Oracle® AI Database

Pro*FORTRAN® Supplement to the Oracle Precompilers Guide

26ai
G44323-01
October 2025

Oracle AI Database Pro*FORTRAN® Supplement to the Oracle Precompilers Guide, 26ai

G44323-01

Primary Author: Jiji Thomas

Contributors: Denis Raphaely, Simon Watt, Arun Desai, Mallikharjun Vemana, Subhranshu Banerjee, Radhakrishnan Hari, Nancy Ikeda, Ken Jacobs, Tim Smith, Scott Urman

Contents

A Operating System Dependencies

Index

Preface

This companion book to the *Oracle Database Developer's Guide to the Oracle Precompilers* shows you how to write FORTRAN programs that use the powerful database language SQL to access and manipulate Oracle data. It provides examples, instructions, and programming tips, as well as several full-length programs to aid your understanding of embedded SQL and demonstrate its usefulness.

This manual, accompanied by the *Oracle Database Developer's Guide to the Oracle Precompilers*, guides you to getting the most from Pro*FORTRAN and embedded SQL.

This preface contains these topics:

Audience

This guide is intended for anyone developing new FORTRAN applications or converting existing FORTRAN applications to run in the Oracle environment. Though written specially for developers, it is also useful to systems analysts, project managers, and others interested in embedded SQL applications.

Related Documents

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

- Applications programming in FORTRAN
- The concepts, terminology, and methods discussed in the *Oracle Database Developer's Guide to the Oracle Precompilers*
- The SQL database language
- Oracle database concepts and terminology

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Writing a Pro*FORTRAN Program

This chapter provides basic information for writing a Pro*FORTRAN program:

> **Note**
>
> The Pro*Fortran Precompiler is NOT supported in 64 bit Oracle Database installations.
>
> The Pro*Fortran Precompiler is only supported and available in Oracle Database 32 bit Client installations.

This chapter contains the following topics:

1.1 Programming Guidelines

This section deals with embedded SQL syntax, coding conventions, and FORTRAN-specific features and restrictions. Topics are arranged alphabetically.

1.1.1 32 bit vs 64 bit

The Pro*Fortran Precompiler is NOT supported in 64 bit Oracle environments.

The Pro*Fortran Precompiler is only supported and available in Oracle Database 32 bit Client installations.

1.1.2 Case-sensitivity

Though the standard FORTRAN character set excludes lowercase alpha characters, many compilers allow them in identifiers, comments, and quoted literals.

The Pro*FORTRAN Precompiler is not case-sensitive; however, some compilers are. If your compiler is case-sensitive, you must declare and reference variables in the same uppercase/lowercase format. Check your FORTRAN compiler user's guide.

You must code EXEC SQL and EXEC ORACLE statements in columns 7 through 72 (columns 73 through 80 are ignored). The other columns are used for the following purposes:

- Column 1 can indicate a comment line or can be part of an optional statement label.
- Columns 2 through 5 can contain an optional statement label.
- Column 6 indicates continuation lines.

On some systems, *terminal format* is supported; that is, entry is not restricted to certain columns. In this manual, the program fragments and sample programs are in ANSI format (FORMAT=ANSI).

No more than one statement can appear on a single line.

1.1.3 Comments

You can place FORTRAN comment lines within SQL statements. FORTRAN comment lines start with the letter C or an asterisk (*) in column 1. You can place ANSI SQL-style comments (- - ...) in SQL statements at the end of a line, and you can also place C-style comments (/* ... */) in SQL statements.

The following example shows all three styles of comments:

```
EXEC SQL SELECT ENAME, SAL
C Assign column values to host variables.
 1 INTO :ENAM, :ESAL -- output host variables
 2 FROM EMP
 3 /* Use input host variable in
 4 search condition */
 5 WHERE DEPTNO = :DNUM
```

> **Note**
>
> You cannot nest comments. Blank lines are treated as comments, but are *not* allowed within a continued statement.

1.1.4 Continuation Lines

You can continue SQL statements from one line to the next, according to the rules of FORTRAN. To code a continuation line, place a nonzero, non-blank character in column 6. In this manual, digits are used as continuation characters, as the following example shows:

```
* Retrieve employee data.
 EXEC SQL SELECT EMPNO, ENAME, JOB, SAL
```

```
 1 INTO :ENUM, :ENAM, :EJOB, :ESAL
 2 FROM EMP
 3 WHERE DEPTNO = :DNUM
```

To continue a string literal from one line to the next, code the literal through column 72. On the next line, code a continuation character and the rest of the literal. An example follows:

```
* Execute dynamic SQL statement.
 EXEC SQL EXECUTE IMMEDIATE 'UPDATE EMP SET COMM = 500 WHERE
 1 DEPTNO=20'
```

Most FORTRAN implementations allow up to 19 continuation lines. Check your FORTRAN compiler user's guide.

1.1.5 Delimiters

Though FORTRAN does not require blanks to delimit keywords, you must use blanks to delimit keywords in SQL statements. FORTRAN uses apostrophes to delimit string literals, as in

```
* Display employee name.
 IF (ENAM .LT. 'ZZZZZ') THEN
 PRINT *, ' Employee Name: ', ENAM
 END IF
```

SQL also uses apostrophes to delimit string literals, as in

```
* Retrieve employee data.
 EXEC SQL SELECT ENAME, SAL
 1 INTO :ENAM, :ESAL
 2 FROM EMP
 3 WHERE JOB = 'CLERK'
```

SQL also uses quotation marks to delimit identifiers containing special or lowercase characters.

1.1.6 Embedded SQL Syntax

To use a SQL statement in your host program, precede the SQL statement with the EXEC SQL clause. Embedded SQL syntax is described in the *Pro*C/C++ Developer's Guide*. The precompiler translates all EXEC SQL statements into calls to the runtime library SQLLIB.

1.1.7 File Length

The Pro*FORTRAN Precompiler cannot process arbitrarily long source files. Some of the variables used internally limit the size of the generated file. There is no absolute limit to the number of lines allowed, but the following aspects of the source files are contributing factors to the file-size constraint:

- complexity of the embedded SQL statements (for example, the number of bind and define variables)
- whether a database name is used (for example, connecting to a database with an AT clause)
- number of embedded SQL statements

To prevent problems related to this limitation, use multiple program units to reduce the size of the source files as required.

1.1.8 File Naming Restrictions

Avoid using filenames starting with "sql," because errors might occur. For example, if you name a file SQLERROR.PFO, some linkers return name conflicts because there will be an array named SQLERD and a common block named SQLERD.

1.1.9 FORTRAN Versions

The Pro*FORTRAN Precompiler supports the standard implementation of FORTRAN for your operating system (usually FORTRAN 77). For more information, see your Oracle system-specific documentation.

1.2 Required Declarations and SQL Statements

Passing data between Oracle and your application program requires host variables and event handling. This section shows you how to meet these requirements.

1.2.1 The Declare Section

You must declare all program variables to be used in SQL statements in the *Declare Section*, which begins with the statement

```
EXEC SQL BEGIN DECLARE SECTION
```

and ends with the statement

```
EXEC SQL END DECLARE SECTION
```

Between these two statements only the following are allowed:

- host variable and indicator variable declarations
- EXEC SQL DECLARE statements
- EXEC SQL INCLUDE statements
- EXEC SQL VAR statements
- EXEC ORACLE statements
- FORTRAN comments

In a Pro*FORTRAN source file, multiple program units can contain SQL statements. So, multiple Declare Sections are allowed for each precompiled unit. Furthermore, a Pro*FORTRAN program can contain multiple files.

1.2.2 Using the INCLUDE Statement

FORTRAN INCLUDEs are processed by the FORTRAN compiler, while EXEC SQL INCLUDE statements are processed by Pro*FORTRAN to copy files into your host program, as illustrated in the following example:

```
* Copy in the SQL Communications Area (SQLCA)
* and the Oracle Communications Area (ORACA).
 EXEC SQL INCLUDE SQLCA
 EXEC SQL INCLUDE ORACA
```

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

1.2.3 Filename Extensions

If your system uses file extensions but you do not specify one, the Pro*FORTRAN Precompiler assumes the default extension for source files (usually FOR or F). The default extension is system dependent. For more information, see your Oracle system-specific documentation.

1.2.4 Search Paths

If your system uses directories, you can set a search path for INCLUDE files using the INCLUDE precompiler option, as follows:

```
INCLUDE=path
```

where *path* defaults to the current directory.

The precompiler first searches the current directory, then the directory specified by the INCLUDE option, and finally the directory for standard INCLUDE files. You need not specify a path for standard files such as the SQLCA and ORACA. However, a path is required for nonstandard files unless they are stored in the current directory.

You can also specify multiple paths on the command line, as follows:

```
... INCLUDE=<path1> INCLUDE=<path2> ...
```

When multiple paths are specified, the precompiler searches the current directory first, then the *path1* directory, then the *path2* directory, and so on. The directory containing standard INCLUDE files is searched last. The path syntax is system specific. Check the Oracle installation or user's guide for your system.

1.2.5 Caution

Remember, the precompiler searches for a file in the current directory first even if you specify a search path. If the file you want to INCLUDE is in another directory, make sure no file with the same name is in the current directory or any other directory that precedes it in the search path. Also, if your operating system is case-sensitive, you must specify the same upper or lower case filename under which the file is stored.

1.2.6 Event and Error Handling

Pro*FORTRAN provides forward and backward compatibility when checking the outcome of executing SQL statements. However, there are restrictions on using SQLCA, SQLCODE, and SQLSTATE depending on the MODE and DBMS option settings. For more information, see Error Handling and Diagnostics.

1.3 Host Variable Names

Host variable names must consist only of letters and digits, and must begin with a letter. They can be of any length, but only the first 31 characters are significant. Some compilers prohibit variable names longer than six characters, or ignore characters after the sixth. Check your FORTRAN compiler user's guide.

1.3.1 Logical and Relational Operators

Logical and relational operators are different for FORTRAN and SQL, as shown in the following tables, respectively. For example, the SQL operators do not have leading and trailing periods, as shown in table 1-1 and table 1-2.

Table 1-1 Logical Operators

SQL Operators	FORTRAN Operators
NOT	.NOT.
AND	.AND.
OR	.OR.
--	.EQV.
--	.NEQV.

Table 1-2 Relational Operator

SQL Operators	FORTRAN operators
=	.EQ.
<>, !=, ^=	.NE.
>	.GT.
<	.LT.
>=	.GE.
<=	.LE.

Logical and relational FORTRAN operators are *not* allowed in SQL statements.

1.3.2 MAXLITERAL Default

With the MAXLITERAL precompiler option, you can specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. For Pro*FORTRAN, the default value is 1000, but you might need to specify a lower value.

For example, if your FORTRAN compiler cannot handle string literals longer than 512 characters, specify MAXLITERAL=512. Check your FORTRAN compiler user's guide.

1.3.3 Nulls

In SQL, a null represents a missing, unknown, or inapplicable column value; it equates neither to zero nor to a blank. Use the NVL function to convert nulls to non-null values, use the IS [NOT] NULL comparison operator to search for nulls, and use indicator variables to insert and test for nulls.

1.3.4 Program Units

In FORTRAN, a *program unit* is a function, subroutine, or main program. In Pro*FORTRAN, an input file contains one or more program units.

If a program unit contains SQL statements, it must

- define all local host variables in its Declare Section
- INCLUDE the SQLCA when MODE={ORACLE|ANSI13}
- declare a variable named SQLATA or SQLCOD inside or outside the Declare Section when MODE={ANSI|ANSI14}
- INCLUDE the ORACA if you specify ORACA=YES

Multiple program units can contain SQL statements. For example, you can DECLARE a cursor in one program unit, OPEN it in another, FETCH from it in yet another, and CLOSE it in still another as long as they are in the same file.

1.4 Scope of Host Variables

The scoping rules for FORTRAN identifiers apply to host variables. Host variables declared in a program unit are local to that unit, and host variables declared in the main program are *not* global. So, all host variables used in a program unit must be declared in that unit in the Declare Section.

1.4.1 Statement Labels

You can associate FORTRAN numeric statement labels (1 - 99999) with SQL statements, as shown in the following example:

```
* Insert row into employee table.
 500 EXEC SQL INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO)
 1 VALUES (:ENUM, :ENAM, :EJOB, :DNUM)
```

And, you can reference statement labels in a WHENEVER DO or WHENEVER GOTO statement, as this example shows:

```
* Handle SQL execution errors.
 EXEC SQL WHENEVER SQLERROR GOTO 900
 ...
* SQLEMC stores the Oracle error code and message.
 900 WRITE (*, 8500) SQLEMC
 8500 FORMAT (1X, 70A1)
 ...
```

Statement labels must be coded in columns 1 through 5, and must *not* appear in continuation lines. Statement labels may consist of alphanumeric characters, only; the special characters, underscore (_), hyphen (-), and dollar sign ($) are not allowed.

The Pro*FORTRAN Precompiler does not use statement labels in generated code. Therefore, the BEGLABEL and ENDLABEL options that were available in earlier Pro*FORTRAN versions are not supported in this version and will return an informational message if found.

1.4.2 Statement Terminator

Embedded SQL statements are terminated by an end-of-line, as the following example shows:

```
* Delete employee.
 EXEC SQL DELETE FROM EMP WHERE EMPNO = :ENUM
```

However, a continuation character on the next line overrides an end-of-line.

1.5 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.

1.5.1 Declaring Host Variables

Host variables are declared according to FORTRAN rules, using the FORTRAN datatypes that are supported by Oracle. FORTRAN datatypes must be compatible with the source/target database column. The supported FORTRAN datatypes are shown in the following table. One-dimensional arrays of FORTRAN types are also supported.

Variable Declaration	Description
BYTE *var* CHARACTER *var*	single character
CHARACTER *var***n* CHARACTER**n var*	*n*-byte character string
CHARACTER(*) *var*	character string
INTEGER *var* INTEGER*2 *var* INTEGER*4 *var*	default-length integer 2-byte integer 4-byte integer
LOGICAL *var* LOGICAL*1 *var* LOGICAL*2 *var* LOGICAL*4 *var*	single character 2-byte character string 4-byte character string
REAL *var* REAL*4 *var* REAL*8 *var* DOUBLE PRECISION *var*	4-byte real number 8-byte real number
VARCHAR**n*	<= 32765-byte, variable length character string (3)
SQLCURSOR	cursor variable

Notes:

1. The size of FORTRAN numeric types is implementation-dependent. The sizes given in the table are typical but not universal. Check your FORTRAN compiler user's guide.
2. CHARACTER(*) variables have no predetermined length. They are used to specify dummy arguments in a subroutine declaration. The maximum length of an actual argument is returned by the LEN intrinsic function.
3. Variables declared with VARCHAR**n* (not native to FORTRAN) are assigned the VARCHAR external datatype. See "Declaring VARCHAR Variables" for more information.

The following table lists the compatible Oracle internal datatypes.

Internal Type	FORTRAN Type	Description
CHAR(*x*) (1)VARCHAR2(*y*) (1)	BYTE CHARACTER CHARACTER**n* VARCHAR**n* *var1*, *var2*, *var3*	single character variable-length string variable-length string variable-length string
NUMBER NUMBER (*p*,*s*) (2)	CHARACTER**n**var* CHARACTER *var* **n* CHARACTER(*) DOUBLE PRECISION INTEGER INTEGER*2 INTEGER*4 LOGICAL *var* LOGICAL*1 *var* LOGICAL*2 *var* LOGICAL*4 *var*REAL REAL*4 REAL*8 VARCHAR**n**var1*, *var2*, *var3*	*n*-byte character string (3) character string (as parameter) 8-byte float number integer (default size) 2-byte integer 4-byte integer single character 2-byte character string 4-byte character string float number 4-byte float number 8-byte float number variable-length string

Internal Type	FORTRAN Type	Description
DATE (4)LONG RAW (1)LONG RAW ROWID (5)MLSLABEL (6)	CHARACTER*_n var_ CHARACTER*_n var_ VARCHAR*_n var1_, _var2_, _var3_	_n_-byte character string _n_-byte variable-length string variable-length string
CURSOR	SQLCURSOR	cursor variable

Notes:

1. _x_ ranges from 1 to 255, and 1 is the default. _y_ ranges from 1 to 2000.
2. _p_ ranges from 2 to 38. _s_ ranges from -84 to 127.
3. Strings can be converted to NUMBERs only if they consist of convertible characters -- 0 to 9, period (.), +, -, E, e. The NLS settings for your system might change the decimal point from a period (.) to a comma (,).
4. When converted to a string type, the default size of a DATE depends on the NLS settings in effect on your system. When converted to a binary value, the length is 7 bytes.
5. When converted to a string type, a ROWID requires from 18 to 256 bytes.

1.5.2 Example Declarations

In the following example, several host variables are declared to be used later in a Pro*FORTRAN program:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM
 CHARACTER*10 ENAM
 REAL*4 ESAL
 INTEGER*2 DNUM
 CHARACTER*15 DNAM
 EXEC SQL END DECLARE SECTION
```

You can also declare one-dimensional arrays of FORTRAN types, as the next example shows:

```
* Declare host arrays.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM(100)
 CHARACTER*10 ENAM(100)
 REAL*4 ESAL(100)
 EXEC SQL END DECLARE SECTION
```

1.5.3 Repeating Definitions

You can use repeating definitions for datatypes, as in the following example:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 REAL*4 ESAL, ECOM, EBON
 EXEC SQL END DECLARE SECTION
```

which is equivalent to

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 ...
```

```
 REAL*4 ESAL
 REAL*4 ECOM
 REAL*4 EBON
 EXEC SQL END DECLARE SECTION
```

1.5.4 Initialization

While it is not necessary to initialize host variables inside the Declare Section, you can use the FORTRAN DATA statement to initialize host variables, as shown in the following example:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 REAL*4 MINSAL
 REAL*4 MAXSAL
 DATA MINSAL, MAXSAL /1000.00, 5000.00/
 EXEC SQL END DECLARE SECTION
```

DATA statements must come before the first executable FORTRAN statement but after any variable and PARAMETER declarations. Later in your program, you can change the values of variables initialized by a DATA statement. You cannot, however, reuse a DATA statement to reset the changed values.

1.5.5 Constants

You can use the FORTRAN PARAMETER statement inside or outside the Declare Section to assign constant values to host variables, as the following example shows:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*5 UID
 CHARACTER*5 PWD
 PARAMETER (UID = 'AAAAA', PWD = 'BBBBB')
 EXEC SQL END DECLARE SECTION
```

1.5.6 COMMON Blocks

Using the FORTRAN COMMON statement, you can keep host variables and arrays in a common storage area as if they were globally defined, so that you can use their values in different program units. The COMMON statement must appear *outside* the Declare Section, and before the first executable FORTRAN statement but after variable declarations. An example follows:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM
 CHARACTER*10 ENAM
 REAL*4 ESAL
 REAL*4 ECOM
 EXEC SQL END DECLARE SECTION
* Define COMMON block.
 COMMON /EMPBLK/ ENUM, ESAL, ECOM
```

In this example, EMPBLK is the COMMON block name. The names of COMMON blocks, subroutines, and functions are the only globally defined identifiers in a FORTRAN program. You should avoid using blank COMMON blocks.

You can make a COMMON block available to other program units by redefining it in those units. You must repeat the type declarations for variables in a COMMON block in all units where the block is used.

Only the order and datatypes of variables in the COMMON block matter, not their names. Therefore, the variable names can differ from unit to unit. However, it is good programming practice to use the same names for corresponding variables in each occurrence of a COMMON block.

The following restrictions apply to COMMON blocks:

- You cannot put VARCHAR variables in a COMMON block.
- Host arrays cannot be dimensioned in a COMMON statement.
- You cannot use a DATA statement to initialize variables in a blank COMMON block.
- With most compilers, CHARACTER variables must appear in their own COMMON blocks; that is, they cannot be mixed with other variables in a COMMON block.

1.5.7 EQUIVALENCE Statement

With the FORTRAN EQUIVALENCE statement, you can use two or more host variable names for the same storage location. The EQUIVALENCE statement must appear before the first executable FORTRAN statement.

You can equivalence CHARACTER variables only to other CHARACTER variables. You *cannot* equivalence VARCHAR variables.

1.5.8 Special Requirements for Subroutines

You must explicitly declare host variables in the Declare Section of the program unit that uses them in SQL statements. Thus, variables passed to a subroutine and used in SQL statements within the subroutine must be declared in the subroutine Declare Section, as illustrated in the following example:

```
...
 CALL LOGON (UID, PWD)
 ...
 SUBROUTINE LOGON (UID, PWD)
* Declare host variables in subroutine.
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 UID
 CHARACTER*10 PWD
 EXEC SQL END DECLARE SECTION
 ...
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
 WRITE(*, 1000) UID
 1000 FORMAT(/,' Connected to Oracle as user: ', A10, /)
 RETURN
 END
```

1.5.9 Restrictions

The following restrictions apply with respect to Declarations:

Implicit Declarations

FORTRAN allows implicit declaration of INTEGER and REAL variables. Unless explicitly declared otherwise, identifiers starting with I, J, K, L, M, or N are assumed to be of type INTEGER, and other identifiers are assumed to be of type REAL.

However, implicit declaration of host variables is *not* allowed; it triggers an "undeclared host variable" error message at precompile time. Every variable referenced in a SQL statement must be defined in the Declare Section.

Complex Numbers

These are numbers including a real and an imaginary part. In FORTRAN, complex numbers are represented using the datatype COMPLEX. Pro*FORTRAN, however, does *not* support the use of COMPLEX host variables in SQL statements.

1.6 About Referencing Host Variables

You use host variables in SQL data manipulation statements. A host variable must be prefixed with a colon (:) in SQL statements but must not be prefixed with a colon in FORTRAN statements, as shown in the following example:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM
 CHARACTER*10 ENAM
 REAL*4 ESAL
 CHARACTER*10 EJOB
 EXEC SQL END DECLARE SECTION
 ...
 WRITE (*, 3100)
 3100 FORMAT (' Enter employee number: ')
 READ (*, 3200) ENUM
 3200 FORMAT (I4)
 EXEC SQL SELECT ENAME, SAL, JOB
 1 INTO :ENAM, :ESAL, :EJOB
 2 FROM EMP
 3 WHERE EMPNO = :ENUM
 BONUS = ESAL / 10
 ...
```

Though it might be confusing, you can provide the same name to a host variable as that of an Oracle table or column, as the following example shows:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM
 CHARACTER*10 ENAM
 REAL*4 ESAL
 EXEC SQL END DECLARE SECTION
 ...
 EXEC SQL SELECT ENAME, SAL
 1 INTO :ENAM, :ESAL
 2 FROM EMP
 3 WHERE EMPNO = :ENUM
```

1.6.1 Restrictions

A host variable cannot substitute for a column, table, or other Oracle objects in a SQL statement and must not be an Oracle reserved word. See Oracle Reserved Words, Keywords, and Namespaces for a list of Oracle reserved words and keywords.

1.7 Indicator Variables

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

You use indicator variables in the VALUES or SET clauses to assign nulls to input host variables and in the INTO clause to detect nulls or truncated values in output host variables.

1.7.1 Declaring Indicator Variables

An indicator variable must be explicitly declared in the Declare Section as a 2-byte integer (INTEGER*2) and must not be an Oracle reserved word. In the following example, you declare two indicator variables (the names IESAL and IECOM are arbitrary):

```
* Declare host and indicator variables.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM
 CHARACTER*10 ENAM
 REAL*4 ESAL
 REAL*4 ECOM
 INTEGER*2 IESAL
 INTEGER*2 IECOM
 EXEC SQL END DECLARE SECTION
```

You can define an indicator variable anywhere in the Declare Section. It need not follow its associated host variable.

1.7.2 About Referencing Indicator Variables

In SQL statements, an indicator variable must be prefixed with a colon and appended to its associated host variable. In FORTRAN statements, an indicator variable must *not* be prefixed with a colon or appended to its associated host variable. An example follows:

```
* Retrieve employee data.
 EXEC SQL SELECT SAL, COMM
 1 INTO :ESAL, :ECOM:IECOM
 2 FROM EMP
 3 WHERE EMPNO = :ENUM
* When an indicator variable equals -1, its associated
* host variable is null.
 IF (IECOM .EQ. -1) THEN
 PAY = ESAL
 ELSE
 PAY = ESAL + ECOM
 END IF
```

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 the expression in your host program.

1.7.3 Restrictions

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.
 IECOM = -1
 EXEC SQL DELETE FROM EMP WHERE COMM = :ECOM:IECOM
```

The correct syntax follows:

```
EXEC SQL DELETE FROM EMP WHERE COMM IS NULL
```

1.7.4 Oracle Restrictions

When DBMS=V6, Oracle does not issue an error if you SELECT or FETCH a null into a host variable 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
```

When precompiling with MODE=ORACLE and DBMS=V7, you can disable the ORA-01405 message by also specifying UNSAFE_NULL=YES on the command line. For more information, see UNSAFE_NULL.

1.7.5 ANSI Requirements

When MODE=ORACLE, if you SELECT or FETCH a truncated column value into a host variable 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. Values for indicator variables are discussed in Meeting Program Requirements.

1.8 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.

1.8.1 About Declaring Host Arrays

You declare and dimension host arrays in the Declare Section. In the following example, three host arrays are declared, each with an upper dimension bound of 50 (the lower bound defaults to 1):

```
* Declare and dimension host arrays.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM(50)
```

```
 CHARACTER*10 ENAM(50)
 REAL*4 ESAL(50)
 EXEC SQL END DECLARE SECTION
```

1.8.2 Restrictions

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

```
* Invalid dimensioning of host array
 EXEC SQL BEGIN DECLARE SECTION
 ...
 REAL*4 VECTOR(0:10)
 EXEC SQL END DECLARE SECTION
```

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

```
* Invalid declaration of host array
 EXEC SQL BEGIN DECLARE SECTION
 ...
 REAL*4 MATRIX(50, 100)
 EXEC SQL END DECLARE SECTION
```

You cannot dimension host arrays using the FORTRAN DIMENSION statement. For example, the following usage is *invalid*:

```
* Invalid use of DIMENSION statement
 EXEC SQL BEGIN DECLARE SECTION
 REAL*4 ESAL
 REAL*4 ECOM
 DIMENSION ESAL(50), ECOM(50)
 EXEC SQL END DECLARE SECTION
```

Also, you cannot dimension a host array in a COMMON statement.

1.8.3 About Referencing Host Arrays

If you use multiple host arrays in a single SQL statement, their dimensions should be the same. However, this is not a requirement because the Pro*FORTRAN Precompiler always uses the *smallest* dimension for the SQL operation. In the following example, only 50 rows are INSERTed:

```
* Declare host arrays.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM(100)
 CHARACTER*10 ENAM(100)
 INTEGER*4 DNUM(100)
 REAL*4 ECOM(50)
 EXEC SQL END DECLARE SECTION
 ...
* Populate host arrays here.
 ...
 EXEC SQL INSERT INTO EMP (EMPNO, ENAME, COMM, DEPTNO)
 1 VALUES (:ENUM, :ENAM, :ECOM, :DNUM)
```

Host arrays must *not* be subscripted in SQL statements. For example, the following INSERT statement is invalid:

```
* Declare host arrays.
 EXEC SQL BEGIN DECLARE SECTION
```

```
 INTEGER*4 ENUM(50)
 REAL*4 ESAL(50)
 INTEGER*4 DNUM(50)
 EXEC SQL END DECLARE SECTION
 ...
 DO 200 J = 1, 50
* Invalid use of host arrays
 EXEC SQL INSERT INTO EMP (EMPNO, SAL, DEPTNO)
 1 VALUES (:ENUM(J), :ESAL(J), :DNUM(J))
 200 CONTINUE
```

You need not process host arrays in a loop. Instead, use 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, see Using Host Arrays.

1.8.4 About 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 with indicator arrays:

```
* Declare host and indicator arrays.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM(50)
 INTEGER*4 DNUM(50)
 REAL*4 ECOM(50)
 INTEGER*2 IECOM(50) -- indicator array
 EXEC SQL END DECLARE SECTION
 ...
* Populate the host and indicator arrays. To insert
* a null into the COMM column, assign -1 to the
* appropriate element in the indicator array.
 ...
 EXEC SQL INSERT INTO EMP (EMPNO, DEPTNO, COMM)
 1 VALUES (:ENUM, :DNUM, :ECOM:IECOM)
```

The dimension of the indicator array must be greater than, or equal to, the dimension of the host array.

1.9 VARCHAR Host Variables

FORTRAN string datatypes are of fixed length. However, Pro*FORTRAN lets you declare a variable-length string pseudotype called VARCHAR.

1.9.1 About Declaring VARCHAR Variables

A VARCHAR is a set of three variables declared using the syntax

```
* Declare a VARCHAR.
 EXEC SQL BEGIN DECLARE SECTION
 VARCHAR*<n> <VARNAM>, <VARLEN>, <VARARR>
 EXEC SQL END DECLARE SECTION
```

where:

`n`

Is the maximum length of the VARCHAR; *n* must be in the range 1 through 32765.

VARNAM

Is the name used to reference the VARCHAR in SQL statements; it is called an *aggregate name* because it identifies a set of variables.

VARLEN

Is a 2-byte signed integer variable that stores the actual length of the string variable.

VARARR

Is the string variable used in FORTRAN statements.

The advantage of using VARCHAR variables is that you can explicitly set and reference VARLEN. With input host variables, Oracle reads the value of VARLEN and uses the same number of characters of VARARR. With output host variables, Oracle sets VARLEN to the length of the character string stored in VARARR.

You can declare a VARCHAR only in the Declare Section. In the following example, you declare a VARCHAR named EJOB with a maximum length of 15 characters:

```
* Declare a VARCHAR.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 VARCHAR*15 EJOB, EJOBL, EJOBA
 EXEC SQL END DECLARE SECTION
```

The precompiler expands this declaration to

```
* Expanded VARCHAR declaration
 INTEGER*2 EJOBL
 LOGICAL*1 EJOBA(15)
 INTEGER*2 SQXXX(2)
 EQUIVALENCE (SQXXX(1), EJOBL), (SQXXX(2), EJOBA(1))
```

where SQXXX is an array generated by the precompiler and XXX denotes three arbitrary characters. Notice that the aggregate name EJOB is not declared. The EQUIVALENCE statement forces the compiler to store EJOBL and EJOBA contiguously.

1.9.2 About Referencing VARCHAR Variables

In SQL statements, you can reference a VARCHAR variable by using the aggregate name prefixed with a colon, as the following example shows:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 INTEGER*4 ENUM
 VARCHAR*15 EJOB, EJOBL, EJOBA
 EXEC SQL END DECLARE SECTION
 ...
 EXEC SQL SELECT JOB
 1 INTO :EJOB
 2 FROM EMP
 3 WHERE EMPNO = :ENUM
```

After the query executes, EJOBL holds the actual length of the character string retrieved from the database and stored in EJOBA. In FORTRAN statements, you reference VARCHAR variables using the length variable and string variable names, as this example shows:

```
* Display job title.
 WRITE (*, 5200) (EJOBA(J), J = 1, EJOBL)
 5200 FORMAT (15A1)
 ...
```

1.9.3 About Overcoming the Length Limit

Recall that the length variable of a VARCHAR must be a 2-byte integer. FORTRAN provides a 2-byte signed integer datatype, which can represent numbers in the range -32768 through 32767. However, FORTRAN lacks a 2-byte *unsigned* integer datatype, which can represent numbers in the range 0 through 65535. Therefore, the maximum length of a VARCHAR character string is 32765 bytes (32767 minus 2 for the length variable).

With other host languages, the maximum length of a VARCHAR character string is 65533 bytes. If you want to use 65533-byte VARCHAR variables, try the technique shown in the following example:

```
* Declare a VARCHAR.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 VARCHAR*65533 BUF, BUFL, BUFA
 EXEC SQL END DECLARE SECTION
 ...
* Equivalence two 2-byte integers to one 4-byte integer.
 INTEGER*2 INT2(2)
 INTEGER*4 INT4
 EQUIVALENCE (INT2(1), INT4)
 INTEGER*4 I
 ...
 INT4 = 65533
* Set the VARCHAR length variable equal to the
* equivalenced value of INT4.
 BUFL = INT2(1)
 DO 100 I = 1, 65533
 BUFA(I) = 32
 100 CONTINUE
 EXEC SQL INSERT INTO LONG_TABLE VALUES (:BUF)
 ...
 BUFL = 0
 EXEC SQL SELECT COL1 INTO :BUF FROM LONG_TABLE
 INT2(1) = BUFL
 ...
```

> **Note**
>
> The way integers are stored varies from system to system. On some systems, the least significant digits are stored at the low address; on other systems they are stored at the high address. In the last example, this determines whether the length is stored in INT2(1) or INT2(2).

1.10 About Handling Character Data

This section explains how the Pro*FORTRAN Precompiler handles character host variables. There are two types of character host variables:

- CHARACTER*_n_

- VARCHAR

Do not confuse VARCHAR, which is a host variable data structure supplied by the precompiler, with VARCHAR2, which is an Oracle column datatype for variable-length character strings.

1.10.1 Effects of the MODE Option

The MODE option has the following effects:

- It determines how the Pro*FORTRAN Precompiler treats data in character arrays and strings. The MODE option allows the program to use ANSI fixed-length strings or to maintain compatibility with previous versions of the Oracle Server and the Pro*FORTRAN Precompiler.
- With respect to character handling, MODE={ANSI14|ANSI13} is equivalent to MODE=ORACLE. The MODE option affects character data on input (from host variables to Oracle) and on output (from Oracle to host variables).

> **Note**
>
> The MODE option does not affect the way Pro*FORTRAN handles VARCHAR host variables.

1.10.2 CHARACTER*n

Character variables are declared using the CHARACTER*n datatype. These types of variables handle character data based on their roles as input or output variables.

1.10.3 On Input

When MODE=ORACLE, the program interface strips trailing blanks before sending the value to the database. If you insert into a fixed-length CHAR column, Oracle re-appends trailing blanks up to the length of the database column. However, if you insert into a variable-length VARCHAR2 column, Oracle never appends blanks.

When MODE=ANSI, trailing blanks are never stripped.

Make sure that the input value is not trailed by extraneous characters. For example, nulls are not stripped and are inserted into the database. Normally, this is not a problem because when a value is READ into or assigned to a CHARACTER*n variable, FORTRAN appends blanks up to the length of the variable.

The following example illustrates the point:

```
* Declare host variables
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER ENAM *10, EJOB *8
 ...
 EXEC SQL END DECLARE SECTION
 ...
 WRITE (*, 300)
 300 FORMAT (/, '$Employee name? ')
* Assume the name 'MILLER' is entered
 READ (*, 400)
 400 FORMAT (A10)
 EJOB = 'SALES'
```

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

If you precompile the last example with MODE=ORACLE and the target database columns are VARCHAR2, the program interface strips the trailing blanks on input and inserts just the 6-character string "MILLER" and the 5-character string "SALES" into the database. However, if the target database columns are CHAR, the strings are blank-padded to the width of the columns.

If you precompile the last example with MODE=ANSI and the JOB column is defined as CHAR(10), the value inserted into that column is "SALES#####" (five trailing blanks). However, if the JOB column is defined as VARCHAR2(10), the value inserted is "SALES###" (three trailing blanks) because the host variable is a CHARACTER*8. This might not be what you want, so be careful.

1.10.4 On Output

The MODE option has no effect on output to character variables. When you use a CHARACTER*_n_ variable as an output host variable, Oracle blank-pads it. In our example, when your program fetches the string "MILLER" from the database, ENAM contains the value "MILLER####" (with four trailing blanks). This character string can be used without change as input to another SQL statement.

1.10.5 VARCHAR Variables

VARCHAR variables handle character data based on their roles as input or output variables

1.10.6 On Input

When you use a VARCHAR variable as an input host variable, your program must assign values to the length and string variables, as shown in the following example:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM
 VARCHAR*15 EJOB, EJOBL, EJOBA
 INTEGER*2 IEJOB
 INTEGER*4 DNUM
 EXEC SQL END DECLARE SECTION
 ...
 WRITE (*, 4300)
 4300 FORMAT (/, ' Enter job title: ')
 READ (*, 4400) EJOBA
 4400 FORMAT (15A1)
* Scan backward for last non-blank character, then
* set length to that position. If input is all blank,
* set indicator variable to -1 to indicate a null.
 DO 5000 J = 15, 1, -1
 IF (EJOBA(J) .NE. ' ') GOTO 5100
 5000 CONTINUE
 J = 0
 5100 IF (J .EQ. 0) THEN
 IEJOB = -1
 ELSE
 IEJOB = 0
 END IF
 EJOBL = J
```

```
EXEC SQL INSERT INTO EMP (EMPNO, JOB, DEPTNO)
1 VALUES (:ENUM, :EJOB:IEJOB, :DNUM)
```

1.10.7 On Output

When you use a VARCHAR variable as an output host variable, Oracle sets the length variable. An example follows:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 INTEGER*4 ENUM
 VARCHAR*15 EJOB, EJOBL, EJOBA
 INTEGER*4 ESAL
 EXEC SQL END DECLARE SECTION
 ...
 EXEC SQL SELECT JOB, SAL INTO :EJOB, :ESAL FROM EMP
 1 WHERE EMPNO = :ENUM
 ...
 IF (EJOBL .EQ. 0) GOTO ...
 ...
```

An advantage of VARCHAR variables over fixed-length strings is that the length of the value returned by Oracle is available immediately. With fixed-length strings, to get the length of the value, your program must count the number of characters. (The intrinsic function LEN returns the length of a string including blanks, not its current length.)

1.11 The Oracle Datatypes

Oracle recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle stores data in database columns. Oracle also uses internal datatypes to represent database pseudocolumns. An external datatype specifies how data is stored in a host variable. For descriptions of the Oracle datatypes, see Oracle Datatypes.

1.11.1 Internal Datatypes

For values stored in database columns, Oracle uses the following internal datatypes:

Name	Code	Description
CHAR	96	<= 255-byte, fixed-length string
DATE	12	7-byte, fixed-length date/time value
LONG	8	<= 2147483647-byte, variable-length string
LONG RAW	24	<= 2147483647-byte, variable-length binary data
MLSLABEL	105	<= 5-byte, variable-length binary label
NUMBER	2	fixed or floating point number
RAW	23	<= 255-byte, variable-length binary data
ROWID	11	fixed-length binary value
VARCHAR2	1	<= 2000-byte, variable-length string

Table 1 - 5. Internal Datatypes

These internal datatypes can be quite different from FORTRAN datatypes. For example, FORTRAN has no equivalent to the NUMBER datatype, which was specially designed for portability and high precision.

1.11.2 External Datatypes

As the following table shows, the external datatypes include all the internal datatypes plus several datatypes found in other supported host languages. For example, the STRING external datatype refers to a C null-terminated string. You use the datatype names in datatype equivalencing, and you use the datatype codes in dynamic SQL Method 4.

Name	Code	Description
CHAR	1 96	<= 65535-byte, variable-length character string (1)<= 65535-byte, fixed-length character string (1)
CHARF	96	<= 65535-byte, fixed-length character string
CHARZ	97	<= 65535-byte, fixed-length, null-terminated string (2)
DATE	12	7-byte, fixed-length date/time value
DECIMAL	7	COBOL packed decimal
DISPLAY	91	COBOL numeric character string
FLOAT	4	4-byte or 8-byte floating-point number
INTEGER	3	2-byte or 4-byte signed integer
LONG	8	<= 2147483647-byte, fixed-length string
LONG RAW	24	<= 217483647-byte, fixed-length binary data
LONG VARCHAR	94	<= 217483643-byte, variable-length string
LONG VARRAW	95	<= 217483643-byte, variable-length binary data
MLSLABEL	106	2..5-byte, variable-length binary data
NUMBER	2	integer or floating-point number
RAW	23	<= 65535-byte, fixed-length binary data (2)
ROWID	11	(typically) 13-byte, fixed-length binary value
STRING	5	<= 65535-byte, null-terminated character string (2)
UNSIGNED	68	2-byte or 4-byte unsigned integer
VARCHAR	9	<= 65533-byte, variable-length character string

Name	Code	Description
VARCHAR2	1	<= 65535-byte, variable-length character string (2)
VARNUM	6	variable-length binary number
VARRAW	15	<= 65533-byte, variable-length binary data

Notes:

1. CHAR is datatype 1 when MODE={ORACLE|ANSI13|ANSI14} and datatype 96 when MODE=ANSI.
2. Maximum size is 32767 (32K) on some platforms.

1.12 Datatype Conversion

At precompile time, an external datatype is assigned to each host variable in the Declare Section. For example, the precompiler assigns the FLOAT external datatype to host variables of type REAL. 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 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 INTEGER*2 value. You cannot, however, convert a CHAR value of "65543" (number too large) or "10F" (number not decimal) to a INTEGER*2 value. Likewise, you cannot convert a CHARACTER*_n_ value that contains alphabetic characters to a NUMBER value.

For more information about datatype conversion, see Meeting Program Requirements.

1.13 Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle interprets input data and the way Oracle formats output data. You can equivalence supported FORTRAN datatypes to Oracle external datatypes on a variable-by-variable basis.

1.13.1 Host Variable Equivalencing

By default, the Pro*FORTRAN Precompiler assigns a specific external datatype to every host variable. The default assignments are shown in the following table. For more information about datatype equivalencing, see Datatype Equivalencing.

Host Type	External Type	Code
BYTE *var* LOGICAL *var* LOGICAL*1 *var* LOGICAL*2 *var* LOGICAL*4 *var* CHARACTER *var* CHARACTER *var*n* CHARACTER*n *var* CHARACTER(*) *var*	VARCHAR2 CHARF	1 (when MODE != ANSI) 96 (when MODE=ANSI)

Host Type	External Type	Code
VARCHAR*n*	VARCHAR	9
INTEGER *var* INTEGER*2 *var* INTEGER*4 *var*	INTEGER	3
REAL *var* REAL*4 *var* REAL*8 *var* DOUBLE PRECISION *var*	FLOAT	4

With 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 <ext_type_name> [({<length> | <precision>,<scale>})]
```

where *host_variable* is an input or output host variable (or host array) declared earlier in the Declare Section, *ext_type_name* is the name of a valid external datatype, and *length* is an integer literal specifying a valid length in bytes.

When *ext_type_name* is FLOAT, use *length*; when *ext_type_name* is DECIMAL, you must specify *precision* and *scale* instead of *length*.

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
 INTEGER*4 ENUM(50)
 ...
* Reset default datatype (INTEGER) to RAW.
 EXEC SQL VAR ENUM IS RAW (200);
 EXEC SQL END DECLARE SECTION
 ...
```

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

For more information about datatype equivalencing, see Datatype Equivalencing.

1.14 Embedding PL/SQL

The Pro*FORTRAN Precompiler treats a PL/SQL block like a single embedded SQL statement. As a result, 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, declare the variables to be shared with PL/SQL and bracket the PL/SQL block with the EXEC SQL EXECUTE and END-EXEC keywords.

1.14.1 Host Variables

Inside a PL/SQL block, host variables are 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.

1.14.2 VARCHAR Variables

When entering a PL/SQL block, Oracle automatically checks the length fields of VARCHAR host variables. So, you must set the length fields *before* the block is entered. For input variables, set the length field to the length of the value stored in the string field. For output variables, set the length field to the maximum length allowed by the string field.

1.14.3 Indicator Variables

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

1.14.4 About Handling Nulls

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

1.14.5 About 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.

1.14.6 SQLCHECK

You must specify SQLCHECK=SEMANTICS when precompiling a program with an embedded PL/SQL block. You must also use the USERID option. For more information, see SQLCHECK.

1.14.7 Cursor Variables

Starting with Release 1.7 of the Pro*FORTRAN Precompiler, you can use *cursor variables* in your Pro*FORTRAN programs to process multi-row queries using static embedded SQL. A cursor variable identifies a *cursor reference* that is defined and opened on the Oracle Database Server, using PL/SQL. See the *Oracle Database PL/SQL Language Reference* for complete information about cursor variables.

The advantages of cursor variables are:

- *Encapsulation*: queries are centralized in the stored procedure that opens the cursor variable.
- *Ease of maintenance*: only the stored procedure needs to be changed if the table changes.
- *Security*: the user of the application (the username when the Pro*FORTRAN application connected to the database) must have execute permission on the stored procedure that opens the cursor. This user, however, does not need to have read permission on the tables used in the query. This capability can be used to limit access to the columns in the table.

1.15 About Declaring a Cursor Variable

You declare a Pro*FORTRAN cursor variable using the SQLCURSOR pseudotype. For example:

```
EXEC SQL BEGIN DECLARE SECTION
 ...
 SQLCURSOR CURVAR
 ...
 EXEC SQL END DECLARE SECTION
```

A SQLCURSOR variable is implemented using a FORTRAN INTEGER*4 array in the code that Pro*FORTRAN generates. A cursor variable is just like any other Pro*FORTRAN host variable.

1.15.1 About Allocating a Cursor Variable

Before you can OPEN or FETCH a cursor variable, you must allocate it by using the Pro*FORTRAN ALLOCATE command. For example, to allocate the cursor variable CURVAR that was declared in the previous section, write the following statement:

```
EXEC SQL ALLOCATE :CURVAR
```

Allocating a cursor variable does *not* require a call to the server, either at precompile time or at run time.

> ⚠ **Caution**
>
> Allocating a cursor variable *does* cause heap memory to be used. For this reason, avoid allocating a cursor variable in a program loop.

1.15.2 About Opening a Cursor Variable

You must use an embedded anonymous PL/SQL block to open a cursor variable on the Oracle Server. The anonymous PL/SQL block may open the cursor either indirectly by calling a PL/SQL stored procedure that opens the cursor (and defines it in the same statement) or directly from the Pro*FORTRAN program.

1.15.3 About Opening Indirectly through a Stored PL/SQL Procedure

Consider the following PL/SQL package stored in the database:

```
CREATE PACKAGE demo_cur_pkg AS
 TYPE EmpName IS RECORD (name VARCHAR2(10));
 TYPE cur_type IS REF CURSOR RETURN EmpName;
 PROCEDURE open_emp_cur (
 curs IN OUT curtype,
 dept_num IN NUMBER);
END;
CREATE PACKAGE BODY demo_cur_pkg AS
 CREATE PROCEDURE open_emp_cur (
 curs IN OUT curtype,
 dept_num IN NUMBER) IS
 BEGIN
 OPEN curs FOR
```

```
 SELECT ename FROM emp
 WHERE deptno = dept_num
 ORDER BY ename ASC;
 END;
END;
```

After this package has been stored, you can open the variable *curs* by calling the *open_emp_cur* stored procedure from your Pro*FORTRAN program, and FETCH from the cursor variable ECUR in the program. For example:

```
EXEC SQL BEGIN DECLARE SECTION
 SQLCURSOR ECUR
 INTEGER*4 DNUM
 VARCHAR*10 ENAM, ENAML, ENAMA
 EXEC SQL END DECLARE SECTION
 ...
* Allocate the cursor variable.
 EXEC SQL ALLOCATE :ECUR
 ...
 DNUM=30
* Open the cursor on the Oracle Database Server.
 EXEC SQL EXECUTE
 1 BEGIN
 2 demo_cur_pkg.open_emp_cur(:ECUR, :DNUM);
 3 END;
 4 END-EXEC
 EXEC SQL WHENEVER NOTFOUND DO CALL SIGNOFF
*
 1000 EXEC SQL FETCH :ECUR INTO :ENAM
 PRINT *, "Employee Name: ", ENAM
 GOTO 1000
 ...
```

1.15.4 About Opening Directly from Your Pro*FORTRAN Application

To open a cursor by using a PL/SQL anonymous block in a Pro*FORTRAN program, define the cursor in the anonymous block. Consider the following example:

```
EXEC SQL EXECUTE
 1 BEGIN
 2 OPEN :ECUR FOR SELECT ENAME FROM EMP
 3 WHERE DEPTNO = :DNUM;
 4 END;
 5 END-EXEC
 ...
```

1.15.5 Return Types

When you define a reference cursor (REF CURSOR) in a PL/SQL stored procedure, you must declare the type that the cursor returns. The return types allowed for reference cursors are described in the *PL/SQL User's Guide and Reference*.

1.15.6 About Fetching from a Cursor Variable

Use the embedded SQL FETCH .... INTO command to retrieve the rows SELECTed when you opened the cursor variable. For example:

```
EXEC SQL FETCH :ECUR INTO :EINFO:IEINFO
```

Before you can FETCH from a cursor variable, the variable must be initialized and opened. You cannot FETCH from an unopened cursor variable.

1.15.7 About Closing a Cursor Variable

Use the embedded SQL CLOSE command to close a cursor variable. For example:

```
EXEC SQL BEGIN DECLARE SECTION
* Declare the cursor variable.
 SQLCURSOR ECUR
 ...
 EXEC SQL END DECLARE SECTION
* Allocate and open the cursor variable, then
* fetch one or more rows.
 ...
* Close the cursor variable.
 EXEC SQL CLOSE :ECUR
```

1.15.8 Restrictions

The following restrictions apply to the use of cursor variables:

- You can only use cursor variables with the ALLOCATE, FETCH, and CLOSE commands. The DECLARE CURSOR command does *not* apply to cursor variables.
- You cannot FETCH from a CLOSEd or unALLOCATEd cursor variable.
- If you precompile with MODE=ANSI, it is an error to close a cursor variable that is already closed.
- You cannot use the AT clause with the ALLOCATE command.

1.15.9 Error Conditions

Do *not* perform any of the following operations:

- FETCH from a closed cursor variable
- use a cursor variable that is not ALLOCATEd
- CLOSE a cursor variable that is not open

These operations on cursor variables result in errors.

1.15.10 Sample Programs

The following sample programs -- a SQL script (SAMPLE11.SQL) and a Pro*FORTRAN program (SAMPLE11.PFO) -- demonstrate how you can use cursor variables in Pro*FORTRAN.

> **Note**
>
> For simplicity in demonstrating this feature, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

1.15.11 SAMPLE11.SQL

Following is the PL/SQL source code for a creating a package that declares and opens a cursor variable:

```
CONNECT AAAAA/BBBBB
CREATE OR REPLACE PACKAGE emp_demo_pkg AS
 TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
 PROCEDURE open_cur (
 cursor IN OUT emp_cur_type,
 dept_num IN number);
END emp_demo_pkg;
/
CREATE OR REPLACE PACKAGE BODY emp_demo_pkg AS
 PROCEDURE open_cur (
 cursor IN OUT emp_cur_type,
 dept_num IN number) IS
 BEGIN
 OPEN cursor FOR SELECT * FROM emp
 WHERE deptno = dept_num
 ORDER BY ename ASC;
 END;
END emp_demo_pkg;
/
```

1.15.12 SAMPLE11.PFO

Following is a Pro*FORTRAN sample program that uses the cursor declared in the SAMPLE11.SQL example to fetch employee names, salaries, and commissions from the EMP table.

```
PROGRAM SAMPLE11
 EXEC SQL BEGIN DECLARE SECTION
* Declare the cursor variable.
 SQLCURSOR ECUR
* EMPINFO
 INTEGER ENUM
 CHARACTER*10 ENAM
 VARCHAR*9 EJOB, EJOBL, EJOBA
 INTEGER EMGR
 VARCHAR*10 EDAT, EDATL, EDATA
 REAL ESAL
 REAL ECOM
 INTEGER EDEP
* EMPINFO INDICATORS
 INTEGER*2 IENUM
 INTEGER*2 IENAM
 INTEGER*2 IEJOB
 INTEGER*2 IEMGR
 INTEGER*2 IEDAT
 INTEGER*2 IESAL
 INTEGER*2 IECOM
 INTEGER*2 IEDEP
 EXEC SQL END DECLARE SECTION
 EXEC SQL INCLUDE SQLCA
 COMMON /CURSOR/ ECUR
 EXEC SQL WHENEVER SQLERROR DO CALL SQLERR

* LOG ON TO ORACLE.
 CALL LOGON
```

```
* Initialize the cursor variable.
 EXEC SQL ALLOCATE :ECUR
 TYPE 1000
 1000 FORMAT (/, 'Enter department number (0 to exit): ', $)
 ACCEPT 1100, EDEP
 1100 FORMAT (I10)
 IF (EDEP .LE. 0) THEN
 CALL SIGNOFF
 ENDIF

* Open the cursor by calling a PL/SQL stored procedure.
 EXEC SQL EXECUTE
 1 BEGIN
 2 emp_demo_pkg.open_cur (:ECUR, :EDEP);
 3 END;
 4 END-EXEC
 PRINT 1200, EDEP
 1200 FORMAT (/, 'For department ', I, ':',/)
 PRINT 1300
 1300 FORMAT (/, 'EMPLOYEE SALARY COMMISSION',
 + /, '---------- ---------- ----------')

* Fetch data from the cursor into the host variables.
 2000 EXEC SQL WHENEVER NOT FOUND DO CALL SIGNOFF
 EXEC SQL FETCH :ECUR
 1 INTO :ENUM:IENUM,
 2 :ENAM:IENAM,
 3 :EJOB:IEJOB,
 4 :EMGR:IEMGR,
 5 :EDAT:IEDAT,
 6 :ESAL:IESAL,
 7 :ECOM:IECOM,
 8 :EDEP:IEDEP

* Check for commission and print results.
 IF (IECOM .EQ. 0) THEN
 PRINT 2100, ENAM, ESAL, ECOM
 2100 FORMAT (A10, 2X, F10.2, 2X, F10.2)
 ELSE
 PRINT 2200, ENAM, ESAL
 2200 FORMAT (A10, 2X, F10.2, 2X, ' N/A')
 ENDIF
 GOTO 2000
 END

* LOG ON TO ORACLE.
 SUBROUTINE LOGON
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 UID
 CHARACTER*10 PWD
 EXEC SQL END DECLARE SECTION
 EXEC SQL INCLUDE SQLCA
 UID = 'AAAAA'
 PWD = 'BBBBB'
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
 PRINT 3000, UID
 3000 FORMAT (/, 'CONNECTED TO ORACLE AS USER: ', A)
 END
* Close the cursor variable.
 SUBROUTINE SIGNOFF
 EXEC SQL BEGIN DECLARE SECTION
```

```
SQLCURSOR ECUR
EXEC SQL END DECLARE SECTION
EXEC SQL INCLUDE SQLCA
COMMON /CURSOR/ ECUR
EXEC SQL CLOSE :ECUR
PRINT 4100
4100 FORMAT (/, 'HAVE A GOOD DAY.', /)
EXEC SQL COMMIT WORK RELEASE
STOP
END

SUBROUTINE SQLERR
EXEC SQL INCLUDE SQLCA
EXEC SQL WHENEVER SQLERROR CONTINUE
PRINT*, ' '
PRINT *, 'ORACLE ERROR DETECTED: '
PRINT '(70A1)', SQLEMC
PRINT*, ' '
EXEC SQL ROLLBACK WORK RELEASE
STOP
END
```

1.16 Connecting to Oracle

Your Pro*FORTRAN program must log on to Oracle before querying or manipulating data. To log on, you use the CONNECT statement, as in

```
* Log on to Oracle.
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
```

where UID and PWD are CHARACTER or VARCHAR host variables. Alternatively, you can use the statement

```
* Log on to Oracle.
 EXEC SQL CONNECT :UIDPWD
```

where the host variable UIDPWD contains your username and password separated by a slash (/).

The CONNECT statement must be the first SQL statement executed by the program. That is, other executable SQL statements can positionally, but not logically, precede the CONNECT statement.

To supply the Oracle username and password separately, you define two host variables in the Declare Section as character strings or VARCHAR variables. If you supply a userid containing both username and password, only one host variable is needed.

Make sure to set the username and password variables before the CONNECT is executed or it will fail. Your program can prompt for the values or you can hard code them as follows:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*5 UID
 CHARACTER*5 PWD
 ...
 EXEC SQL END DECLARE SECTION
 UID = 'AAAAA'
 PWD = 'BBBBB'
* Handle logon errors.
 EXEC SQL WHENEVER SQLERROR GOTO ...
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
```

However, you cannot hard code a username and password into the CONNECT statement or use quoted literals. For example, both of the following statements are *invalid*:

```
* Invalid CONNECT statements
 EXEC SQL CONNECT AAAAA IDENTIFIED BY BBBBB
 EXEC SQL CONNECT 'AAAAA' IDENTIFIED BY 'BBBBB'
```

1.16.1 Automatic Logons

You can automatically log on to the Oracle using the following userid:

```
<prefix><username>
```

where *prefix* is the value of the Oracle initialization parameter OS_AUTHENT_PREFIX (the default value is OPS$) and *username* is your operating system user or task name. For example, if the prefix is OPS$, your user name is TBARNES, and OPS$TBARNES is a valid Oracle userid, you log on to Oracle as user OPS$TBARNES.

To take advantage of the automatic logon feature, you simply pass a slash (/) character to the precompiler, as follows:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 CHARACTER*1 ORAID
 EXEC SQL END DECLARE SECTION
 ORAID = '/'
 EXEC SQL CONNECT :ORAID
```

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

You can also pass a character string to the precompiler. However, the string cannot contain trailing blanks. For example, the following CONNECT statement will fail:

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 CHARACTER*5 ORAID
 EXEC SQL END DECLARE SECTION
 ORAID = '/ '
 EXEC SQL CONNECT :ORAID
```

For more information about operating system authentication, see the *Oracle Database Administrator's Guide*.

1.16.2 Concurrent Logons

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

```
* Declare host variables.
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*5 UID
 CHARACTER*5 PWD
 CHARACTER*12 DBSTR1
 CHARACTER*12 DBSTR2
 EXEC SQL END DECLARE SECTION
```

```
 UID = 'AAAAA'
 PWD = 'BBBBB'
 DBSTR1 = 'NEWYORK'
 DBSTR2 = 'BOSTON'
* 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 :UID IDENTIFIED BY :PWD
 1 AT DBNAM1 USING :DBSTR1
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
 1 AT DBNAM2 USING :DBSTR2
```

The string syntax in DBSTR1 and DBSTR2 depends on your network driver and how it is configured. DBNAM1 and DBNAM2 name the nondefault connections; they can be undeclared identifiers or host variables.

For step-by-step instructions on connecting to Oracle through Oracle Net Services, see Meeting Program Requirements.

2

Error Handling and Diagnostics

This chapter discusses error reporting and recovery as it applies to Pro*FORTRAN. It supplements Chapter 8 of the *Developer's Guide to the Oracle Precompilers*.

You learn how to declare and use the SQLSTA status variable and the SQLCOD status variable, and how to include the SQL Communications Area (SQLCA). You also learn how to declare and enable the Oracle Communications Area (ORACA).

This chapter contains the following sections:

2.1 Error Handling Alternatives

The Pro*FORTRAN Precompiler supports four status variables that serve as error handling mechanisms:

- SQLCOD
- SQLSTA
- SQLCA (using the WHENEVER statement)
- ORACA

The precompiler MODE option governs ANSI/ISO compliance. The availability of the SQLCOD, SQLSTA, and SQLCA variables depends on the MODE setting. You can declare and use the ORACA variable regardless of the MODE setting. For more information, see "About Using the Oracle Communications Area".

When MODE={ORACLE|ANSI13}, you must declare the SQLCA status variable. SQLCOD and SQLSTA declarations are accepted (not recommended) but are not recognized as status variables. For more information, see "About Using the SQL Communications Area".

When MODE={ANSI|ANSI14}, you can use any one, two, or all three of the SQLCOD, SQLSTA, and SQLCA variables. To determine which variable (or variable combination) is best for your application, see "About Using Status Variables when MODE={ANSI|ANSI14}".

2.1.1 SQLCOD and SQLSTA

SQLCOD stores error codes and the "not found" condition. It is retained only for compatibility with SQL-89 and has been removed in SQL:1999 and subsequent versions of the standard.

Unlike SQLCOD, SQLSTA stores error and warning codes and uses a standardized coding scheme. After executing a SQL statement, the Oracle server returns a status code to the SQLSTA variable currently in scope. The status code indicates whether a SQL statement executed successfully or raised an exception (error or warning condition). To promote

interoperability (the ability of systems to exchange information easily), the SQL standard predefines all the common SQL exceptions.

2.1.2 SQLCA

The SQLCA is a record-like, host-language data structure. Oracle updates the SQLCA after every *executable* SQL statement. (SQLCA values are undefined after a declarative statement.) By checking Oracle return codes stored in the SQLCA, your program can determine the outcome of a SQL statement. This can be done in two ways:

- implicit checking with the WHENEVER statement
- explicit checking of SQLCA variables

You can use WHENEVER statements, code explicit checks on SQLCA variables, or do both. Generally, using WHENEVER statements is preferable because it is easier, more portable, and ANSI-compliant.

2.1.3 ORACA

When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA, which contains cursor statistics, SQL statement data, option settings, and system statistics.

The ORACA is optional and can be declared regardless of the MODE setting. For more information about the ORACA status variable, see "About Using the Oracle Communications Area".

2.2 About Using Status Variables when MODE={ANSI|ANSI14}

When MODE={ANSI|ANSI14}, you must declare at least one -- you may declare two or all three -- of the following status variables:

- SQLCOD
- SQLSTA
- SQLCA

Your program can retrieve the outcome of the most recent executable SQL statement by checking SQLCOD or SQLSTA explicitly with your own code after checking executable SQL and PL/SQL statements. Your program can also check SQLCA implicitly (with the WHENEVER SQLERROR and WHENEVER SQLWARNING statements) or it can check the SQLCA variables explicitly.

> **Note**
>
> When MODE={ORACLE|ANSI13}, you must declare the SQLCA status variable. For more information, see Using the SQL Communications Area.

2.2.1 Some Historical Information

The treatment of status variables and variable combinations by the Oracle Pro*FORTRAN Precompiler has evolved beginning with Release 1.5.

2.2.2 Release 1.5

Pro*FORTRAN Release 1.5 presumed that there was a status variable SQLCOD whether or not it was declared in a Declare Section; in fact, the precompiler never noted whether SQLCOD was declared or not -- it just presumed it was. SQLCA would be used as a status variable only if there was an INCLUDE of the SQLCA.

2.2.3 Release 1.6

Beginning with Pro*FORTRAN Release 1.6, the precompiler no longer presumes that there is a SQLCOD status variable and it is not required. The precompiler requires that *at least* one of SQLCA, SQLCOD, or SQLSTA be declared.

SQLCOD is recognized as a status variable if and only if at least one of the following criteria is satisfied:

- It is declared in a Declare Section with *exactly* the right datatype.
- The precompiler finds no other status variable.

If the precompiler finds a SQLSTA declaration (of *exactly* the right type of course) in a Declare Section or finds an INCLUDE of the SQLCA, it will *not* presume SQLCOD is declared.

2.2.4 Release 1.7

Because Pro*FORTRAN Release 1.5 allowed the SQLCOD variable to be declared outside of a Declare Section while also declaring SQLCA, Pro*FORTRAN Release 1.6 and greater is presented with a compatibility problem. A new option, ASSUME_SQLCODE={YES|NO} (default NO), was added to fix this in Release 1.6.7 and is documented as a new feature in Release 1.7.

When ASSUME_SQLCODE=YES, and when SQLSTA and/or SQLCA are declared as a status variables, the precompiler presumes SQLCOD is declared whether or not it is declared in a Declare Section or of the proper type. This causes Releases 1.6.7 and later to act like Release 1.5 in this regard. For information about the precompiler option ASSUME_SQLCODE, see ASSUME_SQLCODE.

2.2.5 About Declaring Status Variables

This section describes how to declare SQLCOD and SQLSTA. For information about declaring the SQLCA status variable, see "About Declaring the SQLCA".

2.2.6 Declaring SQLCOD

SQLCOD must be declared as a 4-byte integer variable either *inside* or *outside* the Declare Section, In the following example, SQLCOD is declared outside the Declare Section:

```
* Declare host and indicator variables.
 EXEC SQL BEGIN DECLARE SECTION
 ...
 EXEC SQL END DECLARE SECTION
* Declare status variable.
 INTEGER*4 SQLCOD
```

If declared outside the Declare Section, SQLCOD is recognized as a status variable only if ASSUME_SQLCODE=YES. When MODE={ORACLE|ANSI13|ANSI14}, declarations of the SQLCOD variable are ignored.

Access to a local SQLCOD is limited by its scope within your program. After every SQL operation, Oracle returns a status code to the SQLCOD currently in scope. So, your program can learn the outcome of the most recent SQL operation by checking SQLCOD explicitly, or implicitly with the WHENEVER statement.

When you declare SQLCOD instead of the SQLCA in a particular compilation unit, the precompiler allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA.

2.2.7 Declaring SQLSTA

SQLSTA *must* be declared as a *five*-character alphanumeric string *inside* the Declare Section, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION
 ...
 CHARACTER*5 SQLSTA
 ...
 EXEC SQL END DECLARE SECTION
```

When MODE={ORACLE|ANSI13}, SQLSTA declarations are ignored. Declaring the SQLCA is optional.

2.2.8 Status Variable Combinations

When MODE={ANSI|ANSI14}, the behavior of the status variables depends on the following:

- which variables are declared
- declaration placement (*inside* or *outside* the Declare Section)
- ASSUME_SQLCODE setting

The following tables describe the resulting behavior of each status variable combination when ASSUME_SQLCODE=NO and when ASSUME_SQLCODE=YES, respectively.

Declare Section (IN/OUT/ --) SQLCODE	Declare Section (IN/OUT/ --) SQLSTA	Declare Section (IN/OUT/ --) SQLCA	Behavior
OUT	na	na	SQLCOD is declared and is presumed to be a status variable.
OUT	na	OUT	SQLCA is declared as a status variable, and SQLCOD is declared but is not recognized as a status variable.
OUT	na	IN	This status variable configuration is not supported.

Declare Section (IN/OUT/ --) SQLCODE	Declare Section (IN/OUT/ --) SQLSTA	Declare Section (IN/OUT/ --) SQLCA	Behavior
OUT	OUT	na	SQLCOD is declared and is presumed to be a status variable, and SQLSTA is declared but is not recognized as a status variable.
OUT	OUT	OUT	SQLCA is declared as a status variable, and SQLCOD and SQLSTA are declared but are not recognized as status variables.
OUT	OUT	IN	This status variable configuration is not supported.
OUT	IN	na	SQLSTA is declared as a status variable, and SQLCOD is declared but is not recognized as a status variable.
OUT	IN	OUT	SQLSTA and SQLCA are declared as status variables, and SQLCOD is declared but is not recognized as a status variable.
OUT	IN	IN	This status variable configuration is not supported.
IN	na	na	SQLCOD is declared as a status variable.
IN	na	OUT	SQLCOD and SQLCA are declared as a status variables.
IN	na	IN	This status variable configuration is not supported.
IN	OUT	na	SQLCOD is declared as a status variable, and SQLSTA is declared but is not recognized as a status variable.
IN	OUT	OUT	SQLCOD and SQLCA are declared as a status variables, and SQLSTA is declared but is not recognized as a status variable.
IN	OUT	IN	This status variable configuration is not supported.

Declare Section (IN/OUT/ --) SQLCODE	Declare Section (IN/OUT/ --) SQLSTA	Declare Section (IN/OUT/ --) SQLCA	Behavior
IN	IN	na	SQLCOD and SQLSTA are declared as a status variables.
IN	IN	OUT	SQLCOD, SQLSTA, and SQLCA are declared as a status variables.
IN	IN	IN	This status variable configuration is not supported.
na	na	na	This status variable configuration is not supported.
na	na	OUT	SQLCA is declared as a status variable.
na	na	IN	This status variable configuration is not supported.
na	OUT	na	This status variable configuration is not supported.
na	OUT	OUT	SQLCA is declared as a status variable, and SQLSTA is declared but is not recognized as a status variable.
na	OUT	IN	This status variable configuration is not supported.
na	IN	na	SQLSTA is declared as a status variable.
na	IN	OUT	SQLSTA and SQLCA are declared as status variables.
na	IN	IN	This status variable configuration is not supported.

Declare Section (IN/OUT/ --) SQLCODE	Declare Section (IN/OUT/ --) SQLSTA	Declare Section (IN/OUT/ --) SQLCA	Behavior
OUT	na	na	SQLCODE is declared and is presumed to be a status variable.
OUT	na	OUT	SQLCA is declared as a status variable, and SQLCODE is declared and is presumed to be a status variable.
OUT	na	IN	This status variable configuration is not supported.

Declare Section (IN/OUT/ --) SQLCODE	Declare Section (IN/OUT/ --) SQLSTA	Declare Section (IN/OUT/ --) SQLCA	Behavior
OUT	OUT	na	SQLCODE is declared and is presumed to be a status variable, and SQLSTA is declared but is not recognized as a status variable.
OUT	OUT	OUT	SQLCA is declared as a status variable, SQLCODE is declared and is presumed to be a status variable, and SQLSTA is declared but is not recognized as status variable.
OUT	OUT	IN	This status variable configuration is not supported.
OUT	IN	na	SQLSTA is declared as a status variable, and SQLCODE is declared and is presumed to be a status variable.
OUT	IN	OUT	SQLSTA and SQLCA are declared as status variables, and SQLCODE is declared and is presumed to be a status variable.
OUT	IN	IN	This status variable configuration is not supported.
IN	na	na	SQLCODE is declared as a status variable.
IN	na	OUT	SQLCODE and SQLCA are declared as a status variables.
IN	na	IN	This status variable configuration is not supported.
IN	OUT	na	SQLCODE is declared as a status variable, and SQLSTA is declared but not as a status variable.
IN	OUT	OUT	SQLCODE and SQLCA are declared as a status variables, and SQLSTA is declared but is not recognized as a status variable.
IN	OUT	IN	This status variable configuration is not supported.

Declare Section (IN/OUT/ --) SQLCODE	Declare Section (IN/OUT/ --) SQLSTA	Declare Section (IN/OUT/ --) SQLCA	Behavior
IN	IN	na	SQLCODE and SQLSTA are declared as a status variables.
IN	IN	OUT	SQLCODE, SQLSTA, and SQLCA are declared as a status variables.
IN	IN	IN	This status variable configuration is not supported.
na	na	na	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.
na	na	OUT	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.
na	na	IN	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.
na	OUT	na	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.
na	OUT	OUT	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.

Declare Section (IN/OUT/ --) SQLCODE	Declare Section (IN/OUT/ --) SQLSTA	Declare Section (IN/OUT/ --) SQLCA	Behavior
na	OUT	IN	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.
na	IN	na	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.
na	IN	OUT	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.
na	IN	IN	This status variable configuration is not supported. SQLCODE must be declared either inside or outside the Declare Section when ASSUME_SQLCODE=YES.

2.3 About Using the SQL Communications Area

Oracle uses the SQL Communications Area (SQLCA) to store status information passed to your program at run time. The SQLCA is a record-like, FORTRAN data structure that is updated after each executable SQL statement, so it 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 FORTRAN code or implicitly with the WHENEVER statement.

When MODE={ORACLE|ANSI13}, the SQLCA is required; if the SQLCA is not declared, compile-time errors will occur. The SQLCA is optional when MODE={ANSI|ANSI14}, but you cannot use the WHENEVER SQLWARNING statement without the SQLCA. So, if you want to use the WHENEVER SQLWARNING statement, you must declare the SQLCA.

When MODE={ANSI|ANSI14}, you must declare either SQLSTA (see "Declaring SQLSTA") or SQLCOD (see "Declaring SQLCOD") or both. The SQLSTA status variable supports the SQLSTA status variable specified by the SQL standard. You can use the SQLSTA status variable with or without SQLCOD. For more information see Error Handling and Diagnostics.

2.3.1 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.

Figure 2-1 shows all the variables in the SQLCA. However, SQLWN2, SQLWN5, SQLWN6, SQLWN7, and SQLEXT are not currently in use.

Figure 2-1 SQLCA Variable Declarations for Pro*FORTRAN

```
      LOGICAL*1    SQLAID(8)
      INTEGER*4    SQLABC
      INTEGER*4    SQLCDE
*     SQLERRM
      INTEGER*2    SQLEML
      LOGICAL*1    SQLEMC(70)
      LOGICAL*1    SQLERP(8)
      INTEGER*4    SQLERD(6)
*     SQLWRN(8)
      LOGICAL*1    SQLWN0, SQLWN1, SQLWN2, SQLWN3,
     1             SQLWN4, SQLWN5, SQLWN6, SQLWN7
      LOGICAL*1    SQLEXT(8)

      COMMON /SQLCA/
     1   SQLAID,
     2   SQLABC,
     3   SQLCDE,
     4   SQLEML,
     5   SQLEMC,
     6   SQLERP,
     7   SQLERD,
     8   SQLWN0, SQLEN1, SQLWN2, SQLWN3,
     9   SQLWN4, SQLEN5, SQLWN6, SQLWN7,
     1   SQLEXT

      INTEGER*4    DSC2N
      INTEGER*4    DSC2V
      INTEGER*4    DSC2L
      INTEGER*4    DSC2T
      INTEGER*4    DSC2I
      INTEGER*4    DSC2F
      INTEGER*4    DSC2S
      INTEGER*4    DSC2M
      INTEGER*4    DSC2C
      INTEGER*4    DSC2X
      INTEGER*4    DSC2Y
      INTEGER*4    DSC2Z

      COMMON /DSC2/   DSC2N, DSC2V, DSC2L, DSC2T, DSC2I, DSC2F
     1                DSC2S, DSC2M, DSC2C, DSC2X, DSC2Y, DSC2Z
```

To ensure portability, LOGICAL variables are used in the SQLCA instead of CHARACTER variables. For a full description of the SQLCA, its fields, and the values its fields can store, see Error Handling and Diagnostics.

2.3.2 About Declaring the SQLCA

To declare the SQLCA, simply include it (using an EXEC SQL INCLUDE statement) in your Pro*FORTRAN source file outside the Declare Section as follows:

```
* Include the SQL Communications Area (SQLCA).
 EXEC SQL INCLUDE SQLCA
```

Because it is a COMMON block, the SQLCA must be declared *outside* the Declare Section. Furthermore, the SQLCA must come before the CONNECT statement and the first executable FORTRAN statement.

You must declare the SQLCA in each subroutine and function that contains SQL statements. Every time a SQL statement in one of the subroutines or functions is executed, Oracle updates the SQLCA held in the COMMON block.

Ordinarily, only the order and datatypes of variables in a COMMON-list matter, not their names. However, you cannot rename the SQLCA variables because the precompiler generates code that refers to them. Thus, all declarations of the SQLCA must be identical.

When you precompile your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow Oracle to communicate with the program.

2.3.3 Key Components of Error Reporting

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

2.3.4 Status Codes

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

2.3.5 Warning Flags

Warning flags are returned in the SQLCA variables SQLWN0 through SQLWN7, which you can check with WHENEVER SQLWARNING or with your own FORTRAN code. These warning flags are useful for detecting runtime conditions that are not considered errors by Oracle.

2.3.6 Rows-Processed Count

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

2.3.7 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 SQLERD(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 tenth character.

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

2.3.8 Error Message Text

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

```
Handle SQL execution errors.
 WRITE (*, 10000) SQLEMC
10000 FORMAT (1X, 70A1)
```

```
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, which is discussed next.

2.3.9 About 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 function. If connected to Oracle, you can call SQLGLM using the syntax

```
CALL SQLGLM (MSGBUF, BUFLEN, MSGLEN)
```

where:

`MSGBUF`

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

`BUFLEN`

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

`MSGLEN`

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

The maximum length of an Oracle error message is 512 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 BUFLEN. In the following example, you use SQLGLM to get an error message of up to 200 characters in length:

```
* Declare variables for function call.
 LOGICAL*1 MSGBUF(200)
 INTEGER*4 BUFLEN
 INTEGER*4 MSGLEN
 DATA BUFLEN /200/
 EXEC SQL WHENEVER SQLERROR GO TO 9000
 ...
* Handle SQL execution errors.
 9000 WRITE (*,9100)
 9100 FORMAT (1X, ' >>> Oracle error detected', /)
* Get and display the full text of the error message.
 CALL SQLGLM (MSGBUF, BUFLEN, MSGLEN)
 WRITE (*, 9200) (MSGBUF(J), J = 1, MSGLEN)
 9200 FORMAT (1X, 200A1, /)
 ...
```

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

2.3.10 About Using the WHENEVER Statement

By default, the Pro*FORTRAN 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 subroutine, branching to a labeled statement, or stopping.

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, which are described in Error Handling and Diagnostics:

- 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 *subroutine_call*
- GOTO *statement_label*
- STOP

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

In the following example, WHENEVER SQLERROR DO statements are used to handle specific errors:

```
EXEC SQL WHENEVER SQLERROR DO CALL INSERR
 EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
 VALUES (:MYEMPNO, :MYENAME, :MYDEPTNO)
 EXEC SQL WHENEVER SQLERROR DO CALL DELERR
 EXEC SQL DELETE FROM DEPT
 WHERE DEPTNO = :MYDEPTNO
 ...
* Error-handling subroutines
 SUBROUTINE INSERR
* Check for "duplicate key value" Oracle error.
 IF (SQLCDE .EQ. -1) THEN
 ...
* Check for "value too large" Oracle error.
 ELSE IF (SQLCDE .EQ. -1401) THEN
 ...
 ELSE
 ...
 END IF
 ...
 SUBROUTINE DELERR
* Check for the number of rows processed.
 IF (SQLERD(3) .EQ. 0) THEN
```

```
...
ELSE
...
END IF
...
```

Notice how the subroutines check variables in the SQLCA to determine a course of action. For more information about the WHENEVER conditions and actions, see Error Handling and Diagnostics.

2.3.11 Scope

Because WHENEVER is a declarative statement, its scope is positional, not logical. It tests all executable SQL statements that follow it in the source file, not in the flow of program logic. 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.

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

2.3.12 Careless Usage: Examples

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

```
* Improper use of WHENEVER
 EXEC SQL WHENEVER NOT FOUND GOTO 7000
 6000 EXEC SQL FETCH EMPCUR INTO :MYENAME, :MYSAL
 ...
 GOTO 6000
 7000 EXEC SQL DELETE FROM EMP WHERE EMPNO = :MYEMPNO
 ...
```

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 7000
 6000 EXEC SQL FETCH EMPCUR INTO :MYENAME, :MYSAL
 ...
 GOTO 6000
 7000 EXEC SQL WHENEVER NOT FOUND GOTO 8000
 EXEC SQL DELETE FROM EMP WHERE EMPNO = :MYEMPNO
 ...
 8000 CONTINUE
```

Verify that all SQL statements governed by a WHENEVER ... GOTO statement can branch to the GOTO label. The following code results in a compilation error because the label 5000 in subroutine DELROW is not within the scope of the INSERT statement in subroutine INSROW:

```
SUBROUTINE DELROW
 ...
 EXEC SQL WHENEVER SQLERROR GOTO 5000
 EXEC SQL DELETE FROM EMP WHERE DEPTNO = :MYDEPTNO
 ...
 5000 WRITE (*, 10000) SQLEMC
10000 FORMAT (1X, 70A1)
 RETURN
 END
 SUBROUTINE INSROW
 ...
 EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
 VALUES (:MYEMPNO, :MYENAME, :MYDEPTNO)
 ...
```

2.4 About Using the Oracle Communications Area

The SQLCA handles standard SQL communications. The Oracle Communications Area (ORACA) is a similar structure that you can include in 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.

2.4.1 What's in the ORACA?

The ORACA contains option settings, system statistics, and extended diagnostics. Figure 2-2 shows all the variables in the ORACA.

Figure 2-2 ORACA Variable Declarations for Pro*FORTRAN

```
      LOGICAL*1    ORAAID(8)
      INTEGER*4    ORAABC
      INTEGER*4    ORACHF
      INTEGER*4    ORADBF
      INTEGER*4    ORAHPF
      INTEGER*4    ORATXF
      INTEGER*2    ORATXL
      LOGICAL*1    ORATXC(70)
      INTEGER*1    ORAFNL
      LOGICAL*1    ORAFNC(70)
      INTEGER*4    ORASLN
      INTEGER*4    ORAHOC
      INTEGER*4    ORAMOC
      INTEGER*4    ORACOC
      INTEGER*4    ORANOR
      INTEGER*4    ORANPR
      INTEGER*4    ORANEX

      COMMON /ORACA/
     1  ORAAID,ORAABC,
     2  ORACHF,ORADBF,ORAHPF,ORATXF
     3  ORATXL,ORATXC,ORAFNL,ORAFNC,ORASLN,
     4  ORAHOC,ORAMOC,ORACOC,ORANOR,ORANPR,ORANEX
```

To ensure portability, LOGICAL variables are used in the ORACA instead of CHARACTER variables. For a full description of the ORACA, its fields, and the values its fields can store, see Chapter 8 of the *Developer's Guide to the Oracle Precompilers*.

2.4.2 About Declaring the ORACA

To declare the ORACA, simply include it (using an EXEC SQL INCLUDE statement) in your Pro*FORTRAN source file outside the Declare Section as follows:

```
* Include the Oracle Communications Area (ORACA).
 EXEC SQL INCLUDE ORACA
```

Because it is a COMMON block, the ORACA must be declared *outside* the Declare Section. Furthermore, the ORACA must come before the CONNECT statement and the first executable FORTRAN statement.

You can redeclare the ORACA in any subroutine or function that contains SQL statements. Every time a SQL statement in the subroutine or function is executed, Oracle updates the ORACA held in COMMON.

Ordinarily, only the order and datatypes of variables in a COMMON-list matter, not their names. However, you *cannot* rename the ORACA variables because the precompiler generates code that refers to them. Thus, all declarations of the ORACA must be identical.

2.4.3 About Enabling the ORACA

To enable the ORACA, you must set the ORACA precompiler option to YES on the command line or in a configuration file 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.

3
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*FORTRAN programming and demonstrate techniques that let you take full advantage of SQL's power and flexibility.

Each sample program in this chapter is available online. Table 3-1 shows the usual filenames of the sample programs. However, the exact filenames are system-dependent. For specific filenames, see your Oracle system-specific documentation.

This chapter contains the following sections:

Table 3-1 Pro*FORTAN Sample Programs

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

3.1 Sample Program 1: Simple 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. The program ends when the user enters a zero employee number.

```
PROGRAM QUERY

 EXEC SQL BEGIN DECLARE SECTION
```

```
      CHARACTER*10 UID
      CHARACTER*10 PWD
      INTEGER EMPNO
      CHARACTER*10 ENAME
      REAL SAL
      REAL COMM
      INTEGER*2 ICOMM
      EXEC SQL END DECLARE SECTION

      INTEGER TOTAL

      EXEC SQL INCLUDE SQLCA
      EXEC SQL WHENEVER SQLERROR DO CALL SQLERR

*     LOG ON TO ORACLE.
      UID = 'SCOTT'
      PWD = 'TIGER'
      EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
      PRINT *, 'CONNECTED TO ORACLE AS USER: ', UID

*     QUERY LOOP REPEATS UNTIL THE USER ENTERS A 0
      TOTAL = 0
2000  CONTINUE

      PRINT *, '\NENTER EMPLOYEE NUMBER (0 TO QUIT): '
      READ '(I10)', EMPNO
      IF (EMPNO .EQ. 0) CALL SIGNOFF (TOTAL)

      EXEC SQL WHENEVER NOT FOUND GOTO 7000
      EXEC SQL SELECT ENAME, SAL, COMM
      1 INTO :ENAME, :SAL, :COMM:ICOMM
      2 FROM EMP
      3 WHERE EMPNO = :EMPNO

      PRINT *, 'EMPLOYEE SALARY COMMISSION\N',
      +'---------- ------- ----------'


     IF (ICOMM .EQ. -1) THEN
      PRINT '(A10, 2X, F7.2, A12)', ENAME, SAL, ' NULL'
      ELSE
      PRINT '(A10, 2X, F7.2, 5X, F7.2)', ENAME, SAL, COMM
      END IF

      TOTAL = TOTAL + 1
      GOTO 2000

     7000 CONTINUE

      PRINT *, 'NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.'
      GOTO 2000
      END

      SUBROUTINE SIGNOFF (NUMQ)
      INTEGER NUMQ
      EXEC SQL INCLUDE SQLCA
      PRINT *, 'TOTAL NUMBER QUERIED WAS: ', NUMQ
      PRINT *, 'HAVE A GOOD DAY.'
      EXEC SQL COMMIT WORK RELEASE
      STOP
      END
```

```
      SUBROUTINE SQLERR
      EXEC SQL INCLUDE SQLCA
      EXEC SQL WHENEVER SQLERROR CONTINUE
      PRINT *, 'ORACLE ERROR DETECTED:'
      PRINT '(70A1)', SQLEMC
      EXEC SQL ROLLBACK WORK RELEASE
      STOP
      END
```

3.2 Sample Program 2: Cursor Operations

This program connects to Oracle, declares and opens a cursor, fetches the names, salaries, and commissions of all salespeople, displays the results, then closes the cursor.

```
      PROGRAM CURSOR

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*10 UID
      CHARACTER*10 PWD
      CHARACTER*10 ENAME
      REAL SAL
      REAL COMM
      EXEC SQL END DECLARE SECTION

      EXEC SQL INCLUDE SQLCA
      EXEC SQL WHENEVER SQLERROR DO CALL SQLERR

*     LOG ON TO ORACLE.
      UID = 'SCOTT'
      PWD = 'TIGER'
      EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
      PRINT *, 'CONNECTED TO ORACLE AS USER:', UID

*     DECLARE THE CURSOR.
      EXEC SQL DECLARE SALESPEOPLE CURSOR FOR
      1 SELECT ENAME, SAL, COMM
      2 FROM EMP
      3 WHERE JOB LIKE 'SALES%'
      EXEC SQL OPEN SALESPEOPLE

      PRINT *, 'SALESPERSON SALARY COMMISSION\N',
      +'----------- ------- ----------'

*     LOOP, FETCHING ALL SALESPERSON'S STATISTICS
      EXEC SQL WHENEVER NOT FOUND DO CALL SIGNOFF
3000  EXEC SQL FETCH SALESPEOPLE INTO :ENAME, :SAL, :COMM
      PRINT '(1X, A10, 3X, F7.2, 5X, F7.2)', ENAME, SAL, COMM
      GOTO 3000
      END

      SUBROUTINE SIGNOFF
      EXEC SQL INCLUDE SQLCA
      EXEC SQL CLOSE SALESPEOPLE
      PRINT *, 'HAVE A GOOD DAY.'
      EXEC SQL COMMIT WORK RELEASE
      STOP
      END

      SUBROUTINE SQLERR
      EXEC SQL INCLUDE SQLCA
      EXEC SQL WHENEVER SQLERROR CONTINUE
```

```
 PRINT *, 'ORACLE ERROR DETECTED:'
 PRINT '(70A1)', SQLEMC
 EXEC SQL ROLLBACK WORK RELEASE
 STOP
 END
```

3.3 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 subroutine PRTRES.

```
PROGRAM ARRAYS

 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 UID
 CHARACTER*10 PWD
 CHARACTER*10 ENAME(5)
 INTEGER EMPNO(5)
 REAL SAL(5)
 EXEC SQL END DECLARE SECTION

* NUMBER OF ROWS RETURNED, AND NUMBER TO PRINT
 INTEGER NUMRET
 INTEGER NUMP
 EXEC SQL INCLUDE SQLCA
 EXEC SQL WHENEVER SQLERROR DO CALL SQLERR

* LOG ON TO ORACLE.
 UID = 'SCOTT'
 PWD = 'TIGER'
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
 PRINT *, 'CONNECTED TO ORACLE AS USER: ', UID

* DECLARE THE CURSOR, THEN OPEN IT.
 EXEC SQL DECLARE C1 CURSOR FOR
 1 SELECT EMPNO, ENAME, SAL
 2 FROM EMP
 EXEC SQL OPEN C1
 NUMRET = 0

* LOOP, FETCHING AND PRINTING BATCHES,
* UNTIL NOT FOUND BECOMES TRUE.
 EXEC SQL WHENEVER NOT FOUND GOTO 3000
2000 EXEC SQL FETCH C1 INTO :EMPNO, :ENAME, :SAL
 NUMP = SQLERD(3) - NUMRET
 CALL PRTRES (NUMP, EMPNO, ENAME, SAL)
 NUMRET = SQLERD(3)
 GOTO 2000

* PRINT FINAL SET OF ROWS, IF ANY.
3000 NUMP = SQLERD(3) - NUMRET
 IF (NUMP .GT. 0) CALL PRTRES (NUMP, EMPNO, ENAME, SAL)
 CALL SIGNOFF
 END
 SUBROUTINE PRTRES (NUMP, EMPNO, ENAME, SAL)
 INTEGER NUMP
 INTEGER EMPNO(NUMP)
 CHARACTER*10 ENAME(NUMP)
 REAL SAL(NUMP)

* PRINT HEADER.
```

```
 PRINT *, 'EMPLOYEE NUMBER EMPLOYEE NAME SALARY\N',
 +'--------------- ------------ -------'

* PRINT BATCH OF ROWS.
 DO 7000 I = 1, NUMP
 PRINT '(1X, I4, 13X, A10, 5X, F7.2)',
 + EMPNO(I), ENAME(I), SAL(I)
7000 CONTINUE
 RETURN
 END

 SUBROUTINE SIGNOFF
 EXEC SQL INCLUDE SQLCA
 EXEC SQL CLOSE C1
 PRINT *, 'HAVE A GOOD DAY.'
 EXEC SQL COMMIT WORK RELEASE
 STOP
 END

 SUBROUTINE SQLERR
 EXEC SQL INCLUDE SQLCA
 EXEC SQL WHENEVER SQLERROR CONTINUE
 PRINT *, 'ORACLE ERROR DETECTED:'
 PRINT '(70A1)', SQLEMC
 EXEC SQL ROLLBACK WORK RELEASE
 STOP
 END
```

3.4 Sample Program 4: Datatype Equivalencing

After connecting to Oracle, this program creates a database table named IMAGE in the SCOTT account, then simulates the insertion of bitmap images of employee numbers into the table. Datatype equivalencing lets the program use the Oracle external datatype LONG RAW to represent the images. Later, when the user enters an employee number, the number's "bitmap" is selected from the IMAGE table and pseudo-displayed on the terminal screen.

```
PROGRAM DTYEQV
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 UID
 CHARACTER*10 PWD
 INTEGER EMPNO
 CHARACTER*10 ENAME
 REAL SAL
 REAL COMM
 CHARACTER*8192 BUFFER
 EXEC SQL VAR BUFFER IS LONG RAW
 INTEGER SELECTION
 EXEC SQL END DECLARE SECTION

 CHARACTER*10 REPLY

 EXEC SQL INCLUDE SQLCA
 EXEC SQL WHENEVER SQLERROR DO CALL SQLERR

* LOG ON TO ORACLE.
 UID = 'SCOTT'
 PWD = 'TIGER'
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
 PRINT *, 'CONNECTED TO ORACLE AS USER: ', UID

 PRINT *, 'PROGRAM IS ABOUT TO DROP THE IMAGE ',
```

```
 +'TABLE - OK [Y/N]? '
 READ '(A10)', REPLY
 IF ((REPLY(1:1) .NE. 'Y') .AND. (REPLY(1:1) .NE. 'Y'))
 1 CALL SIGNOFF

 EXEC SQL WHENEVER SQLERROR CONTINUE
 EXEC SQL DROP TABLE IMAGE
 IF (SQLCDE .EQ. 0) THEN
 PRINT *, 'TABLE IMAGE HAS BEEN DROPPED - ',
 + 'CREATING NEW TABLE.'
 ELSE IF (SQLCDE .EQ. -942) THEN
 PRINT *, 'TABLE IMAGE DOES NOT EXIST - ',
 + 'CREATING NEW TABLE.'

ELSE
 CALL SQLERR
 END IF

 EXEC SQL WHENEVER SQLERROR DO CALL SQLERR
 EXEC SQL CREATE TABLE IMAGE
 1 (EMPNO NUMBER(4) NOT NULL, BITMAP LONG RAW)
 EXEC SQL DECLARE EMPCUR CURSOR FOR
 1 SELECT EMPNO, ENAME FROM EMP
 EXEC SQL OPEN EMPCUR
 PRINT *, 'INSERTING BITMAPS INTO IMAGE FOR ALL EMPLOYEES...'

7000 CONTINUE
 EXEC SQL WHENEVER NOT FOUND GOTO 10000
 EXEC SQL FETCH EMPCUR INTO :EMPNO, :ENAME
 CALL GETIMG (EMPNO, BUFFER)
 EXEC SQL INSERT INTO IMAGE VALUES (:EMPNO, :BUFFER)
 PRINT *, 'EMPLOYEE ', ENAME, '.......... IS DONE!'
 GOTO 7000

10000 EXEC SQL CLOSE EMPCUR
 EXEC SQL COMMIT WORK
 PRINT *, 'DONE INSERTING BITMAPS. NEXT, LETS DISPLAY SOME.'

* BEGINNING OF DISPLAY LOOP
12000 SELECTION = 0
 PRINT *, '\NENTER EMPLOYEE NUMBER (0 TO QUIT):'
 READ '(I10)', SELECTION
 IF (SELECTION .EQ. 0) CALL SIGNOFF
 EXEC SQL WHENEVER NOT FOUND GOTO 16000

 EXEC SQL SELECT EMP.EMPNO, ENAME, SAL, NVL(COMM,0), BITMAP
 1 INTO :EMPNO, :ENAME, :SAL, :COMM, :BUFFER
 2 FROM EMP, IMAGE
 3 WHERE EMP.EMPNO = :SELECTION
 4 AND EMP.EMPNO = IMAGE.EMPNO
 CALL SHWIMG (BUFFER)

 PRINT *, '\NEMPLOYEE ', ENAME, ' HAS SALARY ', SAL,
 + ' AND COMMISSION ', COMM
 GOTO 12000

16000 PRINT *, 'NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.'
 GOTO 12000
 END


 SUBROUTINE GETIMG (ENUM, BUF)
```

```
 INTEGER ENUM
 CHARACTER*8192 BUF
 INTEGER I

 DO 18000 I = 1, 8192
 BUF(I:I) = '*'
18000 CONTINUE
 END

 SUBROUTINE SHWIMG (BUF)
 CHARACTER*8192 BUF
 INTEGER I

 PRINT *, ' ***************************'
 DO 22000 I = 1, 9
 PRINT *, ' ***************************'
22000 CONTINUE
 END

 SUBROUTINE SIGNOFF
 EXEC SQL INCLUDE SQLCA
 PRINT *, 'HAVE A GOOD DAY.'
 EXEC SQL COMMIT WORK RELEASE
 STOP
 END

 SUBROUTINE SQLERR
 EXEC SQL INCLUDE SQLCA
 EXEC SQL WHENEVER SQLERROR CONTINUE
 PRINT *, 'ORACLE ERROR DETECTED:'
 PRINT '(70A1)', SQLEMC
 EXEC SQL ROLLBACK WORK RELEASE
 STOP
 END
```

3.5 Sample Program 5: Oracle Forms User Exit

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

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

where *user_exit* is a packaged procedure supplied with Oracle Forms and CONCAT is the name of the user exit. A sample CONCAT form invokes the user exit. For more information about Oracle Forms user exits, see Writing User Exits.

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

```
INTEGER FUNCTION CONCAT (CMD,CMDL,ERR,ERRL,INQRY)

 EXEC SQL BEGIN DECLARE SECTION
 LOGICAL*1 VALUE(81)
 LOGICAL*1 FINAL(241)
 LOGICAL*1 FIELD(81)
 EXEC SQL END DECLARE SECTION

 EXEC SQL INCLUDE SQLCA
 EXEC SQL WHENEVER SQLERROR GO TO 999

 LOGICAL*1 CMD(80)
```

```
      LOGICAL*1 ERR(80)
      INTEGER*2 CMDL, ERRL, INQRY

*     CERR IS A DYNAMICALLY BUILT ERROR MESSAGE RETURNED
*     TO SQL*FORMS.

      LOGICAL*1 CERR(80)

*     TEMPORARY VARIABLES TO DO STRING MANIPULATIONS.

      INTEGER*2 CMDCNT
      INTEGER*2 FLDCNT
      INTEGER*2 FNLCNT

*     INITIALIZE VARIABLES.

      DO 1 I = 1, 81
      FIELD(I) = ' '
    1 VALUE(I) = ' '
      DO 2 I = 1, 241
    2 FINAL(I) = ' '
      FNLCNT = 0
*     STRIP CONCAT FROM COMMAND LINE.

      CMDCNT = 7
      I = 1

*     LOOP UNTIL END OF COMMAND LINE.

      DO WHILE (CMDCNT .LE. CMDL)

*     PARSE EACH FIELD DELIMITED BY A COMMA.

      FLDCNT = 0
      DO WHILE ((CMD(CMDCNT) .NE. ',').AND.(CMDCNT .LE. CMDL))
      FLDCNT = FLDCNT + 1
      FIELD(FLDCNT) = CMD(CMDCNT)
      CMDCNT = CMDCNT + 1
      END DO
      IF (CMDCNT .LT. CMDL) THEN

*     WE HAVE FIELD1...FIELDN. THESE ARE NAMES OF
*     SQL*FORMS FIELDS; GET THE VALUE.

      EXEC TOOLS GET :FIELD INTO :VALUE

*     REINITIALIZE FIELD NAME.

      DO 20 K = 1, FLDCNT
   20 FIELD(K) = ' '

*     MOVE VALUE RETRIEVED FROM FIELD TO A CHARACTER
*     TO FIND LENGTH.

      DO WHILE (VALUE(I) .NE. ' ')
      FNLCNT = FNLCNT + 1
      FINAL(FNLCNT) = VALUE(I)
      I = I + 1
      END DO
      I = 1
      CMDCNT = CMDCNT + 1
      ELSE
```

```
* WE HAVE RESULT_FIELD; STORE IN SQL*FORMS FIELD.

 EXEC TOOLS SET :FIELD VALUES (:FINAL)
 END IF
 END DO

* ALL OK. RETURN SUCCESS CODE.

 CONCAT = IAPSUC
 RETURN

* ERROR OCCURRED. PREFIX NAME OF USER EXIT TO ORACLE
* ERROR MESSAGE, SET FAILURE RETURN CODE, AND EXIT.

999 CERR(1) = 'C'
 CERR(2) = 'O'
 CERR(3) = 'N'
 CERR(4) = 'C'
 CERR(5) = 'A'
 CERR(6) = 'T'
 CERR(7) = ':'
 CERR(8) = ' '
 DO 1000 J = 1, 70
 CERR(J + 8) = SQLEMC(J)
1000 CONTINUE
 ERRL = 78
 CALL TOOLS MESSAGE (CERR, ERRL)
 CONCAT = IAPFAI
 RETURN
 END
```

3.6 Sample Program 6: Dynamic SQL Method 1

This program uses dynamic SQL Method 1 to create a table, insert a row, commit the insert, then drop the table.

```
PROGRAM DYN1

 EXEC SQL INCLUDE SQLCA
 EXEC SQL INCLUDE ORACA
 EXEC ORACLE OPTION (ORACA=YES)
 EXEC ORACLE OPTION (RELEASE_CURSOR=YES)

 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 USERNAME
 CHARACTER*10 PASSWORD
 CHARACTER*80 DYNSTM
 EXEC SQL END DECLARE SECTION

 EXEC SQL WHENEVER SQLERROR GOTO 9000

 ORATXF = 1

 USERNAME = 'SCOTT'
 PASSWORD = 'TIGER'
 EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
 PRINT *, 'CONNECTED TO ORACLE.'

 PRINT *, 'CREATE TABLE DYN1 (COL1 CHAR(4))'
 EXEC SQL EXECUTE IMMEDIATE
```

```
 1 'CREATE TABLE DYN1 (COL1 CHAR(4))'

 DYNSTM = 'INSERT INTO DYN1 VALUES (''TEST'')'
 PRINT *, DYNSTM
 EXEC SQL EXECUTE IMMEDIATE :DYNSTM
 EXEC SQL COMMIT WORK

 DYNSTM = 'DROP TABLE DYN1'
 PRINT *, DYNSTM
 EXEC SQL EXECUTE IMMEDIATE :DYNSTM
 EXEC SQL COMMIT RELEASE

 PRINT *, 'HAVE A GOOD DAY!'
 GOTO 9999

9000 PRINT *, '\N-- ORACLE ERROR:'
 PRINT '(70A)', SQLEMC
 PRINT '(3A, 70A)', 'IN ', ORATXC
 PRINT *, 'ON LINE', ORASLN
 PRINT '(3A, 70A)', 'OF ', ORAFNC
 EXEC SQL WHENEVER SQLERROR CONTINUE
 EXEC SQL ROLLBACK RELEASE

9999 CONTINUE
 END
```

3.7 Sample Program 7: Dynamic SQL Method 2

This program uses dynamic SQL Method 2 to insert two rows into the EMP table, then delete them.

```
PROGRAM DYN2

 EXEC SQL INCLUDE SQLCA

 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 USERNAME
 CHARACTER*10 PASSWORD
 CHARACTER*80 DYNSTM
 INTEGER*2 EMPNO
 INTEGER*2 DEPTNO1
 INTEGER*2 DEPTNO2
 EXEC SQL END DECLARE SECTION

 EXEC SQL WHENEVER SQLERROR GOTO 9000

 USERNAME = 'SCOTT'
 PASSWORD = 'TIGER'
 EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
 PRINT *, 'CONNECTED TO ORACLE.'

 DYNSTM = 'INSERT INTO EMP (EMPNO,DEPTNO) VALUES(:V1, :V2)'
 PRINT *, DYNSTM
 EMPNO = 1234
 DEPTNO1 = 97
 PRINT *, 'V1 = ', EMPNO
 PRINT *, 'V2 = ', DEPTNO1
 EXEC SQL PREPARE S FROM :DYNSTM
 EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO1
 PRINT *, 'INSERT STATEMENT EXECUTED.\N'
```

```
 EMPNO = EMPNO + 1
 DEPTNO2 = 99
 PRINT *, 'CHANGED BIND VARIABLES V1 AND V2\NV1 = ', EMPNO
 PRINT *, 'V2 = ', DEPTNO2
 PRINT *, 'EXECUTING STATEMENT AGAIN WITH NEW BIND ',
 + 'VARIABLES.'
 EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO2
 PRINT *, 'DONE, NOW DELETING...\N'

 DYNSTM =
 + 'DELETE FROM EMP WHERE DEPTNO = :V1 OR DEPTNO = :V2'


 PRINT *, DYNSTM
 PRINT *, 'V1 = ', DEPTNO1
 PRINT *, 'V2 = ', DEPTNO2
 EXEC SQL PREPARE S FROM :DYNSTM
 EXEC SQL EXECUTE S USING :DEPTNO1, :DEPTNO2

 EXEC SQL COMMIT RELEASE
 PRINT *, 'HAVE A GOOD DAY!'
 GOTO 9999

9000 PRINT '(70A1)', SQLEMC
 EXEC SQL WHENEVER SQLERROR CONTINUE
 EXEC SQL ROLLBACK RELEASE

9999 CONTINUE
 END
```

3.8 Sample Program 8: Dynamic SQL Method 3

This program uses dynamic SQL Method 3 to retrieve the names of all employees in a given department from the EMP table.

```
PROGRAM DYN3

 EXEC SQL INCLUDE SQLCA
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 USERNAME
 CHARACTER*10 PASSWORD
 CHARACTER*80 DYNSTM
 CHARACTER*10 ENAME
 INTEGER*2 DEPTNO
 EXEC SQL END DECLARE SECTION
 EXEC SQL WHENEVER SQLERROR GOTO 9000

 USERNAME = 'SCOTT'
 PASSWORD = 'TIGER'
 EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
 PRINT *, 'CONNECTED TO ORACLE.\N'

 DYNSTM = 'SELECT ENAME FROM EMP WHERE DEPTNO = :V1'
 PRINT *, DYNSTM
 DEPTNO = 10
 PRINT *, 'V1 = ', DEPTNO
 EXEC SQL PREPARE S FROM :DYNSTM
 EXEC SQL DECLARE C CURSOR FOR S
 EXEC SQL OPEN C USING :DEPTNO
 EXEC SQL WHENEVER NOT FOUND GOTO 110
```

```
 PRINT *, '\NEMPLOYEE NAME\N-------------'
100 EXEC SQL FETCH C INTO :ENAME
 PRINT *, ENAME
 GOTO 100

110 PRINT *, '\NQUERY RETURNED', SQLERD(3), ' ROWS.'
 EXEC SQL CLOSE C
 EXEC SQL COMMIT RELEASE
 PRINT *, '\NHAVE A GOOD DAY.'
 GOTO 9999

9000 PRINT '(70A1)', SQLEMC
 EXEC SQL WHENEVER SQLERROR CONTINUE
 EXEC SQL CLOSE C
 EXEC SQL ROLLBACK RELEASE

9999 CONTINUE
 END
```

3.9 Sample Program 9: Calling a Stored Procedure

Before trying the sample program, you must create a PL/SQL package named *calldemo*, by running a script named CALLDEMO.SQL, which is supplied with Pro*FORTRAN and shown in the following example. The script can be found in the Pro*FORTRAN demo library. Check your Oracle system-specific documentation for exact spelling of the script.

```
CREATE OR REPLACE PACKAGE calldemo AS

 TYPE name_array IS TABLE OF emp.ename%type
 INDEX BY BINARY_INTEGER;
 TYPE job_array IS TABLE OF emp.job%type
 INDEX BY BINARY_INTEGER;
 TYPE sal_array IS TABLE OF emp.sal%type
 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 name_array,
 job OUT job_array,
 sal OUT sal_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 name_array,
 job OUT job_array,
 sal OUT sal_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;
/
```

The following sample program connects to Oracle, prompts the user for a department number, then calls a PL/SQL procedure named *get_employees*, which is stored in package *calldemo*. The procedure declares three PL/SQL tables as OUT formal parameters, then fetches a batch of employee data into the PL/SQL tables. The matching actual parameters are host tables. When the procedure finishes, row values in the PL/SQL tables are automatically assigned to the corresponding elements in the host tables. The program calls the procedure repeatedly, displaying each batch of employee data, until no more data is found.

```
PROGRAM CALLSP

 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*10 UID
 CHARACTER*10 PWD
 INTEGER DEPTNO
 CHARACTER*10 ENAME(10)
 CHARACTER*10 JOB(10)
 REAL SAL(10)
 INTEGER ENDFLG
 INTEGER ARYSIZ
 INTEGER NUMRET
 INTEGER*4 SQLCOD
 EXEC SQL END DECLARE SECTION

 EXEC SQL INCLUDE SQLCA
 EXEC SQL WHENEVER SQLERROR DO CALL SQLERR
```

```
 UID = 'SCOTT'
 PWD = 'TIGER'
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
 PRINT *, 'CONNECTED TO ORACLE AS USER ', UID

 PRINT *, 'ENTER DEPARTMENT NUMBER: '
 READ '(I10)', DEPTNO

* INITIALIZE VARIABLES AND ARRAYS.
 ENDFLG = 0
 ARYSIZ = 10
 NUMRET = 0
 DO 4000 I = 1, ARYSIZ
 ENAME(I) = ' '
 JOB(I) = ' '
 SAL(I) = 0
4000 CONTINUE

* DISPLAY HEADER.
 PRINT *, 'EMPLOYEE NAME JOB TITLE SALARY\N',
 +'------------ --------- ------'

* LOOP, FETCHING AND PRINTING BATCHES UNTIL END-FLAG IS SET.
6000 EXEC SQL EXECUTE
 1 BEGIN
 2 CALLDEMO.GET_EMPLOYEES (:DEPTNO, :ARYSIZ,
 3 :NUMRET, :ENDFLG, :ENAME, :JOB, :SAL);
 4 END;
 5 END-EXEC

 CALL PBATCH (NUMRET, ENAME, JOB, SAL)
 IF (ENDFLG .EQ. 0) GOTO 6000
 CALL SIGNOFF
 END

********************** SUBROUTINES *********************
* DISPLAY A BATCH OF ROWS.

 SUBROUTINE PBATCH (ROWS, ENAME, JOB, SAL)
 INTEGER ROWS
 CHARACTER*10 ENAME(ROWS)
 CHARACTER*10 JOB(ROWS)
 REAL SAL(ROWS)

 DO 8000 I = 1, ROWS
 PRINT '(1X, A10, 5X, A10, 1X, F7.2)', ENAME(I), JOB(I), SAL(I)
8000 CONTINUE
 RETURN
 END

* LOG OFF ORACLE.

 SUBROUTINE SIGNOFF
 EXEC SQL INCLUDE SQLCA
 PRINT *, 'HAVE A GOOD DAY.'
 EXEC SQL COMMIT WORK RELEASE
 STOP
 END

* HANDLE SQL ERRORS.
```

```
      SUBROUTINE SQLERR
      EXEC SQL INCLUDE SQLCA
      EXEC SQL WHENEVER SQLERROR CONTINUE
      PRINT *, 'ORACLE ERROR DETECTED:'
      PRINT '(70A1)', SQLEMC
      EXEC SQL ROLLBACK WORK RELEASE
      STOP
      END
```

4

Implementing Dynamic SQL Method 4

This chapter shows you how to implement dynamic SQL Method 4, which lets your program accept or build dynamic SQL statements that contain a varying number of host variables.

> **Note**
>
> For a discussion of dynamic SQL Methods 1, 2, and 3, and an overview of Method 4, see Using Dynamic SQL.

This chapter contains the following sections:

4.1 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 (input) 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 = :DNUM AND LOC = :DLOC
```

> **Note**
>
> Placeholders cannot reference table or column names.

4.1.1 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
- take explicit control over datatype conversion between Oracle and FORTRAN types

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

4.1.2 What Information Does Oracle Need?

The Pro*FORTRAN 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 for bind variables, as shown in 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 select-list items and/or placeholders for bind variables, Oracle needs information about the program variables that will hold output or input values. Specifically, Oracle needs the following information:

- the number of select-list items and the number of bind variables
- the length of each select-list item and bind variable
- the datatype of each select-list item and bind variable
- the memory address of each output variable that will store the value of a select-list item, and the address of each bind variable

For example, to write the value of a select-list item, Oracle needs the address of the corresponding output variable.

4.1.3 Where Is the Information Stored?

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

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

The values of select-list items are stored in output buffers; the values of bind variables are stored in input buffers. You use the library routine SQLADR to store the addresses of these data buffers 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 data buffers? Output values are FETCHed using a cursor, and input values are filled in by your program, typically from information entered interactively by the user.

4.1.4 How Is the Information Obtained?

You use the DESCRIBE statement to help get the information Oracle needs. The DESCRIBE SELECT LIST statement examines each select-list item to determine its name, datatype, constraints, length, scale, and precision, then stores this information in the select SQLDA for your use. For example, you might use select-list names as column headings in a printout. DESCRIBE also stores the total number of select-list items in the SQLDA.

The DESCRIBE BIND VARIABLES statement examines each placeholder to determine its name and length, then stores this information in an input buffer and bind SQLDA for your use. For example, you might use placeholder names to prompt the user for the values of bind variables.

4.2 Understanding the SQL Descriptor Area (SQLDA)

This section describes the SQLDA data structure in detail. You learn how to declare it, what variables it contains, how to initialize them, and how to use them in your program.

4.2.1 Purpose of the SQLDA

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or placeholders for bind variables. To process this kind of dynamic SQL statement, your program must explicitly declare SQLDAs, also called *descriptors* (not to be confused with the CHARACTER variable descriptors generated by some FORTRAN compilers). Each descriptor is a named COMMON block, which you must copy or hard code into your program.

A *select descriptor* holds descriptions of select-list items and the addresses of output buffers where the names and values of select-list items are stored.

> **Note**
>
> The name of a select-list item can be a column name, a column alias, or the text of an expression such as SAL + COMM.

A *bind descriptor* holds descriptions of bind variables and indicator variables and the addresses of input buffers where the names and values of bind variables and indicator variables are stored.

Remember, some descriptor variables contain addresses, not values. So, you must declare data buffers to hold the values. You decide the sizes of the required input and output buffers. Because FORTRAN does not support pointers, you must use the library subroutine SQLADR to get the addresses of input and output buffers. You learn how to call SQLADR in the section "About Using SQLADR".

4.2.2 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.

4.2.3 Naming Conventions

You can name select and bind descriptors anything you like. Typically, the names SEL and BND are used. The precompiler references descriptor variables by appending single-character suffixes to the descriptor name (see Table 4 - 1). You use the descriptor name in the DESCRIBE, OPEN, and FETCH statements.

For example, the statement

```
* Open a cursor.
 EXEC SQL OPEN CUR1 USING DESCRIPTOR BND
* Fetch select-list values.
 EXEC SQL FETCH CUR1 USING DESCRIPTOR SEL
```

fetches select-list values into output data buffers.

You decide the names and sizes of the required data buffers. The variable and buffer names shown in the following tables, respectively, are used in the following discussion. For example, the elements of descriptor array SELS address the elements of data buffer array SELSB.

Suffix	Host Datatype	Description
N	INTEGER var	maximum number of select-list items or placeholders
F	INTEGER var	actual number of select-list items or placeholders
S	INTEGER*4 var(n)	addresses of select-list or placeholder names
M	INTEGER*2 var(n)	maximum lengths of select-list or placeholder names
C	INTEGER*2 var(n)	actual lengths of select-list or placeholder names
L	INTEGER*4 var(n)	lengths of select-list or bind-variable values
T	INTEGER*2 var(n)	datatypes of select-list or bind-variable values
V	INTEGER*4 var(n)	addresses of select-list or bind-variable values
I	INTEGER*4 var(n)	addresses of indicator-variable values (1)
X (2)	INTEGER*4 var(n)	addresses of indicator-variable names (1)
Y (2)	INTEGER*2 var(n)	maximum lengths of indicator-variable names (1)
Z (2)	INTEGER*2 var(n)	actual lengths of indicator-variable names (1)

> **Note**
>
> 1. Indicator-variable names apply only in a bind SQLDA.
> 2. These suffixes apply only to bind variables.

Buffer	Host Datatype	Description
SELSB	LOGICAL*1 var(m,n)	select-list names
SELVB	LOGICAL*1 var(m,n)	select-list names
SELIV	INTEGER*2 var(n)	indicator-variable values
BNDSB	LOGICAL*1 var(m,n)	placeholder names
BNDVB	LOGICAL*1 var(m,n)	bind-variable values
BNDXB	LOGICAL*1 var(m,n)	indicator-variable names
BNDIV	INTEGER*2 var(n)	indicator-variable names

> **Note**
>
> There is no SELXB buffer because indicator-variable names cannot be associated with select-list items.

4.2.4 About Declaring a SQLDA

To declare select and bind SQLDAs, you can hardcode them into your program using the sample SQLDA shown in Figure 4-1.

Figure 4-1 Sample Pro*FORTRAN SQLDA Variables and Data Buffers

```
*      Sample Select Descriptors and Data Buffers

*      Descriptors
       INTEGER       SELN
       INTEGER       SELF
       INTEGER*4     SELS(20)
       INTEGER*2     SELM(20)
       INTEGER*2     SELC(20)
       INTEGER*4     SELL(20)
       INTEGER*2     SELT(20)
       INTEGER*4     SELV(20)
       INTEGER*4     SELI(20)

*      Buffers
       LOGICAL*1     SELSB(30,20)
       LOGICAL*1     SELVB(80,20)
       INTEGER       SELIV(20)

       COMMON /SELDSC/
      1  SELN,   SELF,   SELS,   SELM,   SELC,   SELL,
      2  SELT,   SELV,   SELI,  SELSB,  SELVB,  SELIV


*      Sample Bind Descriptors and Data Buffers

*      Descriptors
       INTEGER       BNDN
       INTEGER       BNDF
       INTEGER*4     BNDS(20)
       INTEGER*2     BNDM(20)
       INTEGER*2     BNDC(20)
       INTEGER*4     BNDL(20)
       INTEGER*2     BNDT(20)
       INTEGER*4     BNDV(20)
       INTEGER*4     BNDI(20)
       INTEGER*4     BNDX(20)
       INTEGER*4     BNDY(20)
       INTEGER*4     BNDZ(20)

*      Buffers
       LOGICAL*1     BNDSB(30,20)
       LOGICAL*1     BNDVB(80,20)
       LOGICAL*1     BNDXB(30,20)
       INTEGER       BNDIV(20)

       COMMON /BNDDSC/
      1    BNDN,   BNDF,   BNDS,   BNDM,   BNDC,   BNDL,
      2    BNDT,   BNDV,   BNDI,   BNDX,   BNDY,   BNDZ,
      3  BNDSB,  BNDVB,  BNDXB,  BNDIV
```

You can modify the array dimensions to suit your needs. The following example uses a parameter to specify array dimensions; which makes changing the dimensions easy:

```
INTEGER SIZE
* Set dimension of descriptor arrays.
 PARAMETER (SIZE = 25)
* Declare select descriptor.
 INTEGER SELN
 INTEGER SELF
 INTEGER*4 SELV(SIZE)
 INTEGER*4 SELL(SIZE)
 ...
```

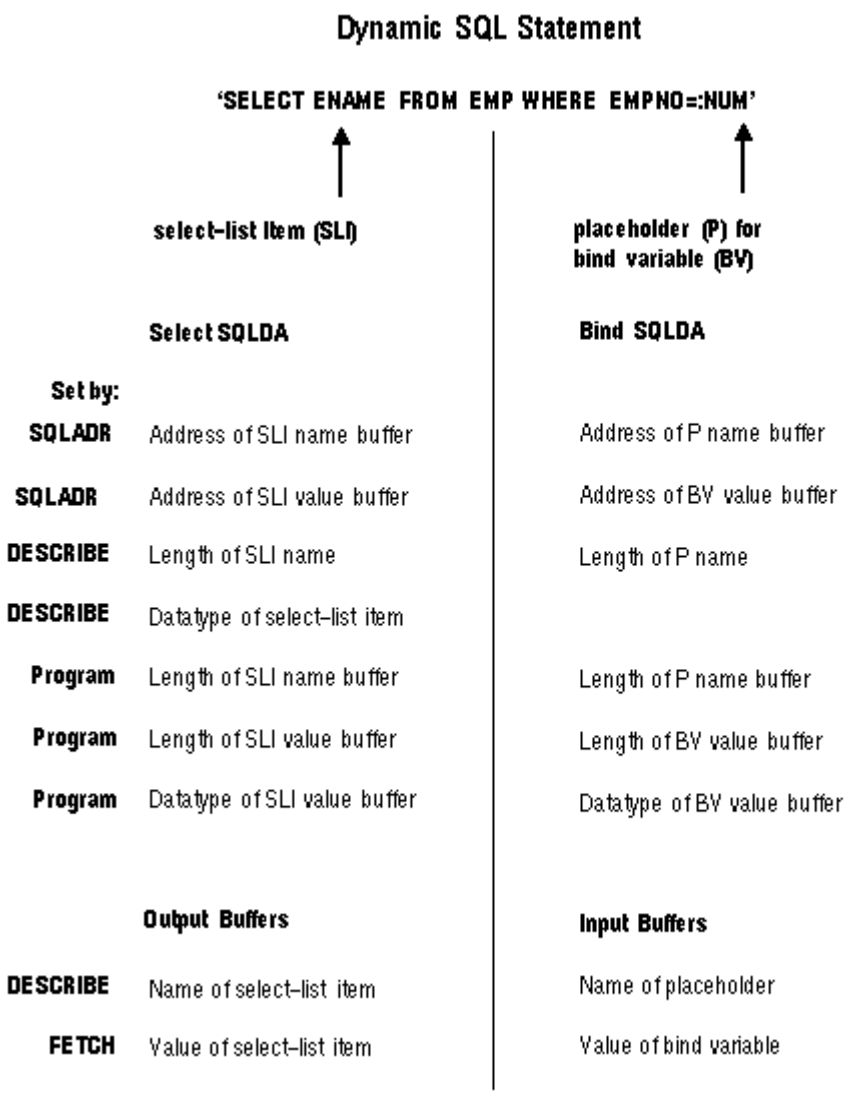
You might want to store the SQLDAs in files (named SELDSC and BNDDSC, for example), revise them as needed, then copy the files into your program with the INCLUDE statement as follows:

```
* Declare select and bind SQLDAs.
 EXEC SQL INCLUDE SELDSC
 EXEC SQL INCLUDE BNDDSC
```

Because they are COMMON blocks, SQLDAs must be declared *outside* the Declare Section. How the data buffers are declared is up to you. You need not include them in the SQLDA COMMON blocks. For example, you might want to declare one large data buffer to store all names and values, then access them using byte offsets.

Figure 4-2 shows whether variables are set by SQLADR calls, DESCRIBE commands, FETCH commands, or program assignments.

Figure 4-2 How Variables Are Set



4.3 About Using the SQLDA Variables and Arrays

This section explains the purpose and use of each SQLDA variable. In examples, the arbitrary SQLDA file names, descriptor names, and data buffer names given earlier are used.

4.3.1 The N Variable

This variable specifies the maximum number of select-list items or placeholders that can be DESCRIBEd. For example, SELN determines the number of elements in the select descriptor arrays.

Before issuing a DESCRIBE command, you must set this variable to the dimension of the descriptor arrays. After the DESCRIBE, you must reset it to the actual number of variables DESCRIBEd, which is stored in the F variable.

4.3.2 The F Variable

This is the actual number of select-list items or placeholders found by the DESCRIBE command.

The F variable is set by DESCRIBE. If the F variable is negative, the DESCRIBE command found too many select-list items or placeholders for the size of the descriptor. For example, if you set SELN to 10 but DESCRIBE finds 11 select-list items, SELF is set to -11. If this happens, you cannot process the SQL statement without reallocating the descriptor.

After the DESCRIBE command, you must set the N variable equal to the F variable.

4.3.3 The S Array

This array contains the addresses of data buffers that store select-list or placeholder names as they appear in dynamic SQL statements.

You must set the elements of the S array using SQLADR before issuing the DESCRIBE command.

DESCRIBE directs Oracle to store the name of the Jth select-list item or placeholder in the buffer addressed by SELS(J) or BNDS(J). If the elements of SELS and BNDS address elements of data buffer arrays named SELSB and BNDSB, Oracle stores the Jth select-list or placeholder name in SELSB(J) or BNDSB(J).

4.3.4 The M Array

This array contains the lengths of the data buffers that store select-list or placeholder names. The buffers are addressed by elements of the S array.

You must set the elements of the M array before issuing the DESCRIBE command. Each select-list or placeholder name buffer can have a different length.

4.3.5 The C Array

This array contains the actual lengths of select-list or placeholder names. DESCRIBE sets the array of actual lengths to the number of characters in each select-list or placeholder name.

4.3.6 The L Array

This array contains the lengths of select-list or bind-variable values stored in the data buffers.

4.3.7 Select Descriptors

DESCRIBE sets the array of lengths to the maximum expected for each select-list item. However, you might want to reset some lengths before issuing a FETCH command. FETCH returns at most *n* characters, where *n* is the value of SELL(J) before the FETCH.

The format of the length differs among Oracle datatypes. For character select-list items, DESCRIBE sets SELL(J) to the maximum length in bytes of the select-list item. For NUMBER select-list items, scale and precision are returned respectively in the low and next-higher bytes of the variable. You can use the library subroutine SQLPRC to extract precision and scale values from SELL. See the section Extracting Precision and Scale.

You must reset SELL(J) to the required length of the data buffer before the FETCH. For example, when coercing a NUMBER to a FORTRAN CHARACTER string, set SELL(J) to the precision of the number plus two for the sign and decimal point. When coercing a NUMBER to a FORTRAN REAL, set SELL(J) to the length of REALs on your system. For more information about the lengths of coerced datatypes, see the section "Converting Data".

4.3.8 Bind Descriptors

You must set the array of lengths before issuing the OPEN command.

Because Oracle accesses a data buffer indirectly, using the address in SELV(J) or BNDV(J), it does not know the length of the value in that buffer. If you want to change the length Oracle uses for the Jth select-list or bind-variable value, reset SELL(J) or BNDL(J) to the length you need. Each input or output buffer can have a different length.

4.3.9 The T Array

This array contains the datatype codes of select-list or bind-variable values. These codes determine how Oracle data is converted when stored in the data buffers addressed by elements of SELV. This topic is covered in the section "Converting Data".

4.3.10 Select Descriptors

DESCRIBE sets the array of datatype codes to the *internal* datatype (for example, VARCHAR2, CHAR, NUMBER, or DATE) of the items in the select list.

Before FETCHing, you might want to reset some datatypes because the internal format of Oracle datatypes can be difficult to handle. For display purposes, it is usually a good idea to coerce the datatype of select-list values to VARCHAR2. For calculations, you might want to coerce numbers from Oracle to FORTRAN format. See the section "Coercing Datatypes".

The high bit of SELT(J) is set to indicate the null/not null status of the Jth select-list column. You must always clear this bit before issuing an OPEN or FETCH command. You use the library subroutine SQLNUL to retrieve the datatype code and clear the null/not null bit. See the section "Handling Null/Not Null Datatypes".

You should change the Oracle NUMBER internal datatype to an external datatype compatible with that of the FORTRAN data buffer addressed by SELV(J).

4.3.11 Bind Descriptors

DESCRIBE sets the array of datatype codes to zeros. You must reset the datatype code stored in each element before issuing the OPEN command. The code represents the external (FORTRAN) datatype of the data buffer addressed by BNDV(J). Often, bind-variable values are stored in character strings, so the datatype array elements are set to 1 (the VARCHAR2 datatype code).

To change the datatype of the Jth select-list or bind-variable value, reset SELT(J) or BNDT(J) to the datatype you want.

4.3.12 The V Array

This array contains the addresses of data buffers that store select-list or bind-variable values. You must set the elements of the V array using SQLADR.

4.3.13 Select Descriptors

You must set this array before issuing the FETCH command. The following statement

```
* Fetch select-list values.
 EXEC SQL FETCH ... USING DESCRIPTOR SEL
```

directs Oracle to store FETCHed select-list values in the data buffers addressed by SELV(1) through SELV(SELN). If the elements of SELV address elements of a data buffer array named SELVB, Oracle stores the Jth select-list value in SELVB(J).

4.3.14 Bind Descriptors

You must set this array before issuing the OPEN command. The following statement

```
* Open cursor.
 EXEC SQL OPEN ... USING DESCRIPTOR BND
```

directs Oracle to execute the dynamic SQL statement using the bind-variable values addressed by BNDV(1) through BNDV(BNDN). If the elements of BNDV address elements of a data buffer array named BNDVB, Oracle finds the Jth bind-variable value in data buffer BNDVB(J).

4.3.15 The I Array

This array contains the addresses of data buffers that store indicator-variable values.

You must set the elements of the I array using SQLADR.

4.3.16 Select Descriptors

You must set this array before issuing the FETCH command. When Oracle executes the statement

```
* Fetch select-list values.
 EXEC SQL FETCH ... USING DESCRIPTOR SEL
```

if the Jth returned select-list value is null, the buffer addressed by SELI(J) is set to -1. Otherwise, it is set to zero (the value is not null) or a positive integer (the value was truncated). For example, if the elements of SELI address elements of a data buffer array named SELIV, and the Jth returned select-list value is null, SELIV(J) is set to -1.

4.3.17 Bind Descriptors

You must initialize this array and set the associated indicator variables before issuing the OPEN command. When Oracle executes the following statement

```
* Open cursor.
 EXEC SQL OPEN ... USING DESCRIPTOR BND
```

the buffer addressed by BNDI(J) determines whether the Jth bind variable is a null. If the value of an indicator variable is -1, its associated host variable is null. For example, if the elements of BNDI address elements of a data buffer array named BNDIV, and the value of BNDIV(J) is -1, the value of the Jth bind variable is set to NULL.

4.3.18 The X Array

This array contains the addresses of data buffers that store indicator-variable names. You can associate indicator-variable *values* with select-list items and bind variables. However, you can associate indicator-variable *names* only with bind variables. So, you can use the X array only with bind descriptors.

You must set the elements of the X array using SQLADR before issuing the DESCRIBE command.

DESCRIBE directs Oracle to store any indicator-variable names in the buffers addressed by BNDX(1) through BNDX(BNDN). If the elements of BNDX address elements of a data buffer array named BNDXB, Oracle stores the Jth indicator-variable name in BNDXB(J).

4.3.19 The Y Array

This array contains the maximum lengths of the data buffers that store indicator-variable names. The buffers are addressed by elements of the X array. Like the X array, you can use the Y array only with bind descriptors.

You must set the elements BNDY(1) through BNDY(BNDN) before issuing the DESCRIBE command. Each indicator-variable name buffer can have a different length.

4.3.20 The Z Array

This array contains the actual lengths of indicator-variable names. Like the X and Y arrays, you can use the Z array only with bind descriptors.

DESCRIBE sets the array of actual lengths to the number of characters in each indicator-variable name.

4.4 Some Preliminaries

You need a working knowledge of the following subjects to implement dynamic SQL Method 4:

- using the library subroutine SQLADR
- converting data
- coercing datatypes
- handling null/not null datatypes

4.4.1 About Using SQLADR

You must call the library subroutine SQLADR to get the addresses of data buffers that store input and output values. You store the addresses in a select or bind SQLDA so that Oracle knows where to read bind-variable values or write select-list values.

Call SQLADR using the syntax

```
CALL SQLADR (BUFF, ADDR)
```

where:

BUFF

Is a data buffer that stores the value or name of a select-list item, bind variable, or indicator variable.

ADDR

Is an integer variable that returns the address of the data buffer.

A call to SQLADR stores the address of BUFF in ADDR. In the following example, you use SQLADR to initialize the select descriptor arrays SELV, SELS, and SELI. Their elements address data buffers for select-list values, select-list names, and indicator values.

```
* Initialize select descriptor arrays.
 DO 100 J = 1, SELN
 CALL SQLADR (SELVB(1, J), SELV(J))
 CALL SQLADR (SELSB(1, J), SELS(J))
 CALL SQLADR (SELIV(J), SELI(J))
100 CONTINUE
```

4.4.2 Restriction

You cannot use CHARACTER variables with SQLADR if your FORTRAN compiler generates descriptors for CHARACTER variables and passes the descriptor address (rather than the data address) to SQLADR. Check your FORTRAN compiler user's guide. In such cases, SQLADR gets the wrong address. Instead, use LOGICAL*1 variables, because they always have simple addresses.

However, you can (cautiously) use SQLADR with CHARACTER variables if your compiler provides a built-in function to access the data address. For example, if your compiler provides a function named %REF, and X is a CHARACTER variable, you call SQLADR as follows:

```
* Use %REF built-in function.
 CALL SQLADR (%REF(X), ...)
```

4.4.3 Converting Data

This section provides more detail about the datatype descriptor array. In host programs that use neither datatype equivalencing nor dynamic SQL Method 4, the conversion between Oracle internal and external datatypes is determined at precompile time. By default, the precompiler assigns a specific external datatype to each host variable in the Declare Section. For example, the precompiler assigns the FLOAT external datatype to host variables of type REAL.

However, Method 4 lets you control data conversion and formatting. You specify conversions by setting datatype codes in the datatype descriptor array.

4.4.4 Internal Datatypes

Internal datatypes specify the formats used by Oracle to store column values in database tables and the formats to represent pseudocolumn values.

When you issue a DESCRIBE SELECT LIST command, Oracle returns the internal datatype code for each select-list item to the SELT (datatype) descriptor array. For example, the datatype code for the Jth select-list item is returned to SELT(J).

The following table shows the Oracle internal datatypes and their codes.

Oracle Internal Datatype	Code
VARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHAR	96
MLSLABEL	105

4.4.5 External Datatypes

External datatypes specify the formats used to store values in input and output host variables.

The DESCRIBE BIND VARIABLES command sets the BNDT array of datatype codes to zeros. So, you must reset the codes *before* issuing the OPEN command. The codes tell Oracle which external datatypes to expect for the various bind variables. For the Jth bind variable, reset BNDT(J) to the external datatype you want.

The following table shows the Oracle external datatypes and their codes, as well as the corresponding FORTRAN datatypes:

Name	Code	FORTRAN Datatype
VARCHAR2	1	CHARACTER*_n_ when MODE != ANSI
NUMBER	2	CHARACTER*_n_
INTEGER	3	INTEGER
FLOAT	4	REAL
STRING (1)	5	CHARACTER*(_n_+1)
VARNUM	6	CHARACTER*_n_
DECIMAL	7	CHARACTER*_n_
LONG	8	CHARACTER*_n_
VARCHAR (2)	9	CHARACTER*_n_
ROWID	11	CHARACTER*_n_
DATE	12	CHARACTER*_n_
VARRAW (2)	15	CHARACTER*_n_
RAW	23	CHARACTER*_n_
LONG RAW	24	CHARACTER*_n_
UNSIGNED	68	INTEGER
DISPLAY	91	CHARACTER*_n_
LONG VARCHAR (2)	94	CHARACTER*_n_
LONG VARRAW (2)	95	CHARACTER*_n_
CHARF	96	CHARACTER*_n_ when MODE = ANSI

Name	Code	FORTRAN Datatype
CHARZ (1)	97	CHARACTER*(*n*+1)
CURSOR	102	SQLCURSOR
MLSLABEL	106	CHARACTER**n*

Note

1. For use in an EXEC SQL VAR statement only.
2. Include the *n*-byte length field.

For more information about the Oracle datatypes and their formats, see Meeting Program Requirements.

4.4.6 PL/SQL Datatypes

PL/SQL provides a variety of predefined scalar and composite datatypes. A *scalar* type has no internal components. A *composite* type has internal components that can be manipulated individually. The following table shows the predefined PL/SQL scalar datatypes and their Oracle internal datatype equivalences.

PL/SQL Datatype	Oracle Internal Datatype
VARCHAR VARCHAR2	VARCHAR2
BINARY_INTEGER DEC DECIMAL DOUBLE PRECISION FLOAT INT INTEGER NATURAL NUMBER NUMERIC POSITIVE REAL SMALLINT	NUMBER
LONG	LONG
ROWID	ROWID
DATE	DATE
RAW	RAW
LONG RAW	LONG RAW
CHAR CHARACTER STRING	CHAR
MLSLABEL	MLSLABEL

4.4.7 Coercing Datatypes

For a select descriptor, DESCRIBE SELECT LIST can return any of the Oracle internal datatypes. Often, as in the case of character data, the internal datatype corresponds exactly to the external datatype you want to use. However, a few internal datatypes map to external datatypes that can be difficult to handle. So, you might want to reset some elements in the SELT descriptor array.

For example, you might want to reset NUMBER values to FLOAT values, which correspond to REAL values in FORTRAN. Oracle does any necessary conversion between internal and external datatypes at FETCH time. So, be sure to reset the datatypes *after* the DESCRIBE SELECT LIST but *before* the FETCH.

For a bind descriptor, DESCRIBE BIND VARIABLES does *not* return the datatypes of bind variables, only their number and names. Therefore, you must explicitly set the BNDT array of datatype codes to tell Oracle the external datatype of each bind variable. Oracle does any necessary conversion between external and internal datatypes at OPEN time.

When you reset datatype codes in the SELT or BNDT descriptor array, you are "coercing datatypes." For example, to coerce the Jth select-list value to VARCHAR2, use the following statement:

```
* Coerce select-list value to VARCHAR2.
 SELT(J) = 1
```

When coercing a NUMBER select-list value to VARCHAR2 for display purposes, you must also extract the precision and scale bytes of the value and use them to compute a maximum display length. Then, before the FETCH, you must reset the appropriate element of the SELL (length) descriptor array to tell Oracle the buffer length to use. To specify the length of the Jth select-list value, set SELL(J) to the length you need.

For example, if DESCRIBE SELECT LIST finds that the Jth select-list item is of type NUMBER, and you want to store the returned value in a FORTRAN variable declared as REAL, simply set SELT(J) to 4 and SELL(J) to the length of REAL numbers on your system.

4.4.8 Exceptions

In some cases, the internal datatypes that DESCRIBE SELECT LIST returns might not suit your purposes. Two examples of this are DATE and NUMBER. When you DESCRIBE a DATE select-list item, Oracle returns the datatype code 12 to the SELT array. Unless you reset the code before the FETCH, the date value is returned in its 7-byte internal format. To get the date in its default character format, you must change the datatype code from 12 to 1 (VARCHAR2), and increase the SELL value from 7 to 9.

Similarly, when you DESCRIBE a NUMBER select-list item, Oracle returns the datatype code 2 to the SELT array. Unless you reset the code before the FETCH, the numeric value is returned in its internal format, which is probably not what you want. So, change the code from 2 to 1 (VARCHAR2), 3 (INTEGER), 4 (FLOAT), or some other appropriate datatype.

4.4.9 About Extracting Precision and Scale

The library subroutine SQLPRC extracts precision and scale. Normally, it is used after the DESCRIBE SELECT LIST, and its first argument is SELL(J). You call SQLPRC using the syntax

```
CALL SQLPRC (LENGTH, PREC, SCALE)
```

where:

LENGTH

Is an integer variable that stores the length of an Oracle NUMBER value. The scale and precision of the value are stored in the low and next-higher bytes, respectively.

PREC

Is an integer variable that returns the *precision* of the NUMBER value. Precision is the number of significant digits. It is set to zero if the select-list item refers to a NUMBER of unspecified size. In this case, because the size is unspecified, you might want to assume the maximum precision, 38.

SCALE

Is an integer variable that returns the *scale* of the NUMBER value. Scale specifies where rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of -3 means the number is rounded to the nearest thousand (3456 becomes 3000).

The following example shows how SQLPRC is used to compute maximum display lengths for NUMBER values that will be coerced to VARCHAR2:

```
* Declare variables for function call.
 INTEGER PREC
 INTEGER SCALE
 EXEC SQL DESCRIBE SELECT LIST FOR S INTO SEL
 DO 1300 J = 1, SELN
 IF (SELT(J) .NE. 2) GOTO 1300
* If datatype is NUMBER, extract precision and scale.
 CALL SQLPRC (SELL(J), PREC, SCALE)
* If no precision was specified, assign a maximum.
 IF (PREC .NE. 0) GOTO 1100
 SELL(J) = 10
 GOTO 1300
 1100 CONTINUE
 SELL(J) = PREC
* Allow for possible sign and decimal point.
 SELL(J) = SELL(J) + 2
 1300 CONTINUE
 ...
```

The SQLPRC subroutine returns zero as the precision and scale values for certain SQL datatypes. The SQLPR2 subroutine is similar to SQLPRC in that it has the same syntax and returns the same binary values, except for the datatypes shown in the following table.

SQL Datatype	Binary Precision	Binary Scale
FLOAT	126	-127
FLOAT(*n*)	*n* (range is 1 .. 126)	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

4.4.10 About Handling Null/Not Null Datatypes

For every select-list column (not expression), DESCRIBE SELECT LIST returns a null/not null indication in the datatype array (SELT) of the select descriptor. If the Jth select-list column is constrained to be not null, the high-order bit of SELT(J) is clear; otherwise, it is set.

Before using the datatype in an OPEN or FETCH statement, if the null status bit is set, you must clear it. Never set the bit.

You can use the library subroutine SQLNUL to find out if a column allows nulls, and to clear the datatype's null status bit. You call SQLNUL using the syntax

```
CALL SQLNUL (VALTYP, TYPCODE, NULSTAT)
```

where:

VALTYP

Is a 2-byte integer variable that stores the datatype code of a select-list column.

TYPCODE

Is a 2-byte integer variable that returns the datatype code of the select-list column with the high-order bit cleared.

NULSTAT

Is an integer variable that returns the null status of the select-list column. 1 means the column allows nulls; 0 means it does not.

The following example shows how to use SQLNUL:

```
* Declare variable for subroutine call.
 INTEGER*2 DATYPE
 INTEGER NULLOK
 DO 1500 J = 1, SELN
* Find out if column is NOT NULL, and
* clear high-order bit.
 CALL SQLNUL (SELT(J), DATYPE, NULLOK)
 SELT(J) = DATYPE
* If NULLOK equals 1, nulls are allowed.
 ...
 1500 CONTINUE
 ...
```

The first argument in the subroutine is the Jth element of the SELT datatype array before its null/not null bit is cleared. Though some systems let you use SELT(J) as the second argument too, it is poor programming practice to use the same variable as multiple arguments.

4.5 The Basic Steps

Method 4 can be used to process *any* dynamic SQL statement. In the example, a query is processed so that you can see how both input and output host variables are handled. Again, the arbitrary SQLDA file names, descriptor names, and data buffer names given earlier are used.

To process the dynamic query, our sample program performs the following:

1. Declare a host string in the Declare Section to hold the query text.
2. Declare select and bind descriptors.
3. Set the maximum number of select-list items and placeholders that can be DESCRIBEd.
4. Initialize the select and bind descriptors.
5. Store the query text in the host string.
6. PREPARE the query from the host string.
7. DECLARE a cursor FOR the query.
8. DESCRIBE the bind variables INTO the bind descriptor.
9. Reset the number of placeholders to the number actually found by the DESCRIBE command.
10. Get values for the bind variables found by DESCRIBE.
11. OPEN the cursor USING the bind descriptor.

12. DESCRIBE the select list INTO the select descriptor.
13. Reset the number of select-list items to the number actually found by the DESCRIBE command.
14. Reset the length and datatype of each select-list item for display purposes.
15. FETCH a row from the database INTO data buffers using the select descriptor.
16. Process the select-list values 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 steps are unnecessary.

4.6 A Closer Look at Each Step

This section discusses each step in more detail. Also, at the end of this chapter is a full-length program illustrating Method 4.

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> CURSOR FOR <statement_name>
EXEC SQL
 DESCRIBE BIND VARIABLES FOR <statement_name>
 INTO <bind_descriptor_name>
EXEC SQL
 OPEN <cursor_name>
 [USING DESCRIPTOR <bind_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>
```

If the number of select-list items in a dynamic query is known, you can omit DESCRIBE SELECT LIST and use the following Method 3 FETCH statement:

```
EXEC SQL FETCH <cursor_name> INTO <host_variable_list>
```

If the number of placeholders for bind variables in a dynamic SQL statement 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>]
```

Next, you see how these statements allow your host program to accept and process a dynamic SQL statement using descriptors.

> **Note**
>
> Several figures accompany the following discussion. To avoid cluttering the figures, it was necessary to confine descriptor arrays to 3 elements and to limit the maximum length of names and values to 5 and 10 characters, respectively.

4.6.1 Declare a Host String

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

```
EXEC SQL BEGIN DECLARE SECTION
 ...
 CHARACTER*120 SELSTM
 EXEC SQL END DECLARE SECTION
```

4.6.2 Declare the SQLDAs

Because the query in our example might contain an unknown number of select-list items or placeholders, you must declare select and bind descriptors. Instead of hard coding the SQLDAs, you use the INCLUDE statement to copy them into your program, as follows:

```
EXEC SQL INCLUDE SELDSC
 EXEC SQL INCLUDE BNDDSC
```

4.6.3 Set the Maximum Number to DESCRIBE

Next, you set the maximum number of select-list items or placeholders that can be DESCRIBEd, as follows:

```
SELN = 3
BNDN = 3
```

4.6.4 Initialize the Descriptors

You must initialize several descriptor variables; some require the library subroutine SQLADR. In our example, you store the maximum lengths of name buffers in the M and Y arrays, and use SQLADR to store the addresses of value and name buffers in the V, S, I, and X arrays:

```
* Initialize select descriptor arrays.
* Store addresses of select-list value and name
* buffers in SELV and SELS, addresses of indicator
* value buffers in SELI, and maximum length of
* select-list name buffers in SELM.
 DO 100 J = 1, SELN
 CALL SQLADR (SELVB(1, J), SELV(J))
 CALL SQLADR (SELSB(1, J), SELS(J))
 CALL SQLADR (SELIV(J), SELI(J))
 SELM(J) = 5
 100 CONTINUE
* Initialize bind descriptor arrays.
* Store addresses of bind-variable value and name
* buffers in BNDV and BNDS, addresses of indicator
* value and name buffers in BNDI and BNDX, and maximum
* lengths of placeholder and indicator name buffers in
* BNDM and BNDY.
```

```
DO 200 J = 1, BNDN
CALL SQLADR (BNDVB(1, J), BNDV(J))
CALL SQLADR (BNDSB(1, J), BNDS(J))
CALL SQLADR (BNDIV(J), BNDI(J))
CALL SQLADR (BNDXB(1, J), BNDX(J))
BNDM(J) = 5
BNDY(J) = 5
200 CONTINUE
...
```

Figure 4-3 and Figure 4-4represent the resulting descriptors.

Figure 4-3 Initialized Select Descriptor

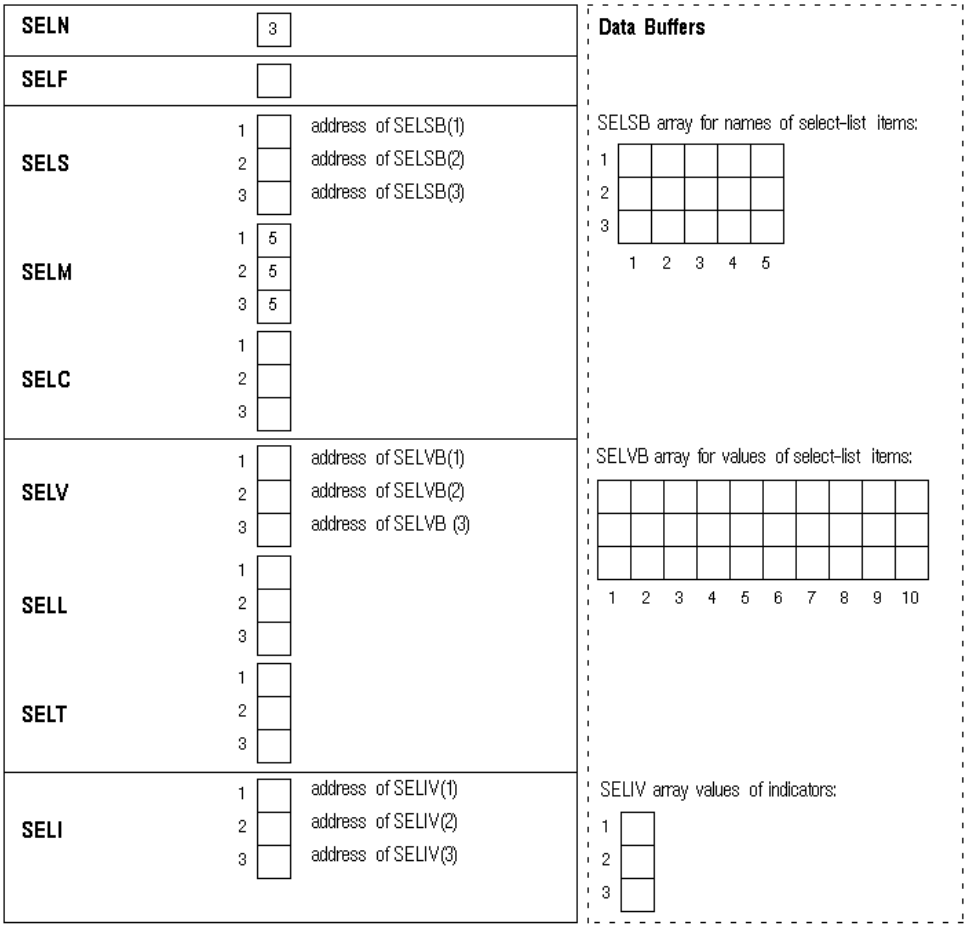
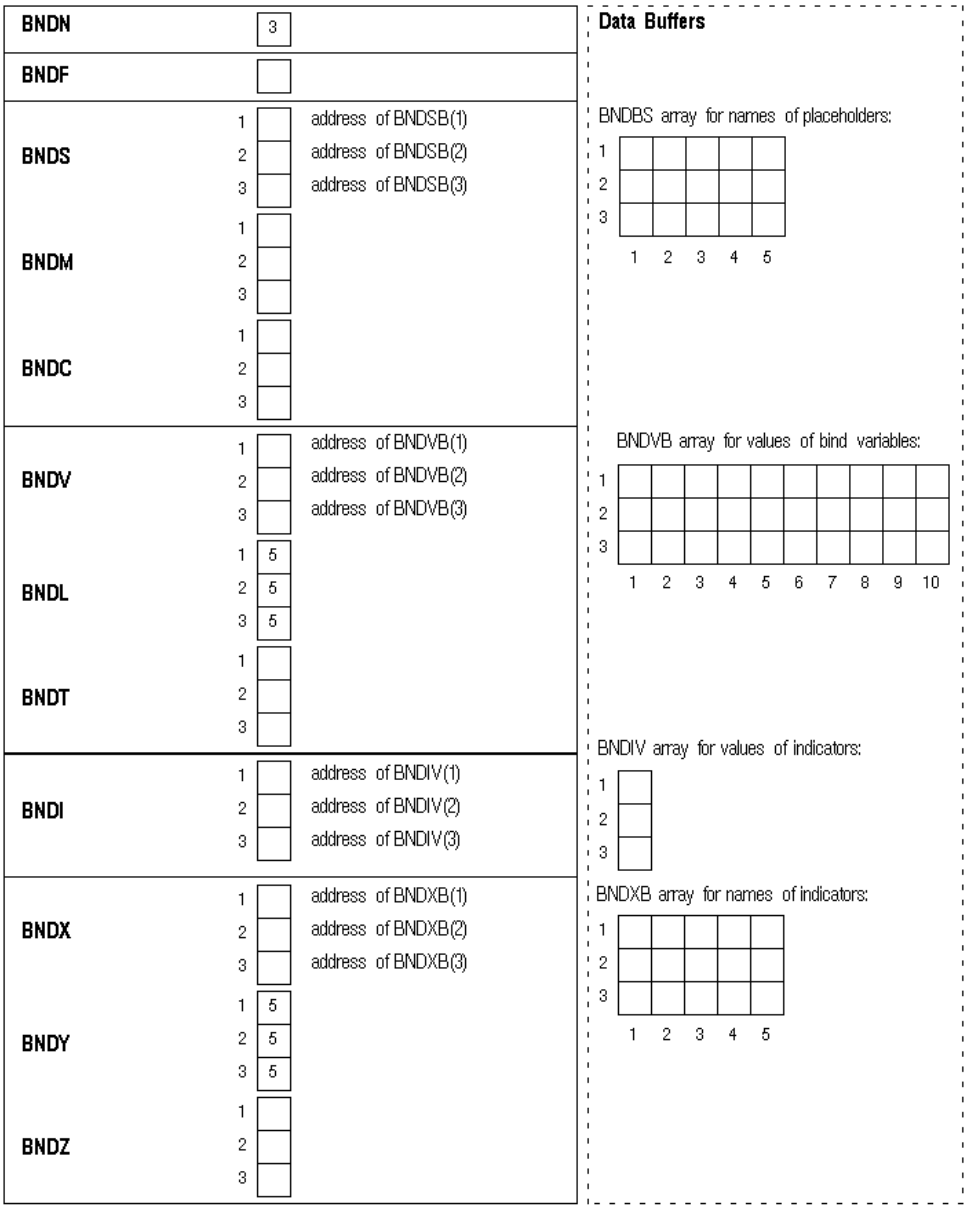


Figure 4-4 Initialized Bind Descriptor



4.6.5 Store the Query Text in the Host String

Continuing our example, you prompt the user for a SQL statement, then store the input string in SELSTM as follows:

```
WRITE (*, 1900)
 1900 FORMAT (' Enter query: ')
 READ (*, 2000) SELSTM
 2000 FORMAT (A120)
```

We assume the user entered the following string:

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

4.6.6 PREPARE the Query from the Host String

PREPARE parses the SQL statement and gives it a name. In our example, PREPARE parses the host string SELSTM and gives it the name DYNSTMT, as follows:

```
EXEC SQL PREPARE DYNSTMT FROM :SELSTM
```

4.6.7 DECLARE a Cursor

DECLARE CURSOR defines a cursor by giving it a name and associating it with a specific SELECT statement.

To declare a cursor for *static* queries, you use the following syntax:

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

To declare a cursor for *dynamic* queries, you substitute the statement name given to the dynamic query by PREPARE for the static query. In our example, DECLARE CURSOR defines a cursor named EMPCUR and associates it with DYNSTMT, as follows:

```
EXEC SQL DECLARE EMPCUR CURSOR FOR DYNSTMT
```

Note: You must declare a cursor for all dynamic SQL statements, not just queries. With non-queries, OPENing the cursor executes the dynamic SQL statement.

4.6.8 DESCRIBE the Bind Variables

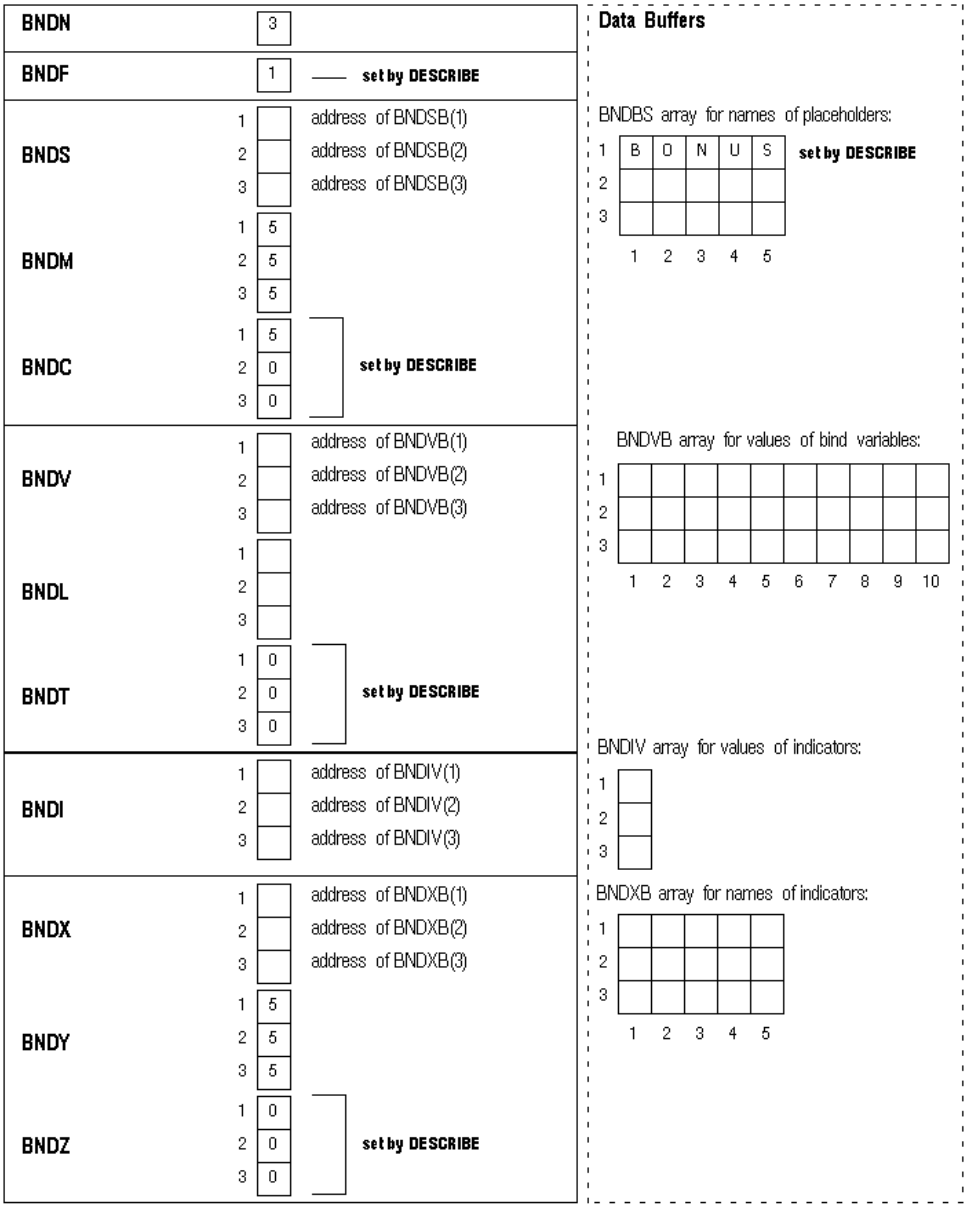
DESCRIBE BIND VARIABLES puts descriptions of bind variables into a bind descriptor. In our example, DESCRIBE readies BND as follows:

```
EXEC SQL DESCRIBE BIND VARIABLES FOR DYNSTMT INTO BND
```

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

Figure 4-5 shows the bind descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set BNDF to the actual number of placeholders found in the processed SQL statement.

Figure 4-5 Bind Descriptor after the DESCRIBE



4.6.9 Reset Number of Placeholders

Next, you must reset the maximum number of placeholders to the number actually found by DESCRIBE, as follows:

```
BNDN = BNDF
```

4.6.10 Get Values for Bind Variables

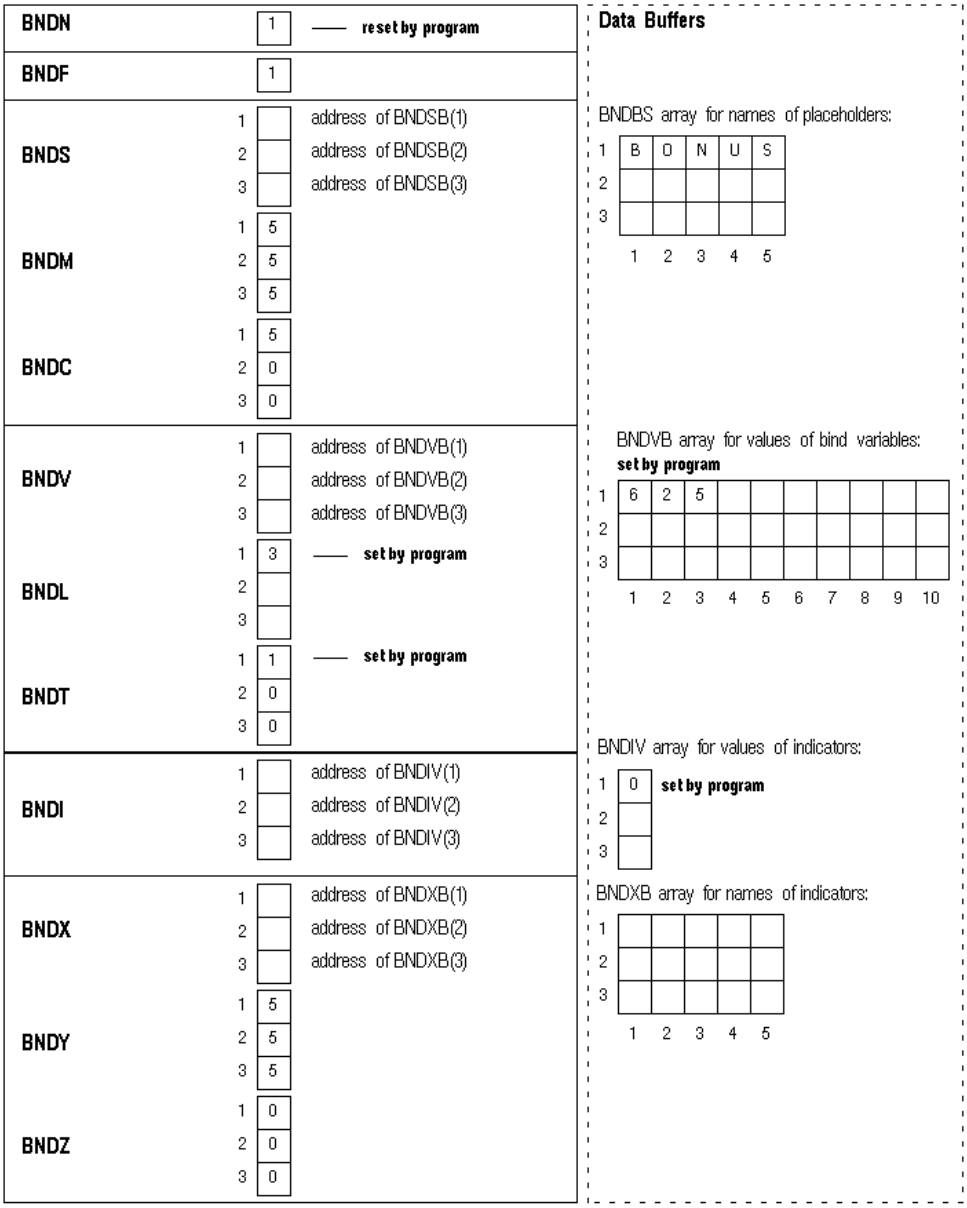
Your program must get values for the bind variables in the SQL statement. How the program gets the values is up to you. For example, they can be hard coded, read from a file, or entered interactively.

In our example, a value must be assigned to the bind variable that replaces the placeholder BONUS in the query's WHERE clause. Prompt the user for the value, then process it as follows:

```
CHARACTER*1 COLON
 COLON = ':'
* BNDN was set equal to BNDF after the DESCRIBE.
 DO 500 J = 1, BNDN
* Prompt user for value of bind variable.
 WRITE (*, 10200) (BNDSB(K,J), K = 1, BNDC(J)), COLON
10200 FORMAT (1X, 'Enter value for ', 6A1)
* Get value for bind variable.
 READ (*, 10300) (BNDVB(K,J), K = 1, 10)
10300 FORMAT (10A1)
* Find length of value by scanning backward for first
* non-blank character.
 DO 200 K = 1, 10
 IF (BNDVB(BNDL(J),J) .NE. ' ') GOTO 300
 BNDL(J) = BNDL(J) - 1
 200 CONTINUE
* Set datatype of bind variable to VARCHAR2 (code 1), and set
* indicator variable to NOT NULL.
 300 BNDT(J) = 1
 BNDIV(J) = 0
 500 CONTINUE
```

Assuming that the user supplied a value of 625 for BONUS, Figure 4-6 shows the resulting bind descriptor.

Figure 4-6 Bind Descriptor After Assigning Values



4.6.11 OPEN the Cursor

The OPEN statement for dynamic queries is similar to the one for static queries, except the cursor is associated with a bind descriptor. Values determined at run time and stored in buffers addressed by elements of the bind descriptor arrays are used to evaluate the SQL statement. With queries, the values are also used to identify the active set.

In our example, OPEN associates EMPCUR with BND as follows:

```
EXEC SQL OPEN EMPCUR USING DESCRIPTOR BND
```

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

Then, OPEN executes the SQL statement. With queries, OPEN also identifies the active set and positions the cursor at the first row.

4.6.12 DESCRIBE the Select List

If the dynamic SQL statement is a query, the DESCRIBE SELECT LIST statement must follow the OPEN statement and must precede the FETCH statement.

DESCRIBE SELECT LIST puts descriptions of select-list items into a select descriptor. In our example, DESCRIBE readies SEL as follows:

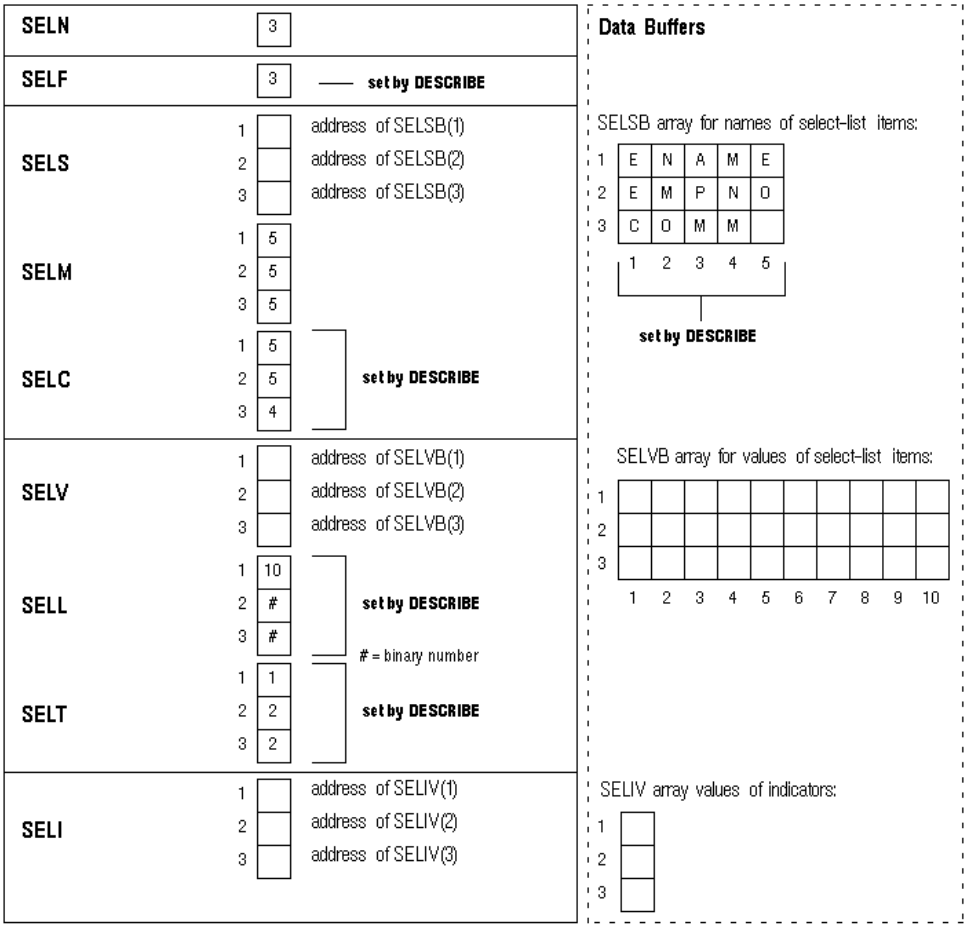
```
EXEC SQL DESCRIBE SELECT LIST FOR DYNSTMT INTO SEL
```

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

Figure 4-7 shows the select descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set SELF to the actual number of items found in the query select list. If the SQL statement is not a query, SELF is set to zero.

Also notice that the NUMBER lengths are not usable yet. For columns defined as NUMBER, you must use the library subroutine SQLPRC to extract precision and scale. See the section "Coercing Datatypes".

Figure 4-7 Select Descriptor after the DESCRIBE



4.6.13 Reset Number of Select-List Items

Next, you must reset the maximum number of select-list items to the number actually found by DESCRIBE, as follows:

```
SELN = SELF
```

4.6.14 Reset Length/Datatype of Each Select-List Item

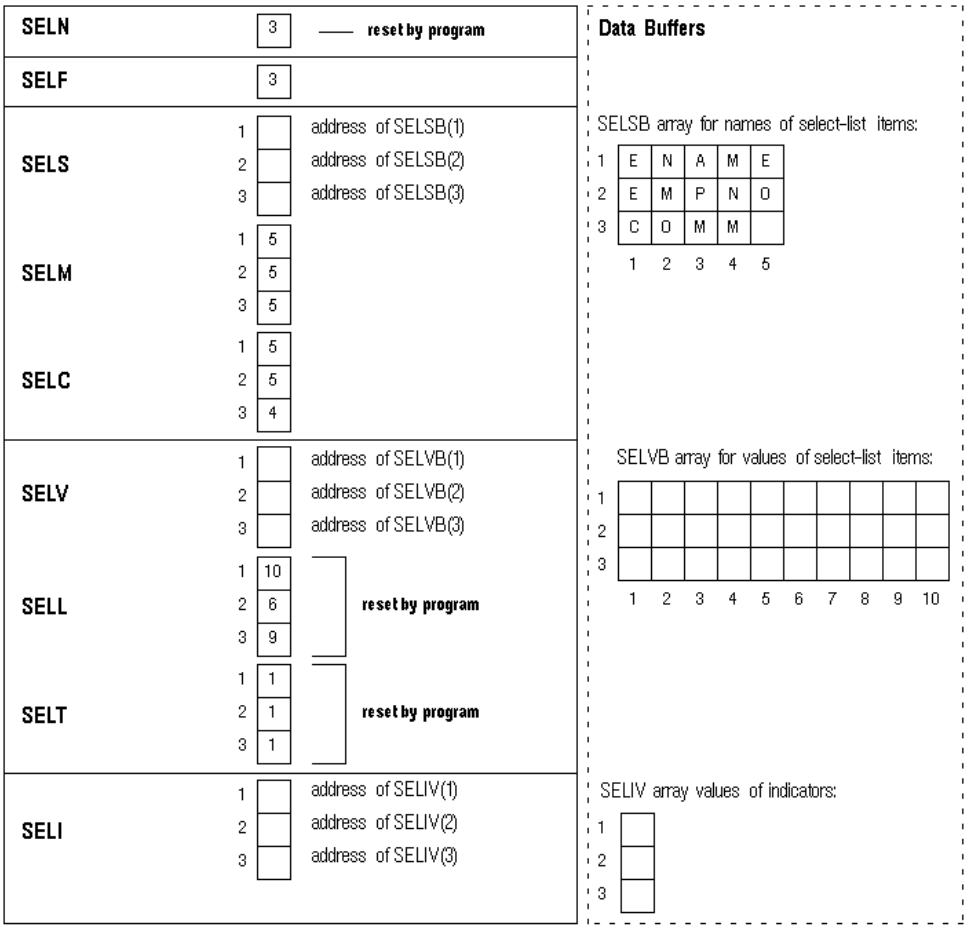
In our example, before fetching the select-list values, you reset some elements in the length and datatype arrays for display purposes.

```
DO 500 J = 1, SELN
* Clear null/not null bit.
 CALL SQLNUL (SELT(J), DATYPE, NULLOK)
 SELT(J) = DATYPE
* If datatype is NUMBER, extract precision and scale.
 IF (SELT(J) .NE. 2) GOTO 400
 CALL SQLPRC (SELL(J), PREC, SCALE)
* Allow for maximum precision.
 IF (PREC .NE. 0) GOTO 200
* Although maximum precision is 38, we use 10 because
* that is our buffer size.
 SELL(J) = 10
 GOTO 400
 200 CONTINUE
 SELL(J) = PREC
* Allow for possible sign and decimal point.
 SELL(J) = SELL(J) + 2
* Adjust length if it exceeds size of buffer. This
* applies to character as well as numeric data.
 400 IF (SELL(J) .GT. 10) SELL(J) = 10
* Coerce datatype to VARCHAR2.
 SELT(J) = 1
 500 CONTINUE
```

Figure 4-8 shows the resulting select descriptor. Notice that the NUMBER lengths are now usable and that all the datatypes are VARCHAR2. The lengths in SELL(2) and SELL(3) are 6 and 9 because we increased the DESCRIBEd lengths of 4 and 7 by two to allow for a possible sign and decimal point.

> **Note**
>
> When the datatype code returned by DESCRIBE is 2 (NUMBER), it must be coerced to a compatible FORTRAN type. The FORTRAN type need not be CHARACTER. For example, you can coerce a NUMBER to a REAL by setting SELT(J) to 4, and SELL(J) to the length of REALs on your system.

Figure 4-8 Select Descriptor before the FETCH



4.6.15 FETCH Rows from the Active Set

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

```
EXEC SQL FETCH EMPCUR USING DESCRIPTOR SEL
```

Figure 4-9 shows the select descriptor in our example after the FETCH. Notice that Oracle has stored the select-list and indicator values in the data buffers addressed by the elements of SELV and SELI.

For output buffers of datatype 1, Oracle, using the lengths stored in SELL, left-justifies CHAR or VARCHAR2 data, and right-justifies NUMBER data.

The value 'MARTIN' was retrieved from a VARCHAR2(10) column in the EMP table. Using the length in SELL(1), Oracle left-justifies the value in a 10-byte field, filling the buffer.

The value 7654 was retrieved from a NUMBER(4) column and coerced to "7654." However, the length in SELL(2) was increased by two to allow for a possible sign and decimal point, so Oracle right-justifies the value in a 6-byte field.

The value 482.50 was retrieved from a NUMBER(7,2) column and coerced to "482.50." Again, the length in SELL(3) was increased by two, so Oracle right-justifies the value in a 9-byte field.

4.6.16 Get and Process Select-List Values

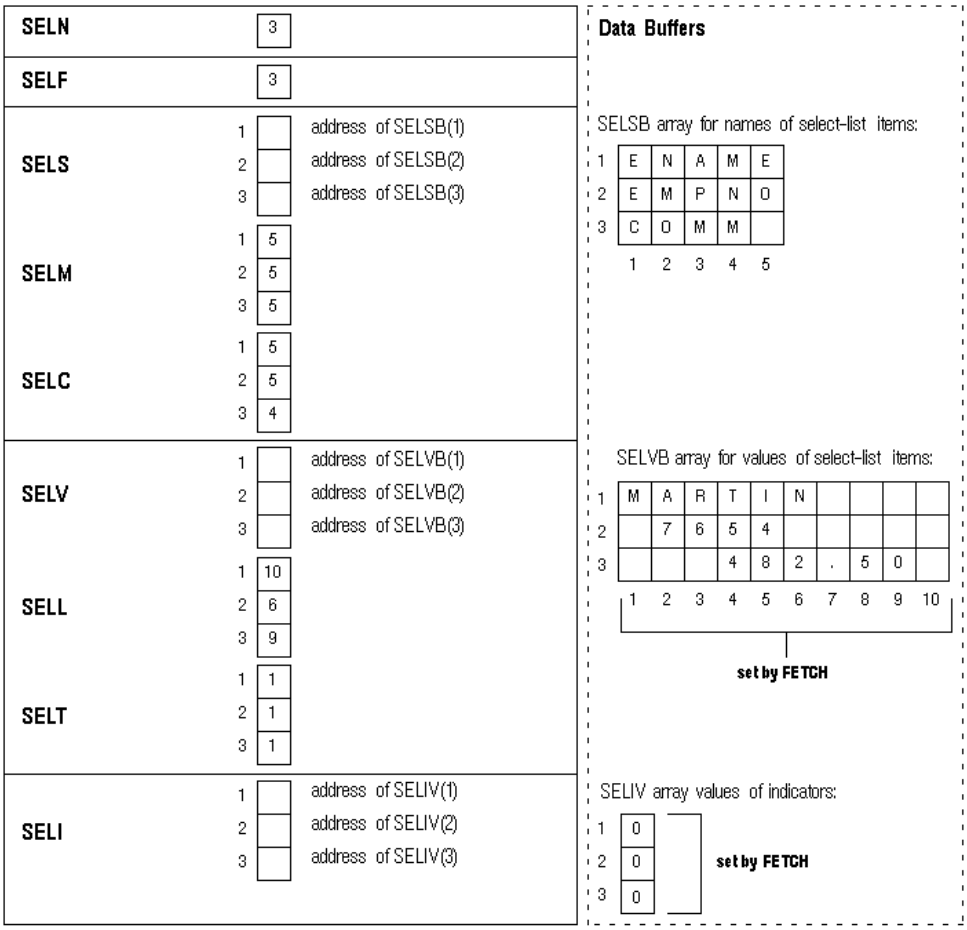
After the FETCH, your program can process the select-list values returned by FETCH. In our example, values for columns ENAME, EMPNO, and COMM are processed.

4.6.17 CLOSE the Cursor

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

```
EXEC SQL CLOSE EMPCUR
```

Figure 4-9 Select Descriptor after the FETCH



4.7 Using Host Arrays with Method 4

To use input or output host arrays with Method 4, you must use the optional FOR clause to tell Oracle the size of your host array. For more information about the FOR clause, see About Using the FOR Clause.

Set descriptor entries for the Jth select-list item or bind variable, but instead of addressing a single data buffer, SELV(J) or BNDV(J) addresses the first element of a data buffer array. Then

use a FOR clause in the EXECUTE or FETCH statement, as appropriate, to tell Oracle the number of table elements you want to process.

This procedure is necessary, because Oracle has no other way of knowing the size of your host ARRAY.

In the following example, an input host array is used to DELETE rows from the EMP table. Note that EXECUTE can be used for non-queries with Method 4.

```
* Use host arrays with Method 4.
 PROGRAM DYN4HA
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*20 UID
 CHARACTER*20 PWD
 CHARACTER*60 STMT
 INTEGER*4 SIZE
 EXEC SQL END DECLARE SECTION
 EXEC SQL INCLUDE SQLCA
 CHARACTER*10 NAMES(5)
 INTEGER*2 NUMBERS(5)
 INTEGER*2 DEPTS(5)
 EXEC SQL INCLUDE BNDDSC
 EXEC SQL WHENEVER SQLERROR GOTO 9000
 UID = 'SCOTT'
 PWD = 'TIGER'
* Log on to Oracle.
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD
 WRITE (*, 10000)
10000 FORMAT (' Connected to Oracle')
 SIZE = 5
 STMT = 'INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
 1 VALUES (:E, :N, :D)'
* Prepare and describe the SQL statement.
 EXEC SQL PREPARE S FROM :STMT
 BNDN = 3
 EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO BND
* Initialize bind descriptor items.
 BNDN = BNDF
 CALL SQLADR(NUMBERS(1), BNDV(1))
 BNDL(1) = 2
 BNDT(1) = 3
 BNDI(1) = 0
* %REF is used to pass the address of the data, not
* of the FORTRAN compiler-generated descriptor of
* CHARACTER variable NAMES. (See the section "Using
* SQLADR" earlier in this chapter.)
 CALL SQLADR(%REF(NAMES(1)), BNDV(2))
 BNDL(2) = 10
 BNDT(2) = 1
 BNDI(2) = 0
 CALL SQLADR(DEPTS(1), BNDV(3))
 BNDL(3) = 2
 BNDT(3) = 3
 BNDI(3) = 0
 DO 110 I = 1, SIZE
 BNDM(I) = 0
 BNDY(I) = 0
 BNDX(I) = 0
 110 CONTINUE
* Fill the data buffers. Normally, this data would
* be entered interactively by the user, or read from
* a file.
```

```
 NAMES(1) = 'TRUSDALE'
 NUMBERS(1) = 1014
 DEPTS(1) = 30
 NAMES(2) = 'WILKES'
 NUMBERS(2) = 1015
 DEPTS(2) = 30
 NAMES(3) = 'BERNSTEIN'
 NUMBERS(3) = 1016
 DEPTS(3) = 30
 NAMES(4) = 'FRAZIER'
 NUMBERS(4) = 1017
 DEPTS(4) = 30
 NAMES(5) = 'MCCOMB'
 NUMBERS(5) = 1018
 DEPTS(5) = 30
* Do the INSERT.
 WRITE (*, 10020)
10020 FORMAT(1X, 'Adding to Sales force ...')
 EXEC SQL FOR :SIZE EXECUTE S USING DESCRIPTOR BND
 EXEC SQL COMMIT RELEASE
 GOTO 150
* Here if SQLERROR occurred.
 9000 CONTINUE
 WRITE (*, 10030) SQLEMC
10030 FORMAT (1X, 70A1)
 EXEC SQL WHENEVER SQLERROR CONTINUE
 EXEC SQL ROLLBACK RELEASE
* Here when ready to exit the program.
 150 CONTINUE
 STOP
 END
```

4.8 Sample Program 10: Dynamic SQL Method 4

This program shows the basic steps required to use dynamic SQL Method 4. After logging on to Oracle, the program prompts the user for a SQL statement, PREPAREs the statement, DECLAREs a cursor, checks for any bind variables using DESCRIBE BIND, OPENs the cursor, and DESCRIBEs any select-list items. If the input SQL statement is a query, the program FETCHes each row of data, then CLOSEs the cursor. Notice that a VARCHAR is used to store the dynamic SQL statement.

```
PROGRAM DYN4
 EXEC SQL BEGIN DECLARE SECTION
 CHARACTER*20 UID
 CHARACTER*20 PWD
 VARCHAR *1024 STMT, STMTL, STMTA
 EXEC SQL END DECLARE SECTION
 CHARACTER*1 ANS
 EXEC SQL INCLUDE SQLCA
 EXEC SQL INCLUDE BNDDSC
 EXEC SQL INCLUDE SELDSC

* INITIALIZE.
 CALL INIT

* LOG ON TO ORACLE.
10 PRINT *, 'ENTER USERNAME:'
 READ '(20A)', UID
 PRINT *, 'ENTER PASSWORD:'
 READ '(20A)', PWD
 EXEC SQL WHENEVER SQLERROR GOTO 8500
```

```
 EXEC SQL CONNECT :UID IDENTIFIED BY :PWD

 EXEC SQL WHENEVER SQLERROR GOTO 9000
 PRINT *,
 1'TO EXIT, TYPE NULL SQL STATEMENT (;) AT DSQL PROMPT.'

* GET SQL STATEMENT FROM USER.
100 CONTINUE
 CALL GETSQL (STMTA, STMTL)
 IF (STMTL .EQ. 0) GOTO 9500

* PREPARE THE SQL STATEMENT, AND DECLARE A CURSOR FOR IT.
 EXEC SQL PREPARE S FROM :STMT
 EXEC SQL DECLARE C CURSOR FOR S

* DESCRIBE THE BIND VARIABLES. FIRST, INITIALIZE BNDN TO
* THE MAXIMUM NUMBER OF VARIABLES THAT CAN BE DESCRIBED.
 BNDN = 20
 EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO BND
 IF (BNDF .GE. 0) GOTO 125
 PRINT *, 'TOO MANY BIND VARIABLE - TRY AGAIN...'
 GOTO 300

* HAVE DESCRIBED BIND VARIABLES. GET VALUES FOR ANY
* BIND VARIABLES.
125 BNDN = BNDF
 IF (BNDN .GT. 0) CALL GETBND

* OPEN CURSOR TO EXECUTE THE SQL STATEMENT.
 EXEC SQL OPEN C USING DESCRIPTOR BND

* DESCRIBE THE SELECT-LIST ITEMS. FIRST, INITIALIZE SELN TO
* THE MAXIMUM NUMBER OF ITEMS THAT CAN BE DESCRIBED.
 SELN = 20
 EXEC SQL DESCRIBE SELECT LIST FOR S INTO SEL
 IF (SELF .GE. 0) GOTO 150
 PRINT *, 'TOO MANY SELECT-LIST ITEMS. TRY AGAIN...'
 GOTO 300

* HAVE DESCRIBED SELECT LIST. IF THIS IS A SELECT STATEMENT,
* RESET LENGTHS AND DATATYPES OF FETCHED VALUES, AND OUTPUT
* COLUMN HEADINGS.
150 SELN = SELF
 IF (SELN .EQ. 0) GO TO 300
 CALL PRCOLH

* FETCH EACH ROW, AND PRINT IT.
 EXEC SQL WHENEVER NOT FOUND GOTO 300
200 EXEC SQL FETCH C USING DESCRIPTOR SEL
 CALL PRROW
 GOTO 200

* THERE ARE NO MORE ROWS (ROW NOT FOUND), OR NON-SELECT
* STATEMENT COMPLETED.
300 EXEC SQL CLOSE C
 IF (SELN .EQ. 0) GOTO 310

* THERE WERE SOME SELECT-LIST ITEMS, SO SQL STATEMENT
* MUST BE A SELECT.
 PRINT *, SQLERD(3), ' ROW(S) SELECTED.'
 GOTO 100
```

```
* THERE WERE NO SELECT-LIST ITEMS, SO SQL STATEMENT
* CANNOT BE A SELECT.
310 PRINT *, SQLERD(3), ' ROW(S) PROCESSED.'
 GOTO 100

* A SQL EXECUTION ERROR (SQLERROR) OCCURRED.
* CONNECT ERROR
8500 PRINT '(70A1)', SQLEMC
 PRINT *, 'TRY AGAIN (Y OR N)?'
 READ '(A1)', ANS
 IF ((ANS .EQ. 'Y') .OR. (ANS .EQ. 'Y')) GOTO 10
 GOTO 9500

* OTHER SQL ERRORS
9000 PRINT '(70A1)', SQLEMC
 GOTO 100

* NOW READY TO EXIT PROGRAM.
9500 EXEC SQL WHENEVER SQLERROR CONTINUE
 EXEC SQL COMMIT WORK RELEASE
 PRINT *, 'HAVE A GOOD DAY.'
9600 CONTINUE
 END

*********************************************************
* NAME: INIT (INITIALIZE)
* FUNCTION: INITIALIZES THE BIND AND SELECT DESCRIPTORS.
* RETURNS: NONE
*********************************************************
 SUBROUTINE INIT

 EXEC SQL INCLUDE BNDDSC
 EXEC SQL INCLUDE SELDSC

* INITIALIZE BIND DESCRIPTOR ITEMS.
 DO 100 I = 1, 20
 CALL SQLADR (BNDSB(1,I), BNDS(I))
 CALL SQLADR (BNDVB(1,I), BNDV(I))
 CALL SQLADR (BNDXB(1,I), BNDX(I))
 CALL SQLADR (BNDIV(I), BNDI(I))
 BNDM(I) = 30
 BNDY(I) = 30
100 CONTINUE

* INITIALIZE SELECT DESCRIPTOR ITEMS.
 DO 200 I = 1, 20
 CALL SQLADR (SELSB(1,I), SELS(I))
 CALL SQLADR (SELVB(1,I), SELV(I))
 CALL SQLADR (SELIV(I), SELI(I))
 SELM(I) = 30
200 CONTINUE
 RETURN
 END

*********************************************************
* NAME: GETSQL (GET SQL STATEMENT FROM TERMINAL)
* FUNCTION: ASKS THE USER TO TYPE IN A SQL STATEMENT.
* RETURNS: SARR IS A STRING (LOGICAL*1) CONTAINING
* THE SQL STATEMENT. SLEN IS THE NUMBER OF
* CHARACTERS IN SARR. IF SLEN IS 0, THEN NO
* SQL STATEMENT WAS ENTERED (DSQL USES THIS
* TO INDICATE THAT USER WANTS TO LOG OFF).
```

```
*******************************************************
 SUBROUTINE GETSQL (SARR, SLEN)

 LOGICAL*1 SARR(1)
 INTEGER*2 SLEN
 LOGICAL*1 INP(80)
 INTEGER INPL
 INTEGER CNTLIN

 CNTLIN = 0
 SLEN = 0
 PRINT *, 'DSQL>'
50 READ '(80A1)', (INP(I), I = 1, 80)

* FIND LENGTH OF SQL STATEMENT BY SCANNING BACKWARD FOR
* FIRST NON-BLANK CHARACTER.
 INPL = 80
 DO 100 I = 1, 80
 IF (INP(INPL) .NE. ' ') GOTO 150
 INPL = INPL - 1
100 CONTINUE

* MOVE THIS PIECE OF THE SQL STATEMENT TO SQL STATEMENT
* BUFFER.
150 CONTINUE
 DO 200 I = 1, INPL
 SLEN = SLEN + 1
 IF (SLEN .GT. 1024) GOTO 1000
 SARR(SLEN) = INP(I)
200 CONTINUE
 IF (SARR(SLEN) .EQ. ';') GOTO 1000
* LINE NOT TERMINATED BY ';'. REQUEST CONTINUED LINE.
 CNTLIN = CNTLIN + 1
 WRITE (*, 10300) CNTLIN
10300 FORMAT ('$', I5, ':')

 SLEN = SLEN + 1
 IF (SLEN .GT. 1024) GOTO 1000
 SARR(SLEN) = ' '
 GOTO 50
1000 CONTINUE
 SLEN = SLEN - 1
 RETURN
 END

*******************************************************
* NAME: PRCOLH (PRINT COLUMN HEADINGS)
* FUNCTION: RESETS LENGTH AND DATATYPE ARRAYS IN SELECT
* DESCRIPTOR, AND PRINTS COLUMN HEADINGS FOR
* SELECT-LIST ITEMS.
* NOTES: FOR EXAMPLE, GIVEN THE STATEMENT
*
* SELECT TNAME, TABTYPE FROM TAB
*
* AND ASSUMING TNAME COLUMN IS 30 CHARACTERS
* WIDE AND TABTYPE COLUMN IS 7 CHARACTERS WIDE,
* PRCOLH PRINTS:
*
* TNAME TABTYPE
* ------------------------------ -------
*******************************************************
 SUBROUTINE PRCOLH
```

```
      EXEC SQL INCLUDE SELDSC
      LOGICAL*1 LINE(132)
      INTEGER LINESZ
      INTEGER PREC, SCALE, NULLOK
      INTEGER*2 DATYPE

      PREC = 26
      SCALE = 0
      LINESZ = 132
      L = 0

      DO 500 I = 1, SELN

*     SQLPRC IS USED TO EXTRACT PRECISION AND SCALE FROM THE
*     LENGTH (SELL(I)).

*     SQLNUL IS USED TO RESET HIGH ORDER BIT OF THE DATATYPE
*     AND TO CHECK IF THE COLUMN IS NOT NULL.

*     CHAR DATATYPES HAVE LENGTH, BUT ZERO PRECISION AND
*     SCALE. THE LENGTH IS THAT DEFINED AT CREATE TIME.

*     NUMBER DATATYPES HAVE PRECISION AND SCALE IF DEFINED
*     AT CREATE TIME. HOWEVER, IF THE COLUMN DEFINITION
*     WAS JUST NUMBER, THE PRECISION AND SCALE ARE ZERO,
*     SO WE DEFAULT THE COLUMN WIDTH TO 10.

*     RIGHT JUSTIFY COLUMN HEADING FOR NUMBERS.

      CALL SQLNUL (SELT(I), DATYPE, NULLOK)
      SELT(I) = DATYPE
      IF (SELT(I) .NE. 2) GOTO 150
      CALL SQLPRC (SELL(I), PREC, SCALE)

*     IF NO PRECISION, USE DEFAULT.
      IF (PREC .EQ. 0) PREC = 10
      SELL(I) = PREC

*     ADD 2 FOR POSSIBLE SIGN AND DECIMAL POINT.
      SELL(I) = SELL(I) + 2

*     BLANK-PAD COLUMN NAME TO RIGHT-JUSTIFY COLUMN HEADING.
      NBLANKS = SELL(I) - SELC(I)
      DO 130 J = 1, NBLANKS
      L = L + 1
      IF (L .GT. LINESZ - 1) GOTO 450
      LINE(L) = ' '
130   CONTINUE
      GOTO 190

*     CHECK FOR LONG COLUMN, AND SET DATA BUFFER
*     LENGTH TO 240.
150   IF (SELT(I) .NE. 8) GOTO 153
      SELL(I) = 240
      GOTO 190

*     CHECK FOR LONG RAW COLUMN, AND SET DATA BUFFER
*     LENGTH TO 240.
153   IF (SELT(I) .NE. 24) GOTO 155
      SELL(I) = 240
      GOTO 190
```

```
* CHECK FOR ROWID COLUMN, AND SET DATA BUFFER
* LENGTH TO 18 (DISPLAY LENGTH).
155 IF (SELT(I) .NE. 11) GOTO 160
 SELL(I) = 18
 GOTO 190

* CHECK FOR DATE COLUMN, AND SET DATA BUFFER LENGTH
* TO 9 (DEFAULT FORMAT IS DD-MON-YY).
160 IF (SELT(I) .NE. 12) GOTO 165
 SELL(I) = 9
 GOTO 190

* CHECK FOR RAW COLUMN, AND ADD 1 TO DATA BUFFER LENGTH.
165 IF (SELT(I) .NE. 23) GOTO 190
 SELL(I) = SELL(I) + 1

* COPY COLUMN NAME TO OUTPUT LINE.
190 DO 200 J = 1, MIN (SELC(I), SELL(I))
 L = L + 1
 IF (L .GT. LINESZ - 1) GOTO 450
 LINE(L) = SELSB(J, I)
200 CONTINUE

* PAD COLUMN NAME WITH BLANKS PLUS 1 FOR INTER-COLUMN
* SPACING. NOTE THAT NUMBER COLUMNS ARE RIGHT-JUSTIFIED
* SO JUST ONE BLANK IS NEEDED FOR INTER-COLUMN SPACING.
 NBLANKS = 1
 IF (SELT(I) .EQ. 2) GOTO 210
 NBLANKS = MAX (SELL(I) - SELC(I) + 1, 1)
210 DO 300 J = 1, NBLANKS
 L = L + 1
 IF (L .GT. LINESZ - 1) GOTO 450
 LINE(L) = ' '
300 CONTINUE

* EXCEPT FOR LONG RAW COLUMNS, COERCE COLUMN
* DATATYPE TO VARCHAR2 TO SIMPLIFY PRINTING ROW.
450 IF (SELT(I) .NE. 24) SELT(I) = 1
500 CONTINUE

* NOW READY TO PRINT THE HEADING LINE.
1000 WRITE (*, 10100) (LINE(I), I = 1, L)
10100 FORMAT (/, 1X, 132A1)

* UNDERLINE THE COLUMN HEADINGS.
 L = 0
 DO 1500 I = 1, SELN
 NUNDER = SELL(I)
 DO 1250 J = 1, NUNDER
 L = L + 1
 IF (L .GT. LINESZ - 1) GOTO 2000
 LINE(L) = '-'
1250 CONTINUE
 L = L + 1
 IF (L .GT. LINESZ - 1) GOTO 2000
 LINE(L) = ' '
1500 CONTINUE

* NOW READY TO PRINT THE UNDERLINE.
2000 WRITE (*, 10200) (LINE(I), I = 1, L)
10200 FORMAT (1X, 132A1)
 RETURN
```

```
 END

*******************************************************
* NAME: PRROW (PRINT ROW)
* FUNCTION: PRINTS A SINGLE FETCHED ROW.
*******************************************************
 SUBROUTINE PRROW
 EXEC SQL INCLUDE SELDSC
 LOGICAL*1 LINE(132)
 INTEGER LINESZ

 LINESZ = 132
 L = 0
 DO 500 I = 1, SELN

* CHECK FOR NULL COLUMN. IF NULL, BLANK-PAD COLUMN.
 IF (SELIV(I) .GE. 0) GOTO 100
 DO 90 J = 1, SELL(I)
 L = L + 1
 IF (L .GT. LINESZ - 1) GOTO 1000
 LINE(L) = ' '
90 CONTINUE
 GOTO 250

* COLUMN DATATYPE IS VARCHAR2. COPY COLUMN VALUE TO
* OUTPUT LINE.
100 CONTINUE
 DO 200 J = 1, SELL(I)
 L = L + 1
 IF (L .GT. LINESZ - 1) GOTO 1000
 LINE(L) = SELVB(J, I)
200 CONTINUE

* APPEND ONE BLANK FOR INTER-COLUMN SPACING.
250 CONTINUE
 L = L + 1
 IF (L .GT. LINESZ - 1) GOTO 1000
 LINE(L) = ' '
500 CONTINUE

* NOW READY TO PRINT THE LINE.
1000 WRITE (*, 10100) (LINE(I), I = 1, L)
10100 FORMAT (1X, 132A1)
 RETURN
 END

*******************************************************
* NAME: GETBND (GET BIND VARIABLES)
* FUNCTION: USING THE DESCRIPTOR BND, SET UP BY
* THE DESCRIBE BIND VARIABLES STATEMENT,
* GETBND PROMPTS THE USER FOR VALUES OF BIND
* VARIABLES.
* RETURNS: BNDVB AND BNDL ARRAYS SET UP WITH VALUES
* FOR BIND VARIABLES.
*******************************************************
 SUBROUTINE GETBND
 EXEC SQL INCLUDE BNDDSC
 CHARACTER*1 CLN, SPC

 CLN = ':'
 SPC = ' '
 WRITE (*, 10100)
```

```
10100 FORMAT (/, 'PLEASE ENTER VALUES OF BIND VARIABLES.', /)
 DO 500 I = 1, BNDN
 WRITE (*, 10200)(BNDSB(J, I), J = 1, BNDC(I)), CLN, SPC
10200 FORMAT ('ENTER VALUE FOR ', 32A1)

* GET VALUE FOR BIND VARIABLE.

 READ '(80A1)', (BNDVB(J, I), J = 1, 80)

* FIND LENGTH OF VALUE BY SCANNING BACKWARD
* FOR FIRST NON-BLANK CHARACTER.
 BNDL(I) = 80
 DO 200 J = 1, 80
 IF (BNDVB(BNDL(I), I) .NE. ' ') GOTO 300
 BNDL(I) = BNDL(I) - 1
200 CONTINUE

* SET DATATYPE OF BIND VARIABLE TO VARCHAR2, AND SET
* INDICATOR VARIABLE TO NOT NULL.
300 CONTINUE
 BNDT(I) = 1
 BNDIV(I) = 0
500 CONTINUE
 RETURN
 END
```

A

Operating System Dependencies

Some details of Pro*FORTRAN programming vary from one system to another. This appendix is a collection all system-specific issues regarding Pro*FORTRAN. References are provided, where applicable, to other sources in your document set.

This appendix contains the following sections:

A.1 System-Specific References for Chapter 1

For more information, refer to .

A.1.1 Case-sensitivity

Though the standard FORTRAN character set excludes lowercase alpha characters, many compilers allow them in identifiers, comments, and quoted literals.

The Pro*FORTRAN Precompiler is not case-sensitive; however, some compilers are. If your compiler is case-sensitive, you must declare and reference variables in the same uppercase/lowercase format. Check your FORTRAN compiler user's guide.

A.1.2 Coding Area

You must code EXEC SQL and EXEC ORACLE statements in columns 7 through 72 (columns 73 through 80 are ignored). The other columns are used for the following purposes: column 1 indicates comment lines, columns 1 through 5 contain an optional statement label, and column 6 indicates continuation lines.

On some systems, *terminal format* is supported; that is, entry is not restricted to certain columns. Check your Oracle system-specific documentation.

No more than one statement can appear on a single line.

A.1.3 Continuation Lines

You can continue SQL statements from one line to the next according to the rules of FORTRAN. To code a continuation line, place a nonzero, non-blank character in column 6. In this manual, digits are used as continuation characters, as the following example shows:

```
* Retrieve employee data.
 EXEC SQL SELECT EMPNO, ENAME, JOB, SAL
 1 INTO :MYEMPNO, :MYENAME, :MYJOB, :MYSAL
 2 FROM EMP
 3 WHERE DEPTNO = :MYDEPTNO
```

You can also continue string literals from one line to the next. Code the literal through column 72, then, on the next line, code a continuation character and the rest of the literal. An example follows:

```
* Execute dynamic SQL statement.
 EXEC SQL EXECUTE IMMEDIATE 'UPDATE EMP SET COMM = 500 WHERE
 1 DEPTNO=20'
```

Most FORTRAN implementations allow up to 19 continuation lines. Check your FORTRAN language user's guide.

A.1.4 FORTRAN Versions

The Pro*FORTRAN Precompiler supports the standard implementation of FORTRAN for your operating system (usually FORTRAN 77). Check your Oracle system-specific documentation.

How you declare and name host variables depends on which FORTRAN compiler you use. Check your FORTRAN user's guide for details about declaring and naming host variables.

A.1.5 Declaring

Declare host variables in the Declare Section according to FORTRAN rules, specifying a FORTRAN datatype supported by Oracle. Table 1 - 3 shows the FORTRAN datatypes and pseudotypes you can specify in the Declare Section. However, your FORTRAN implementation might not include all of them.

The host datatypes and pseudotypes you can specify in the Declare Section are shown in the table on page 1-10. However, your implementation might not include all of them. Check your FORTRAN language user's guide.

The size of FORTRAN numeric types is implementation-dependent. The sizes given in the table are typical but not universal. Check your FORTRAN language user's guide.

A.1.6 Naming

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. Some compilers prohibit variable names longer than six characters, or ignore characters after the sixth. Check your FORTRAN compiler user's guide.

A.1.7 INCLUDE Statements

You can INCLUDE any file. When you precompile your Pro*FORTRAN 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*FORTRAN Precompiler assumes the default extension for source files (usually FOR or F). The default extension is system-dependent. Check your Oracle system-specific documentation.

If your system uses directories, you can set a directory path for INCLUDEd files by specifying the precompiler option INCLUDE=*path*. You must use INCLUDE to specify a directory path for nonstandard files unless they are stored in the current directory. The syntax for specifying a directory path is system-specific. Check your Oracle system-specific documentation.

A.1.8 MAXLITERAL Default

With the MAXLITERAL precompiler option you can specify the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded. The MAXLITERAL default value is 1000, but you might have to specify a lower value.

For example, if your FORTRAN compiler cannot handle string literals longer than 512 characters, specify "MAXLITERAL=512." Check your FORTRAN compiler user's guide. For more information about the MAXLITERAL option, see MAXLITERAL.

A.2 System-Specific Reference for Chapter 3

For more information, refer to .

A.2.1 Sample Programs

All the sample programs in this chapter are available online. The names of the online files are shown. However, the exact filenames are system-dependent. For more information, check your Oracle system-specific documentation.

A.3 System-Specific Reference for Chapter 4

For more information, refer to .

A.3.1 SQLADR

You cannot use CHARACTER variables with SQLADR if your FORTRAN compiler generates descriptors of CHARACTER variables and passes the descriptor address (rather than the data address) to SQLADR. Check your FORTRAN compiler user's guide. In such cases, SQLADR gets the wrong address. Instead, use LOGICAL*1 variables, because they always have simple addresses.

You can, however, use (cautiously) SQLADR with CHARACTER variables if your compiler provides a built-in function to access the data address. For example, if the compiler provides a function named %REF, and X is a CHARACTER variable, you call SQLADR as follows:

```
* Use %REF built-in function.
 CALL SQLADR (%REF(X), ...)
```

Index

A

W