORA-02700..02899 ORA-03100..03199 ORA-06200..06249
64000	control file, database file, and redo file errors; archival and media recovery errors	ORA-00200..00369 ORA-01100..01250
65000	PL/SQL errors	ORA-06500..06599
66000	SQL*Net driver errors	ORA-06000..06149 ORA-06250..06429 ORA-06600..06999 ORA-12100..12299 ORA-12500..12599
67000	licensing errors	ORA-00430..00439
69000	SQL*Connect errors	ORA-00570..00599 ORA-07000..07199

Table 2–2   SQLSTATE Status Codes

Code	Condition	OracleError(s)
72000	SQL execute phase errors	ORA-00001 ORA-01000..01099 ORA-01400..01489 ORA-01495..01499 ORA-01500..01699 ORA-02400..02419 ORA-02425..02449 ORA-04060..04069 ORA-08000..08190 ORA-12000..12019 ORA-12300..12499 ORA-12700..21999
82100	out of memory (could not allocate)	SQL-02100
82101	inconsistent cursor cache (UCE/CUC mismatch)	SQL-02101
82102	inconsistent cursor cache (no CUC entry for UCE)	SQL-02102
82103	inconsistent cursor cache (out-or-range CUC ref)	SQL-02103
82104	inconsistent cursor cache (no CUC available)	SQL-02104
82105	inconsistent cursor cache (no CUC entry in cache)	SQL-02105
82106	inconsistent cursor cache (invalid cursor number)	SQL-02106
82107	program too old for runtime library; re-precompile	SQL-02107
82108	invalid descriptor passed to runtime library	SQL-02108
82109	inconsistent host cache (out-or-range SIT ref)	SQL-02109
82110	inconsistent host cache (invalid SQL type)	SQL-02110
82111	heap consistency error	SQL-02111

Table 2–2   SQLSTATE Status Codes

Code	Condition	OracleError(s)
82113	code generation internal consistency failed	SQL-02115
82114	reentrant code generator gave invalid context	SQL-02116
82117	invalid OPEN or PREPARE for this connection	SQL-02122
82118	application context not found	SQL-02123
HZ000	remote database access	

Using SQLSTATE

The following rules apply to using SQLSTATE with SQLCODE or the SQLCA when you precompile with the option setting MODE=ANSI. SQLSTATE must be declared inside a Declare Section; otherwise, it is ignored.

If you declare SQLSTATE

- Declaring SQLCODE is optional. If you declare SQLCODE inside the Declare Section, the Oracle Server returns status codes to SQLSTATE and SQLCODE after every SQL operation. However, if you declare SQLCODE outside the Declare Section, Oracle returns a status code only to SQLSTATE.

- Declaring the SQLCA is optional. If you declare the SQLCA, Oracle returns status codes to SQLSTATE and the SQLCA. In this case, to avoid compilation errors, do *not* declare SQLCODE.

If you do *not declare SQLSTATE*

- You must declare SQLCODE inside or outside the Declare Section. The Oracle Server returns a status code to SQLCODE after every SQL operation.

- Declaring the SQLCA is optional. If you declare the SQLCA, Oracle returns status codes to SQLCODE and the SQLCA.

You can learn the outcome of the most recent executable SQL statement by checking SQLSTATE explicitly with your own code or implicitly with the WHENEVER SQLERROR statement. Check SQLSTATE only after executable SQL statements and PL/SQL statements.

Declaring SQLCODE

When MODE={ANSI | ANSI14} and you have not declared SQLSTATE, you must declare a long integer variable named SQLCODE inside or outside the Declare Section. Upper case is required. An example follows:

```
/* Declare host and indicator variables */
EXEC SQL BEGIN DECLARE SECTION;
    ...
EXEC SQL END DECLARE SECTION;

/* Declare status variable */
    DCL  SQLCODE    FIXED BIN(31);
```

After every SQL operation, Oracle returns a status code to the SQLCODE variable currently in scope. The status code, which indicates the outcome of the SQL operation, can be any of the following numbers:

0	Means that Oracle executed the statement without detecting an error or exception.
> 0	Means that Oracle executed the statement but detected an exception. This occurs when Oracle cannot find a row that meets your WHERE-clause search condition or when a SELECT INTO or FETCH returns no rows.
	When MODE={ANSI	ANSI14	ANSI13}, +100 is returned to SQLCODE after an INSERT of no rows. This can happen when a subquery returns no rows to process.
< 0	Means that Oracle did not execute the statement because of a database, system, network, or application error. Such errors can be fatal. When they occur, the current transaction should, in most cases, be rolled back.
	Negative return codes correspond to error codes listed in the *Oracle7 Server Messages and Codes Manual*.

You can learn the outcome of the most recent SQL operation by checking SQLCODE explicitly with your own code or implicitly with the WHENEVER statement.

When you declare SQLCODE instead of the SQLCA in a particular precompilation unit, the precompiler allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA. If you declare the SQLCA *and* SQLCODE, Oracle returns the same status code to both after every SQL operation.

When MODE={ANSI13 | Oracle}, if you declare SQLCODE, it is not used.

Using the SQLCA

Oracle uses the SQLCA to store status information passed to your program at run time. The SQLCA always reflects the outcome of the most recent SQL operation. To determine that outcome, you can check variables in the SQLCA explicitly with your own PL/I code, or implicitly with the WHENEVER statement.

When MODE=ORACLE (the default) or MODE=ANSI13, you must declare the SQLCA by hardcoding it or by copying it into your program with the INCLUDE statement.

When MODE={ANSI | ANSI14}, declaring the SQLCA is optional. However, you must declare a status variable named SQLCODE. SQL92 specifies a similar status variable named SQLSTATE, which you can use with or without SQLCODE.

After executing a SQL statement, the Oracle Server returns a status code to the SQLSTATE variable currently in scope. The status code indicates whether the SQL statement executed successfully or raised an exception (error or warning condition). To promote *interoperability* (the ability of systems to exchange information easily), SQL92 predefines all the common SQL exceptions.

Unlike SQLCODE, which stores only error codes, SQLSTATE stores error and warning codes. Furthermore, the SQLSTATE reporting mechanism uses a standardized coding scheme. Thus, SQLSTATE is the preferred status variable. Under SQL92, SQLCODE is a "deprecated feature" retained only for compatibility with SQL89 and likely to be removed from future versions of the standard.

Declaring the SQLCA

To declare the SQLCA, copy it into your program with the INCLUDE statement

```
EXEC SQL INCLUDE SQLCA;
```
or hardcode it as shown below:

```
DCL 1 SQLCA,
     2 SQLCAID     CHAR(8)         INIT('SQLCA'),
     2 SQLCABC     FIXED BIN (31) INIT(136),
     2 SQLCODE     FIXED BIN (31),
     2 SQLERRM     CHAR (70) VAR,
     2 SQLERRP     CHAR (8)         INIT('SQLERRP'),
     2 SQLERRD (6)  FIXED BIN (31),
```

```
        2 SQLWARN,
          3 SQLWARN0   CHAR (1),
          3 SQLWARN1   CHAR (1),
          3 SQLWARN2   CHAR (1),
          3 SQLWARN3   CHAR (1),
          3 SQLWARN4   CHAR (1),
          3 SQLWARN5   CHAR (1),
          3 SQLWARN6   CHAR (1),
          3 SQLWARN7   CHAR (1),
        2 SQLEXT,      CHAR (8)       INIT('SQLEXT');
```

Not declaring the SQLCA when MODE=Oracle results in compile time warnings, and causes runtime errors.

Your Pro*PL/I program can have more than one SQLCA. The SQLCA should not be INCLUDEd outside of a procedure block, since the elements in it are not declared with the STATIC storage class. Oracle returns information to the SQLCA that is in the scope of the SQL statement that caused the error or warning condition. The name of this structure must be SQLCA, since it is referred to by precompiler-generated code.

What's in the SQLCA?

The SQLCA contains runtime information about the execution of SQL statements, such as Oracle error codes, warning flags, event information, rows-processed count, and diagnostics.

Key Components of Error Reporting

The key components of Pro*PL/I error reporting depend on several fields in the SQLCA.

Status Codes

Every executable SQL statement returns a status code in the SQLCA variable SQLCODE, which you can check implicitly with WHENEVER SQLERROR, or explicitly with your own PL/I code.

Warning Flags

Warning flags are returned in the SQLCA variables SQLWARN0 through SQLWARN7, which you can check implicitly with WHENEVER SQLWARNING, or explicitly with your own PL/I code. These warning flags are useful for detecting runtime conditions not considered errors by Oracle.

Rows-processed Count

The number of rows processed by the most recently executed SQL statement is recorded in the SQLCA variable SQLERRD(3). For repeated FETCHes on an OPEN cursor, SQLERRD(3) keeps a running total of the number of rows fetched.

Parse Error Offset

Before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. If Oracle finds an error, an offset is stored in the SQLCA variable SQLERRD(5), which you can check explicitly. The offset specifies the character position in the SQL statement at which the parse error begins. The first character occupies position zero. For example, if the offset is 9, the parse error begins at the 10th character.

If your SQL statement does not cause a parse error, Oracle sets SQLERRD(5) to zero. Oracle also sets SQLERRD(5) to zero if a parse error begins at the first character, which occupies position zero. So, check SQLERRD(5) only if SQLCODE is negative, which means that an error has occurred.

Error Message Text

The error code and message for Oracle errors are available in the SQLCA variable SQLERRM. For example, you might place the following statements in an error-handling routine:

```
/* Handle SQL execution errors. */
PUT EDIT(SQLCA.SQLERRM)(A(70));

EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE
...
```

At most, the first 70 characters of message text are stored. For messages longer than 70 characters, you must call the SQLGLM function.

Getting the Full Text of Error Messages

The SQLCA can accommodate error messages of up to 70 characters in length. To get the full text of longer (or nested) error messages, you need the SQLGLM procedure. If connected to Oracle, you can call SQLGLM using the syntax

```
CALL SQLGLM (MSG_BUF, BUF_SIZE, MSG_LENGTH);
```
where:

MSG_BUF

Is the buffer in which you want Oracle to store the error message. Oracle blank-pads to the end of this buffer.

BUF_SIZE

Is an integer variable that specifies the maximum length of MSG_BUF in bytes.

MSG_LENGTH

Is an integer variable in which Oracle stores the actual length of the error message.

The maximum length of an Oracle error message is 196 characters including the error code, nested messages, and message inserts such as table and column names. The maximum length of an error message returned by SQLGLM depends on the value you specify for BUF_SIZE. In the following example, you use SQLGLM to get an error message of up to 200 characters in length:

```
TEST: PROC OPTIONS(MAIN);

/* Declare variables for the function call. */
DCL MSG_BUF CHAR(200),  /* buffer for message text */
BUF_SIZE FIXED BIN(31) INIT(200), /* size in bytes */
MSG_LEN FIXED BIN(31);  /* length of message text  */

WHENEVER SQLERROR GOTO ERROR_PRINT;

...

ERROR_PRINT:
/* Get full text of error message. */
CALL SQLGLM(MSG_BUF, BUF_SIZE, MSG_LEN);
/* Print the text. */
PUT SKIP EDIT (MSG_BUF) (A(MSG_LEN));
...
```

Notice that SQLGLM is called only when a SQL error has occurred. Always make sure SQLCA.SQLCODE is negative *before* calling SQLGLM. If you call SQLGLM when SQLCODE is zero, you get the message text associated with a prior SQL statement.

Using the WHENEVER Statement

By default, the Pro*PL/I Precompiler ignores Oracle error and warning conditions and continues processing if possible. To do automatic condition checking and error handling, you need the WHENEVER statement.

With the WHENEVER statement you can specify actions to be taken when Oracle detects an error, warning condition, or "not found" condition. These actions include continuing with the next statement, calling a procedure, branching to a labeled statement, or stopping.

You code the WHENEVER statement using the following syntax:

```
EXEC SQL WHENEVER <condition> <action>
```
You can have Oracle automatically check the SQLCA for any of the following *conditions*:

- SQLWARNING

- SQLERROR

- NOT FOUND

When Oracle detects one of the preceding conditions, you can have your program take any of the following *actions*:

- CONTINUE

- DO procedure_call

- GOTO statement_label

- STOP

When using the WHENEVER ... DO statement, the usual rules for entering and exiting a procedure apply. However, passing parameters to the subroutine is *not* allowed. Furthermore, the subroutine must *not* return a value.

In the following example, you use WHENEVER SQLERROR DO statements to handle specific errors:

```
...
EXEC SQL WHENEVER SQLERROR DO CALL INSERT_ERROR;
...
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
VALUES(:MY_EMPNO, :MY_ENAME, :MY_DEPTNO);
...
INSERT_ERROR: PROCEDURE;
/* test for "duplicate key value" Oracle error */
IF (SQLCA.SQLCODE = -1) THEN DO;
...
/* test for "value too large" Oracle error */
ELSE IF (SQLCA.SQLCODE = -1401) DO;
...
/* etc. */
```

```
END;
END INSERT_ERROR
```

Notice how the procedure checks variables in the SQLCA to determine a course of action.

For more information about the WHENEVER conditions and actions, see Chapter 7 of the *Programmer's Guide to the Oracle Precompilers*.

Scope of WHENEVER

Because WHENEVER is a declarative statement, its scope is positional, not logical. That is, it tests all executable SQL statements that physically (not logically) follow it in your program. So, code the WHENEVER statement before the first executable SQL statement you want to test.

A WHENEVER statement stays in effect until superseded by another WHENEVER statement checking for the same condition.

Helpful Hint

You might want to place WHENEVER statements at the beginning of each block that contains SQL statements. That way, SQL statements in one block will not reference WHENEVER actions in another block, causing errors at compile or run time.

Caution

Careless use of WHENEVER can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets NOT FOUND because no rows meet the search condition:

```
/* Improper use of WHENEVER */
EXEC SQL WHENEVER NOT FOUND GOTO DO_DELETE;
DO J = 1 TO N_FETCH;
EXEC SQL FETCH EMP_CURSOR INTO :MY_ENAME, :MY_SAL;
...
END;
DO_DELETE:
EXEC SQL DELETE FROM EMP WHERE EMPNO = :MY_EMPNO;
```

In the next example, you handle the NOT FOUND condition properly by resetting the GOTO target:

```
/* Proper use of WHENEVER */
EXEC SQL WHENEVER NOT FOUND GOTO DO_DELETE;
```

```
DO J = 1 TO N_FETCH;
EXEC SQL FETCH EMP_CURSOR INTO :MY_ENAME, :MY_SAL;
...
END;
DO_DELETE:
EXEC SQL WHENEVER NOT FOUND GOTO WHAT_NEXT;
EXEC SQL DELETE FROM EMP WHERE EMPNO = :MY_EMPNO;
...
WHAT_NEXT:
...
```

Also, make sure all SQL statements governed by a WHENEVER ... GOTO statement can branch to the GOTO label. The following code results in a compile time error because the UPDATE statement in PROC2 is not within the scope of LABEL_A in PROC1:

```
PROC1: PROC();
...
EXEC SQL WHENEVER SQLERROR GOTO LABEL_A;
EXEC SQL DELETE FROM EMP WHERE DEPTNO = :DEPT_NUMBER;
...
LABEL_A:
PUT SKIP LIST ('Error occurred');
END PROC1;
PROC2: PROC();
...
EXEC SQL UPDATE EMP SET SAL = SAL * 1.20
WHERE JOB = 'PROGRAMMER';
...
END PROC2;
```

Using the ORACA

The SQLCA handles standard SQL communications. The ORACA is a similar data structure copied or hardcoded into your program to handle Oracle-specific communications. When you need more runtime information than the SQLCA provides, use the ORACA.

Besides helping you to diagnose problems, the ORACA lets you monitor your program's use of Oracle resources, such as the SQL Statement Executor and the *cursor cache*, an area of memory reserved for cursor management.

Declaring the ORACA

To declare the ORACA, you can copy it into your main program with the INCLUDE statement, as follows:

```
/* Copy in the Oracle Communications Area (ORACA). */
EXEC SQL INCLUDE ORACA;
```
Alternatively, you can hardcode it as follows:

```
DCL 1 ORACA
      2 ORACAID   CHAR        (8)   INIT ('ORACA')
      2 ORACABC   FIXED BIN   (31)  INIT (176)
      2 ORACCHF   FIXED BIN   (31)  INIT (0)
      2 ORADBGF   FIXED BIN   (31)  INIT (0)
      2 ORAHCHF   FIXED BIN   (31)  INIT (0)
      2 ORASTXTF  FIXED BIN   (31)  INIT (0)
      2 ORASTXT   CHAR (70)   VAR   INIT ('')
      2 ORASFNM   CHAR (70)   VAR   INIT ('')
      2 ORASLNR   FIXED BIN   (31)  INIT (0)
      2 ORAHOC    FIXED BIN   (31)  INIT (0)
      2 ORAMOC    FIXED BIN   (31)  INIT (0)
      2 ORACOC    FIXED BIN   (31)  INIT (0)
      2 ORANOR    FIXED BIN   (31)  INIT (0)
      2 ORANPR    FIXED BIN   (31)  INIT (0)
      2 ORANEX    FIXED BIN   (31)  INIT (0)
```

Enabling the ORACA

To enable the ORACA, you must set the ORACA precompiler option to YES, either on the command line with

```
ORACA=YES
```
or inline with

```
/* Enable the ORACA. */
EXEC Oracle OPTION (ORACA=YES);
```

Then, you must choose appropriate runtime options by setting flags in the ORACA.

Enabling the ORACA is optional because it adds to runtime overhead. The default setting is ORACA=NO.

What's in the ORACA?

The ORACA contains option settings, system statistics, and extended diagnostics. The listing above shows all the variables in the ORACA.

For a full description of the ORACA, its fields, and the values the fields can store, see Chapter 7 of the *Programmer's Guide to the Oracle Precompilers.*

3

Running the Pro*PL/I Precompiler

This chapter provides the basic information you need to invoke the Pro*PL/I Precompiler.

Precompiler Command

To run the Pro*PL/I Precompiler, you issue the following command:

```
propli
```

The location of the precompiler differs from system to system. The system or database administrator defines environment variables, logicals, or aliases, or uses other operating-system-specific means to make the Pro*PL/I executable accessible.

The INAME= option specifies the source file to be precompiled. For example, the command

```
propli INAME=test_propli
```

precompiles the file *test_propli.ppl* in the current directory, since the precompiler assumes that the filename extension is *.ppl.* You need not use the use a file extension when specifying INAME unless the extension is nonstandard. The INAME options does not have to be the first option on the command line, but if it is, you can omit the option specification. So, the command

```
propli myfile
```

is equivalent to

```
propli INAME=myfile
```

> **Note:** Option names, and option values that do not name specific operating-system objects, such as filenames, are not case-sensitive. In the examples in this guide, option names are written in upper case, and option values are usually in lower case. Filenames, including the name of the Pro*PL/I Precompiler executable itself always follow the case conventions used by your operating system.

Precompiler Options

Many useful options are available at precompile time. They let you control how resources are used, how errors are reported, how input and output are formatted, and how cursors are managed. To specify a precompiler option, you use the following syntax:

```
option_name=value
```
See Chapter 11 of the *Programmer's Guide to the Oracle Precompilers* for a list of precompiler options. The list gives the purpose, syntax, default value, and usage notes for each option.

A handy reference to the precompiler options is available online. To see the online display, enter the precompiler command with no arguments at your operating system prompt. The display gives the name, syntax, default value, and purpose of each option. Options marked with an asterisk (*) can be specified inline as well as on the command line.

The value of an option is a string literal, which can represent text or numeric values. For example, for the option

```
... INAME=my_test
```

the value is a string literal that specifies a filename. But for the option

```
... MAXOPENCURSORS=20
```

the value is numeric.

Some options take Boolean values, and you can represent these with the strings *yes* or *no.* For example, the option

```
... SELECT_ERROR=YES
```

The option value is always separated from the option name by an equals sign, with no whitespace around the equals sign.

Default Values

Many of the options have default values. The default value of an option is determined by:

- a value built in to the precompiler

- a value set in the Pro*PL/I *system configuration file*

- a value set in a Pro*PL/I *user configuration file*

- a value set in an inline specification

For example the option MAXOPENCURSORS specifies the maximum number of cached open cursors. The built-in precompiler default value for this option is 10. However, if MAXOPENCURSORS=32 is specified in the system configuration file, the default now becomes 32. The user configuration file could set it to yet another value, which then overrides the system configuration value. Then, if this option is set on the command line, the new command-line value takes precedence over the precompiler default, the system configuration file specification, and the user configuration file specification. Finally, an inline specification takes precedence over

all preceding defaults. For more information see the section "Configuration Files" on page "Configuration Files" on page 3-4.

Some options, such as USERID, do not have a precompiler default. For more information, see the *Programmer's Guide to the Oracle Precompilers*.

Determining Current Values

You can interactively determine the current value for one or more options by using a question mark on the command line. For example, if you issue the command

```
propli ?
```

the complete set of options, along with their current values, is printed to your terminal. (On a UNIX system running the C shell, escape the '?' with a backslash.) In this case, the values are those built into the precompiler, overridden by any values in the system configuration file. But if you issue the command

```
propli CONFIG=my_config_file.cfg ?
```

and there is a file named *my_config_file.cfg* in the current director, the options are listed. Values in the user configuration file supply missing values, and supersede values built-in to the Pro*PL/I precompiler, or values specified in the system configuration file.

You can also determine the current value of a single option, by simply specifying that option name, followed by =?. For example

```
propli MAXOPENCURSORS=?
```

prints the current default value for the MAXOPENCURSORS option.

Case Sensitivity

In general, you can use either uppercase or lowercase for command-line option names and values However, if your operating system is case sensitive, like UNIX, you must specify filename values, including the name of the Pro*PL/I executable, using the correct combination of upper and lowercase letters.

Configuration Files

A configuration file is a text file that contains precompiler options. Each record (line) in the file contains one option, with its associated value or values. For example, a configuration file might contain
the lines

```
FIPS=YES
MODE=ANSI
CODE-ANSI_C
```

to set defaults for the FIPS, MODE, and CODE options.

There is a single system configuration file for each system. The name of the system configuration file is *pccpli.cfg.*. The location of the file is operating-system-specific. On most UNIX systems, the file specification is usually located in *$ORACLE_ HOME/propli/pccpli.cfg.*

Each Pro*PL/I user can have one or more user configuration files. The name of the configuration file must be specified using the CONFIG= command-line option.

> **Note:** You cannot nest configuration files. This means that CONFIG= is not a valid option inside a configuration file.

Scope of Options

The options specified when you precompile a given Pro*PL/I source file affect only the code generated from that file; they have no effect on other modules that may be linked in to form the final program. For example, if you specify MAXLITERAL for file A but not for file B, SQL statements in file A run with the specified MAXLITERAL value, but SQL statements in file B run with the default value.

There is one exception to this rule: the MAXOPENCURSORS value that is in effect when a connection to a database is made stays in effect for the life of that connection.

An option setting stays in effect until the end-of-file unless you respecify the option. For more information on other options, see the *Programmer's Guide to the Oracle Precompilers.*

DBMS

Purpose
Specifies whether Oracle follows the semantic and syntactic rules of Oracle Version 6, Oracle7, or the native version of Oracle (that is, the version to which the application is connected).

Syntax
DBMS=NATIVE | V6 | V7

Default

NATIVE

Usage Notes

Can be entered only on the command line.

The DBMS option lets you control the version-specific behavior of Oracle. When DBMS=NATIVE (the default), Oracle follows the semantic and syntactic rules of the database version to which the application is connected.

When DBMS=V6, or DBMS=V7, Oracle follows the rules of Oracle Version 6 or Oracle7, respectively. A summary of the differences between DBMS=V6 and DBMS=V7 follows:

- When DBMS=V6, Oracle treats string literals like variable-length character values. However, when DBMS=V7, Oracle treats string literals like fixed-length character values, and CHAR semantics change slightly to comply with the current SQL standard.

- When DBMS=V6, Oracle treats local CHAR variables in a PL/SQL block like variable-length character values. When DBMS=V7, however, Oracle treats the CHAR variables like SQL standard, fixed-length character values.

- When DBMS=V6, Oracle treats the return value of the function USER like a variable-length character value. However, when DBMS=V7, Oracle treats the return value of USER like a SQL standard, fixed-length character value.

- When DBMS=V6, if you process a multirow query that calls a SQL group function such as AVG or COUNT, the function is called at OPEN time. When DBMS=V7, however, the function is called at FETCH time. At OPEN time or FETCH time, if the function call fails, Oracle issues an error message immediately. Thus, the DBMS value affects error reporting slightly.

- When DBMS=V6, no error is returned if a SELECT or FETCH statement selects a null, and there is no indicator variable associated with the output host variable. When DBMS=V7, SELECTing or FETCHing a null column or expression into a host variable that has no associated indicator variable causes an error (SQLSTATE is "22002"; SQLCODE is -01405).

- When DBMS=V6, a DESCRIBE operation of a fixed-length string (in Dynamic SQL Method 4) returns datatype code 1. When DBMS=V7, the DESCRIBE operation returns datatype code 96.

- When DBMS=V6, PCTINCREASE is allowed for rollback segments. When DBMS=V7, PCTINCREASE is not allowed for rollback segments.

- When DBMS=V6, illegal MAXEXTENTS storage parameters are allowed. They are not allowed when DBMS=V7.

- When DBMS=V6, constraints (except NOT NULL) are not enabled. When DBMS=V7, all Oracle7 constraints are enabled.

If you precompile using the DBMS=V6 option, and connect to an Oracle7 database, then a Data Definition Language statement such as

```
CREATE TABLE T1 (COL1 CHAR(10))
```

creates the table using the VARCHAR2 (variable-length) datatype, as if the CREATE TABLE statement had been

```
CREATE TABLE T1 (COL1 VARCHAR2(10))
```

MODE

Purpose
Specifies whether your program observes Oracle practices or complies with the current ANSI/ISO SQL standards.

Syntax
MODE=ANSI | ANSI13 | ANSI14 |ISO | ORACLE

Default
ORACLE

Usage Notes
Can be entered only on the command line.

ISO is a synonym for ANSI.

When MODE=ORACLE (the default), your embedded SQL program observes Oracle practices. When MODE=ANSI, your program complies *fully* with the ANSI standard, and the following changes go into effect:

- CHAR column values, USER pseudocolumn values, character host values, and quoted literals are treated like ANSI fixed-length character strings. And, ANSI-compliant blank-padding semantics are used when you assign, compare, INSERT, UPDATE, SELECT, or FETCH such values.

- Issuing a COMMIT or ROLLBACK closes all explicit cursors.

- You cannot OPEN an already open cursor or CLOSE an already closed cursor. (When MODE=ORACLE, you can reOPEN an open cursor to avoid reparsing.)

- You cannot SELECT or FETCH nulls into a host variable not associated with an indicator variable.

- If you declare SQLSTATE, then you must declare SQLSTATE as *DCL SQLSTATE CHAR(5);.*

- Declaring the SQLCA is optional. You need not include the SQLCA.

- No error message is issued if Oracle assigns a truncated column value to an output host variable.

- The "no data found" Oracle error code returned to SQLCODE becomes +100 instead of +1403. The error message text does not change.

Entering Options

All the precompiler options can be entered on the command line; some can also be entered inline (within the source file).

```
... [option_name=value] [option_name=value] ...
```

Separate each option with one or more spaces.

For example, you might enter

```
... ERRORS=yes LTYPE=long MODE=ANSI13
```

You enter options inline by coding EXEC Oracle statements, using the following syntax:

```
EXEC Oracle OPTION (option_name=value);
```

For example, you might code

```
EXEC Oracle OPTION (AREASIZE=4);
```

An option entered inline overrides the same option entered on the command line. The scope of an inline option is positional, not logical. ("Positional" means that it takes effect for all statements that follow it in the source file, regardless of the flow of control of the program logic.)

An inline option setting stays in effect until overridden by another EXEC Oracle OPTION directive that sets the same option name.

Special PL/I Options

The Pro*PL/I Precompiler supports LMARGIN and RMARGIN controls that allow you to specify the left and right margins of the input source file. Using these controls makes it possible to support the card image format required by some compilers.

By default, the left and right margins on all IBM systems are set to 2 and 72 respectively. On all non-IBM systems, the default setting for the left margin is 1, and the default setting for the right margin is the record length of the input file. To change these defaults (to 5 and 75, for example), specify LMARGIN and RMARGIN on the command line as follows:

```
LMARGIN=5 RMARGIN=75
```

Doing Conditional Precompilations

Conditional precompilation includes (or excludes) sections of code in your Pro*PL/I program based on certain conditions. For example, you may want to include one section of code when precompiling under the VM operating system and another section when precompiling under VMS. Conditional precompilation lets you write programs that can run in different environments. For more information, see Chapter 11 of the *Programmer's Guide to the Oracle Precompilers*.

Doing Separate Precompilations

With the Pro*PL/I Precompiler you can precompile several program files separately, then link them into one executable program. This allows modular programming—required when the functional components of a program are written and debugged by different programmers.

Restrictions

All references to an explicit cursor must be in the same program file. You cannot perform operations on a cursor that was DECLAREd in a different module. See the *Programmer's Guide to the Oracle Precompilers*, Chapter 4, for more information about cursors.

Also, any program file that contains SQL statements must have a SQLCA that is in the scope of the local SQL statements.

Compiling and Linking

To produce an executable program, you must compile the PL/I output files produced by the precompiler, then link the resulting object modules with the Oracle runtime library, SQLLIB.

Compiling and linking are system dependent. For instructions, see the Oracle installation or user's guide for your system.

4

Sample Programs

This chapter provides several embedded SQL programs to guide you in writing your own. These programs illustrate the key concepts and features of Pro*PL/I programming and demonstrate techniques that let you take full advantage of SQL's power and flexibility.

Sample Programs

Each sample program in this chapter is available online. The table below shows the usual filenames of the sample programs. However, the exact filename and storage location of the online files can be system dependent. Check the Oracle installation or user's guide for your system.

File Name	Demonstrates...
SAMPLE1.PPL	a simple query
SAMPLE2.PPL	cursor operations
SAMPLE3.PPL	array fetches
SAMPLE4.PPL	datatype equivalencing
SAMPLE5.PPL	a SQL*Forms user exit
SAMPLE6.PPL	dynamic SQL Method 1
SAMPLE7.PPL	dynamic SQL Method 2
SAMPLE8.PPL	dynamic SQL Method 3
SAMPLE9.PPL	calling a stored procedure

Sample Program 1: Login and Query

```
/***************************************************************
This program connects to Oracle, prompts the user for an employee
number, queries the database for the employee's name, salary,
and commission, then displays the result. It continues until
the user enters a 0 for the employee number.
***************************************************************/

QUERYEX: PROCEDURE OPTIONS(MAIN);

EXEC SQL BEGIN DECLARE SECTION;
    DCL USERNAME     CHAR(10) VARYING,
        PASSWORD     CHAR(10) VARYING,
        EMP_NUMBER   BIN FIXED(31),
        EMP_NAME     CHAR(10) VARYING,
        SALARY       DECIMAL FLOAT(6),
        COMMISSION   DECIMAL FLOAT(6);
EXEC SQL END DECLARE SECTION;

    DCL TOTAL        BIN FIXED(31);

EXEC SQL INCLUDE SQLCA;

/* log in to Oracle */

USERNAME = 'SCOTT';
PASSWORD = 'TIGER';

EXEC SQL WHENEVER SQLERROR DO CALL SQLERR;

EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
PUT SKIP EDIT('Connected to Oracle as user: ',USERNAME)(A, A);

TOTAL = 0;

LOOP:    DO WHILE (1=1);

    PUT SKIP(2) LIST('Enter employee number (0 to exit): ');
    GET LIST(EMP_NUMBER);
    IF (EMP_NUMBER = 0)
        THEN LEAVE LOOP;

    EXEC SQL WHENEVER NOT FOUND GOTO NOTFND;
```

```
        EXEC SQL SELECT ENAME, SAL, NVL(COMM,0)
            INTO :EMP_NAME, :SALARY, :COMMISSION
            FROM EMP
            WHERE EMPNO = :EMP_NUMBER;

        PUT SKIP(2) LIST('Employee Name  Salary  Commission');
        PUT SKIP    LIST('-------------  ------  ----------');
        PUT SKIP    EDIT(EMP_NAME, SALARY, COMMISSION)
            (A(13), X(2), F(7,2), X, F(9,2));

        TOTAL = TOTAL + 1;
        GOTO LOOP;

        NOTFND:
            PUT SKIP LIST('Not a valid employee number - try again.');

        END;

        PUT SKIP(2) LIST('Total number queried was ', TOTAL, '.');
        PUT SKIP(2) LIST('Have a good day.');

        EXEC SQL COMMIT WORK RELEASE;                /* log off Oracle */
        STOP;

SQLERR: PROCEDURE;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    PUT SKIP(2) LIST('Oracle error detected:');
    PUT SKIP(2) LIST(SQLCA.SQLERRM);

    EXEC SQL ROLLBACK WORK RELEASE;
    STOP;

END SQLERR;

END QUERYEX;
```

Sample Program 2: Using a Cursor

```
/***************************************************************
This program logs on to Oracle, declares and opens a cursor,
fetches the names, salaries, and commissions of all salespeople,
displays the results, then closes the cursor.
```

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

        CURSDEM: PROCEDURE OPTIONS(MAIN);

        EXEC SQL BEGIN DECLARE SECTION;
            DCL USERNAME     CHAR(10) VARYING,
                PASSWORD     CHAR(10) VARYING,
                EMP_NAME     CHAR(10) VARYING,
                SALARY       DECIMAL FLOAT(6),
                COMMISSION   DECIMAL FLOAT(6);
        EXEC SQL END DECLARE SECTION;

        EXEC SQL INCLUDE SQLCA;

        /* log in to Oracle */

        USERNAME = 'SCOTT';
        PASSWORD = 'TIGER';

        EXEC SQL WHENEVER SQLERROR DO CALL SQLERR;

        EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
        PUT SKIP EDIT('Connected to Oracle as user: ', USERNAME)(A, A);

        /* Establish the cursor. */

        EXEC SQL DECLARE salespeople CURSOR FOR
            SELECT ENAME, SAL, COMM
            FROM EMP
            WHERE JOB LIKE 'SALES%';

        EXEC SQL OPEN salespeople;

        PUT SKIP(2) LIST('Employee Name  Salary  Commission');
        PUT SKIP    LIST('-------------  ------  ----------');

        LOOP:   DO WHILE (1 = 1);

            EXEC SQL WHENEVER NOT FOUND GOTO NOTFND;

            EXEC SQL FETCH salespeople
                INTO :EMP_NAME, :SALARY, :COMMISSION;

            PUT SKIP EDIT(EMP_NAME, SALARY, COMMISSION)
                (A(13), X(2), F(7,2), X(1), F(9,2));
```

```
                        GOTO LOOP;

                        NOTFND: LEAVE LOOP;

                        END;

                        EXEC SQL CLOSE salespeople;
                        PUT SKIP(2) LIST('Have a good day.');

                        EXEC SQL COMMIT WORK RELEASE;               /* log off Oracle */
                        STOP;

                SQLERR: PROCEDURE;

                        EXEC SQL WHENEVER SQLERROR CONTINUE;

                        PUT SKIP(2) LIST('Oracle error detected:');
                        PUT SKIP(2) LIST(SQLCA.SQLERRM);

                        EXEC SQL ROLLBACK WORK RELEASE;
                        STOP;

                END SQLERR;

                END CURSDEM;
```

Sample Program 3: Fetching in Batches

```
        /************************************************************
        This program logs on to Oracle, declares and opens a cursor,
        fetches in batches using arrays, and prints the results using
        the function print_rows().
        ************************************************************/

        ARRDEM: PROCEDURE OPTIONS(MAIN);

        EXEC SQL BEGIN DECLARE SECTION;
            DCL USERNAME       CHAR(10) VARYING,
                PASSWORD       CHAR(10) VARYING,
                EMP_NAME(5)    CHAR(10) VARYING,
                EMP_NUMBER(5) BIN FIXED(31),
                SALARY(5)      DECIMAL FLOAT(6);
        EXEC SQL END DECLARE SECTION;
```

```
DCL NUM_RET BIN FIXED(31);

EXEC SQL INCLUDE SQLCA;

/* log in to Oracle */

USERNAME = 'SCOTT';
PASSWORD = 'TIGER';

EXEC SQL WHENEVER SQLERROR DO CALL SQLERR;

EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
PUT SKIP EDIT('Connected to Oracle as user: ', USERNAME)(A, A);

/* Establish the cursor. */

EXEC SQL DECLARE c1 CURSOR FOR
    SELECT EMPNO, ENAME, SAL FROM EMP;

EXEC SQL OPEN c1;

NUM_RET = 0;    /* initialize number of rows returned */

LOOP:   DO WHILE(1 = 1); /* terminate when NOT FOUND is raised */

    EXEC SQL WHENEVER NOT FOUND GOTO NOTFND;
    EXEC SQL FETCH c1 INTO :EMP_NUMBER, :EMP_NAME, :SALARY;

    CALL PRINT_ROWS(SQLCA.SQLERRD(3) - NUM_RET);
    NUM_RET = SQLCA.SQLERRD(3);

    END;

NOTFND:

    /* Print remaining rows from last fetch, if any. */
    IF ((SQLCA.SQLERRD(3) - NUM_RET) >> 0) THEN
    CALL PRINT_ROWS(SQLCA.SQLERRD(3) - NUM_RET);

    EXEC SQL CLOSE c1;
    PUT SKIP(2) LIST('Have a good day.');

    EXEC SQL COMMIT WORK RELEASE;    /* log off Oracle */
    STOP;
```

```
                PRINT_ROWS: PROCEDURE(N);

                    DCL (N,I) BIN FIXED (31);

                    PUT SKIP;
                    PUT SKIP(2) LIST('Employee Number  Employee Name  Salary');
                    PUT SKIP    LIST('---------------  -------------  ------');

                    DO I = 1 TO N BY 1;
                    PUT SKIP EDIT(EMP_NUMBER(I), EMP_NAME(I), SALARY(I))
                        (F(4), X(13), A(13), X(2), F(7,2));
                    END;

                END PRINT_ROWS;

                SQLERR: PROCEDURE;

                    EXEC SQL WHENEVER SQLERROR CONTINUE;

                    PUT SKIP    LIST('Oracle error detected:');
                    PUT SKIP(2) LIST(SQLCA.SQLERRM);

                    EXEC SQL ROLLBACK RELEASE;
                    STOP;

                END SQLERR;

                END ARRDEM;
```

Sample Program 4: Datatype Equivalencing

```
                /***********************************************************
                This program features an in-depth example of the use of
                Datatype Equivalencing. After logging in, it creates a new
                table in the SCOTT account, IMAGE, and simulates placement of
                bitmap images of employees in it. Later, when an employee
                number is entered, his/her bitmap is selected back out of the
                IMAGE table, and pseudo-displayed on the terminal screen.
                ***********************************************************/

                DTYEQV: PROCEDURE OPTIONS(MAIN);

                EXEC SQL BEGIN DECLARE SECTION;
```

```
    DCL USERNAME    CHAR(10) VARYING,
        PASSWORD    CHAR(10) VARYING,

        EMP_NUMBER  BIN FIXED(31),
        EMP_NAME    CHAR(10) VARYING,
        SALARY      DECIMAL FLOAT(6),
        COMMISSION  DECIMAL FLOAT(6);

    DCL BUFFER      CHAR(8192);
        EXEC SQL VAR BUFFER IS LONG RAW;
    DCL SELECTION   BIN FIXED(31);

EXEC SQL END DECLARE SECTION;

    DCL REPLY       CHAR(10) VARYING;
EXEC SQL INCLUDE SQLCA;

/* log in to Oracle */

USERNAME = 'SCOTT';
PASSWORD = 'TIGER';

EXEC SQL WHENEVER SQLERROR DO CALL SQLERR;

EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
PUT SKIP EDIT('Connected to Oracle as user: ', USERNAME)(A, A);

PUT SKIP(2)
    LIST('Program is about to drop the IMAGE table - OK [y/N]? ');
GET EDIT(REPLY)(A(1));
IF ((REPLY ^= 'Y') & (REPLY ^= 'y')) THEN CALL SIGNOFF;

EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC SQL DROP TABLE IMAGE;

IF (SQLCA.SQLCODE = 0) THEN
    PUT SKIP(2)
        LIST('Table IMAGE has been dropped - creating new table.');
ELSE IF (SQLCA.SQLCODE = -942) THEN
    PUT SKIP(2)
        LIST('Table IMAGE does not exist - creating new table.');
ELSE CALL SQLERR;

EXEC SQL WHENEVER SQLERROR DO CALL SQLERR;
```

```
EXEC SQL CREATE TABLE IMAGE
    (empno NUMBER(4) NOT NULL, bitmap LONG RAW);

EXEC SQL DECLARE EMPCUR CURSOR FOR
    SELECT EMPNO, ENAME FROM EMP;

EXEC SQL OPEN EMPCUR;

PUT SKIP(2)
    LIST('INSERTing bitmaps into IMAGE for all employees ...');
PUT SKIP;

GLOOP:  DO WHILE (1 = 1);

    EXEC SQL WHENEVER NOT FOUND GOTO GNOTFND;

    EXEC SQL FETCH EMPCUR INTO :EMP_NUMBER, :EMP_NAME;

    PUT SKIP EDIT('Employee ', EMP_NAME)(A, A(10));
    CALL GETIMG(EMP_NUMBER, BUFFER);
    EXEC SQL INSERT INTO IMAGE VALUES (:EMP_NUMBER, :BUFFER);
    PUT EDIT(' is done!')(A);

    GOTO GLOOP;

    GNOTFND: LEAVE GLOOP;

    END;

EXEC SQL CLOSE EMPCUR;
EXEC SQL COMMIT WORK;

PUT SKIP(2)
    LIST('Done INSERTing bitmaps. Next, lets display some.');

SLOOP:  DO WHILE (1 = 1);

    PUT SKIP(2) LIST('Enter employee number (0 to exit): ');
    GET LIST(SELECTION);
    IF (SELECTION = 0) THEN CALL SIGNOFF;

    EXEC SQL WHENEVER NOT FOUND GOTO SNOTFND;

    EXEC SQL SELECT EMP.EMPNO, ENAME, SAL, NVL(COMM,0), BITMAP
```

```
              INTO :EMP_NUMBER, :EMP_NAME, :SALARY, :COMMISSION, :BUFFER
              FROM EMP, IMAGE
              WHERE EMP.EMPNO = :SELECTION AND EMP.EMPNO = IMAGE.EMPNO;

         CALL SHWIMG(BUFFER);

         PUT SKIP(2) EDIT('Employee ', EMP_NAME)(A, A(10));
         PUT EDIT(' has salary ', SALARY)(A, F(7,2));
         PUT EDIT(' and commission ', COMMISSION)(A, F(7,2));

         GOTO SLOOP;

         SNOTFND:
             PUT SKIP LIST('Not a valid employee number - try again.');

         END;

STOP;

GETIMG:  PROCEDURE(ENUM, BUF);

    DCL ENUM        BIN FIXED(31),
        BUF         CHAR(8192);
    DCL I           BIN FIXED(31);

DO I=1 TO 8192 BY 1;
    SUBSTR(BUF,I,1) = '*';
    IF (MOD(I,256) = 0) THEN PUT EDIT('.')(A);
END;

END GETIMG;

SHWIMG:  PROCEDURE(BUF);

    DCL BUF         CHAR(8192);
    DCL I           BIN FIXED(31);

PUT SKIP;
DO I=1 TO 10 BY 1;
    PUT SKIP LIST('                  ************************');
END;

END SHWIMG;

SIGNOFF: PROCEDURE;
```

```
                    PUT SKIP(2) LIST('Have a good day.');

                    EXEC SQL COMMIT WORK RELEASE;
                    STOP;

                END SIGNOFF;

                SQLERR: PROCEDURE;

                    EXEC SQL WHENEVER SQLERROR CONTINUE;

                    PUT SKIP(2) LIST('Oracle error detected:');
                    PUT SKIP(2) LIST(SQLCA.SQLERRM);

                    EXEC SQL ROLLBACK WORK RELEASE;
                    STOP;

                END SQLERR;

                END DTYEQV;
```

Sample Program 5: A SQL*Forms User Exit

This user exit concatenates form fields. To call the user exit from a SQL*Forms
trigger, use the syntax

```
user_exit('CONCAT field1, field2, ..., result_field');
```

where *user_exit* is a packaged procedure supplied with SQL*Forms and CONCAT is
the name of the user exit. A sample form named CONCAT invokes the user exit.
For more information about SQL*Forms user exits, see Chapter 10 in the
Programmer's Guide to the Oracle Precompilers.

Note:

The sample code listed is for a SQL*Forms user exit and is not intended to be
compiled in the same manner as the other sample programs listed in this chapter.

```
CONCAT:

PROCEDURE(CMD,CMDLEN,MSG,MSGLEN,QUERY) RETURNS(FIXED BINARY(31));

EXEC SQL BEGIN DECLARE SECTION;
    DCL FIELD    CHARACTER(81) VARYING,
        VALUE    CHARACTER(81) VARYING,
```

```
           OUTVALUE CHARACTER(241) VARYING;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR GOTO ER_EXIT;

/*  parameter declarations */
DCL CMD     CHAR(80),
    MSG     CHAR(80),
    CMDLEN  FIXED BIN(31),
    MSGLEN  FIXED BIN(31),
    QUERY   FIXED BIN(31),

/*  local declarations */
    CMDCNT  FIXED BIN(31),
    I       FIXED BIN(31),

/*  local copy of cmd */
    LOCCMD  CHAR(80),

/*  dynamically built error message to be
    returned to SQL*Forms */
    ERRMSG  CHAR(80),
    ERRLEN  FIXED BIN(31);


/*  strip off "concat" keyword in the command string */
LOCCMD = SUBSTR(CMD, 8, CMDLEN-7);
OUTVALUE = '';

I = INDEX(LOCCMD,',');
DO WHILE(I >> 0);            /* found field delimited by (,) */
    FIELD = SUBSTR(LOCCMD, 1, I-1); /* field name minus (,) */
    EXEC IAF GET :FIELD INTO :VALUE;
    OUTVALUE = OUTVALUE || VALUE;
/*  skip over (,) and following blank space */
    CMDCNT = I + 2;
/*  take previous field off command line */
    LOCCMD = SUBSTR(LOCCMD, CMDCNT, CMDLEN-I);
    I = INDEX(LOCCMD, ',');
END;
I = INDEX(LOCCMD, ' ');
/*  get last field concat */
FIELD = SUBSTR(LOCCMD, 1, I-1);
EXEC IAF PUT :FIELD VALUES (:OUTVALUE);
```

```
                RETURN(SQL_IAPXIT_SUCCESS);

                ER_EXIT:
                ERRMSG = 'CONCAT: ' || SQLCA.SQLERRM;
                ERRLEN = 80;
                CALL SQLIEM(ADDR(ERRMSG), ADDR(ERRLEN));
                RETURN(SQL_IAPXIT_FAILURE);

                END CONCAT;
```

Sample Program 6: Dynamic SQL Method 1

Dynamic SQL Method 1 executes a SQL statement contained in a host character string that is constructed at runtime. The statement must not be a SELECT and must not contain input or output host variables. Method 1 has only one step:

```
EXEC SQL EXECUTE IMMEDIATE {:string_var | 'string_literal'};
```

This program demonstrates the use of dynamic SQL Method 1 to create a table, insert a row, commit the insert, and drop the table. It accesses Oracle through the SCOTT/TIGER account. It does not require user input or existing database tables. The program displays the SQL statements before their execution.

The program is available online in the file Sample6.

```
DYN1DEM: PROCEDURE OPTIONS(MAIN);

/*  Include the SQL Communications Area, a structure
    through which Oracle makes runtime status information
    such as error codes, warning flags, and diagnostic text
    available to the host program. */

EXEC SQL INCLUDE SQLCA;

/*  Include the Oracle Communications Area, a structure
    through which Oracle makes additional runtime status
    information available to the program. */

EXEC SQL INCLUDE ORACA;

/*  The ORACA=YES option must be specified to enable use
    of the ORACA. */
```

```
EXEC Oracle OPTION (ORACA=YES);

/*  Specifying the RELEASE_CURSOR=YES option instructs
    Pro*PL/I to release resources associated with embedded
    SQL statements after they are executed.
    This ensures that Oracle does not keep parse locks
    on tables after DML operations, so that subsequent DDL
    operations on those tables do not result in a
    "resource locked" Oracle run-time error. */


EXEC Oracle OPTION (RELEASE_CURSOR=YES);

/*  All host variables used in embedded SQL must appear
    in the DECLARE SECTION. */

EXEC SQL BEGIN DECLARE SECTION;
DCL USERNAME     CHAR(10) VARYING,
    PASSWORD     CHAR(10) VARYING,
    SQLSTMT      CHAR(80) VARYING;
EXEC SQL END DECLARE SECTION;

/*  Branch to label 'SQL_ERR' if an Oracle error occurs. */

EXEC SQL WHENEVER SQLERROR GOTO SQL_ERR;

/*  Save text of current SQL statement in the ORACA if
    an error occurs. */

ORACA.ORASTXTF = 1;

/*  Connect to Oracle. */
USERNAME = 'SCOTT';
PASSWORD = 'TIGER';
EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
PUT SKIP LIST('CONNECTED TO Oracle.');

/*  Execute a string literal to create the table. */
PUT SKIP LIST('CREATE TABLE DYN1 (COL1 CHAR(4))');
EXEC SQL EXECUTE IMMEDIATE 'CREATE TABLE DYN1 (COL1 CHAR(4))';

/*  Assign a SQL statement to the character string
    SQLSTMT. */
SQLSTMT = 'INSERT INTO DYN1 VALUES (''TEST'')';
PUT SKIP LIST(SQLSTMT);
```

```
/*  Execute sqlstmt to insert a row. This usage is
    "dynamic" because the SQL statement is a string
    variable whose contents the program may determine
    at runtime. */

EXEC SQL EXECUTE IMMEDIATE :SQLSTMT;

/*  Commit the insert. */
EXEC SQL COMMIT WORK;

/*  Change sqlstmt and execute it to drop the table. */
SQLSTMT = 'DROP TABLE DYN1';
PUT SKIP LIST(SQLSTMT);
EXEC SQL EXECUTE IMMEDIATE :SQLSTMT;

/*  Commit any outstanding changes and disconnect from
    Oracle. */

EXEC SQL COMMIT RELEASE;
PUT SKIP LIST('DISCONNECTED FROM Oracle.');
STOP;

SQL_ERR:

/*  Oracle error handler. Print diagnostic text
    containing error message, current SQL statement,
    line number and file name of error. */

PUT SKIP(2) LIST(SQLCA.SQLERRM);
PUT SKIP EDIT('IN "', ORACA.ORASTXT, '..."')
    (A, A(LENGTH(ORACA.ORASTXT)), A);
PUT SKIP EDIT('ON LINE ', ORACA.ORASLNR, ' OF ', ORACA.ORASFNM)
    (A, F(3), A, A(LENGTH(ORACA.ORASFNM)));

/*  Disable Oracle error checking to avoid an infinite
    loop should another error occur within this routine. */

EXEC SQL WHENEVER SQLERROR CONTINUE;

/*  Roll back any outstanding changes and disconnect
    from Oracle. */

EXEC SQL ROLLBACK RELEASE;
```

```
       END DYN1DEM;
```

Sample Program 7: Dynamic SQL Method 2

Dynamic SQL Method 2 processes a SQL statement contained in a host character string constructed at runtime. The statement must not be a SELECT but may contain input host variables. Method 2 has two steps:

```
EXEC SQL PREPARE statement_name FROM
    { :string_var | 'string_literal' };

EXEC SQL EXECUTE statement_name
    [USING :invar1[, :invar2...]];
```

This program demonstrates the use of dynamic SQL Method 2 to insert two rows into the EMP table and then delete them. It accesses Oracle through the SCOTT/TIGER account and requires the EMP table. It does not require user input. The program displays the SQL statements before their execution.

This program is available online in the file Sample7.

```
DYN2DEM:  PROCEDURE OPTIONS(MAIN);

/*  Include the SQL Communications Area, a structure
    through which Oracle makes runtime status information
    such as error codes, warning flags, and
    diagnostic text available to the program. */

EXEC SQL INCLUDE SQLCA;

/*  All host variables used in embedded SQL must
    appear in the DECLARE SECTION. */

EXEC SQL BEGIN DECLARE SECTION;
    DCL USERNAME CHAR(10) VARYING,
        PASSWORD CHAR(10) VARYING,
        SQLSTMT  CHAR(80) VARYING,
        EMPNO    FIXED DECIMAL(4) INIT(1234),
        DEPTNO1  FIXED DECIMAL(2) INIT(97),
        DEPTNO2  FIXED DECIMAL(2) INIT(99);
EXEC SQL END DECLARE SECTION;

/*  Branch to label 'sqlerror' if an Oracle error
    occurs. */
```

```
EXEC SQL WHENEVER SQLERROR GOTO SQL_ERR;

/*  Connect to Oracle. */
USERNAME = 'SCOTT';
PASSWORD = 'TIGER';
EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
PUT SKIP LIST('CONNECTED TO Oracle.');

/*  Assign a SQL statement to the character string
    SQLSTMT. Note that the statement contains two
    host variable placeholders, V1 and V2, for which
    actual input host variables must be supplied at the
    EXECUTE (see below). */

SQLSTMT = 'INSERT INTO EMP (EMPNO, DEPTNO) VALUES(:V1, :V2)';

/*  Display the SQL statement and the values to be used for
    its input host variables. */

PUT SKIP LIST(SQLSTMT);
PUT SKIP LIST('  V1 = ', EMPNO, ', V2 = ', DEPTNO1);

/*  The PREPARE statement associates a statement name
    with a string containing a SQL statement.
    The statement name is a SQL identifier, not a host
    variable, and therefore does not appear in the
    DECLARE SECTION.
    A single statement name may be PREPAREd more than
    once, optionally FROM a different string variable. */

EXEC SQL PREPARE S FROM :SQLSTMT;

/*  The EXECUTE statement performs a PREPAREd SQL
    statement USING the specified input host variables,
    which are substituted positionally for placeholders
    in the PREPAREd statement. For each occurrence of
    a placeholder in the statement there must be a
    variable in the USING clause, i.e. if a placeholder
    occurs multiple times in the statement then the
    corresponding variable must appear multiple times
    in the USING clause. The USING clause may be
    omitted only if the statement contains no placeholders.
    A single PREPAREd statement may be EXECUTEd more
    than once, optionally USING different
    input host variables. */
```

```
EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO1;

/*  Increment empno and display new input host
    variables. */

EMPNO = EMPNO + 1;
PUT SKIP LIST('  V1 = ', EMPNO, ', V2 = ', DEPTNO2);

/*  ReEXECUTE S to insert the new value of EMPNO and a
    different input host variable, DEPTNO2. A rePREPARE
    is not necessary. */

EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO2;

/*  Assign a new value to sqlstmt. */

SQLSTMT = 'DELETE FROM EMP WHERE DEPTNO = :V1 OR DEPTNO = :V2';

/*  Display the new SQL statement and the values to
    be used for its current input host variables. */

PUT SKIP LIST(SQLSTMT);
PUT SKIP LIST('  V1 = ', DEPTNO1, ', V2 = ', DEPTNO2);

/*  RePREPARE S FROM the new sqlstmt. */

EXEC SQL PREPARE S FROM :SQLSTMT;

/*  EXECUTE the new S to delete the two rows previously
    inserted. */

EXEC SQL EXECUTE S USING :DEPTNO1, :DEPTNO2;

/*  Commit any outstanding changes and disconnect from
    Oracle. */

EXEC SQL COMMIT RELEASE;
PUT SKIP LIST('Disconnected from Oracle.');
STOP;

SQL_ERR:

/*  Oracle error handler. */
```

```
              PUT SKIP(2) LIST(SQLCA.SQLERRM);

          /*  Disable Oracle error checking to avoid an
              infinite loop should another error occur
              within this routine. */

          EXEC SQL WHENEVER SQLERROR CONTINUE;

          /*  Roll back any outstanding changes and disconnect
              from Oracle. */

          EXEC SQL ROLLBACK RELEASE;

          END DYN2DEM;
```

Sample Program 8: Dynamic SQL Method 3

Dynamic SQL Method 3 processes a SQL statement contained in a host character string constructed at runtime. The statement may be a SELECT, and may contain input host variables but not output host variables (the INTO clause is on the FETCH instead). This Dynamic SQL Method 3 example processes a query, and uses the following five steps:

```
EXEC SQL PREPARE statement_name
    FROM { :string_var | 'string_literal' };

EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;

EXEC SQL OPEN cursor_name [USING :invar1[,:invar2...]];

EXEC SQL FETCH cursor_name INTO :outvar1[,:outvar2...];

EXEC SQL CLOSE cursor_name;
```

This program demonstrates the use of dynamic SQL Method 3 to retrieve all the names from the EMP table. It accesses Oracle through the SCOTT/TIGER account and requires the EMP table. It does not require user input. The program displays the query and its results

The program is available online in the file Sample8.

```
DYN3DEM:  PROCEDURE OPTIONS(MAIN);

/*  Include the SQL Communications Area, a structure
    through which Oracle makes runtime status
```

```
        information such as error codes, warning flags, and
        diagnostic text available to the program. */

EXEC SQL INCLUDE SQLCA;

/*  All host variables used in embedded SQL must appear
    in the DECLARE SECTION. */

EXEC SQL BEGIN DECLARE SECTION;
    DCL USERNAME CHAR(10) VARYING,
        PASSWORD CHAR(10) VARYING,
        SQLSTMT  CHAR(80) VARYING,
        ENAME    CHAR(10) VARYING,
        DEPTNO   FIXED DECIMAL(2) INIT(10);
EXEC SQL END DECLARE SECTION;


/*  Branch to label SQL_ERR: if an Oracle error
    occurs. */

EXEC SQL WHENEVER SQLERROR GOTO SQL_ERR;

/*  Connect to Oracle. */

USERNAME = 'SCOTT';
PASSWORD = 'TIGER';
EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
PUT SKIP LIST('CONNECTED TO Oracle.');

/*  Assign a SQL query to the character string SQLSTMT.
    Note that the statement contains one host variable
    placeholder, V1, for which an actual input
    host variable must be supplied at the OPEN
    (see below). */

SQLSTMT = 'SELECT ENAME FROM EMP WHERE DEPTNO = :V1';

/*  Display the SQL statement and the value to be used
    for its current input host variable. */

PUT SKIP LIST(SQLSTMT);
PUT SKIP LIST('  V1 = ', DEPTNO);

/*  The PREPARE statement associates a statement
    name with a string containing an SQL statement.
```

```
                    The statement name is a SQL identifier, not a host
                    variable, and therefore does not appear in the
                    DECLARE SECTION. A single statement name may be
                    PREPAREd more than once, optionally FROM a
                    different string variable. */

             EXEC SQL PREPARE S FROM :SQLSTMT;

             /*   The DECLARE statement associates a cursor with a
                    PREPAREd statement. The cursor name, like the
                    statement name, does not appear in the DECLARE
                    SECTION. A single cursor name may not be DECLAREd
                    more than once. */


             EXEC SQL DECLARE C CURSOR FOR S;

             /*   The OPEN statement evaluates the active set of the
                    PREPAREd query USING the specified input host
                    variables, which are substituted positionally for
                    placeholders in the PREPAREd query. For each
                    occurrence of a placeholder in the statement there
                    must be a variable in the USING clause. That is, if
                    a placeholder occurs multiple times in the statement
                    then the corresponding variable must appear multiple
                    times in the USING clause. The USING clause may be
                    omitted only if the statement contains no placeholders.
                    OPEN places the cursor at the first row of the active
                    set in preparation for a FETCH.

                    A single DECLAREd cursor may be OPENed more than
                    once, optionally USING different input host variables.
             */

             EXEC SQL OPEN C USING :DEPTNO;

             /*   Branch to label 'notfound' when all rows have been
                    retrieved. */

             EXEC SQL WHENEVER NOT FOUND GOTO N_FND;

             /*   Loop until NOT FOUND condition is raised. */

             DO WHILE (1 = 1);
```

```
        /*  The FETCH statement places the SELECT list of the
            current row into the variables specified by the INTO
            clause then advances the cursor to the next row.
            If there are more SELECT list fields than output
            host variables, the extra fields will not be returned.
            More output host variables than SELECT list fields
            will result in an Oracle error. */

        EXEC SQL FETCH C INTO :ENAME;
        PUT SKIP LIST(ENAME);
    END;

    N_FND:

    /*  Print the cumulative number of rows processed by the
        current SQL statement. */

    PUT SKIP LIST('QUERY RETURNED ', SQLCA.SQLERRD(3), ' ROW(S).');

    /*  The CLOSE statement releases resources associated
        with the cursor. */

    EXEC SQL CLOSE C;

    /*  Commit any outstanding changes and disconnect from
        Oracle. */

    EXEC SQL COMMIT RELEASE;
    PUT SKIP LIST('DISCONNECTED FROM Oracle.');
    STOP;

    SQL_ERR:

    /*  Oracle error handler. Print diagnostic text
        containing error message. */

    PUT SKIP(2) LIST(SQLCA.SQLERRM);

    /*  Disable Oracle error checking to avoid an infinite
        loop should another error occur within this routine. */

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /*  Release resources associated with the cursor. */
```

```
                EXEC SQL CLOSE C;

        /*  Roll back any outstanding changes and disconnect
            from Oracle. */

        EXEC SQL ROLLBACK RELEASE;

        END DYN3DEM;
```

Sample Program 9: Calling a Stored procedure

Before trying the sample program, you must create a PL/SQL package named
calldemo. You do that by running a script named CALLDEMO.SQL, which is
supplied with Pro*C and shown below. The script can be found in the Pro*C demo
library.

```
CREATE OR REPLACE PACKAGE calldemo AS

    TYPE char_array IS TABLE OF VARCHAR2(20)
        INDEX BY BINARY_INTEGER;
    TYPE num_array IS TABLE OF FLOAT
        INDEX BY BINARY_INTEGER;

    PROCEDURE get_employees(
      dept_number IN      number,    -- department to query
      batch_size  IN      INTEGER,   -- rows at a time
      found        IN OUT INTEGER,   -- rows actually returned
      done_fetch  OUT      INTEGER,  -- all done flag
      emp_name    OUT      char_array,
      job         OUT      char_array,
      sal         OUT      num_array);

END calldemo;
/

CREATE OR REPLACE PACKAGE BODY calldemo AS

    CURSOR get_emp (dept_number IN number) IS
        SELECT ename, job, sal FROM emp
            WHERE deptno = dept_number;

    -- Procedure "get_employees" fetches a batch of employee
    -- rows (batch size is determined by the client/caller
    -- of the procedure).  It can be called from other
```

```
-- stored procedures or client application programs.
-- The procedure opens the cursor if it is not
-- already open, fetches a batch of rows, and
-- returns the number of rows actually retrieved. At
-- end of fetch, the procedure closes the cursor.

PROCEDURE get_employees(
    dept_number IN      number,
    batch_size  IN      INTEGER,
    found       IN OUT  INTEGER,
    done_fetch  OUT     INTEGER,
    emp_name    OUT     char_array,
    job         OUT     char_array,
    sal         OUT     num_array) IS

BEGIN
    IF NOT get_emp%ISOPEN THEN        -- open the cursor if
        OPEN get_emp(dept_number);  -- not already open
    END IF;

    -- Fetch up to "batch_size" rows into PL/SQL table,
    -- tallying rows found as they are retrieved. When all
    -- rows have been fetched, close the cursor and exit
    -- the loop, returning only the last set of rows found.

    done_fetch := 0;  -- set the done flag FALSE
    found := 0;

    FOR i IN 1..batch_size LOOP
        FETCH get_emp INTO emp_name(i), job(i), sal(i);
        IF get_emp%NOTFOUND THEN      -- if no row was found
            CLOSE get_emp;
            done_fetch := 1;    -- indicate all done
            EXIT;
        ELSE
            found := found + 1;  -- count row
        END IF;
    END LOOP;
    END;
END;
/
/*
 * This program connects to Oracle, prompts the user for a
 * department number, uses a stored procedure to fetch Oracle
 * data into PL/SQL tables, returns the data in host arrays, then
```

```
              * displays the name, job title, and salary of each employee in
              * the department.
              * For this example to work, the package CALLDEMO must be in
              * the SCOTT schema, or SCOTT must have execute privileges on the
              * package.
              */

          EXEC SQL BEGIN DECLARE SECTION;
             DCL USERNAME         STATIC   CHAR(10) VARYING,
                 PASSWORD         STATIC   CHAR(10) VARYING,

                 TABLE_SIZE       STATIC   BIN FIXED(31),
                 DEPT_NUMBER      STATIC   BIN FIXED(31),
                 DONE_FLAG        STATIC   BIN FIXED(31),
                 NUM_RET          STATIC   BIN FIXED(31),
                 EMP_NAME(10)     STATIC   CHAR(20) VARYING,
                 JOB(10)          STATIC   CHAR(20) VARYING,
                 SALARY(10)       STATIC   DECIMAL FLOAT(6);
          EXEC SQL END DECLARE SECTION;

          SAMP9: PROCEDURE OPTIONS(MAIN);

              /* connect to Oracle */

              EXEC SQL INCLUDE SQLCA;

              USERNAME = 'SCOTT';
              PASSWORD = 'TIGER';

              EXEC SQL WHENEVER SQLERROR DO CALL SQLERR;

              EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD;
              PUT SKIP EDIT
                  ('Connected to Oracle as user: ', USERNAME)(A, A);

              PUT SKIP(2) LIST('Enter the department number: ');
              GET LIST (DEPT_NUMBER);
              PUT SKIP;
              TABLE_SIZE = 2;
              DONE_FLAG = 0;

          CLOOP:  DO WHILE (1 = 1);
                      EXEC SQL EXECUTE
                          BEGIN
                              CALLDEMO.GET_EMPLOYEES (
```

```
                          :DEPT_NUMBER, :TABLE_SIZE, :NUM_RET,
                          :DONE_FLAG, :EMP_NAME, :JOB, :SALARY);
                   END;
               END-EXEC;
               CALL PRINT_ROWS(NUM_RET);
               IF (DONE_FLAG ^= 0) THEN
                   CALL SIGNOFF;
               ELSE
                   GOTO CLOOP;
         END;
STOP;

PRINT_ROWS:  PROCEDURE(N);
    DCL  N   BIN FIXED(31),
         I   BIN FIXED(31);


    IF N = 0 THEN DO;
        PUT SKIP(2) LIST('No rows retrieved.');
    END;
    ELSE DO;
        PUT SKIP(2) EDIT('Got', N, ' rows.') (A, F(3));
        PUT SKIP(2) LIST
            ('Employee name      Job                Salary');
        PUT SKIP LIST
            ('---------------------------------------------');
        DO I = 1 TO N;
    PUT SKIP EDIT(EMP_NAME(I)) (A(20));
            PUT EDIT    (JOB(I))    (A(20));
            PUT EDIT    (SALARY(I))  (F(7,2));
        END;
    END;
END PRINT_ROWS;

SIGNOFF: PROCEDURE;
    PUT SKIP(2) LIST('Have a good day.');
    EXEC SQL COMMIT WORK RELEASE;
    STOP;
END SIGNOFF;

SQLERR: PROCEDURE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    PUT SKIP(2) LIST('Oracle error detected:');
    PUT SKIP(2) LIST(SQLCA.SQLERRM);
    EXEC SQL ROLLBACK WORK RELEASE;
```

```
             STOP;
        END SQLERR;

        END SAMP9;
```

5

Implementing Dynamic SQL Method 4

This chapter provides the information you need to implement Dynamic SQL Method 4 in your Pro*PL/I application. You learn the following things:

- what the SQL descriptor area is used for
- how to declare descriptors
- elements of the PL/I select and bind descriptors
- how to initialize and use each element of a descriptor
- how to write code for Dynamic Method 4

 Note: For a discussion of dynamic SQL Methods 1, 2, and 3, and an overview of Method 4, see Chapter 9 of the *Programmer's Guide to the Oracle Precompilers.*

Meeting the Special Requirements of Method 4

Before looking into the requirements of Method 4, you should feel comfortable with the terms *select-list item* and *placeholder*. Select-list items are the columns or expressions following the keyword SELECT in a query. For example, the following dynamic query contains three select-list items:

```
'SELECT ename, job, sal + comm FROM emp WHERE deptno = 20'
```

Placeholders are dummy bind variables that hold places in a SQL statement for actual bind variables. You do not declare placeholders, and can name them anything you like.

Placeholders for bind variables are most often used in the SET, VALUES, and WHERE clauses. For example, the following dynamic SQL statements each contain two placeholders:

```
'INSERT INTO emp (empno, deptno) VALUES (:E, :D)'
'DELETE FROM dept WHERE deptno = :NUM AND loc = :LOC'
```

Placeholders, such as bind variables, cannot reference table or column names.

What Makes Method 4 Special?

Unlike Methods 1, 2, and 3, dynamic SQL Method 4 lets your program

- accept or build dynamic SQL statements that contain an unknown number of select-list items or placeholders, and

- take explicit control over datatype conversion

To add this flexibility to your program, you must provide additional information to the Oracle runtime library.

What Information Does Oracle Need?

The Pro*PL/I Precompiler generates calls to Oracle for all executable dynamic SQL statements. If a dynamic SQL statement contains no select-list items or placeholders, Oracle needs no additional information to execute the statement. The following DELETE statement falls into this category:

```
/* Dynamic SQL statement. */
STMT = 'DELETE FROM emp WHERE deptno = 30';
```

However, most dynamic SQL statements contain select-list items or placeholders, as does the following UPDATE statement:

```
/* Dynamic SQL statement with placeholders. */
STMT = 'UPDATE emp SET comm = :C WHERE empno = :E';
```

To execute a dynamic SQL statement that contains placeholders for bind variables or select-list items, Oracle needs information about the program variables that hold the input (bind) values, and that will hold the FETCHed values when a query is executed. The information needed by Oracle is

- the number of bind variables and select-list items

- the length of each bind variable and item

- the datatype of each bind variable and item

- the address of the each bind variable and program variable that will hold a received select-list item

Where Is the Information Stored?

All the information Oracle needs about select-list items or placeholders for bind variables, except their actual values, is stored in a program data structure called the SQL Descriptor Area (SQLDA).

Descriptions of select-list items are stored in a *select* descriptor, and descriptions of bind variables are stored in a *bind* descriptor.

The values of select-list items are stored in output variables; the values of bind variables are stored in input variables. You store the addresses of these variables in a select or bind SQLDA so that Oracle knows where to write output values and read input values.

How do values get stored in these variables? Output values are FETCHed using a cursor, and input values are typically filled in by the program, often from information entered interactively by the user.

How is the Information Obtained?

DESCRIBE helps you provide the information Oracle needs by storing descriptions of select-list items or placeholders in a SQLDA.

You use the DESCRIBE statement to help obtain the information Oracle needs.

The DESCRIBE SELECT LIST statement examines each select-list item and determines its name, datatype, constraints, length, scale, and precision. It then stores this information in the select SQLDA, and in program variables pointed to by

fields in the SQLDA. The total number of select-list items is also stored in the SQLDA by the DESCRIBE statement.

The DESCRIBE BIND VARIABLES statement obtains the number of placeholders in the SQL statement, and the names and lengths of each placeholder. The program must then fill in the datatype and length of the associated bind variables in the SQLDA, and obtain the bind variable values, which are stored in the program variables pointed to by fields in the SQLDA.

See the section "The Basic Steps" later in this chapter for a complete description of the steps that you perform to declare, allocate, and use the SQLDA.

The SQLDA

This section describes the SQL Descriptor Area in detail. You learn what elements the descriptor structure contains, how they should be initialized, and how they are used in your program.

Introducing the PL/I SQLDA

The SQLDA is a PL/I structure that contains two top-level elements and an array of substructures. Each substructure contains information about a single input or output variable. You declare a separate SQLDA major structure for the select-list items, and for the bind (or input) variables. These are called the *select descriptor* and the *bind descriptor.*

All SQLDA elements that hold an address are declared as FIXED BINARY (31). This datatype, rather than the more natural PL/I POINTER type, is used to achieve compatibility among different implementations of PL/I. You initialize these elements using the **SQLADR** procedure. The syntax of this procedure call is

```
CALL SQLADR(YOUR_BUFFER_ADDRESS, SQLDA_ELEMENT_ADDRESS);
```
as shown in the following example:

```
DCL SELECT_DATA_VALUE CHARACTER (10);
DCL SQLADR EXTERNAL ENTRY(PTR VALUE, PTR VALUE);
...
CALL SQLADR(ADDR(SELECT_DATA_VALUE),
ADDR(SELDSC.SQLDSC(1).SQLDV));
```

In this example, the address of the buffer SELECT_DATA_VALUE is stored in the SQLDV element in the first substructure of the array SQLDSC.

Note: Under IBM operating systems (MVS and VM/CMS), there is an alternate form of **SQLADR** called **SQ3ADR**. With **SQ3ADR**, the arguments are passed without using the PL/I **ADDR** built-in function. See the section "Sample 9: Dynamic SQL Method 4 Program" later in this chapter to see how to use **SQ3ADR**.

The SQLDVLN and SQLDVTYP elements contain the length and the datatype code for the select-list item or the bind variable. For a select descriptor, these elements are set when the SQL DESCRIBE statement is executed. You may reset them before actually fetching the values. For more information, see the later section "The SQLDA Variables." For a bind descriptor, you must set the length and datatype.

Declaring a SQLDA

To declare a SQLDA, copy it into your program with the statement

```
EXEC SQL INCLUDE DESCRIPTOR_NAME;
```

where "DESCRIPTOR_NAME" is the name of the file containing the text of the descriptor declaration. Or hardcode it as shown in Figure 5–1.

Note that the example below shows a SQLDA named SELDSC. It is a select descriptor. A bind descriptor is identical.
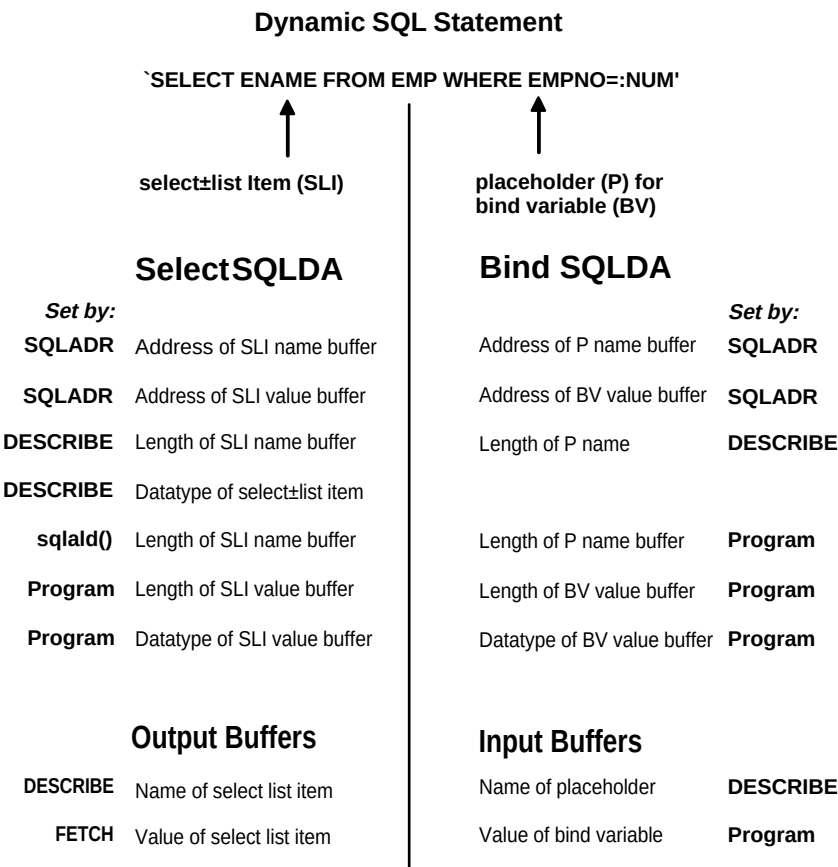
Figure 5–1   The SQL Descriptor Area

```
DCL  1 SELDSC,
       2 SQLDNUM            FIXED BIN (31),
       2 SQLDFND            FIXED BIN (31),
       2 SQLDSC (N)
         3 SQLDV            FIXED BIN (31),
         3 SQLDFMT          FIXED BIN (31),
         3 SQLDVLN          FIXED BIN (31),
         3 SQLDFMTL         FIXED BIN (15),
         3 SQLDVTYP         FIXED BIN (15),
         3 SQLDI            FIXED BIN (31),
         3 SQLDH_VNAME      FIXED BIN (31),
         3 SQLDH_MAX_VNAMEL FIXED BIN (15),
         3 SQLDH_CUR_VNAMEL FIXED BIN (15),
         3 SQLDI_VNAME      FIXED BIN (31),
         3 SQLDI_MAX_VNAMEL FIXED BIN (15),
         3 SQLDI_CUR_VNAMEL FIXED BIN (15),
         3 SQLDFCLP         FIXED BIN (31),
         3 SQLDFCRCP        FIXED BIN (31),
```

In the examples used in this section, the structures are named SELDSC for the select SQLDA, and BNDDSC for the bind SQLDA. These structures are identical, except for their names.

Figure 5–2 shows whether variables are set by calls to SQLADR, DESCRIBE commands, FETCH commands, or program assignments.

Figure 5–2   How Variables Are Set

Dynamic SQL Statement

`SELECT ENAME FROM EMP WHERE EMPNO=:NUM'

select±list Item (SLI)	placeholder (P) for bind variable (BV)

	Select SQLDA	**Bind SQLDA**	
Set by:			***Set by:***
SQLADR	Address of SLI name buffer	Address of P name buffer	**SQLADR**
SQLADR	Address of SLI value buffer	Address of BV value buffer	**SQLADR**
DESCRIBE	Length of SLI name buffer	Length of P name	**DESCRIBE**
DESCRIBE	Datatype of select±list item		
sqlald()	Length of SLI name buffer	Length of P name buffer	**Program**
Program	Length of SLI value buffer	Length of BV value buffer	**Program**
Program	Datatype of SLI value buffer	Datatype of BV value buffer	**Program**

	Output Buffers	**Input Buffers**	
DESCRIBE	Name of select list item	Name of placeholder	**DESCRIBE**
FETCH	Value of select list item	Value of bind variable	**Program**

Multiple SQLDAs

If your program has more than one active dynamic SQL statement, each statement must have its own SQLDA(s). You can declare any number of SQLDAs with different names. For example, you might declare three select SQLDAs named SEL1, SEL2, and SEL3, so that you can FETCH from three concurrently OPEN cursors. (However, non-concurrent cursors can reuse SQLDAs.)

The SQLDA Variables

This section explains the purpose and use of each element in the SQLDA structure.

{SELDSC | BNDDSC}

The name of the structure. The name is arbitrary; however, the examples in this manual use BNDDSC for a bind descriptor and SELDSC for a select descriptor. The names of the structure elements are not arbitrary; they should not be changed.

SQLDNUM

Contains the number of bind variables or select-list items that can be DESCRIBEd. Before issuing a DESCRIBE statement, your program must set this variable to the maximum size of the SQLDSC substructure array. After the DESCRIBE, SQLDNUM must be reset to the actual number of variables described, which is contained in SQLDFND.

SQLDFND

Set when Oracle performs a DESCRIBE statement. It contains the number of bind variables or select-list items DESCRIBEd. If after a DESCRIBE, SQLDFND is less than 0, there were more bind variables or select-list items than specified in SQLDNUM. In this case, SQLDFND is set to the negative of the number of variables DESCRIBEd. For example, if SQLDNUM is 10, and SQLDFND contains the value -11 after a DESCRIBE, 11 select-list items or bind variables were found. If this happens, you cannot process the statement.

SQLDSC(N)

Defines an array of N substructures. In a select descriptor, each substructure holds information about a select-list item, and the addresses of the buffers that will hold the data retrieved from the item. In a bind descriptor, each substructure holds information about a bind variable, and the address of the data buffer that contains the value of the bind variable.

You must set N before compiling your program. It should be large enough to handle the maximum number of expected bind variables (for the bind descriptor) or the maximum number of expected select-list items (for the select descriptor).

SQLDV

Holds the address of the buffer that contains the value for this bind variable, or that will hold the data retrieved from the select-list item.

Your program must place the address in this element using the **SQLADR** procedure. Set this element in a select descriptor before doing a FETCH. In a bind descriptor, set it before doing the OPEN.

SQLDFMT

Holds the address of a format conversion string, used only with decimal datatypes (FIXED DECIMAL(P,S) or FIXED DECIMAL(P)). They have the Oracle external datatype code 7. The format conversion string is of the form 'PP.+SS' or 'PP.-SS', where PP is the precision of the decimal number, and SS is the scale. The '+' and '-' signs are mandatory. For example, a FIXED DECIMAL(6,2) format string would be '06.+02'. Refer to the "Coercing Datatypes after DESCRIBE" section later in this chapter for more information on the use of precision and scale for Oracle NUMBER data.

Set this element using **SQLADR**. Set it before a FETCH in a select descriptor, and before an OPEN in a bind descriptor. You must also set the length of the format conversion string in the SQLDFMTL element.

SQLDVLN

Select Descriptor
Oracle sets this element when it executes the DESCRIBE statement. It contains the length of the select-list value. The format of the length differs among Oracle datatypes. For character datatypes (VARCHAR2 and CHAR), SQLDVLN is set to the number of bytes in the select-list item. For NUMBER datatypes, the scale is returned in the low-order byte of the variable, and the precision in the next highest-order byte.

If, for a NUMBER datatype, SQLDVLN is set to zero, this means that the column in the table was established with no explicit precision and scale, or the precision and scale for the select-list expression was not determined. In this case, you must decide on an appropriate value (between 0 and 38) and put it in SQLDVLN before the FETCH.

For maximum portability, use the **SQLPRC** or **SQLPR2** library procedures (described in the "Datatypes in the SQLDA" section later in this chapter) to retrieve the precision and scale values from SQLDVLN.

You must reset SQLDVLN to the required length of the data buffer before executing the FETCH statement. For example, if you coerce a described NUMBER to a PL/I CHARACTER string, set SQLDVLN to the precision of the number, and add two to account for the sign and decimal point.

If you coerce a NUMBER to a PL/I FLOAT, set SQLDVLN to the length in bytes of the appropriate FLOAT type in your implementation of PL/I (usually 4 for a FLOAT(7) or less, and 8 for a FLOAT(8) or greater). See the "Datatypes in the

SQLDA" section later in this chapter for more information about the lengths of coerced datatypes.

Bind Descriptor

Your program must set the correct SQLDVLN value, which is the length in bytes of the bind variable data buffer that SQLDV points to. Set the length before executing the OPEN statement.

SQLDFMTL

Contains the length of the decimal format conversion string pointed to by the SQLDFMT element. Set it to zero if SQLDFMT does not point

to a format conversion string. Set it before the FETCH in a select descriptor, and before the OPEN in a bind descriptor.

SQLDVTYP

Select Descriptor

Oracle sets this element when it performs the DESCRIBE statement. It contains the datatype code for the select-list value. This datatype code determines how the Oracle data will be converted into the PL/I data buffer or variable that SQLDV points to. This topic is covered in detail in the "Datatypes in the SQLDA" section later in this chapter.

> **Note:** In a select descriptor, the high-order bit of this element is set to indicate the NULL/NOT NULL status of the field. You should retrieve the datatype code using the **SQLNUL** procedure. See the "Handling NULL/NOT NULL Datatypes" section later in this chapter for a description of this procedure.

Bind Descriptor

The DESCRIBE statement sets this element to zero. You must set the datatype code in this element before executing the OPEN statement. The code indicates the external PL/I type of the buffer or variable that SQLDV points to. Refer to the section "Datatypes in the SQLDA" later in this chapter for more information on the datatype codes.

SQLDI

Holds the address of an indicator variable, declared in your program. The indicator variable must be a FIXED BIN(15).

You put the address in this element using the **SQLADR** procedure.

Select Descriptor
You must initialize this element before doing a FETCH. Oracle sets the indicator values after fetching select-list items.

For select descriptors, when Oracle executes the statement

```
EXEC SQL FETCH ... USING DESCRIPTOR SELDSC;
```

if the Jth returned select-list item is NULL, the indicator-variable value to which SELDSC.SQLDSC(J).SQLDI points is set to -1. If not NULL, it is set to zero or a positive integer.

Bind Descriptor
You must initialize this element and set the variable values before doing an OPEN.

For bind descriptors, when Oracle executes the statement

```
EXEC SQL OPEN ... USING DESCRIPTOR BNDDSC;
```

the indicator-variable value to which BNDDSC.SQLDSC(J).SQLDI points determines whether the Jth bind variable is NULL. If the value of an indicator variable is -1, the value of its associated variable is NULL.

SQLDH_VNAME

Holds the address of a buffer used to store the name of a select-list item (for select descriptors) or the name of a bind variable (for bind descriptors). After a DESCRIBE, the appropriate names will be placed in the strings pointed to by the SQLDH_VNAME elements.

Your host program code must initialize the SQLDH_VNAME elements with the addresses of the strings you have declared before executing the DESCRIBE statement. Use the **SQLADR** procedure to do this initialization.

SQLDH_MAX_VNAMEL

Contains the maximum length of the buffer pointed to by SQLDH_VNAME. Your program must set this value before issuing a DESCRIBE statement. A value placed in the string pointed to by SQLDH_VNAME will be truncated to this length if necessary.

SQLDH_CUR_VNAMEL

Contains the actual number of characters in the string pointed to by SQLDH_VNAME after a DESCRIBE.

SQLDI_VNAME

Holds the address of a string that stores the name of a bind indicator variable. It is set using **SQLADR**. It must be set before the DESCRIBE statement is executed. This element is not used in a select descriptor.

SQLDI_MAX_VNAMEL

Contains the maximum number of characters in the SQLDI_VNAME string. Your program must initialize this value before performing a DESCRIBE statement. A value placed in the string pointed to by SQLDI_VNAME will be truncated to this length if necessary. This element is not used in a select descriptor.

SQLDI_CUR_VNAMEL

Contains the number of characters contained in the string pointed to by SQLDI_VNAME after the DESCRIBE statement. This element is not used in a select descriptor.

SQLDFCLP

Unused element; reserved by Oracle. It must be set to zero when your program starts.

SQLDFCRCP

Unused element; reserved by Oracle. It must be set to zero when your program starts.

Datatypes in the SQLDA

This section provides more information about using the SQLDVTYP datatype element in the SQLDA. In host programs that do not use datatype equivalencing or Dynamic SQL Method 4, the precompiler predefines how to convert between Oracle data and your program host variables. When you SELECT data in a table into a PL/I variable, the type of the PL/I variable determines the conversion. For example, if you SELECT data from an Oracle column having the type NUMBER into a PL/I CHARACTER array, the numeric data is converted to a character (ASCII or EBCDIC) representation. If you select the same numeric data into an integer (FIXED BINARY) variable, the data is converted to a binary integer representation. When you INSERT or UPDATE data, the data in the host variable is converted to the correct type for the column in the table.

But, when you write a Dynamic Method 4 program, you must specify the conversion by doing the following:

- set the datatype code for each bind variable in the bind descriptor
- change some of the datatype codes that Oracle returns when you DESCRIBE a select list into a select descriptor

Internal and External Datatypes

Oracle defines a number of datatypes, and for each datatype, there is a *datatype code*. There is a distinction between *internal* datatypes and *external* datatypes. Internal datatypes are the types that you can assign to an Oracle column in a table, and that Oracle can return from a table. The internal datatypes are CHAR, VARCHAR2, NUMBER, DATE, LONG, RAW, and LONG RAW. There are additional entities that you can SELECT from a table, such as LEVEL, CURRVAL, NEXTVAL, ROWNUM, UID, USER, and SYSDATE, but these entities are always returned as one of the basic internal types. LEVEL, CURRVAL, NEXTVAL, UID, and ROWNUM are NUMBERs; SYSDATE is a DATE type; and USER is a CHAR.

External datatypes include the internal datatypes, and in addition provide extended conversion capabilities. For example, when you DESCRIBE a select list that contains a LONG RAW item, the length of the value is not returned in the SQLDVLN element. You can coerce the LONG RAW internal type to a VARRAW external type, by placing an external datatype code for VARRAW in the SQLDVTYP element after doing the DESCRIBE, but before doing the FETCH. The data returned on the FETCH will then include the length of the LONG RAW item in the first two bytes of the output buffer.

Coercing Datatypes After DESCRIBE

In some cases, the internal datatype codes that a DESCRIBE operation returns in SQLDVTYP might not be the ones you want for your program's purposes. Two examples of this are DATE and NUMBER. When you DESCRIBE a select list containing a DATE item, the datatype code 12 for DATE is returned in the SQLDVTYP element. If you do not change this before the FETCH, the date value is returned as 7 binary bytes that represent the date in the Oracle internal format. To receive the date in a character format (DD-MON-YY), you must change the 12 datatype code to 1 (VARCHAR2), and increase the SQLDVLN (length) value, which was returned as 7, to 9.

Similarly, when you DESCRIBE a select list that contains a NUMBER item, the datatype code 2 is returned in SQLDVTYP. If you do not change this before the FETCH, the numeric value is returned as an array representing the number in its

Oracle internal format, which is probably not what you want. So, change the 2 code to 1 (VARCHAR2), 3 (INTEGER), 4 (FLOAT), or some other appropriate type.

Extracting Precision and Scale

When coercing NUMBER (2) values to VARCHAR2 (1) for display purposes, you also have to extract the precision and scale bytes from the value that the DESCRIBE statement places in the SQLDVLN variable. You then use the precision and scale to compute a maximum length for the conversion into a PL/I CHARACTER string, placing the value back into the SQLDVLN element before the FETCH.

The library procedure **SQLPRC** extracts precision and scale. You call **SQLPRC** using the syntax

```
CALL SQLPRC(LENGTH, PRECISION, SCALE);
```

where:

LENGTH	Is a FIXED BIN(31) variable that holds the length of the NUMBER value. The scale and precision of the value are stored respectively in low and next-higher bytes.
PRECISION	Is an output parameter that returns the precision of the NUMBER value. Precision is the total number of significant digits in the number.
	If precision is returned as zero, the size of the number is not specified. In this case, you should set the precision to a value (between 0 and 38) that is appropriate for the size of your data buffer.
SCALE	Is an output parameter that returns the scale of the NUMBER value. If positive, scale specifies the number of digits to the right of the decimal point. If negative, scale indicates the position of the first significant digit to the left of the decimal point. For example, a scale of -2 indicates that the number is rounded to the nearest hundreds.
	When the scale is negative, increase the length by the absolute value of the scale. For example, precision and scale values of 4 and -2 can accommodate a number as large as 999900. The following example shows how to use **SQLPRC**:

```
/* include a select descriptor SQLDA */
EXEC SQL INCLUDE SELDSC;
```

```
/* Declare variables for the function call. */
DCL (J, PRECISION, SCALE) FIXED BIN(31),
/* Declare some data buffers. */
    SEL_DV(3)              CHARACTER (10) VARYING,
/* Declare library function. */
    SQLPRC EXTERNAL       ENTRY(ANY, ANY, ANY);
...
/* Extract precision and scale. */
CALL SQLPRC(SELDSC.SQLDSC(J).SQLDVLN, PRECISION, SCALE);

/* Set the desired precision if 0 is returned
.   Note that size of the buffer is 10. */
IF PRECISION = 0 THEN
   PRECISION = 6;

/* Allow for possible decimal point and sign. */
SELDSC.SQLDSC(J).SQLDVLN = PRECISION + 2;

/* Increase SQLDVLN if scale is negative. */
IF SCALE < 0 THEN
   SELDSC.SQLDSC(J).SQLDVLN = SELDSC.SQLDSC(J).SQLDVLN
   + (-SCALE);
```

Notice that the first parameter in the **SQLPRC** procedure call points to the
SQLDVLN element in the Jth minor structure of the SQLDSC array, and that the
precision and scale parameters must be 4 bytes in size.

The SQLPRC procedure returns zero as the precision and scale values for certain
SQL datatypes. The SQLPR2 procedure is similar to SQLPRC, having the same
syntax, and returning the same binary values, except for these datatypes:

SQL Datatype	Binary Precision	Scale
FLOAT	126	-127
FLOAT (N)	N (range is 1 to 126)	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

Datatype Codes

The table below lists the datatypes and datatype codes, as well as the type of host
variable normally used for that external type. See the *Programmer's Guide to the*

Oracle Precompilers for detailed information about the format of the external datatypes.

External Datatype	Code	PL/I Host Variable
VARCHAR2	1	CHARACTER(N)
NUMBER	2	CHARACTER(N)
INTEGER	3	FIXED BINARY (31)
FLOAT	4	FLOAT DECIMAL(P,S)
STRING	5	CHARACTER(N)
VARNUM	6	CHARACTER(N)
DECIMAL	7	FIXED DECIMAL(P,S)
LONG	8	CHARACTER(N)
VARCHAR	9	CHARACTER(N) VARYING
ROWID	11	CHARACTER(N)
DATE	12	CHARACTER(N)
VARRAW	15	CHARACTER(N)
RAW	23	CHARACTER(N)
LONG RAW	24	CHARACTER(N)
UNSIGNED	68	(not used in PL/I)
DISPLAY	91	FIXED DECIMAL(P,S)
LONG VARCHAR	94	CHARACTER(N)
LONG VARRAW	95	CHARACTER(N)
CHAR	96	CHARACTER(N)
CHARZ	97	(not used in PL/I)
MLSLABEL	106	CHARACTER(N)

The datatype codes listed in the table above are the ones that you should set in the SQLDVTYP element of the SQLDA for data conversion.

Handling NULL/NOT NULL Datatypes

DESCRIBE returns a NULL/NOT NULL indication in the SQLDVTYP element of the select descriptor, defined as a FIXED BINARY (15). If a column is declared to be NOT NULL, the high-order bit of the variable is clear; otherwise, it is set.

Before using the datatype in an OPEN or FETCH statement, if the NULL/NOT NULL bit is set, you must clear it. (Never set the bit.) You can use the library procedure **SQLNUL** to find out whether a column allows NULLs, and to clear the datatype's NULL/NOT NULL bit. You call **SQLNUL** using the syntax

```
CALL SQLNUL(TYPE_VALUE, TYPE_CODE, NULL_STATUS);
```

where:

TYPE_VALUE	Is the FIXED BIN(15) variable that holds the datatype code of a select-list value, as returned by the DESCRIBE.
TYPE_CODE	Is a variable that returns the datatype code of the select-list item, with the NULL bit cleared.
NULL_STATUS	Is a variable that returns set to zero if the column was declared to be NOT NULL, or set to 1 if the column allows NULL values.

The following example shows how to use **SQLNUL**:

```
/* Declare variables for the function call. */
DCL (NULL_OK, TYPE_CODE)  FIXED BIN (15),
    SQLNUL EXTERNAL       ENTRY(ANY, ANY, ANY);

/* Find out whether column is NOT NULL. */
CALL SQLNUL(SELDSC.SQLDSC(J).SQLDVTYP,
    TYPE_CODE, NULL_OK);

IF NULL_OK ^= 0 THEN
    PUT LIST ('Nulls OK for this column.');
```

> **Note:** After SQLNUL returns, the second parameter contains the type code with the NULL bit cleared. This is the value you must use when checking for an Oracle internal datatype code. You should also make sure to reset the SQLDVTYP element in the SQLDA (before the FETCH) with a datatype code that has the high-order bit cleared. For example
>
> ```
> SELDSC.SQLDSC(J).SQLDVTYP = TYPE_CODE;
> ```

The Basic Steps

Method 4 can be used to process *any* dynamic SQL statement. In this example, a query is processed so you can see how both input and output variables are handled. Steps that are common to all embedded SQL programs, such as connecting to Oracle and including the SQLCA, are not described here.

To process a dynamic query using Method 4, our example program takes the following steps:

1. Declare a host string to hold the query text in the SQL Declare Section.

2. Set the maximum number of select-list items and bind variables that can be described in the INCLUDEd SQLDAs.

3. INCLUDE the select and bind SQLDAs.

4. Declare the data buffers to hold the input and output values.

5. Initialize the select and bind descriptors.

6. Get the query text into the host string.

7. PREPARE the query from the host string.

8. DECLARE a cursor FOR the query.

9. DESCRIBE the bind variables INTO the bind descriptor.

10. Reset the maximum number of bind variables to the number actually found by DESCRIBE.

11. Get values for the input bind variables found by DESCRIBE.

12. OPEN the cursor USING the bind descriptor.

13. DESCRIBE the select list INTO the select descriptor.

14. Adjust the N, length, and datatype values in the select descriptor after the DESCRIBE (SQLDNUM, SQLDVTYP, and SQLDVLN).

15. FETCH a row from the database INTO the buffers pointed to by the select descriptor.

16. Process the select-list items returned by FETCH.

17. CLOSE the cursor when there are no more rows to fetch.

    Note: If the dynamic SQL statement is **not** a query or contains a known number of select-list items or placeholders, then some of the above steps are unnecessary.

A Closer Look at Each Step

With Method 4, you use the following sequence of embedded SQL statements:

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };

EXEC SQL DECLARE cursor_name CURSORFOR statement_name;

EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
    INTO bind_descriptor_name;

EXEC SQL OPEN cursor_name
    [USING DESCRIPTOR descriptor_name];

EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
    INTO select_descriptor_name;

EXEC SQL FETCH cursor_name
    USING DESCRIPTOR select_descriptor_name;

EXEC SQL CLOSE cursor_name;
```

Note that if the number of select-list items is known, you can omit DESCRIBE SELECT LIST and use the following Method 3

FETCH statement:

```
EXEC SQL FETCH emp_cursor INTO host_variable_list;
```

If the number of bind variables is known, you can omit DESCRIBE BIND VARIABLES and use the following Method 3 OPEN statement:

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

The following sections show how these statements allow your host program to accept and process a dynamic query using descriptors.

Declare a Host String

Your program needs a variable to store the text of the dynamic query. The variable (SELECT_STMT in our example) must be declared as a character string.

```
EXEC SQL BEGIN DECLARE SECTION;
   ..
.  DCL SELECT_STMT  CHARACTER (120);
EXEC SQL END DECLARE SECTION;
```

Set the Size of the Descriptors

Before you include the files that contain the select and bind descriptor declarations, you should set the size of the descriptor in each file. This is set by changing the N variable for the SQLDSC array of substructures.

You normally set this to a value high enough to accommodate the maximum number of select-list items and bind variables that you expect to have to process. The program will not be able to process the SQL statement if there are more select-list items or bind variables than the number of substructures. In our example, a low number of three is used so that the structures can be easily illustrated.

Declare the SQLDAs

Use INCLUDE to copy the files containing the SQLDA declarations into your program, as follows:

```
EXEC SQL INCLUDE SELDSC;  /* select descriptor */
EXEC SQL INCLUDE BNDDSC;  /* bind descriptor */
```

Declare the Data Buffers

You must declare data buffers to hold the bind variables and the returned select-list items. In our examples, arbitrary names are used for the buffers used to hold the following:

- names of select-list items (SEL_DH_VNAME) or bind variables (BND_DH_VNAME)

- data retrieved from the query (SEL_DV)

- values of bind variables (BND_DV)

- values of indicator variables (SEL_DI and BND_DI)

- names of indicator variables used with bind variables (BND_DI_VNAME)

```
DCL SEL_DH_VNAME (3)  CHARACTER (5),
    BND_DH_VNAME (3)  CHARACTER (5),
    SEL_DV (3)        CHARACTER (10),
    BND_DV (3)        CHARACTER (10)
    SEL_DI (3)        FIXED BIN (15),
    BND_DI (3)        FIXED BIN (15),
    BND_DI_VNAME (3)  CHARACTER (5);
```

Note that an *array* of data buffers is declared, and the dimension of the array (3) is the same as the number N of substructures (SQLDSC(N)) in each descriptor area.

Initialize the Descriptors

You must initialize several elements in each descriptor. Some are simply set to numeric values; some require the library procedure **SQLADR** to place an address in the element.

In our example, you first initialize the select descriptor. Set SQLDNUM to the number of substructures (3). Then, in each substructure, set the SQLDH_MAX_ VNAMEL element to the length (5) of the name data buffer (SEL_DH_VNAME). Set the SQLDVLN element to the length (10) of the value data buffer (SEL_DV). Put the addresses of the data buffers in the SQLDH_VNAME, SQLDV, and SQLDI elements using **SQLADR**. Finally, set the reserved and unused elements to zero.

```
SELDSC.SQLDNUM = 3;
DO J = 1 TO SELDSC.SQLDNUM;
    SELDSC.SQLDSC(J).SQLDH_MAX_VNAMEL = 5;
    SELDSC.SQLDSC(J).SQLDVLN = 10;

    /* setup the pointers */
    CALL SQLADR(ADDR(SEL_DH_VNAME(J)),
        ADDR(SELDSC.SQLDSC(J).SQLDH_VNAME));
    CALL SQLADR(ADDR(SEL_DV(J)),
        ADDR(SELDSC.SQLDSC(J).SQLDV));
    CALL SQLADR(ADDR(SEL_DI(J)),
        ADDR(SELDSC.SQLDSC(J).SQLDI));

    /* initialize unused elements to 0 */
    SEL_DI(J) = 0;
    SELDSC.SQLDSC(J).SQLDFMT = 0;
    SELDSC.SQLDSC(J).SQLDFCLP = 0;
    SELDSC.SQLDSC(J).SQLDFCRCP = 0;
END;
```

The bind descriptor is initialized in almost the same way. The difference is that SQLDI_MAX_VNAMEL must also be initialized.

```
BNDDSC.SQLDNUM = 3;
DO J = 1 TO BNDDSC.SQLDNUM;
    BNDDSC.SQLDSC(J).SQLDH_MAX_VNAMEL = 5;
    BNDDSC.SQLDSC(J).SQLDVLN = 10;

    /* length of indicator variable name */
    BNDDSC.SQLDSC(J).SQLDI_MAX_VNAMEL = 5;

    /* setup the pointers */
```

```
            CALL SQLADR(ADDR(BND_DH_VNAME(J)),
                ADDR(BNDDSC.SQLDSC(J).SQLDH_VNAME));

            /* address of indicator variable name */
            CALL SQLADR(ADDR(BND_DI_VNAME(J)),
                ADDR(BNDDSC.SQLDSC(J).SQLDI_VNAME));
            CALL SQLADR(ADDR(BND_DV(J)),
                ADDR(BNDDSC.SQLDSC(J).SQLDV));
            CALL SQLADR(ADDR(BND_DI(J)),
                ADDR(BNDDSC.SQLDSC(J).SQLDI));

            /* set unused elements to 0 */
            BND_DI(J) = 0;
            BNDDSC.SQLDSC(J).SQLDFMT = 0;
            BNDDSC.SQLDSC(J).SQLDFCLP = 0;
            BNDDSC.SQLDSC(J).SQLDFCRCP = 0;
END;
```

The descriptors that result after the initialization are shown in Figure 5–3 and Figure 5–4. In these pictures, the left-hand box represents the descriptor structure, and the boxes on the right represent the data buffers (such as SEL_DV) that you declared in your program. The arrows represent pointers, showing which data buffers the SQLDA elements point to.

The data buffers are empty after initialization (except SEL_DI and BND_DI, which were set to zero in the example code above). As our example progresses, and the DESCRIBE or FETCH statements begin to fill in the data buffers, the values will be shown in later figures. Whenever these boxes are empty, it indicates that the variable is either uninitialized or was not filled in by a DESCRIBE or FETCH statement. Unused or reserved fields in the descriptors (SQLDFMT, SQLDFMTL, SQLDFCLP, and SQLDFCRCP) are not shown in these figures.

Note: To save space, the SQLDA element names in the left hand columns of Figures 5-3 through 5-9 are abbreviated. Each structure element name must be preceded by the structure and substructure names. For example, S2.SQLDV must be written as SELDSC.SQLDSC(2).SQLDV in the PL/I code. B3.SQLDVTYP stands for BNDDSC.SQLDSC(3).SQLDVTYP.

Figure 5–3   Initialized Select Descriptor

SQLDNUM	3
SQLDFND	
S1.SQLDV	
S2.SQLDV	
S3.SQLDV	
S1.SQLDVLN	10
S2.SQLDVLN	10
S3.SQLDVLN	10
S1.SELDVTYP	
S2.SELDVTYP	
S3.SELDVTYP	
S1.SQLDI	
S2.SQLDI	
S3.SQLDI	
S1.SQLDH±VNAME	
S2.SQLDH±VNAME	
S3.SQLDH±VNAME	
S1.SQLDH±MAX±VNAMEL	5
S2.SQLDH±MAX±VNAMEL	5
S3.SQLDH±MAX±VNAMEL	5
S1.SQLDH±CUR±VNAMEL	
S2.SQLDH±CUR±VNAMEL	
S3.SQLDH±CUR±VNAMEL	

Data Buffers

SEL_DV

SEL_DI

SEL_DH_VNAME
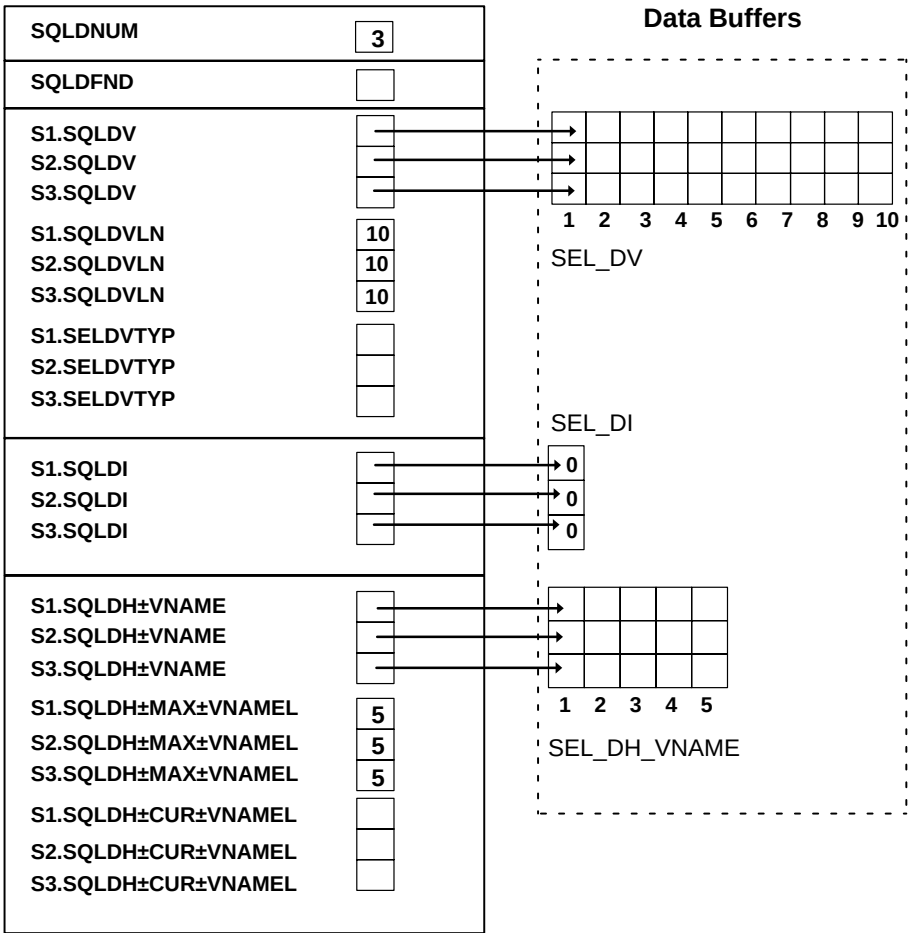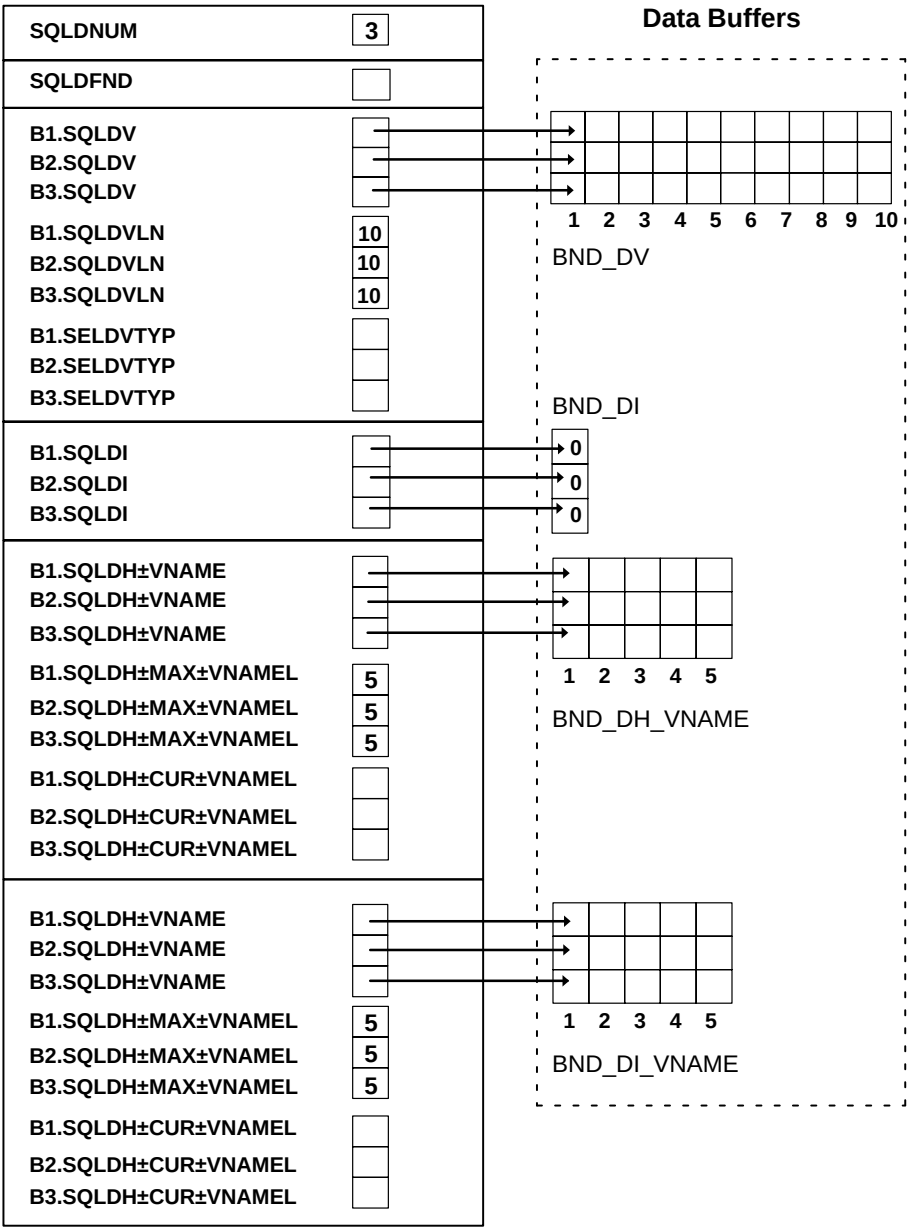
Figure 5–4   Initialized Bind Descriptor

Get the Query Text into the Host String

Continuing our example, you prompt the user for a SQL statement, and then store the input in SELECT_STMT.

```
PUT LIST ('Enter SQL statement: ');
GET EDIT (SELECT_STMT) (A(120));
```

In this example, it is assumed that the user typed the string

```
'SELECT ENAME, EMPNO, COMM FROM EMP WHERE COMM < :BONUS'
```

PREPARE the Query from the Host String

PREPARE parses the query and gives it a name. In our example, PREPARE parses the host string SELECT_STMT and gives it the name SQL_STMT, as follows:

```
EXEC SQL PREPARE SQL_STMT FROM :SELECT_STMT;
```

DECLARE a Cursor

DECLARE CURSOR defines a cursor by giving it a name and associating it with a specific query. When declaring a cursor for static queries, you use the following syntax:

```
EXEC SQL DECLARE CURSOR_NAME CURSOR FOR SELECT ...
```

When declaring a cursor for dynamic queries, the statement name given to the dynamic query by PREPARE is substituted for the static query. In our example, DECLARE CURSOR defines a cursor named EMP_CURSOR and associates it with SQL_STMT, as follows:

```
EXEC SQL DECLARE EMP_CURSOR CURSOR FOR SQL_STMT;
```

> **Note:** You must declare a cursor for all Dynamic SQL statements, not just queries. For non-query statements, opening the cursor executes the statement.

DESCRIBE the Bind Variables

DESCRIBE BIND VARIABLES fills in fields in a bind descriptor that describe the bind variables in the SQL statement. In our example, DESCRIBE fills in a bind descriptor named BNDDSC. The DESCRIBE statement is
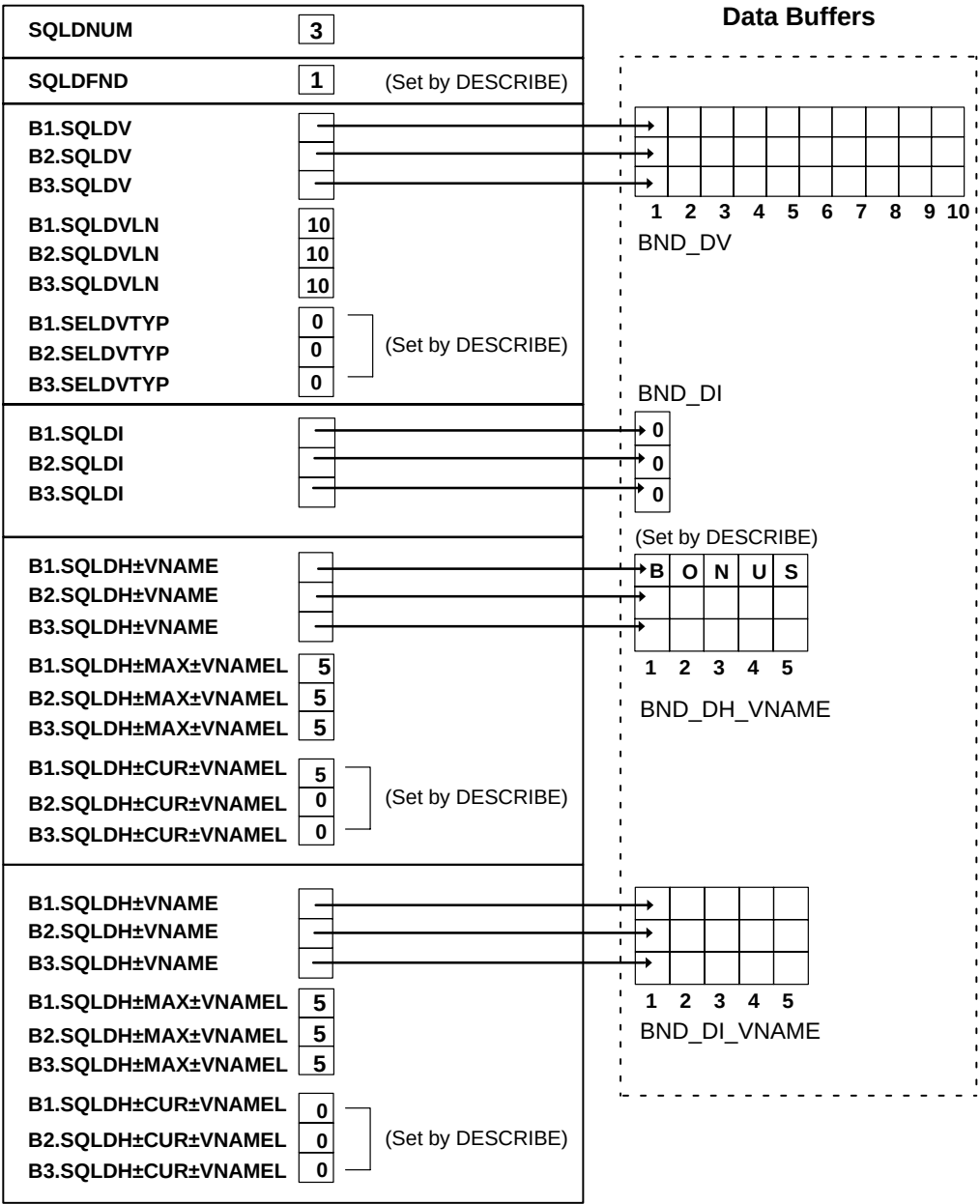
```
EXEC SQL DESCRIBE BIND VARIABLES FOR SQL_STMT INTO BNDDSC;
```

Note that BNDDSC must *not* be prefixed with a colon.

The DESCRIBE BIND VARIABLES statement must follow the PREPARE statement but precede the OPEN statement.

Figure 5–5 shows the bind descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of input bind variables found in the query's WHERE clause.

Figure 5–5   Bind Descriptor After the DESCRIBE

VariablesReset Maximum Number of Bind

Next, you must test the actual number of bind variables found by DESCRIBE, as follows:

```
IF BNDDSC.SQLDFND <0 THEN DO;
    PUT LIST ('Too many input variables were described.');
    GOTO NEXT_SQL_STMT; /* try again */
END;
/* Set number of bind variables DESCRIBEd. */
BNDDSC.SQLDNUM = BNDDSC.SQLDFND;
```
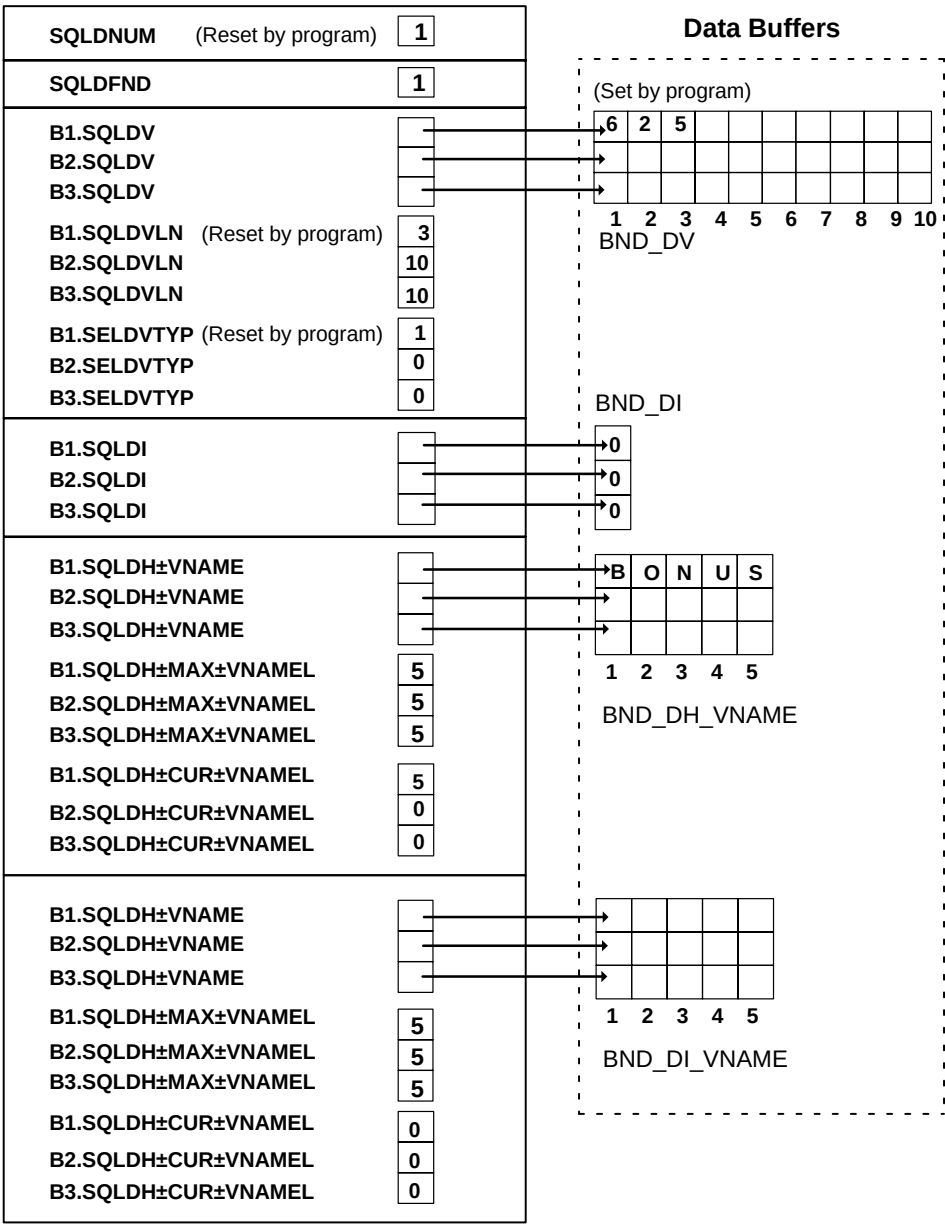
Get Values for Bind Variables

Your program must get values for any bind variables found in the SQL statement. In our example, a value must be assigned to the placeholder BONUS in the query's WHERE clause. So, you prompt the user for the value, and then process it as follows:

```
DCL (BN, BV) POINTER,
    BNAME    CHARACTER (10) BASED(BN),
    BVAL     CHARACTER (10) BASED(BV);
...
PUT SKIP LIST ('Enter values of bind variables');
DO J = 1 TO BNDDSC.SQLDNUM;
    /* Display the name. Use UNSPEC to get an integer
       (really a pointer) into the buffer pointer. */
    UNSPEC(BN) = UNSPEC(BNDDSC.SQLDSC(J).SQLDH_VNAME);
    PUT SKIP EDIT (BN->BNAME, ': ')
        (A(BNDDSC.SQLDSC(J).SQLDH_CUR_VNAMEL),A(2));
    /* Get bind variable value from user. */
    UNSPEC(BV) = UNSPEC(BNDDSC.SQLDSC(J).SQLDV);
    GET LIST (BV->BVAL);
    /* Set the length. */
    BNDDSC.SQLDSC(J).SQLDVLN = LENGTH(BV->BVAL);
    /* Make the datatype VARCHAR2. */
    BNDDSC.SQLDSC(J).SQLDVTYP = 1;
END;
```

Assuming that the user supplied a value of 625 for BONUS, Figure 5–6 shows the resulting bind descriptor.

Figure 5–6   Bind Descriptor After Assigning Values

OPEN the Cursor

The OPEN statement used for dynamic queries is similar to that used for static queries except that the cursor is associated with a bind descriptor. Values determined at runtime and stored in the bind descriptor are used to evaluate the query and identify its active set.

In our example, OPEN associates EMP_CURSOR with BNDDSC, as follows:

```
EXEC SQL OPEN EMP_CURSOR USING DESCRIPTOR BNDDSC;
```

Remember, BNDDSC must *not* be prefixed with a colon.

The OPEN executes the query, identifies its active set, and positions the cursor at the first row.

DESCRIBE the SelectList

The DESCRIBE SELECT LIST statement must follow the OPEN statement but precede the FETCH statement.

DESCRIBE SELECT LIST fills in a select descriptor to hold descriptions of items in the query's select list. In our example, DESCRIBE fills in a select descriptor named SELDSC, as follows:

```
EXEC SQL DESCRIBE SELECT LIST FOR SQL_STMT INTO SELDSC;
```
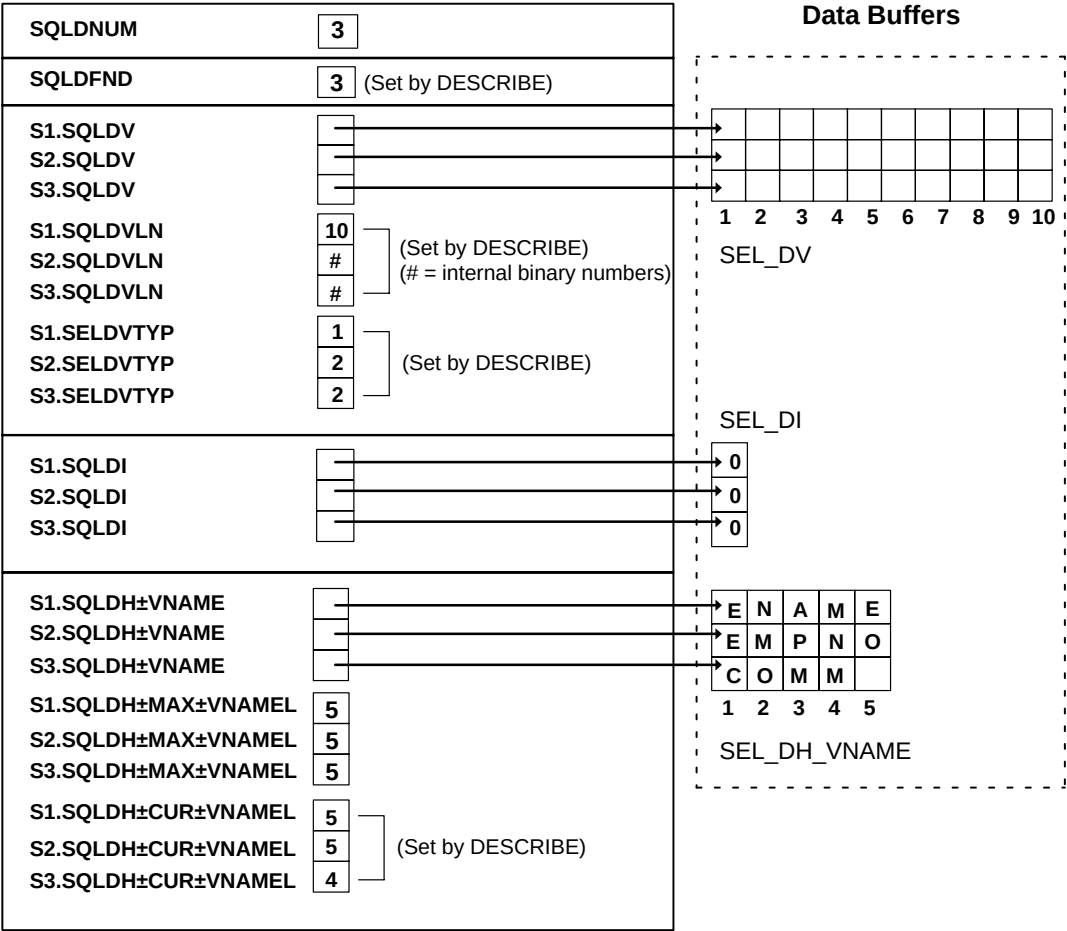
SELDSC must *not* be prefixed with a colon.

Accessing the data dictionary, DESCRIBE sets the length and datatype of each select-list item.

Figure 5–7 shows the select descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SQLDFND to the actual number of items found in the query's select list.

Also notice that the NUMBER lengths in the second and third SQLDVLN elements are not usable yet. For select-list items defined as NUMBER, you should use the library procedure **SQLPRC** to extract precision and scale, as explained in the next section.

Figure 5–7   Select Descriptor After the DESCRIBE

Adjust the Select Descriptor Values

First you must check the SELDSC.SQLDFND variable that was set by the
DESCRIBE statement. If it is negative, too many select-list items were described. If it
is not, set SQLDNUM to the number of select-list items described, as follows:

```
IF SQLDFND <0 THEN DO;
    PUT LIST ('Too many select-list items. Try again.');
    GOTO NEXT_STMT;
END;
```
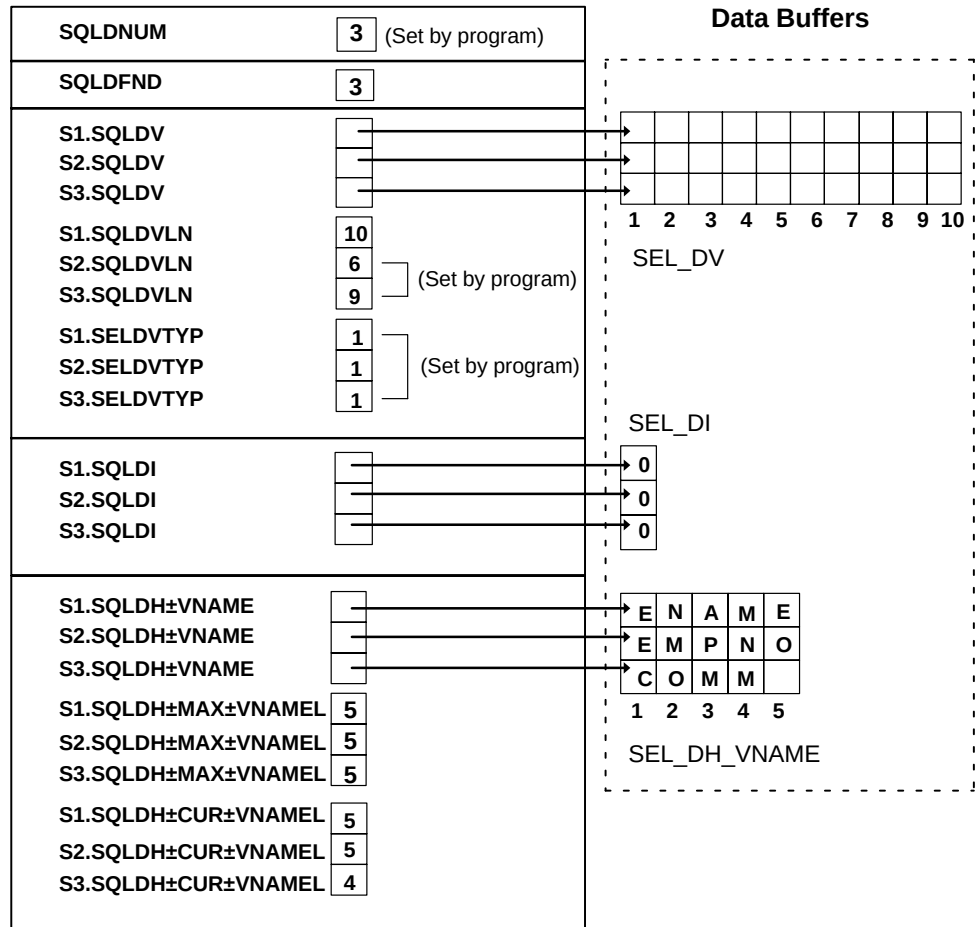
```
ELSE
    SELDSC.SQLDNUM = SELDSC.SQLDFND;
```

In our example, before FETCHing the select-list values, you reset some length elements for display purposes. You also reset the datatype value to avoid dealing with Oracle datatypes.

```
...
DCL J FIXED BIN(15),
    (SCALE, PRECISION) FIXED BIN(31);
DCL SQLPRC EXTERNAL ENTRY(ANY, ANY, ANY);
...
/* Process each field value */

DO J = 1 TO SELDSC.SQLDNUM;
    /* If the datatype is NUMBER (datatype code 2)
       extra processing is required. */
    IF SELDSC.SQLDSC(J).SQLDVTYP = 2 THEN DO;
        /* get precision and scale */
        CALL SQLPRC(SELDSC.SQLDSC(J).SQLDVLN,
            PRECISION, SCALE);
        /* Allow for the size of a number with
           no precision in the table. */
        IF PRECISION = 0 THEN
            PRECISION = 6;
        SELDSC.SQLDSC(J).SQLDVLN = PRECISION + 2;
        IF SCALE <0 THEN
            SELDSC.SQLDSC(J).SQLDVLN =
                SELDSC.SQLDSC(J).SQLDVLN + (-SCALE);
    END;
    /* If datatype is a DATE (datatype code 12)
       set length to 9. */
    IF SELDSC.SQLDSC(J).SQLDVTYP = 12 THEN
        SELDSC.SQLDSC(J).SQLDVLN = 9;
    /* Coerce all datatypes to VARCHAR2. */
    SELDSC.SQLDSC(J).SQLDVTYP = 1;
END;
```

Figure 5–8 shows the resulting select descriptor. Notice that the lengths for the buffers that will hold the EMPNO and COMM fields are set to 6 and 9. These values were set in the DO-loop above from the EMP table column lengths of 4 and 7 by the statement that adds 2 to PRECISION (for possible minus sign and decimal point). Notice also that the datatypes are set to 1 (VARCHAR2).

Figure 5–8   Select Descriptor Before the FETCH

Note: When the datatype code returned on a DESCRIBE is 2 (Oracle internal number) it must be coerced to a PL/I type that a NUMBER can be converted to; this does not have to be CHARACTER. You could also coerce a NUMBER to a PL/I FLOAT, in which case you would put the datatype code number 4 in the SQLDVTYP element, and put the length (size of a PL/I float in bytes) in the SQLDVLN element.
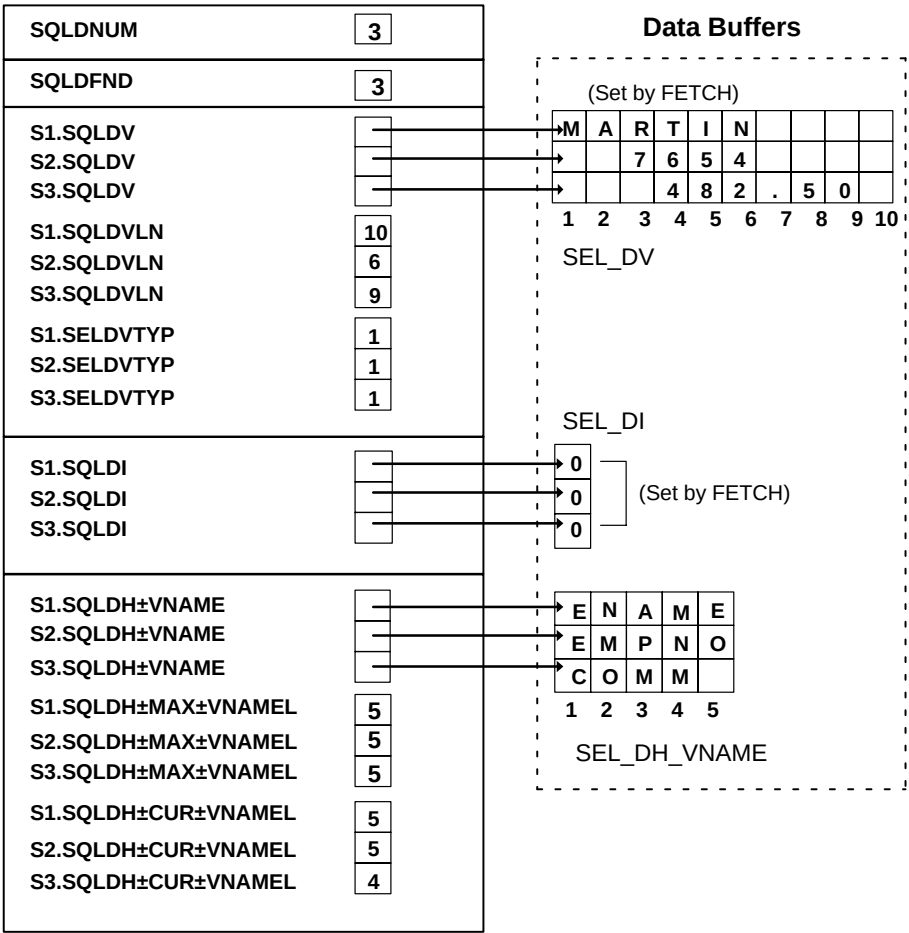
FETCH A Row from the Active Set

FETCH returns a row from the active set, stores select-list values in the select descriptor, and advances the cursor to the next row in the active set. If there are no more rows, FETCH sets SQLCA.SQLCODE, the SQLCODE variable, or the SQLSTATE variable to the "no data found" Oracle error code. In our example, FETCH returns the values of columns ENAME, EMPNO, and COMM to the data buffers pointed to by the SQLDV elements:

```
...
EXEC SQL WHENEVER NOT FOUND GOTO N_FND;

NXT_ROW:
EXEC SQL FETCH EMP_CURSOR USING DESCRIPTOR SELDSC;
CALL PRINT_ROW(SELDSC); /* proc. to print values */
GOTO NXT_ROW;
N_FND:
...
```

Figure 5–9 shows the select descriptor in our example after the FETCH. Notice that Oracle has stored the column and indicator-variable values in the data buffers pointed to by the descriptor.

Figure 5–9   Select Descriptor After the FETCH

Process the Select-List Items

After the FETCH, your program can scan the select descriptor for the select-list values returned by FETCH and process them. In our example, values for columns ENAME, EMPNO, and COMM can be processed.

Note how Oracle converts these values into the SEL_DV data buffers. The select-list value 'MARTIN' is from an Oracle VARCHAR2 column. It is left justified in the 10-byte field of SEL_DV(1).

The EMPNO value is a NUMBER(4) in the Oracle table. The program adds 2 to this (for decimal point and possible sign), resulting in a length of 6. Since EMPNO is a NUMBER, the value '7654' is *right* justified in a 6-byte field in the output buffer.

The COMM column has a length of 7. The program adds 2 (for decimal point and sign) for a total length of 9. The value '482.50' is right justified on conversion into the 9-byte field of SEL_DV(3).

CLOSE the Cursor

CLOSE disables the cursor. In our example, CLOSE disables EMP_CURSOR as follows:

```
EXEC SQL CLOSE EMP_CURSOR;
```

Using Host Arrays

To use input or output host arrays with Method 4, you must use the optional FOR clause of the EXECUTE statement to tell Oracle the size of the host array. (For more information about the FOR clause, see Chapter 8 of the *Programmer's Guide to the Oracle Precompilers.*)

First, you set up a descriptor for the host arrays. Set each SQLDV element to point to the start of the array, the SQLDVLN element to contain the length of each member of the array, and the SQLDVTYP element to contain the type of the members.

Then, you use a FOR clause in the EXECUTE or FETCH statement (whichever is appropriate) to tell Oracle the number of array elements you want to process. This is necessary because Oracle has no other way of knowing the size of your host array. Note that EXECUTE can be used for non-queries with Method 4. In the following program, three input host arrays are used to add data to the emp table:

```
HSTARRS: PROCEDURE OPTIONS(MAIN);
/* Using the FOR clause with Method 4 */
EXEC SQL BEGIN DECLARE SECTION;
    DCL USR           CHARACTER (21) VARYING INIT('SCOTT'),
        PWD           CHARACTER (21) VARYING INIT('TIGER'),
        ARRAY_SIZE    FIXED BIN(31) INIT(5),
        SQL_STMT      CHARACTER (120) VARYING
         INIT('INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
               VALUES (:E, :N, :D)');
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL INCLUDE SQLCA; /* SQLCAPLI on IBM systems */
/* Declare a bind descriptor. */
EXEC SQL INCLUDE BNDDSC;

DCL  NAMES(5)     CHARACTER (15),
     NUMBERS(5)   FIXED BIN(31),
     DEPTS(5)     FIXED BIN(31);

EXEC SQL WHENEVER SQLERROR GOTO ERROR;

EXEC SQL CONNECT :USR IDENTIFIED BY :PWD;
PUT SKIP LIST ('Connected to Oracle.');

/*    Set up the descriptors. */
BNDDSC.SQLDNUM = 3;

EXEC SQL PREPARE S FROM :SQL_STMT;
EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO BNDDSC;
/*    Initialize the descriptors
.      Use SQ3ADR on IBM systems. */
CALL SQLADR(ADDR(NUMBERS(1)),
            ADDR(BNDDSC.SQLDSC(1).SQLDV));
BNDDSC.SQLDSC(1).SQLDVLN = 4;    /* 4-byte... */
BNDDSC.SQLDSC(1).SQLDVTYP = 3;   /* ...integers */

CALL SQLADR(ADDR(NAMES(1)),
            ADDR(BNDDSC.SQLDSC(2).SQLDV));
BNDDSC.SQLDSC(2).SQLDVLN = 15;   /* 15...        */
BNDDSC.SQLDSC(2).SQLDVTYP = 1;   /* char arrays */
CALL SQLADR(ADDR(DEPTS(1)),
            ADDR(BNDDSC.SQLDSC(3).SQLDV));
BNDDSC.SQLDSC(3).SQLDVLN = 4;    /* 4-byte... */
BNDDSC.SQLDSC(3).SQLDVTYP = 3;   /* ...integers */

/*    Now initialize the data buffers. */
NAMES(1)   = 'TRUSDALE';
NUMBERS(1) = 1010;
DEPTS(1)   = 30;
NAMES(2)   = 'WILKES';
NUMBERS(2) = 1020;
DEPTS(2)   = 30;
NAMES(3)   = 'BERNSTEIN';
NUMBERS(3) = 1030;
DEPTS(3)   = 30;
NAMES(4)   = 'FRAZIER';
```

```
NUMBERS(4) = 1040;
DEPTS(4)   = 30;
NAMES(5)   = 'MCCOMB';
NUMBERS(5) = 1050;
DEPTS(5)   = 30;

PUT SKIP LIST ('Adding to the Sales force...');
EXEC SQL FOR :ARRAY_SIZE EXECUTE S USING DESCRIPTOR BNDDSC;
EXEC SQL COMMIT RELEASE;
PUT SKIP EDIT (SQLCA.SQLERRD(3), ' new salespeople added.')
    (F(4), A);
RETURN;

ERROR:
PUT SKIP EDIT (SQLERRM)(A(70));
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK RELEASE;
RETURN;

END HSTARRS;
```

Sample 10: Dynamic SQL Method 4 Program

This section presents a complete program that illustrates the steps required to use Dynamic SQL Method 4. These steps were outlined in the "Basic Steps" section earlier in this chapter, and were discussed in greater detail in the sections following.

In this demo program, each step as outlined in "The Basic Steps" section earlier in this chapter is noted in comments in the source code. (Note that because of the block structure of PL/I, the steps do not follow in order.)

This program is available online as SAMPLE10.PPL.

```
DYN4DEM: PROCEDURE OPTIONS(MAIN);

/*  On IBM systems you must call SQ3ADR
    rather than SQLADR. This is set up here. */
EXEC Oracle IFDEF CMS;
EXEC Oracle DEFINE SQ3LIB;
EXEC Oracle ENDIF;

EXEC Oracle IFDEF MVS;
EXEC Oracle DEFINE SQ3LIB;
EXEC Oracle ENDIF;
```

```
/*
 *
 *   STEP 1 -- Declare a host string
 *              (Host variables for the logon process
 *               must also be declared in the SQL Declare Section)
 */
EXEC SQL BEGIN DECLARE SECTION;
        /* host string */
     DCL STMT        CHARACTER (1000) VAR,
        /* strings for logon */
        (USR, PWD)  CHARACTER (40)<N>VAR;
EXEC SQL END DECLARE SECTION;
EXEC Oracle IFDEF SQ3LIB;
    EXEC SQL INCLUDE SQLCAPLI;
EXEC Oracle ELSE;
    EXEC SQL INCLUDE SQLCA;
EXEC Oracle ENDIF;

/*
 *  STEP 3 -- INCLUDE the bind and select descriptors
 *              (Remember STEP 2, and set N in SQLDSC
 *               in the INCLUDEd files before the INCLUDE.)
 */
EXEC SQL INCLUDE BNDDSC;
EXEC SQL INCLUDE SELDSC;

/*
 *  STEP 4 -- Declare the data buffers for input and output
 *           variables, and the buffers for names.
 *           (NOTE: these are *not* host variables!)
 */
DCL BND_DI(20)        FIXED BINARY (15),
    BND_DI_VNAME(20)  CHARACTER(80),
    BND_DV(20)        CHARACTER(80),
    BND_DH_VNAME(20)  CHARACTER(80),
    SEL_DI(20)        FIXED BINARY (15),
    SEL_DV(20)        CHARACTER(80),
    SEL_DH_VNAME(20)  CHARACTER(80),
    /* index variable and flags */
    (I, TRY, DONE)    FIXED BIN(15);

/*  Local procedures */

/*
 *  STEP 6 --  Get a SQL Statement from the user.
```

```
 */
GET_SQL: PROCEDURE (S);
DCL S      CHARACTER (*) VAR,
    L      FIXED BINARY (15),
    LINE   CHARACTER (80) VAR,
    DONE   FIXED BINARY (15);

S = '';
L = 1;

PUT SKIP LIST ('DSQL>> ');

/*  A statement can occur over multiple lines.
    Keep getting input until the terminating ';' */
DONE = 0;
DO UNTIL (DONE ^= 0);
    GET EDIT (LINE) (A (80));
    DO WHILE (SUBSTR(LINE,LENGTH(LINE),1) = ' ');
        LINE = SUBSTR(LINE,1,LENGTH(LINE)-1);
    END;
    S = S || LINE || ' ';
    IF SUBSTR(LINE,LENGTH(LINE),1) = ';' THEN DO;
/*  Set "all done" flag; Throw away terminating '; '. */
        DONE = 1;
        S = SUBSTR(S,1,LENGTH(S)-2);
    END;
    ELSE
    DO;
        L = L + 1;
        PUT EDIT ( L, ' ') ( F(5), A(2) );
    END;
END;
END GET_SQL;


/*
 *  STEP 11 -- Get values for the bind variables
 */
GET_BIND_VARS: PROCEDURE (BNDDSC);

EXEC SQL INCLUDE BNDDSC;

DCL BN                  POINTER,
    BNAME CHARACTER(80)  BASED (BN),
    BV                  POINTER,
```

```
         BVAL  CHARACTER(80)  BASED (BV),
         I     FIXED BINARY (15);

PUT SKIP LIST ( 'Please enter values for Bind Vars.' );
PUT SKIP;

DO I = 1 TO BNDDSC.SQLDNUM;
/*  Display bind variable name. Use UNSPEC to get an
    integer into a pointer. */
    UNSPEC(BN) = UNSPEC(BNDDSC.SQLDSC(I).SQLDH_VNAME);
    PUT EDIT ( BN ->> BNAME, ': ' )
        (A(BNDDSC.SQLDSC(I).SQLDH_CUR_VNAMEL), A(2));

/*  Get value for this bind variable. */
    UNSPEC(BV) = UNSPEC(BNDDSC.SQLDSC(I).SQLDV);
    GET LIST ( BV ->> BVAL );

/*  Declare the bind variable to be type VARCHAR2. */
    BNDDSC.SQLDSC(I).SQLDVTYP = 1;
END;
END GET_BIND_VARS;

/*
 *  This procedure prints column headings for
    select-list items.
 */
PRINT_COL_HEADINGS: PROCEDURE (SELDSC);
EXEC SQL INCLUDE SELDSC;
DCL (I,J)    FIXED BINARY (15),
     LINE    CHARACTER (132) BASED (P),
     BLANKS  FIXED BINARY (15),
     P       POINTER;

/*
 * STEP 14 -- Readjust TYPE and LENGTH elements in the SQLDA
 *            Output column names as column headings.
 */
PUT SKIP(2);
DO I = 1 TO SELDSC.SQLDNUM;
    UNSPEC(P) = UNSPEC(SELDSC.SQLDSC(I).SQLDH_VNAME);
    IF SELDSC.SQLDSC(I).SQLDVTYP = 1 THEN DO;
/*  Have Oracle VARCHAR2 type. Left justify.
    Compute number of blanks required for padding. */
        BLANKS = MAX(SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL,
            SELDSC.SQLDSC(I).SQLDVLN) -
```

```
                             SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL;
                   PUT EDIT ( P ->> LINE, ' ' )
                       (A (SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL),
                       X(BLANKS), A(1));
              END;
              ELSE DO;
/*  Assume we have Oracle NUMBER type. Right-justify.
    Also, force column width to be the maximum
    of column heading and 9. */
                   SELDSC.SQLDSC(I).SQLDVLN =
                       MAX(SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL, 9);
                   BLANKS = SELDSC.SQLDSC(I).SQLDVLN
                       - SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL;
                   PUT EDIT ( P ->> LINE, ' ' ) ( X (BLANKS),
                       A(SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL), A(1));
/*  Coerce select-list names to
    type VARCHAR2 to simplify printing. */
                   SELDSC.SQLDSC(I).SQLDVTYP = 1;
              END;
         END;


         /*  Underline the column headings. */
         PUT SKIP;

         DO I = 1 TO SELDSC.SQLDNUM;
             IF I >> 1 THEN
                 PUT EDIT (' ') (A(1));
             IF SELDSC.SQLDSC(I).SQLDVTYP = 1 THEN
                 DO J = 1 TO MAX(SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL,
                     SELDSC.SQLDSC(I).SQLDVLN);
                     PUT EDIT ('-') (A (1));
                 END;
             ELSE
                 DO J = 1 TO MAX(SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL, 9);
                     PUT EDIT ('-') (A(1));
             END;
         END;

         PUT SKIP;
         END PRINT_COL_HEADINGS;

         /*
          * Print out some help for the user at
          * program start-up.
          */
```

```
PRINT_HELP: PROCEDURE;

PUT SKIP;
PUT SKIP LIST ( 'Terminate all SQL stmts w/'';''(semi-colon).');
PUT SKIP LIST ( 'Type ''/EXIT''; to exit DSQL' );
PUT SKIP;
END PRINT_HELP;

/*
 *  STEP 16 -- Process the select-list items.
 */
PRINT_ROW: PROCEDURE (SELDSC);
EXEC SQL INCLUDE SELDSC;

DCL BLANKS     FIXED BINARY (15),
    DI         POINTER,
    I          FIXED BINARY (15),
    LINE       CHARACTER(132) BASED(P),
    P          POINTER,
    SELDI      FIXED BINARY (15) BASED(DI);

DO I = 1 TO SELDSC.SQLDNUM;

/*  Check if the select-list item is NULL. */
    UNSPEC(DI) = UNSPEC(SELDSC.SQLDSC(I).SQLDI);
    IF DI ->> SELDI << 0 THEN

/*  This item is NULL. Set the length of
    buf to zero so output spacing (blank pad)
    comes out correctly. */
        SELDSC.SQLDSC(I).SQLDVLN = 0;

/*  Compute number of required blanks
    for appropriate spacing. */
    BLANKS = MAX( MAX(SELDSC.SQLDSC(I).SQLDH_CUR_VNAMEL,
        SELDSC.SQLDSC(I).SQLDVLN), 9) -
        SELDSC.SQLDSC(I).SQLDVLN;

/*  Print column value, with blank padding. */
    UNSPEC(P) = UNSPEC(SELDSC.SQLDSC(I).SQLDV);
    PUT EDIT ( P ->> LINE, ' ')
        (A(SELDSC.SQLDSC(I).SQLDVLN), X(BLANKS), A(1));
END;

PUT SKIP;
```

```
        END PRINT_ROW;

        /*
         *  Begin the MAIN program here.
         *
         */
        EXEC SQL WHENEVER SQLERROR GOTO LOGERR;
        TRY  = 0;

        DO UNTIL (TRY = 3);
            TRY = TRY + 1;
            PUT SKIP LIST ( 'Username: ' );
            GET EDIT (USR) (A(8));
            PUT SKIP LIST ( 'Password: ' );
            GET EDIT (PWD) (A(8));
            EXEC SQL CONNECT :USR IDENTIFIED BY :PWD;
            GOTO CONNECTED_OK;

        LOGERR:
            PUT SKIP;
            PUT SKIP LIST ( SQLCA.SQLERRM );
            PUT SKIP;
        END;   /* DO UNTIL */

        PUT SKIP LIST ( 'Aborting login after 3 attempts.' );
        RETURN;

        CONNECTED_OK:
        /*  Initialization. */
        CALL PRINT_HELP;

        /*
         *  STEP 5 -- Initialize the select and bind descriptors.
         */

        DO I = 1 TO 20;
            SELDSC.SQLDSC(I).SQLDH_MAX_VNAMEL = 80;
            EXEC Oracle IFDEF SQ3LIB;
            CALL SQ3ADR
                (SEL_DH_VNAME(I),SELDSC.SQLDSC(I).SQLDH_VNAME);
            EXEC Oracle ELSE;
            CALL SQLADR
                (ADDR(SEL_DH_VNAME(I)),
                 ADDR(SELDSC.SQLDSC(I).SQLDH_VNAME));
            EXEC Oracle ENDIF;
```

```
            SELDSC.SQLDSC(I).SQLDI_MAX_VNAMEL = 80;
            SELDSC.SQLDSC(I).SQLDVLN = 80;

            EXEC Oracle IFDEF SQ3LIB;
            CALL SQ3ADR(SEL_DV(I),SELDSC.SQLDSC(I).SQLDV);
            EXEC Oracle ELSE;
            CALL SQLADR
                (ADDR(SEL_DV(I)), ADDR(SELDSC.SQLDSC(I).SQLDV));
            EXEC Oracle ENDIF;

            SEL_DI(I) = 0;

            EXEC Oracle IFDEF SQ3LIB;
            CALL SQ3ADR(SEL_DI(I),SELDSC.SQLDSC(I).SQLDI);
            EXEC Oracle ELSE;
            CALL SQLADR
                (ADDR(SEL_DI(I)), ADDR(SELDSC.SQLDSC(I).SQLDI));
            EXEC Oracle ENDIF;

            SELDSC.SQLDSC(I).SQLDFMT   = 0;
            SELDSC.SQLDSC(I).SQLDFCLP  = 0;
            SELDSC.SQLDSC(I).SQLDFCRCP = 0;
        END;

        DO I = 1 TO 20;
            BNDDSC.SQLDSC(I).SQLDH_MAX_VNAMEL = 80;

            EXEC Oracle IFDEF SQ3LIB;
            CALL SQ3ADR
                (BND_DH_VNAME(I),BNDDSC.SQLDSC(I).SQLDH_VNAME);
            EXEC Oracle ELSE;
            CALL SQLADR
               (ADDR(BND_DH_VNAME(I)),
                ADDR(BNDDSC.SQLDSC(I).SQLDH_VNAME));
            EXEC Oracle ENDIF;

            BNDDSC.SQLDSC(I).SQLDI_MAX_VNAMEL = 80;

            EXEC Oracle IFDEF SQ3LIB;
            CALL SQ3ADR
                (BND_DI_VNAME(I),BNDDSC.SQLDSC(I).SQLDI_VNAME);
            EXEC Oracle ELSE;
            CALL SQLADR
                (ADDR(BND_DI_VNAME(I)),
```

```
                ADDR(BNDDSC.SQLDSC(I).SQLDI_VNAME));
        EXEC Oracle ENDIF;

        BNDDSC.SQLDSC(I).SQLDVLN = 80;

        EXEC Oracle IFDEF SQ3LIB;
        CALL SQ3ADR(BND_DV(I),BNDDSC.SQLDSC(I).SQLDV);
        EXEC Oracle ELSE;
        CALL SQLADR
            (ADDR(BND_DV(I)), ADDR(BNDDSC.SQLDSC(I).SQLDV));
        EXEC Oracle ENDIF;

        BND_DI(I) = 0;

        EXEC Oracle IFDEF SQ3LIB;
        CALL SQ3ADR(BND_DI(I),BNDDSC.SQLDSC(I).SQLDI);
        EXEC Oracle ELSE;
        CALL SQLADR
            (ADDR(BND_DI(I)), ADDR(BNDDSC.SQLDSC(I).SQLDI));
        EXEC Oracle ENDIF;

        BNDDSC.SQLDSC(I).SQLDFMT   = 0;
        BNDDSC.SQLDSC(I).SQLDFCLP  = 0;
        BNDDSC.SQLDSC(I).SQLDFCRCP = 0;
    END;

    /*  Main Executive Loop: Get and execute SQL statement. */

    DONE = 0;

    DO UNTIL ( DONE ^= 0 );
        EXEC SQL WHENEVER SQLERROR GOTO SQLERR;

    /*
     *  Call routine to do STEP 6.
     */
        CALL GET_SQL(STMT);
        IF STMT = '/EXIT' | STMT = '/exit' THEN
            DONE = 1;
        ELSE  DO;

    /*
     *  STEPS 7 & 8 - Prepare the SQL statement and
     *               declare a cursor for it.
     */
```

```
        EXEC SQL PREPARE S FROM :STMT;
        EXEC SQL DECLARE C CURSOR FOR S;

/*
 *  STEP 9 -- Describe the bind variables
 *           in this SQL statement.
 */
        BNDDSC.SQLDNUM = 20;
        EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO BNDDSC;

        IF BNDDSC.SQLDFND  << 0 THEN DO;
            PUT SKIP LIST ('Too many Bind Vars in this SQL stmt.');
            PUT LIST ('  Try again...');
            GOTO NXT_STM;
        END;
/*
 *  STEP 10 -- Reset N of bind variables.
 */
        BNDDSC.SQLDNUM = BNDDSC.SQLDFND;

        IF BNDDSC.SQLDNUM ^= 0 THEN
            CALL GET_BIND_VARS (BNDDSC); /* do STEP 11 */


/*
 *  STEP 12 -- Open the cursor using the bind descriptor.
 */
        EXEC SQL OPEN C USING DESCRIPTOR BNDDSC;
        SELDSC.SQLDNUM = 20;


/*
 *  STEP 13 -- Describe the select list.
 */

        EXEC SQL DESCRIBE SELECT LIST FOR S INTO SELDSC;
        IF SELDSC.SQLDFND  << 0 THEN DO;
            PUT SKIP LIST
            ('Too many Select Vars in this SQL stmt.');
            PUT LIST (' Try again...');
            GOTO NXT_STM;
        END;
        SELDSC.SQLDNUM = SELDSC.SQLDFND;

        /*  If this is a SELECT statement, then
            display rows. Else, all done... */
        IF SELDSC.SQLDNUM ^= 0 THEN DO;
```

```
                    CALL PRINT_COL_HEADINGS (SELDSC);
                    /*  Fetch each row, and print it. */
                    EXEC SQL WHENEVER NOT FOUND GOTO N_FND;
        NXT_ROW:

        /*
         *  STEP 15 -- Fetch the data into the buffers
         *            (buffers are pointed to by SELDSC).
         */
                    EXEC SQL FETCH C USING DESCRIPTOR SELDSC;
                    CALL PRINT_ROW (SELDSC);  /* do STEP 16 */
                    GOTO NXT_ROW;
        N_FND:
              END;
              IF SQLCA.SQLERRD(3) ^= 0 THEN DO;
                  PUT EDIT ( SQLCA.SQLERRD(3), ' Row(s) processed.' )
                      ( SKIP(1), F(4), A );
                  PUT SKIP;
              END;
        /*
         *  STEP 17 -- Close the cursor.
         */
              EXEC SQL CLOSE C;
           END;

           GOTO NXT_STM;

        SQLERR:
           PUT SKIP LIST ( SQLCA.SQLERRM );
           PUT SKIP;

        NXT_STM:
        END;   /* DO UNTIL */

        EXEC SQL COMMIT WORK RELEASE;
        RETURN; /* exit program */

        END DYN4DEM;
```

A

Differences from Previous Release

This appendix lists differences between Pro*PL/I precompiler release 1.5 and 1.6. Each feature modification is followed by one or more page numbers referencing the relevant section in this Manual. Bold page numbers refer to the section with the main description of the feature.

Topics

Configuration files on page 3-3, **on page 3-4**

DBMS Option on page 1-13, on page 1-15, **on page 3-5**

INLINE option **on page 3-2**

MODE Option on page 1-13, **on page 3-7**

sqlglm() **on page 2-15**

SQLSTATE variable on page 1-7, **on page 2-2**

B

Operating System Dependencies

Some details of Pro*PL/I programming vary from system to system. So, occasionally you are referred to the *Oracle Installation and User's Guide* for your system. For convenience, this appendix collects all these external references to system-specific information.

Topics

Index

E

F

FETCH statement
    use in Dynamic SQL Method 4,   5-34
file extension
    for INCLUDE files,   1-6
flags
    warning flags,   2-14
FLOAT
    Oracle external datatype,   5-16
functions
    precompiling,   3-9

G

GOTO action
    in the WHENEVER statement,   2-17

H

host arrays
    declaring,   1-13
    multidimensional,   1-14
    restrictions on,   1-14, 1-15
host variables
    attribute factoring in declaration of,   1-9
    compatibility with database objects,   1-8
    declaring,   1-8
    definition of,   1-2
    naming,   1-10
    passed to a subroutine,   1-9
    referencing,   1-10
    restrictions on,   1-16
    rules for naming,   1-2
    scope of,   1-5
    using with PL/SQL,   1-24
    where to declare,   1-5, 1-9

I

INAME option
    when a file extension is required,   3-2
INCLUDE statement
    effect of,   1-6
    not like PL/I %INCLUDE directive,   1-7
    using to declare the ORACA,   2-20
    using to declare the SQLCA,   2-13
    using to declare the SQLDA,   5-5
indicator variables,   1-12
    association with host variables,   1-12
    declaring,   1-12
    function of,   1-12
    referencing,   1-12
    required size of,   1-12
    using with PL/SQL,   1-24
insert of no rows
    cause of,   2-12
internal datatype,   5-13
    defined,   5-13
internal datatypes
    list of,   1-19
IS NULL operator
    for testing nulls,   1-3

L

LIKE attribute
    cannot be used for host variables,   1-9
linking,   3-10
LMARGIN,   3-9
logical operators,   1-3
LONG
    Oracle external datatype,   5-16
LONG RAW
    Oracle external datatype,   5-16
LONG VARRAW
    Oracle external datatype,   5-16

M

MAXLITERAL
    default value,   1-3
message text
    error message text,   2-15
MLSLABEL
    Oracle external datatype,   5-16
MODE option
    default value for,   3-7
    effects of,   1-17
    purpose of,   3-7
    syntax for,   3-7

Z