Programmer's Guide to the Oracle Call Interface™

Release 7.3

February 1996
Part No. A32546–1

Programmer's Guide to the Oracle Call Interface™, Release 7.3

Part No. A32546–1

Contributing Authors: James F. Bisso, Phil Locke, Tim Smith
Contributors: Debashish Chatterjee, Luxi Chidambaran, Ziyad Dahbour, Jacco Draaijer, Jack Godwin, Ken Jacobs, Sandeep Jain, Shoaib Lari, Andrew Mendelsohn, Mark Moore, Valarie Moore, Jacqui Pons, Tom Portfolio, Tuomas Pystynen, Gael Turk, Scott Urman, Peter Vasterd

Preface

The Oracle Call Interface (OCI) Release 7.3 is an application programming interface (API) that allows applications written in a third generation programming language, such as C, COBOL, or FORTRAN, to issue SQL statements to one or more Oracle Servers. The OCI gives your program the capability to perform the full range of operations that are possible with Oracle Server, Release 7.3.

This Guide will give you a sound basis for developing applications using the OCI. It includes the following information:

- the structure of an OCI application
- conversion of data between the server and variables in your OCI application
- reference sections that describe OCI calls in detail
- sample programs that illustrate the features of the OCI

Audience

The *Programmer's Guide to the Oracle Call Interface* is intended for those developing new applications or converting existing applications to run in the Oracle environment. Written especially for programmers, this comprehensive treatment of the OCI will also be valuable to systems analysts, project managers, and others interested in the development of database applications.

To use this Guide effectively, you need a working knowledge of applications programming in either C, COBOL, or FORTRAN, and knowledge of the SQL relational database language. This Guide describes all concepts and terminology specific to the Oracle Call Interface.

For information about SQL, refer to the *Oracle7 Server SQL Reference* and the *Oracle7 Server Administrator's Guide*. For information about basic Oracle concepts, see *Oracle7 Server Concepts*. For information about the Oracle Precompilers, which enable you to embed SQL commands in a 3GL application, refer to the appropriate Precompiler manual.

How The *Programmer's Guide to the Oracle Call Interface* Is Organized

This Guide contains six chapters and eight appendices. A brief summary of what you will find in each chapter and appendix follows:

Chapter 1: Introduction
This chapter introduces you to the Oracle Call Interfaces and describes special terms and typographical conventions that are used in describing the interfaces. It also lists new OCI routines, as well as routines that are now obsolete or obsolescent.

Chapter 2: Writing an OCI Program
This chapter gives you the basic concepts that are used in developing an OCI program.

Chapter 3: Datatypes
Understanding how data is converted between Oracle tables and variables in your host program is essential for using the OCI interfaces. All the information you need is in Chapter 3.

Chapter 4: The OCI Functions for C
The OCI routines (functions) for C are documented in this chapter. Included are examples in C that show how they are used.

Chapter 5: The OCI Routines for COBOL
The OCI routines for COBOL are documented in this chapter. Included are examples in COBOL that show how they are used.

Chapter 6: The OCI Routines for FORTRAN
The OCI routines for FORTRAN are documented in this chapter. Included are examples in FORTRAN that show how they are used.

Appendix A: Sample Programs in C
This appendix contains five demonstration programs that show how to use the OCI routines in C programs.

Appendix B: Sample Programs in COBOL
This appendix contains three demonstration programs that show how to use the OCI routines in COBOL programs.

Appendix C: Sample Programs in FORTRAN
This appendix contains three demonstration programs that show how to use the OCI routines in FORTRAN programs.

Appendix D: Additional OCI Functions (C)
This appendix documents older OCI functions that are still available, but have been superseded by new routines that offer increased functionality or performance. These routines might become obsolete in future versions, so they should not be used in new OCI programs.

Appendix E: Additional OCI Functions (COBOL)
This appendix documents older OCI routines that are still available, but have been superseded by new routines that offer increased functionality or performance. These routines might become obsolete in future versions, so they should not be used in new OCI programs.

Appendix F: Additional OCI Functions (FORTRAN)
This appendix documents older OCI routines that are still available, but have been superseded by new routines that offer increased functionality or performance. These routines might become obsolete in future versions, so they should not be used in new OCI programs.

Appendix G: Operating System Dependencies
This appendix contains a convenient list of the aspects of OCI programming that depend on the system you are using. In these cases, your Oracle system–specific documentation spells out the appropriate values or options.

Appendix H: Oracle Reserved Words, Keywords and Namespaces
This appendix lists words that have a special meaning to Oracle, and namespaces reserved by Oracle products.

Conventions Used in this Guide

The following notational and text formatting conventions are used in this Guide:

[ ]	Square brackets indicate that the enclosed item is optional. Do not type the brackets.
< >	Angle brackets indicate that the enclosed item is an unused parameter. Do not type the brackets.
{ }	Braces enclose items of which only one is required.
|	A vertical bar separates items within braces.
...	In code fragments, an ellipsis means that code not relevant to the discussion has been omitted.
`font change`	Code examples are shown in monospaced font.
italics	Italics are used for lowercase OCI parameters, C OCI routines, filenames, and data fields.
UPPERCASE	Uppercase is used for SQL keywords. OCI routine names and parameter names are also in uppercase, except in C, where lowercase italic is used.
bold	Bold face type is used to identify datatype names in descriptions of C language calls.

Your Comments are Welcome

Your opinions are the most important feedback we receive. Please use the reader's comment form at the back of this book to tell us what you like and dislike about this Guide. If you prefer, send us a FAX at 415–506–7200, or write to us at:

Oracle Languages Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, California 94065

Contents

CHAPTER

1 Introduction

This chapter introduces you to the Oracle Call Interface, Release 7.3. It gives you the background information that you need to develop applications using the interface. It also introduces special terms that are used in discussing the interface. The following topics are covered:

- the Oracle Call Interface (OCI)
- special terms
- compiling and linking OCI programs
- new OCI routines
- obsolete OCI routines
- obsolescent OCI routines

The Oracle Call Interface

Structured Query Language (SQL) is a non–procedural language. A program in a non–procedural language specifies the set of data to be operated on, but does not specify precisely how the operations are to be carried out. The non–procedural nature of SQL makes it an easy language to learn and to use to perform simple database transactions.

However, third–generation programming languages such as C / C++, COBOL, and FORTRAN are procedural. The execution of most statements depends on preceding or following statements and on control structures, such as loops or conditional branches. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.

The OCI allows you to develop applications that take advantage of the non–procedural capabilities of SQL and the procedural capabilities of a third–generation language. You can also take advantage of PL / SQL, Oracle7's procedural extensions to SQL, in your OCI application. Thus, the applications you develop can be more powerful and flexible than applications written in SQL alone.

What is the OCI?

The OCI is a set of Application Programming Interfaces that allow you to manipulate data and schema in an Oracle database. Subroutine libraries supporting the OCI are offered for most popular high–level programming languages. As Figure 1 – 1 shows, you compile and link an OCI program in the same way that you compile and link a non–database application. There is no need for a separate preprocessing or precompilation step.

> **Note:** On some platforms, it may be necessary to include other libraries, in addition to the OCI library, to properly link your OCI programs. Check your Oracle system–specific documentation for further information about extra libraries which may be required.

The OCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through the Oracle7 Server.

Additionally, the OCI allows you to process PL / SQL statements.

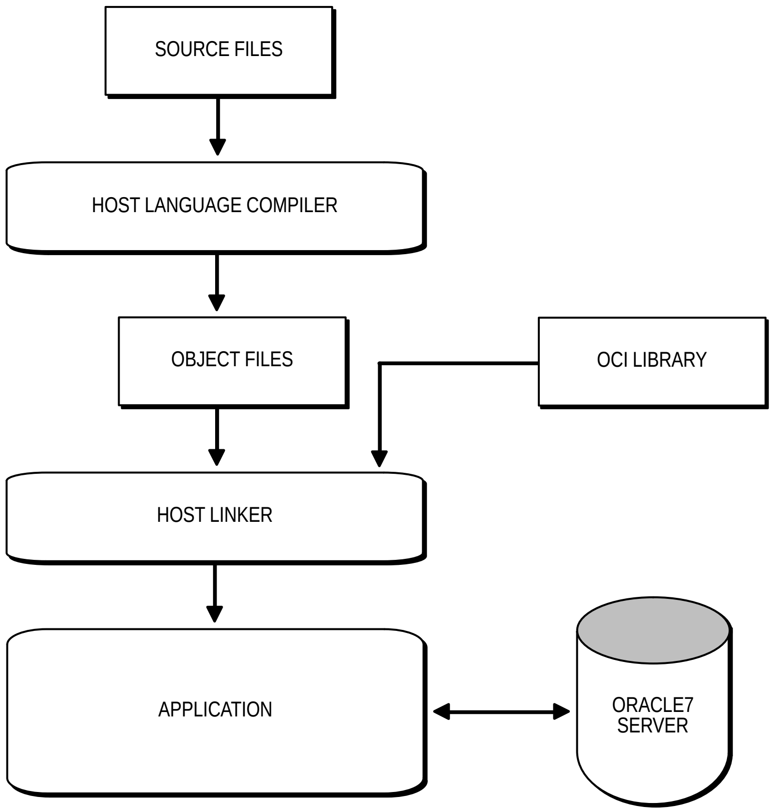

Figure 1 – 1 The OCI Development Process

Language Alternatives

There is a call–level interface for each of the following high–level languages:

- Ada
- C/C++
- COBOL
- FORTRAN
- PL/I

Meant for different application areas and reflecting different design philosophies, these languages offer a broad range of programming solutions.

> **Note:** The PL/I OCI is available only on a limited number of platforms, and is not documented in this guide. Also, because of basic differences in the Ada OCI implementation, it is documented separately in the *Programmer's Guide to the Oracle*

Call Interface for Ada. In the remainder of this guide, references to the OCI routines do not comprise the PL/I and Ada OCIs. Consult your Oracle system–specific documentation for more information.

Special Terms

A SQL statement such as

```
SELECT course_name, instructor
    FROM courses
    WHERE quarter = 'SPRING'
      AND dept = :deptno
```

contains the following parts:

- a SQL command: `SELECT`
- two *select–list items*: `course_name` and `instructor`
- a table name in the FROM clause: `courses`
- two column names in the WHERE clause: `quarter` and `dept`
- a literal input value in the WHERE clause: `'SPRING'`
- a *placeholder* for an input variable in the second part of the WHERE clause: `:deptno`

When you develop your OCI application, you call routines that specify to the Oracle7 Server the address (location) of input and output variables in your program. In this guide, specifying the address of an input variable for a placeholder is called a *bind* operation. Specifying the address of an output variable is called a *define* operation. For PL/SQL, both input and output specifications are referred to as *bind* operations.

These terms and operations are described in detail in Chapter 2.

Compiling and Linking

Oracle Corporation supplies runtime libraries for most popular third–party compilers. The details of linking an OCI program vary from system to system. See Appendix G and Chapter 2 of this Guide for general information about deferred mode linking. See your Oracle system–specific documentation for more information about compiling and linking an OCI application. for your specific platform.

New OCI Routines

Several new OCI routines are being introduced in release 7.3. Each of these routines is listed below, with a reference which points to more information about the functionality provided by that routine. Routine descriptions and parameter lists are found in Chapter 4 (for C), Chapter 5 (for COBOL) and Chapter 6 (for FORTRAN).

OPINIT
This is the new OCI process initialization call. It is described in the section "Thread Safety" in Chapter 2.

OBINDPS, **ODEFINPS**
These are new calls for binding and defining variables. They are discussed in the sections "Piecewise Insert, Update and Fetch" and "Arrays of Structures" in Chapter 2.

OGETPI, **OSETPI**
These calls are used for piecewise database operations. They are covered in detail in the section "Piecewise Insert, Update and Fetch" in Chapter 2.

Obsolete OCI Routines

Some OCI routines that were available in previous versions of the Oracle OCI are no longer supported for the Oracle7 OCI. They are listed below.

Obsolete OCI Routine	*Replacement in Oracle7*
OBIND	OBNDRN or OBNDRV
OBINDN	OBNDRN or OBNDRV
ODFINN	ODEFIN
ODSRBN	ODESCR
OLOGON	OLOG
OSQL	OPARSE

If you have an application written for an earlier version of Oracle that uses the obsolete routines, you must recode the application using the new routines shown in the table. Alternatively, you can write interface routines to map the old calls to the new calls, allowing you to relink the application without modifying the original source code or to handle cases where the original source code is not available.

Obsolescent OCI Routines

You should not use some OCI routines that are still available with the Oracle7 OCI in new programs. Replacing each of these routines is a new routine that offers improved performance or functionality.

These older routines are described in Appendix D, E, or F of this Guide. Oracle will not support these calls in future versions of the OCI. They are as follows:

Older OCI Routine	***Recommended for New Oracle7 Programs***
ODSC	ODESCR
OERMSG	OERHMS
OLON	OLOG
ORLON	OLOG
ONAME	ODESCR
OSQL3	OPARSE

CHAPTER

2

Writing an OCI Program

This chapter introduces you to the basic concepts involved in writing a program using the Oracle Call Interface (OCI). The following topics are covered:

- basic structure of an OCI program
- special data structures used in OCI programs
- SQL statement processing
- deferred statement execution
- steps in developing an OCI program
- general coding rules for OCI programs
- non–blocking mode
- thread safety
- piecewise insert, update, and fetch
- arrays of structures
- using PL/SQL in an OCI program
- developing X/Open DTP applications

Basic Program Structure

When you write an application using the Oracle Call Interface, there are certain steps that you follow to ensure that the program works properly. At a minimum, your OCI application must perform the following steps

Step 1. Allocate data structures that allow you to connect to an Oracle Server / database and process cursors.

Step 2. Connect to one or more Oracle databases.

Step 3. Open one or more cursors to process SQL or PL / SQL statements, as needed by the program.

Step 4. Process the SQL or PL / SQL statements required to perform the application's tasks.

Step 5. Close the cursors.

Step 6. Disconnect from the databases.

OCI Data Structures

In an OCI program you define data structures that enable the program to connect to Oracle and process SQL and PL / SQL statements. The data structures that you use to connect to an Oracle database are called the *logon data area* (LDA) and the *host data area* (HDA). You declare one LDA–HDA pair for each concurrent connection that your program requires, and pass references to them in the OLOG call that makes the connection. Your program can reuse an LDA and associated HDA after it severs the database connection controlled by that LDA–HDA. See the introductory material on pages 4 – 2, 5 – 2, or 6 – 2 for examples of declaring these data structures.

To process a SQL or PL / SQL statement, you must define a *cursor*. The cursor is defined using a *cursor data area* (CDA) together with the OOPEN routine. Each concurrently active cursor requires a separate CDA structure. When you close a cursor, you can reuse the CDA that was associated with the old cursor for a new cursor.

Both the LDA and the CDA have a field called the *return code field*. This field holds a binary 16–bit value. It contains zero after an OCI call that referenced the LDA or CDA returns without error. Otherwise, the return code field contains the Oracle error code. See the *Oracle7 Server*

Messages for a listing of the error codes and the associated error messages. In the program, you can also call the OERHMS routine to obtain the error message corresponding to an error return code.

Logon Data Area (LDA)

A logon data area (LDA) is a data area that you associate with an active connection to Oracle using the OLOG call. The format of the LDA for a typical 32–bit system is shown in Figure 2 – 1.

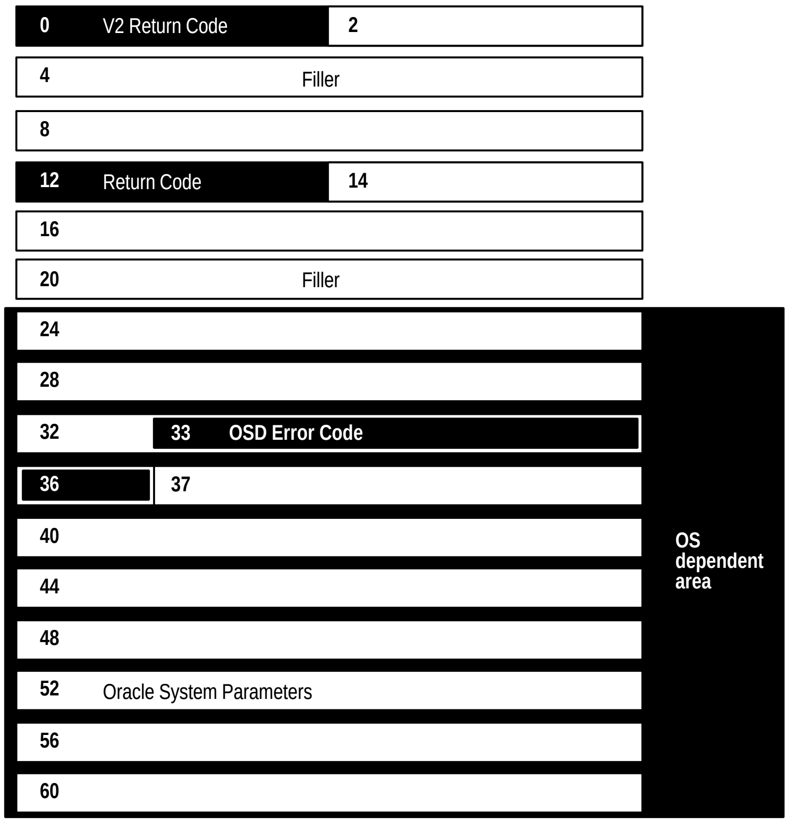

Figure 2 – 1 A Typical LDA (Logon Data Area)

The lengths and byte offsets of the fields in the LDA are system dependent. However, all fields are present for all systems. Check your Oracle system–specific documentation to see exactly how the LDA is configured. C programmers should use the definition of the LDA in the file *ocidfn.h*, which is listed in Appendix A and is available online. *ocidfn.h* is defined for each OCI platform, with the correct offsets.

The most commonly used field in the LDA is *return code*. Other named fields in the LDA are the same as the corresponding fields in the cursor data area. *In new OCI programs, do not check the V2 return code field for error information*. This field is present only for backward compatibility.

> **Note:** Once you have established a connection, do *not* move the HDA or LDA data areas. The Oracle Server uses the address of these areas in processing OCI calls, and these addresses must remain the same during the life of a connection.

Host Data Area (HDA)

A host data area (HDA) is usually a 256–byte data structure that you must allocate in your program when you connect to an Oracle Server. You allocate one LDA and one HDA for each simultaneous connection to Oracle which passes the addresses of these data areas to the server when you log on to the database with the OCI OLOG routine.

The HDA is 256 bytes long on 32–bit systems only. On 64–bit systems the HDA is typically 512 bytes long. If your system is of a different size, check your Oracle system–specific documentation for the correct size of the HDA. Even on 32–bit systems it is possible to allocate a 512–byte HDA if memory permits. This may increase the portability of applications

> **Warning:** The HDA must be properly declared and initialized before it is used in an OCI program. The HDA must be initialized to all zeros (binary zeros, not the "0" character) before the first call to OLOG, or runtime errors will occur. See the descriptions of the OLOG call in Chapter 4 (for C), Chapter 5 (for COBOL) and Chapter 6 (for FORTRAN) for language–specific methods to perform the initialization.

Many existing OCI programs, including the demos and sample code in this manual, have defined the HDA as a block of 256 one–byte integers (e.g., **ub1**[256] in C). On some platforms this may cause errors or unpredictable behavior as a result of the integers in the data block not being properly aligned. If your system automatically aligns four–byte integers, you can eliminate the problem by defining the HDA as a block of 64 four–byte integers (e.g., **ub4**[64] in C).

Cursor Data Area (CDA)

A cursor data area (CDA) provides a mapping between the user cursor in your program and the parsed representation of the SQL statement in the server. Information about a SQL or PL/SQL statement is preserved in the system global area (SGA) and the private SQL area. As Oracle processes the SQL statements in your program, it updates fields in the CDA to show the progress and status of the statement processing.

Figure 2 – 2 shows the structure of a CDA for a typical 32–bit system. The lengths of the fields in the CDA, and hence the offsets of the fields,

are system dependent. However, all fields are present for all systems. The CDA is always 64 bytes, but check your Oracle system–specific documentation for the exact configuration of the CDA.

> **Note:** C programmers should use the definition of the CDA listed in "Calling OCI Routines" on page 4 – 2 (and available online in the header file *ocidfn.h*). Also, see the example programs in Appendix A.

In particular, the size of the Oracle ROWID field in the CDA can be system dependent. Figure 2 – 2 shows a 13–byte ROWID, typical on systems that byte–align C structure fields. Check your Oracle system–specific documentation for the exact size and offsets of the members of the CDA data structure for your system. Also, see the description of ROWID in "External Datatypes" on page 3 – 8.

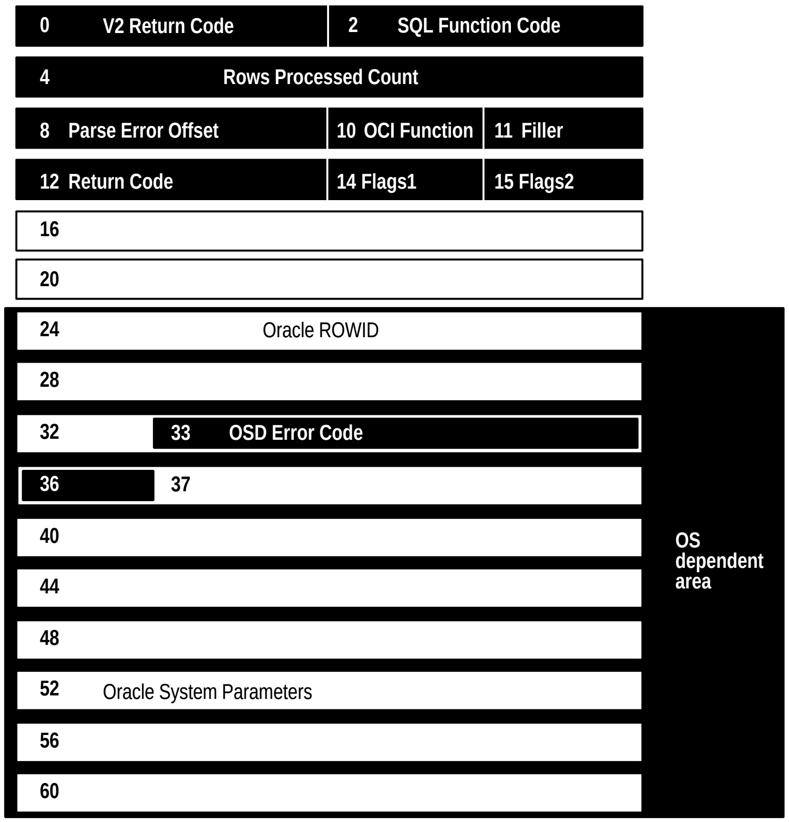

Figure 2 – 2 The CDA (Cursor Data Area)

Each of the fields in the CDA that an OCI program requires is described in the following sections.

V2 Return Code

In C, OCI calls return a two–byte binary integer. If the *V2 return code* is zero, no error has occurred, and if it is non–zero an error has occurred. At present this return code is the same as the *Oracle V2* (version 2) *Return Code*. The *V2 Return Code* field contains the same number.

Note: This field is for backward compatibility only. It might not be supported in future OCI versions. Use the *return code* field described below in new OCI applications.

Code	SQL FUNCTION	Code	SQL FUNCTION	Code	SQL FUNCTION
01	CREATE TABLE	26	ALTER TABLE	51	DROP TABLESPACE
02	SET ROLE	27	EXPLAIN	52	ALTER SESSION
03	INSERT	28	GRANT	53	ALTER USER
04	SELECT	29	REVOKE	54	COMMIT
05	UPDATE	30	CREATE SYNONYM	55	ROLLBACK
06	DROP ROLE	31	DROP SYNONYM	56	SAVEPOINT
07	DROP VIEW	32	ALTER SYSTEM SWITCH LOG	57	CREATE CONTROL FILE
08	DROP TABLE	33	SET TRANSACTION	58	ALTER TRACING
09	DELETE	34	PL/SQL EXECUTE	59	CREATE TRIGGER
10	CREATE VIEW	35	LOCK TABLE	60	ALTER TRIGGER
11	DROP USER	36	(not used)	61	DROP TRIGGER
12	CREATE ROLE	37	RENAME	62	ANALYZE TABLE
13	CREATE SEQUENCE	38	COMMENT	63	ANALYZE INDEX
14	ALTER SEQUENCE	39	AUDIT	64	ANALYZE CLUSTER
15	(not used)	40	NOAUDIT	65	CREATE PROFILE
16	DROP SEQUENCE	41	ALTER INDEX	66	DROP PROFILE
17	CREATE SCHEMA	42	CREATE EXTERNAL DATABASE	67	ALTER PROFILE
18	CREATE CLUSTER	43	DROP EXTERNAL DATABASE	68	DROP PROCEDURE
19	CREATE USER	44	CREATE DATABASE	69	(not used)
20	CREATE INDEX	45	ALTER DATABASE	70	ALTER RESOURCE COST
21	DROP INDEX	46	CREATE ROLLBACK SEGMENT	71	CREATE SNAPSHOT LOG
22	DROP CLUSTER	47	ALTER ROLLBACK SEGMENT	72	ALTER SNAPSHOT LOG
23	VALIDATE INDEX	48	DROP ROLLBACK SEGMENT	73	DROP SNAPSHOT LOG
24	CREATE PROCEDURE	49	CREATE TABLESPACE	74	CREATE SNAPSHOT
25	ALTER PROCEDURE	50	ALTER TABLESPACE	75	ALTER SNAPSHOT
				76	DROP SNAPSHOT

Table 2 – 1 SQL Function Codes

SQL Function Code

The *SQL function code* is a two–byte binary integer used internally by Oracle. There is a SQL function code for each SQL command. The SQL function codes are subject to change between OCI versions. They are shown in Table 2 – 1.

> **Note:** The *SQL function code* is not valid until the parse is performed. This happens on the call to OPARSE, unless the parse has been deferred, in which case it happens on the next describe or execute call. If the statement you parse is a PL/SQL block, the SQL function code is 34.

Rows Processed Count

This field contains a four–byte binary integer that counts the number of rows processed by a SQL statement. The count indicates the number of rows inserted, updated, or deleted by a data manipulation statement, or the cumulative number of rows fetched for the result set of a query.

The *rows processed count* field is valid *only* after an OEXEC, OEXN, OEXFET, OFEN, or OFETCH call. For queries, it is reset to zero when OEXEC or OEXN is called and is incremented after OFETCH or OFEN. For OEXFET, the count is reset to zero on the execute part of the call and is set when the fetch completes.

> **Note:** If a query returns a number of rows which is too large to fit into a four–byte integer, the contents of the *rows processed* field is undefined. Additionally, it is undesirable to issue queries which will return such a large number of rows, due to the extensive time required for Oracle to process such a query.

Parse Error Offset

This field contains a two–byte binary integer that indicates the starting byte position in the SQL statement where a parse error was detected. The first character of the SQL statement is at position zero. If the statement is longer than 64K bytes, the parse error offset is undefined.

A parse error can have many causes. Among them are a syntax error in the statement, a security violation, or a non–existent table or column. The *parse error offset* field is valid only after an OPARSE call. OCI calls other than OPARSE might leave a value in this field, but it is not meaningful.

When dealing with national language support (NLS) servers, be aware that the parse error offset may be incorrect due to a difference in character length between the client and server machines.

> **Note:** The *parse error offset* field is not valid until the parse is performed. This happens on the call to OPARSE, unless the parse has been deferred, in which case it happens on the next describe or execute call.

OCI Function Code

The *OCI function code* field contains a one–byte binary integer that indicates the most recently completed OCI routine. There is a function code for each OCI routine that uses the cursor data area. Routines that reference only the LDA (like OLOG) do not have a function code.

Table 2 – 2 lists the OCI function codes for routines that use the CDA. Codes for which no OCI routine is listed are unused.

#	OCI ROUTINE	#	OCI ROUTINE	#	OCI ROUTINE
04	OEXEC, OEXN	26	OSQL3	58	OFLNG
08	ODEFIN	28	OBNDRV	60	ODESCR
12	OFETCH, OFEN	30	OBNDRN	62	OBNDRA
14	OOPEN	34	OOPT	63	OBINDPS
16	OCLOSE	52	OCAN	64	ODEFINPS
22	ODSC	54	OPARSE	65	OGETPI
24	ONAME	56	OEXFET	66	OSETPI

Table 2 – 2 OCI Function Codes

Return Code

The *return code* is a two–byte positive binary integer that contains the Oracle error code for the most recently executed statement. Error codes and messages are listed in *Oracle7 Server Messages*. Use the OERHMS call to retrieve error message text associated with the return code.

Warning Flags

The *warning flags* field contains bit warning flags. More than one bit can be set. The table below lists the flags by bit position.

Bit Value	***Hex***	***Description***
1	1	There is a warning. This is set when any other bit in warning flags is set.
2	2	Set if any data item was truncated on a fetch.
4	4	This is set if a NULL was encountered during aggregate function evaluation.
8	8	This bit is not used.
16	10	Set if an UPDATE or DELETE statement does not contain a WHERE clause. This is set by OPARSE when the parse is performed (it may be deferred).
32	20	A PL/SQL package or procedure was compiled and entered in the database; however, there were compilation errors. This flag is set when a CREATE PROCEDURE, CREATE FUNCTION, CREATE TRIGGER, CREATE PACKAGE, or CREATE PACKAGE BODY statement caused the compilation error.
64	40	Set when a fatal error occurred and a transaction was completely rolled back. Not used in Version 6 and later versions of the Oracle Server.
128	80	This bit is not used.

Oracle ROWID

This field holds the ROWID in Oracle internal binary format (equivalent to external datatype 11) and is valid after INSERT, UPDATE, DELETE, and SELECT FOR UPDATE operations. It is not valid after a SELECT that does not contain the FOR UPDATE clause.

If a multi–row operation is performed, the *Oracle ROWID* field gets set to the ROWID of the last row that was operated on.

Note: The contents of the *Oracle ROWID* field are also undefined for OCI programs that are connected to non–Oracle data managers using an Oracle Open Gateway. These programs must use the ROWID pseudocolumn directly in SELECT statements and then use the returned ROWID in INSERT, UPDATE, and DELETE statements to identify specific rows in the non–Oracle data manager. See the section “Using ROWID” on page 2 – 31 for an example of this.

See the descriptions of ROWID in “External Datatypes” on page 3 – 14 for more information. The size of the *Oracle ROWID* field is system dependent. You can determine the size in several ways:

- C programmers can use the *sizeof()* operator on the *rid* substructure of the CDA, as described on page A – 8 and in the online header file *ocidfn.h*.
- Programmers using any language can determine the size of the Oracle ROWID field by describing the select–list item ROWID in a query such as “SELECT ROWID FROM dual”. Use the ODESCR routine to describe the ROWID select–list item, and if the DBTYPE parameter returns the datatype code 11, then the DBSIZE parameter contains the correct binary length of ROWID. (If the DBTYPE parameter returns the datatype code 1, then the OCI program is connected to a non–Oracle data manager. See “External Datatypes” on page 3 – 8 for more information.)
- See your Oracle system–specific documentation for the length of the *Oracle ROWID* field.

OSD Error Code

This field contains an operating system–dependent (OSD) error code associated with an Oracle error. For example, if Oracle7 receives an error when trying to perform disk I/O, the *OSD error code* field is set to an operating system–dependent I/O system failure code. These codes are not documented in this manual.

SQL Statement Processing

An OCI application processes SQL statements differently depending on the kind of statement. You must remember the kind of SQL statement being processed when writing your OCI code. If the application is processing dynamic statements (statements whose contents are not known at compile time), you can check the SQL function code after the statement is parsed to determine the steps needed to process the statement. The second sample program on page A – 27 (for C), B – 11 (for COBOL), and C – 12 (for FORTRAN) demonstrates this.

There are eight kinds of SQL statements in Oracle7:

- Data Definition Language
- Control Statements (3 types)
  - Transaction Control
  - Session Control
  - System Control
- Data Manipulation Language (DML)
- PL/SQL
- Queries
- Embedded SQL

Queries represent an additional kind of statement when using the OCI. Queries are often classified as DML statements, but OCI applications process queries differently, so they are considered separately here.

Data Definition Language Statements

Data Definition Language (DDL) statements manage entities in the database. DDL statements create new tables, drop old tables, and establish other schema objects. They also control access to schema objects. For example:

```
CREATE TABLE wine_list
    (name CHAR(20), type CHAR(20), year NUMBER(4),
     bin NUMBER(4))

DROP TABLE wine_list
GRANT UPDATE, INSERT, DELETE ON wine_list TO scott

REVOKE UPDATE ON wine_list FROM scott
```

Control Statements

OCI applications treat Transaction Control, Session Control, and System Control statements like DML statements. See *Oracle7 Server SQL Reference* for information about these statements.

Data Manipulation Language Statements

Data Manipulation Language (DML) statements can change data in the database tables. These statements are used to

- INSERT new rows into a table
- UPDATE column values in existing rows
- DELETE rows from a table
- LOCK a table in the database
- EXPLAIN the execution plan for a SQL statement

DML statements can require the program to input data to the database using input (bind) variables.

PL/SQL

PL/SQL is Oracle's procedural language extension to the SQL language. PL/SQL processes tasks that are more complicated than simple queries and SQL Data Manipulation Language statements. PL/SQL allows a number of constructs to be grouped into a single block and executed as a unit. Among these constructs are

- one or more SQL statements
- variable declarations
- assignment statements
- procedural control statements such as IF–THEN–ELSE statements and loops
- exception handling

You can use PL/SQL blocks in your OCI program to

- call Oracle stored procedures and stored functions
- combine procedural control statements with several SQL statements, to be executed as a single unit
- access special PL/SQL features such as records, tables, CURSOR FOR loops, and exception handling
- use cursor variables

See the *PL/SQL User's Guide and Reference* for information about coding PL/SQL blocks.

Queries

Queries are statements that retrieve data from a database. A query can return zero, one, or many rows of data. All queries begin with the SQL keyword SELECT. Queries access data in tables; they are often classified with DML statements. However, OCI applications process queries differently, so they are considered separately in this guide.

Embedded SQL Statements

An OCI application never uses embedded SQL statements. They are used for the Oracle Precompilers (Pro*C / C++, Pro*COBOL, Pro*FORTRAN, Pro*Ada, Pro*PL / I, Pro*Pascal) only.

The Steps in Processing a Statement

When you write OCI code to process a SQL statement, there are several distinct steps you must accomplish. Some SQL statements require only one step, but others can require four or five steps. Piecewise operations require extra steps, which are described in the section "Piecewise Insert, Update and Fetch" later in this chapter.

For information on deferring the parse, see the next section, "Deferred Statement Execution."

Once you have connected to Oracle and opened a cursor, the basic steps in processing a SQL statement are the following:

Step 1. Parse the statement using OPARSE. DDL statements, which do not accept input values or return results, can be executed directly by OPARSE if the program is linked in non–deferred mode or if it is linked in deferred mode and the OPARSE parameter DEFFLG is zero. No further processing is required. Transaction control statements must execute after the parse.

Step 2. For DML statements and queries, call OBNDRA, OBNDRV, OBNDRN or OBINDPS to bind the address of each input variable (or PL / SQL output variable) or array to each placeholder in the statement. OBINDPS is valid only in deferred mode and is necessary if piecewise operations or arrays of structures are used.

Step 3. For queries, describe the select–list items using ODESCR. This is an optional step; it is not required if the number of select–list items and the attributes of each item (such as its length and datatype) are known at compile time. If the parse was deferred, it is performed when the query is described. If the SQL statement being processed is not a query, ODESCR will generate an error.

Step 4. For queries, call ODEFIN or ODEFINPS to define an output variable for each select–list item in the SQL statement. Note that you do *not* use ODEFIN or ODEFINPS to define the output variables in an anonymous PL / SQL block, OBNDRV or OBNDRA is used instead. ODEFINPS is valid only in deferred mode and is necessary if piecewise operations or arrays of structures are used.

Step 5. For DML and transaction control statements, call OEXN to execute the statement. If the parse was deferred, it will be performed at this point.

Step 6. For queries, call OEXFET or the combination of OEXEC and OFETCH to execute the statement and then fetch the rows that satisfy the statement. If all the rows in the result set are not retrieved by an OEXFET call, it is necessary to call OFEN, perhaps more than once, to fetch the remaining rows. If the parse was deferred, and ODESCR was *not* called, the parse is performed at this point.

Following these steps, the application can close the statement cursor and log off of Oracle. Each of the steps above is described in detail in the section "Steps in Developing an OCI Program" on page 2 – 19.

Deferred Statement Execution

Before Oracle7 Server, Release 7.0, each call to an OCI routine required a corresponding call to the server. For example, when you parsed a SQL statement using OSQL3, the text of the SQL statement was transmitted to the server, and the statement was parsed and stored in the process global area (PGA). If placeholders were present in the statement, addresses of the input variables were bound to the placeholders by passing their addresses to the server. This required one call to the server for each input variable. A query required additional server calls to define addresses of program variables to hold select–list items. Finally, when the statement was executed, Oracle would request the values of any input variables, execute the statement, and, for a query, return the results. This required an additional server call.

When the OCI application and the Oracle Server are running on the same machine, multiple calls to Oracle have only a slight impact on performance. In a networked client/server environment, in which the OCI program runs on a client machine and the Oracle Server to which the program is connected runs on a different system (that might be thousands of kilometers away), separate calls to the database server may decrease performance.

To enhance performance, the OCI and Oracle7 now allow you to defer the execution of one or more steps in the processing of a SQL statement. For example, you can defer the processing of the step that parses the SQL statement and the steps that bind input variables and define output variables until the statement is actually executed.

If there is no describe (ODESCR) call, an entire query can be executed and the results fetched in one server call requiring only a single network round–trip.

Controlling Deferred Execution

Two factors control deferred execution of the parse, bind, and define steps:

- the way you link the OCI program
- how you set the DEFFLG parameter in the OPARSE call

Deferred Mode Linking

When you link your OCI program using *deferred mode linking* and your program is connected to an Oracle7 Server, the bind and define steps are always deferred until the statement executes. This behavior does not depend upon the particular OCI calls that your application uses, only on the link option that you select. For example, if you relink existing Version 6 OCI programs with the deferred mode link option, the bind and define calls are always deferred, (but not necessarily the parse).

Deferred mode linking selects new Oracle7 OCI libraries that buffer bind and define variable information on the client system using dynamically allocated memory, until that information is required by Oracle to process a SQL statement. This method does require additional memory on the client system.

Deferred mode linking is the default. See your Oracle system–specific documentation for information on setting non–deferred mode linking.

Deferring the Parse Step

When you link your program in deferred mode and you use the Oracle7 OPARSE call to parse a SQL statement with the DEFFLG flag set to a non–zero value, the parse step is similarly deferred. Thus, you can defer all OCI operations until the statement actually executes. Version 6 OCI programs must be recoded using OPARSE (replacing OSQL3) to obtain deferred parse behavior.

> **Note:** If you use ODESCR to describe properties of select–list items of a query, any pending bind and define call information for that cursor is sent to Oracle immediately. The statement is parsed if necessary, then the describe operation is performed.

Non–Deferred Linking

Non–deferred linking results in behavior similar to that of Version 6 OCI applications. Version 6 programs can be relinked using non–deferred linking and run without change against the Oracle7 Server.

Deferred mode linking requires additional memory on the client system to buffer call data. If memory resources on the client system are scarce, you might want to use non–deferred linking.

The new release 7.3 routines for binding and defining variables, OBINDPS and ODEFINPS are not valid when an application is linked in non–deferred mode. Deferred linking is necessary to utilize these calls and the functionality they provide for piecewise operations and arrays of structures.

Relinking Version 6 OCI Applications

You must be careful when relinking existing Version 6 OCI applications if you wish to take advantage of deferred OCI statement execution.

While existing applications can be relinked in deferred mode, changes in the time that errors are reported can affect program behavior. For example, some bind and define errors, which formerly were identified immediately when the bind or define call was executed, now will not be identified until the SQL statement is executed or described.

You should examine existing version 6 OCI applications carefully to determine if you can use deferred mode execution without changes to application logic.

Comparing Deferred and Non–deferred Linking

The figures on the following pages show how the parse, bind, define, and execute / fetch OCI calls communicate with Oracle using both deferred and non–deferred parsing.

The numbers in the figures indicate the order in which the particular calls are made. The hollow circles in Figure 2 – 4 indicate calls which are made before execution (as demonstrated in Figure 2 – 3) but which are deferred (not sent to the server) until after Step 4.

In this example, the OPARSE, OBNDRA and ODEFIN calls are deferred until an execute or describe call is performed.

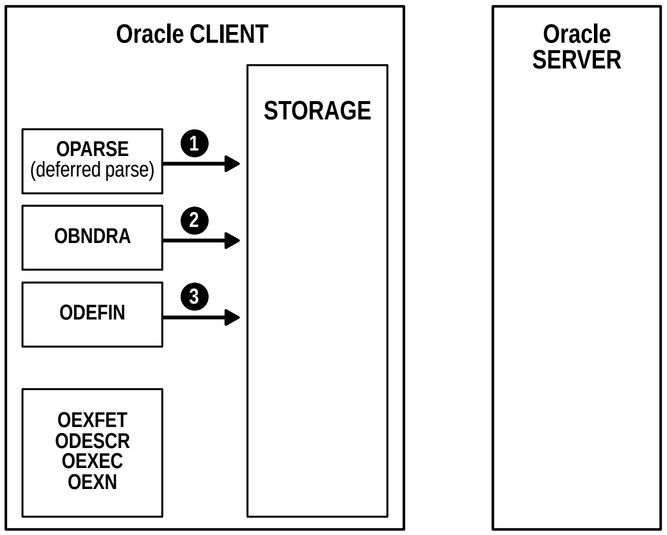

Figure 2 – 3 Statement Processing Deferred Parse (Before Execution)

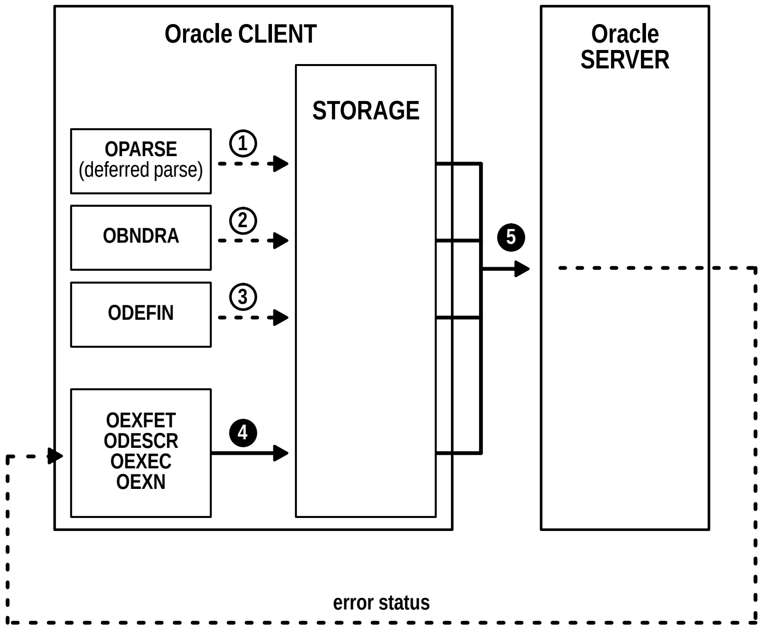

Figure 2 – 4 Statement Processing Deferred Parse (After Execution)

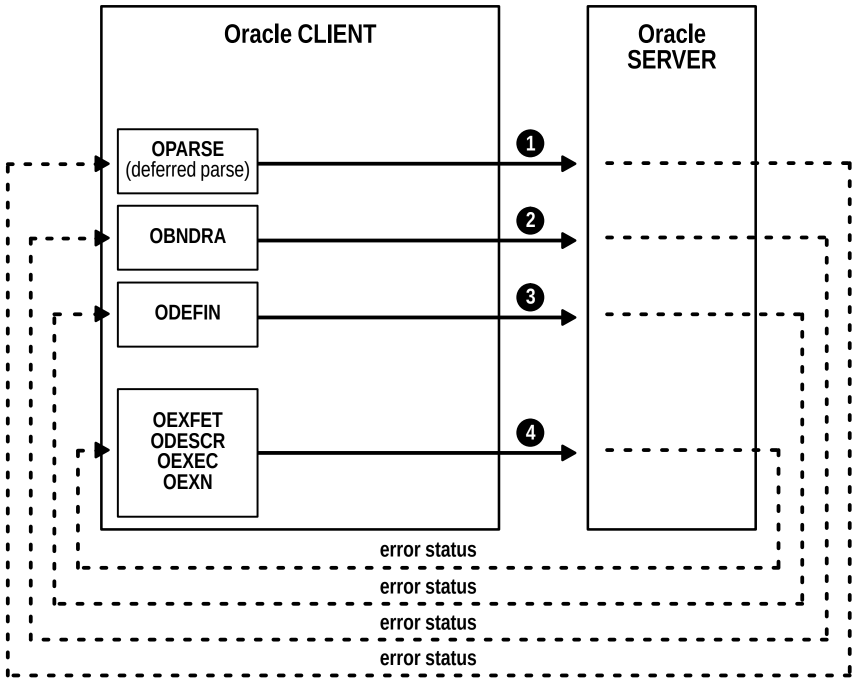

Figure 2 – 5 Statement Processing Non–Deferred Parse

As Figure 2 – 5 demonstrates, when the parse is not deferred, error codes are returned as soon as each call is made to the server. When the parse is deferred, as in Figure 2 – 3 and Figure 2 – 4, error codes for the parse, bind, and define steps are not returned until the describe or execute call is performed.

Developing an OCI Program

Each of the steps that you perform to develop an OCI application is described in greater detail in this section. Some of the steps are optional. For example, you do not need to describe or define select–list items if the statement is not a query.

The special case of piecewise inserts, updates and fetches is described in detail in the section "Piecewise Insert, Update and Fetch" later in this chapter.

Special considerations for operations involving arrays of structures are described in the section "Arrays of Structures" later in this chapter.

For special information and calls related to the use of multi–threaded programming in the C language, see the section "Thread Safety" later in this chapter.

Refer to the section "The Steps in Processing a Statement" on page 2 – 13 for an outline of the steps involved in processing a SQL statement within an OCI program.

The following sections explain how to perform these steps:

- define the OCI data structures
- connect to the Oracle server
- open the cursors
- parse the statement
- bind the addresses of input variables
- describe select–list items
- execute the statement
- define select–list items
- fetch the rows for the query
- close the cursors
- commit or rollback
- disconnect from Oracle

Define the OCI Data Structures

Before connecting to Oracle, your program must define at least one LDA. If the program requires multiple *simultaneous* connections, define one LDA for each simultaneous connection. The way you define these data areas depends on the language you are using. You must also define a HDA for each LDA. There must be one HDA / LDA pair per database connection. For examples showing the use of the LDA and HDA, see the description of the OLOG call on page 4 – 83 for C, 5 – 76 for COBOL, or 6 – 70 for FORTRAN, and the sample OCI programs in Appendix A, B, and C.

To process SQL statements, you define one or more CDAs, one for each SQL statement that is *simultaneously* active. If your program processes SQL statements serially, you might need only one CDA. The CDAs are defined in the same way as the LDAs.

Connect to the Oracle Server

Your program establishes communication with one or more Oracle databases by calling OLOG. To connect to Oracle, first define a LDA and a HDA in your program. Communication with Oracle is established at connect time; it takes place using the LDA and HDA that you define.

If your OCI program requires only a single database connection at a time, you can use OLOG and OLOGOF alternately to make and terminate that connection.

Any number of simultaneous connections can be made using OLOG.

Restrictions on Connections

When you link a program using the single–task driver, it can only connect to one database at a time, which is implicitly the default database. If another database is desired, it must be explicitly referenced in the OLOG call. Additional connections to the same database, if desired, must be made using OLOG.

There are two communications modes of connecting to an Oracle database, *blocking* and *non–blocking*. With the blocking mode, an OCI call returns only when it completes, either successfully or in error. With the non–blocking mode, control is immediately returned to the OCI program if the call could not complete (with ORA–03123 message). In this case, the OCI client can continue to process other statements while waiting to retry the OCI call to the server.

> **Note:** The non–blocking mode is based on a polling paradigm, which means that the client application must check whether the *pending* call has finished at the server by executing the call again with the same parameters.

See the section “Non–Blocking Mode” on page 2 – 32 for more information about connection modes.

Open the Cursors

To process a SQL statement, you must have an open cursor. You make the association between the data structures representing a valid cursor that are maintained by Oracle and a CDA in your program by calling the OOPEN routine. Perform this step after connecting to Oracle, because a valid LDA is a required parameter for the OOPEN call. This step must be performed before the CDA can be used to parse a SQL statement.

Each open cursor in an OCI application is associated with a particular server / database. If an OCI program has connections to more than one database, the total number of cursors which may be open concurrently within the program is the sum of the OPEN_CURSORS parameters of the two databases. For example, if an OCI client is connected to both *db1*, which has OPEN_CURSORS set to 50, and *db2*, which has OPEN_CURSORS set to 100, the OCI client can have up to 150 open cursors. It is important to keep in mind, however, that no more than 50 of those can be associated with *db1*, and no more than 100 can be associated with *db2*.

You can use cursors to execute the same SQL statement repeatedly or to execute a new SQL statement. When a cursor is reused, the contents of the corresponding CDA in your program are reset when the new SQL statement is parsed. It is never necessary to close and reopen a cursor before reusing it.

Parse the Statement

Every SQL statement must be parsed using the OPARSE routine.

Parsing the statement associates it with the CDA in your program. The exact semantics of the OPARSE routine are documented on page 4 – 98 for C, 5 – 87 for COBOL, or 6 – 80 for FORTRAN.

Data Definition Language statements are executed on the parse if you have linked in non–deferred mode or if you have linked with the deferred option and the DEFFLG parameter of OPARSE is zero. If you have linked in deferred mode and the DEFFLG parameter is non–zero, you must call OEXN or OEXEC to execute the statement.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when OPARSE is called in non–deferred mode are not detected in deferred mode until the first non–deferred call is made (usually an execute or describe call).

All DML statements, PL / SQL blocks, and queries require further processing after the parse step.

Bind the Addresses of Input Variables

Most DML statements, and some queries (such as those having a WHERE clause), require that program data be passed to Oracle as part of a SQL or PL/SQL statement.

Such data can be constant or literal data, known when your program is compiled. For example, the SQL statement

```
INSERT INTO wine_list (name, type, year, bin_no) VALUES
    ('Joseph Swan Vineyards', 'ZINFANDEL', NULL, 112)
```

contains several literals, such as 'ZINFANDEL' and 112.

This kind of statement is very limited, to say the least. You would not want to change and recompile the program each time a new bottle is added to the cellar. Instead, you write the program so that the input data is supplied at runtime.

Placeholders

When you define a SQL statement or PL/SQL block that contains input data to be supplied at runtime, placeholders in the SQL statement or PL/SQL block mark where data must be supplied. For example, the SQL statement

```
INSERT INTO wine_list (name, type, year, bin_no) VALUES
    (:Name, :Type, :Year, :Bin_Number)
```

contains four placeholders, indicated by the leading colons, that show where input data must be supplied by the program.

The following short PL/SQL block contains two placeholders:

```
char plsql_statement[] = "BEGIN\
                           RAISE_SALARY(:EMP_NUMBER, :NEW_SAL);\
                          END;" ;
```

You can use placeholders for input variables in any DELETE, INSERT, SELECT, or UPDATE statement, or PL/SQL block, in any position in the statement where you can use an expression or a literal value.

> **Note:** Placeholders *cannot* be used to name other Oracle objects such as tables or columns.

For each placeholder in the SQL statement or PL/SQL block, you must call an OCI routine that binds the address of a variable in your program to the placeholder. Thus when the statement executes, Oracle gets the data that your program placed in the input, or bind, variables.

Data does not have to be in a bind variable when you perform the bind step. At the bind step, you are only telling Oracle the address, datatype, and length of the variable. Make sure, however, that the variable contains valid data when you execute the SQL statement or PL/SQL block.

Note: If you change only the value of a bind variable, it is not necessary to rebind in order to execute the statement again. The bind is a *bind by reference*, so as long as the address of the bind variable remains valid, it is possible to re–execute a statement that references the variable without rebinding.

Routines that Bind Addresses

There are four OCI routines that you can use to bind addresses to placeholders: OBNDRV, OBNDRN, OBNDRA and OBINDPS.

OBNDRV

When you use OBNDRV, you must specify the name of the placeholder. Thus, for the statement above, you specify ":Year" as the name of the placeholder for the year value.

OBNDRV can be used in interactive applications, where the user will enter a SQL statement at runtime. In this case, however, your program must scan the SQL statement to obtain the placeholder names. For an example, see the second sample program on page A – 27 for C, B – 11 for COBOL, and C – 12 for FORTRAN.

The name of a placeholder for the OBNDRV routine cannot be a reserved word. For example, the following SQL statement is *not* legal, because ROWID is a reserved word:

```
SELECT ename FROM emp WHERE rowid = :ROWID
```

A list of Oracle reserved words, keywords and namespaces can be found in Appendix H.

OBNDRN

To use OBNDRN, each placeholder must be in the form :N, where N is a literal integer between 1 and 255. Consider the following example:

```
SELECT ename, sal FROM emp
    WHERE (job = :1 AND sal > :2)
    OR
    (job != :1 AND sal < :2)
```

is a valid SQL statement for the OBNDRN routine. Note that in this statement there are four instances of a placeholder, but really only two placeholders. Thus only two bind variables are needed. You only need to call OBNDRN twice for this statement. All occurrences of a placeholder in a SQL statement are bound on a single call.

The OBNDRN routine allows you to use an index variable to iterate through a set of placeholders.

Note: You cannot use OBNDRN to bind variables in a PL/SQL block. You must use OBNDRA or OBNDRV.

OBNDRA

The OBNDRA routine binds addresses of scalars or arrays in your program to placeholders in a SQL statement or a PL/SQL block. OBNDRA is similar to OBNDRV, but it provides additional parameters that indicate the maximum size of an array, the number and lengths of array elements returned, and return errors on a column–by–column basis. OBNDRA is also used to bind C arrays or COBOL tables to PL/SQL tables.

OBINDPS

The OBINDPS routine subsumes much of the functionality of the OBNDRA and OBNDRN routines for binding placeholders in SQL statements or PL/SQL blocks. Additionally, the OPCODE parameter can signal that an application will be providing inserted or updated data incrementally at runtime. OBINDPS is also used when the application will be inserting data stored in an array of structures.

OBINDPS is supported only when applications are linked in deferred mode and run against Oracle Server release 7.3 or later. If applications are linked in non–deferred mode or run against a release 7.2 or earlier server, another bind routine must be used. In that case, the ability to handle piecewise operations and arrays of structures is not supported.

For information about using OBINDPS with piecewise operations or arrays of structures, see the sections "Piecewise Insert, Update and Fetch" and "Arrays of Structures" later in this chapter.

Describe Select–List Items

If the SQL statement is a query, you might need to obtain more information about the select–list items. This is particularly true for dynamic queries, that is, queries whose contents are not known until runtime. In this case, the program might not have prior information about the datatypes, column lengths, or display sizes of the select–list items.

For example, a user might enter a query such as

```
SELECT * FROM wine_list
```

where the program has no prior information about the columns in the table WINE_LIST.

You can obtain this information using the ODESCR (describe) routine. ODESCR returns information about the *n*th select–list item, where *n* is an IN parameter. You can use this information to determine how to convert, display, or store the data that will be returned when the rows are fetched for the query.

To process dynamic select lists, call ODESCR in a loop. Set an index variable to one at the start of the loop, then increment it, doing the describe at each iteration, until a "variable not in select list" error (ORA–01007) is returned in the return code field of the CDA. The following C language code fragment demonstrates this process. For a more complete example, see the description of the *odescr()* routine on page 4 – 48, or the *cdemo2* example program in Appendix A (for C), Appendix B (for COBOL) or Appendix C (for FORTRAN).

```
for (pos = 1; pos <= NPOS; pos++) {
  cbufl[pos] = sizeof cbuf[pos];
  if (odescr(&cda, pos, &dbsize[pos], &dbtype[pos],
             &cbuf[pos], &cbufl[pos], &dsize[pos],
             &prec[pos], &scale[pos], &nullok[pos])) {
    if (cda.rc == 1007)
      break;
    oci_error();
    continue;
  }
```

Note: If you have deferred the parse, then the statement will be parsed when ODESCR is called.

Execute the Statement

If the SQL statement is a DML statement, you must execute the statement. The execute operation inputs the values in all bind variables to Oracle.

There are different ways to input data to Oracle. You can execute a SQL statement repeatedly using the OEXEC routine and supply different input values on each iteration. Alternatively, you can use the Oracle array interface and input many values with a single statement by using the OEXN routine. (You can also use OEXN to execute a statement that processes only a single row of data.)

Note: If you change only the value of a bind variable, it is not necessary to rebind in order to execute the statement again. The bind is a *bind by reference*, so as long as the address of the bind variable remains valid, it is possible to re–execute a statement that references the variable without rebinding.

The array interface significantly reduces communications traffic with Oracle when you need to update or insert a large volume of data. This can lead to considerable performance gains, especially in a client/server environment. For example, consider an application that needs to insert 10 rows into the database. Calling OEXEC ten times with different values results in ten network round–trips to insert all the data. The same result is possible with a single call to OEXN which involves only one network round–trip.

Define Select–List Items

For a query, you use the ODEFIN or ODEFINPS routine to associate the address of an output variable (or array) in your program with each select–list item in the query. If you do not know in advance the number of select–list items, as in the case

```
SELECT * FROM wine_list
```

you might first call ODESCR repeatedly to determine the number. You can also call ODESCR and then ODEFIN or ODEFINPS in the same loop, and exit the loop when ODESCR returns the “variable not in select–list” error. See the *cdemo2* example program in Appendix A (for C), Appendix B (for COBOL) or Appendix C (for FORTRAN), for an example showing the use of ODESCR and ODEFIN in a loop.

> **Note:** You do not use ODEFIN or ODEFINPS to define select–list items in a SQL SELECT statement in a PL / SQL block. You must use OBNDRA or OBINDPS (or OBNDRV) in this case.

You can call ODEFIN or ODEFINPS again to redefine the output variables without having to reparse or re–execute the SQL statement.

ODEFINPS provides additional functionality for piecewise fetches and fetches into arrays of structures. For information about using ODEFINPS in these situations, see the sections “Piecewise Insert, Update and Fetch” and “Arrays of Structures” later in this chapter.

Fetch the Rows for the Query

After you have defined the addresses of output variables, you can fetch the rows that satisfy a query by calling one of the following routines:

- OFETCH, to fetch a single row. Before calling OFETCH, you must call OEXEC to execute the statement. You can call OFETCH in a loop to fetch multiple rows, but this is less efficient than using OEXFET or OFEN to select into arrays.
- OFEN, to fetch a single row or to fetch multiple rows into arrays on a single call. You must execute the SQL statement first by calling OEXEC.
- OEXFET, which combines the functionality of OEXEC and OFEN / OFETCH into a single statement. OEXFET can execute the statement, fetch data from one or multiple rows into scalar or array variables in your program, and, optionally, cancel the cursor (releasing resources) when the fetch completes. You can combine a deferred parse (using OPARSE) and deferred binds and defines with OEXFET to perform a complete query operation, from statement parse to canceling the cursor, in one call to the Oracle server.

If you plan to use OEXFET or OFEN to fetch multiple rows, you must make sure that the output variables you define for the select–list items are arrays. There are also optional OUT parameters that must be arrays if you are using the array interface. For example, output variables for indicator variables and column return code values must also be arrays. See the descriptions of OEXFET, page 4 – 65 for C, 5 – 59 for COBOL, or 6 – 56 for FORTRAN, and OFEN, page 4 – 70 for C, 5 – 64 for COBOL, or 6 – 61 for FORTRAN, for more information.

Close the Cursors

Before your program exits, close each open cursor using the OCLOSE routine. Once a cursor is closed, the CDA is no longer associated with the Oracle Server, and any memory areas in the server used by that cursor are freed.

> **Note:** After using a cursor to execute a SQL statement or PL/SQL block, you can reuse that cursor for a new SQL statement or PL/SQL block without closing and reopening it.

If it is necessary to issue an OOPEN call on a cursor that has already been opened by your application and used to execute a SQL statement or PL/SQL block, be sure to call OCLOSE to close that cursor before making the call to OOPEN.

> **Note:** The SQL92 standard requires that a cursor be closed on a commit. Repeatedly opening the same cursor when committing small transactions is inefficient and is a performance issue. Oracle7 permits a fetch after a commit without closing a cursor as a performance enhancement. This applies only to interoperating Oracle7, release 7.0 or higher servers. This does not apply to Oracle gateways accessing non–Oracle data sources. When an Oracle server interoperates with an Oracle gateway server, a cursor must be explicitly closed and opened again before another fetch can occur.

Commit or Rollback

Disconnecting from Oracle using OLOGOF causes an implicit commit. You can force a commit by using the OCOM routine. If you want to roll back the transactions, use the OROL routine.

> **Note:** If an application disconnects from Oracle in some way other than a call to OLOGOF (for example, losing a network connection), and OCOM has not been called, the transaction is rolled back automatically.

Disconnect from Oracle

Call OLOGOF before the program exits to close connections to Oracle. Call OLOGOF for each LDA that was referenced in an OLOG call.

Coding Rules

This section explains some of the general rules that you should follow when coding an OCI application.

Parameter Datatypes

OCI calls use the following types of parameters:

- variable addresses or references
- binary integers
- short binary integers
- character strings
- one–byte integers

Address parameters pass the address of the variable to Oracle. You should be careful when developing in C, which normally passes scalar parameters by value, to make sure that the parameter is an address. For example, if a parameter for a routine is specified as the address of a short integer, and *rcode* is the variable in your program for that parameter, be sure to pass it as &*rcode* in C.

Binary integer parameters are numbers whose size is system dependent. Short binary integer parameters are smaller numbers whose size is also system dependent. See your Oracle system–specific documentation for the size of these integers on your system.

For language–specific information about parameter datatypes and parameter passing conventions, refer to the introductory section in the chapter that covers the language you are using: page 4 – 2 for C, 5 – 2 for COBOL, or 6 – 2 for FORTRAN.

Character Strings

Character strings are a special type of address parameter. The following discussion describes additional rules that apply to character string address parameters.

Each OCI routine that allows a character string to be passed as a parameter also has a string length parameter. The length parameter should be set to the exact length of the string. If the string is terminated by a null character (as is often the case in C), you can specify –1 for the length parameter (do *not* use zero).

Literal character strings can be passed if permitted by the compiler. Note, however, that since character strings are address parameters, your compiler must actually pass the address of the literal.

Indicator Variables

The bind and define OCI calls (OBNDRA, OBNDRV, OBNDRN, OBINDPS, ODEFINPS and ODEFIN) each have a parameter that allows you to associate an indicator variable, or an array of indicator variables if you are using arrays, with a DML statement, PL/SQL statement, or query.

Because host languages do not have the concept of a null, you associate indicator variables with input variables to specify whether the associated placeholder is a NULL value.

For output variables, indicator variables are used to determine whether the value returned from Oracle is in fact a NULL or a truncated value.

For input host variables, the values the OCI program can assign to an indicator variable have the following meanings:

–1	Oracle assigns a null to the column, ignoring the value of the input variable.
>=0	Oracle assigns the value of the input variable to the column.

On output, the values Oracle can assign to an indicator variable have the following meanings:

–2	The length of the item is greater than the length of the output variable; the item has been truncated. Additionally, the original length is longer than the maximum data length that can be returned, which is the maximum value for an unsigned short integer, minus one (usually 2^16 – 1).
–1	The selected value is null, and the value of the output variable is unchanged.
0	Oracle assigned an intact value to the host variable.
>0	The length of the item is greater than the length of the output variable; the item has been truncated. The positive value returned in the indicator variable is the actual length before truncation.

Nulls

You can insert a null into a database column in several ways. One method is to use a literal NULL in the text of an INSERT or UPDATE statement. For example, the SQL statement

```
INSERT INTO emp (ename, empno, deptno)
    VALUES (NULL, 8010, 20)
```

makes the ENAME column null.

Another method is to use indicator variables in the OCI bind call. See "Indicator Variables" on page 2 – 29 for more information.

To detect when nulls are fetched from the database, you can specify indicator parameters in the bind or define routine and then check the values returned after OEXFET, OFEN, or OFETCH. Nulls can also be detected using the column–level RCODE parameter. See the description of OFEN in Chapter 4, 5, and 6 for an example of this.

> **Note:** Following SQL92 requirements, Oracle7 returns an error if an attempt is made to fetch a null select–list item into a variable that does not have an associated indicator variable specified in the define call. Use LNGFLG for Version 6 behavior with no error.

Canceling Calls

On most platforms, the user can interactively cancel a long–running or repeated OCI call, such as OEXN, OEXEC, OEXFET, OFETCH, or OFEN. You do this by entering the operating system's interrupt character (usually CTRL–C) from the keyboard. Programs that are linked single–task and two–task support this interrupt capability.

When you cancel the long–running or repeated call using the operating system interrupt, the error code ORA–01013 ("user requested cancel of current operation") is returned in the return code field of the CDA.

The OCI program can use an OCAN call to cancel a query once the desired number of rows have been fetched.

If the OCI program needs to cancel a long–running call with a mechanism such as a timer, you might be able to use the OBREAK routine. See the description of *obreak()* on page 4 – 27 for an example. Note that you cannot use OBREAK with all operating systems, nor with all supported languages, nor with all transport protocols.

The same effect may be achieved more efficiently through the use of non–blocking calls. See the section "Non–Blocking Mode" on page 2 – 32 for more information.

Maximum Array Size

The maximum size of an array is 32512 items. That is, the ITERS or NROWS parameters of OEXN, OFEN, or OEXFET cannot be set to a value greater than this limit, regardless of the datatypes of these parameters.

Positioned Updates and Deletes

You can use the binary ROWID that is returned in the CDA after a SELECT ... FOR UPDATE OF ... statement in a later UPDATE or DELETE statement. For example, for a SQL statement such as

```
SELECT ename FROM emp WHERE empno = 7499 FOR UPDATE OF sal
```

when the FETCH is performed, the ROWID field in the CDA contains the row identifier of the SELECTed row. You can copy this ROWID into a buffer in your program, then use the saved ROWID in a DELETE or UPDATE statement. For example, if MY_ROWID is the buffer in which the row identifier has been saved, you can later process a SQL statement such as

```
UPDATE emp SET sal = :1 WHERE rowid = :2
```

by binding the new salary to the :1 placeholder and MY_ROWID to the :2 placeholder. Be sure to use datatype code 11 (ROWID) when binding MY_ROWID to :2.

Optimizing Compilers

Many compilers optimize the generated code so that program variable addresses do not accurately reflect the actual location of a variable in storage at all times. For example, optimizers frequently place commonly used variables in machine registers, and only store them in memory locations when they are referenced in a subroutine call.

When the address of a variable used in a subsequent call is passed to Oracle as a parameter, you must be certain that the addressed variable is actually at the specified location when it is used in the subsequent execute or fetch call. This applies to the ODEFIN, ODEFINPS, OBNDRN, OBNDRV, OBNDRA and OBINDPS calls.

> **Caution:** This rule applies to all local variables whose addresses are passed as parameters to these routines. For example, if the variable *value* is declared as a local variable and its address is passed to OBNDRV, program errors may occur as a result of the address of *value* not being in the specified location when it is used in a subsequent call.

The simplest way to ensure currency of variable addresses is to disable the compiler's optimizer. Many compilers provide mechanisms to disable optimizations selectively. For example, there might be options to disable certain optimizations for local sections or routines. For most ANSI C compilers, declaring variables as **volatile** disables optimization for them. Refer to your compiler's manual for more information.

If you cannot switch your compiler's optimization on and off within a single file, an alternative is to put all OCI code in a separate file that you compile with optimization turned off. Then you can link it in with the rest of your program.

Non–Blocking Mode

Before Oracle7 Server, Release 7.2, connections between Oracle and an OCI program were only in blocking mode. Release 7.2 included a new *non–blocking* mode.

> **Note:** To use the non–blocking feature with Oracle7 Server release 7.2, you need version 7.2 of the OCI libraries and version 2.2 of SQL*Net. To use the non–blocking feature with Oracle7 Server release 7.3, you need version 7.2 *or* 7.3 of the OCI libraries and version 2.3 of SQL*Net. Release 7.3 of the OCI libraries is not compatible with release 7.2 of the Server. Non–blocking calls are not supported against release 7.1 or earlier of the Server.

The non–blocking mode returns control to an OCI program so that it may perform other computations while the OCI call is being processed by the server. This mode is particularly useful in Graphical User Interface (GUI) applications, realtime applications, and in distributed environments.

This new mode is not interrupt–driven. Rather, it is based on a polling paradigm, which means that the client application has to check whether the *pending* call is finished at the server.

The following three OCI routines are used specifically with non–blocking connections:

- ONBTST, to test whether a database connection is in blocking or non–blocking mode
- ONBSET, to place a database connection in non–blocking mode for all subsequent OCI calls on that connection
- ONBCLR, to place a database connection in blocking mode

These new calls are described for C in Chapter 4 (on pages 4 – 88, 4 – 89, and 4 – 94), for COBOL in Chapter 5 (on pages 5 – 81, 5 – 82, and 5 – 83), and for FORTRAN in Chapter 6 (on pages 6 – 74, 6 – 75, and 6 – 76).

The OLOG logon call allows the programmer to specify whether a database connection is to be made in blocking or non–blocking mode. The *mode* parameter can take one of two values OCI_LM_DEF (default, for blocking mode) or OCI_LM_NBL (for non–blocking mode). These values are defined in *ocidfn.h*.

Making a Non–Blocking Connection

You can establish a non–blocking connection between Oracle and your OCI program by using either:

- OLOG with the mode parameter set to non–blocking, or
- ONBSET to change an existing communications channel from blocking to non–blocking mode.

For an example program in C illustrating the use of the non–blocking mode, see page 4 – 89.

Thread Safety

The introduction of thread safety in release 7.3 of the Oracle7 Server and OCI libraries allows developers to use the OCI in a multi–threaded environment. OCI code can now be reentrant, with multiple threads of a user program making OCI calls without side effects from one thread to another. Previous releases had a non–reentrant architecture in which only one thread of execution could make OCI calls.

> **Note:** The availability of thread safety for the OCI is subject to the following limitations:
>
> – Thread safety is primarily applicable to programs written in C. Programs in COBOL and FORTRAN may be able to utilize this functionality by incorporating a thread–safe C library and calling C functions. Check the documentation for your specific COBOL or FORTRAN package to see if this is possible. This manual only discusses the use of thread safety with C.
>
> – Thread safety is not available on every platform. Check your Oracle system–specific documentation for more information.

The following sections describe how you can use the OCI to develop multi–threaded applications.

Advantages of Thread Safety in the OCI

The implementation of thread safety in the Oracle Call Interface provides the following benefits and advantages:

- Multiple threads of execution can make OCI calls with the same result as successive calls made by a single thread.
- When multiple threads make OCI calls there are no side effects between threads.
- Users who do not write multi–threaded programs do not pay a performance penalty for using thread–safe OCI calls.
- Cursors linked to statements in a session can be shared between threads. This means a single database connection can be used by several different threads of execution, although care must be taken to manage the use of the connection by different threads.
- Use of multiple threads can improve program performance. Gains may be seen on multiprocessor systems where threads run concurrently on separate processors, and on single processor systems where overlap can occur between slower operations and faster operations. Although this could be accomplished with multiple processes and non–blocking calls, threads are “cheaper” in terms of system resources and are easier to program.

Thread Safety and Three–tier Architectures

In addition to client–server applications, where the client can be a multi–threaded program, a typical use of multi–threaded applications is in three–tier (also called client–agent–server) architectures. In this architecture the client is only concerned with presentation services. The agent (or application server) processes the application logic for the client application. Typically, this relationship is a many–to–one relationship, with multiple clients sharing the same application server.

The server tier in this scenario is an Oracle database. The applications server (agent) is very well suited to being a multi–threaded application server, with each thread serving a client application. In an Oracle environment this application server is an OCI or Precompiler program.

This type of architecture is similar to Oracle’s XA interface used in X/Open DTP applications. See the section “Developing X/Open DTP Applications” on page 2 – 53 for more information.

Basic Concepts of Multi–threaded Development

Threads are lightweight processes which exist within a larger process. Threads share the same code and data segments, but have their own program counters, machine registers and stack. Global and static variables are common to all threads, and a mutual exclusivity mechanism may be required to manage access to these variables from multiple threads within an application.

The sharing of resources by program threads is analogous to a shared system resource area of a database accessed by multiple processes. Access to shared resources by different processes must be synchronized using mutual exclusivity locking and latching mechanisms to prevent conflicts. In contrast, if a process owns a private structure in the resource area, it does not need a lock to access that structure safely.

Once spawned, threads run asynchronously to one another. They can access common data elements and make OCI calls in any order. In Oracle Servers through release 7.3, the host can process only one call at a time for a given database connection. Therefore, if multiple threads within an application are sharing the same database connection and can make independent calls, their access to the connection must be serialized. Only one thread at a time may access the connection.

> **Note:** Application developers are responsible for managing access to the database connection by multiple threads. Care must be taken to insure that different threads access the connection serially, rather than concurrently.

Managing Access to the Database Connection

The mechanism to manage access to the connection may take the form of a semaphore which keeps track of in–progress calls to the host on a particular connection. Any thread wishing to make an OCI call must wait to get the semaphore, make the call, and then release the semaphore. Any other thread trying to get the semaphore while a call is in progress will be blocked from execution by the operating system. This semaphore is one way to implement the mutual exclusivity locking necessary to insure that there are no conflicts between multiple threads which are accessing shared resources within an application.

Single vs. Multiple Connections

The situation of multiple threads accessing a single connection from within a program can be contrasted with an application which has multiple threads but also multiple connections. Figure 2 – 6 on the following page shows an application running in a multi–threaded environment through a single connection.

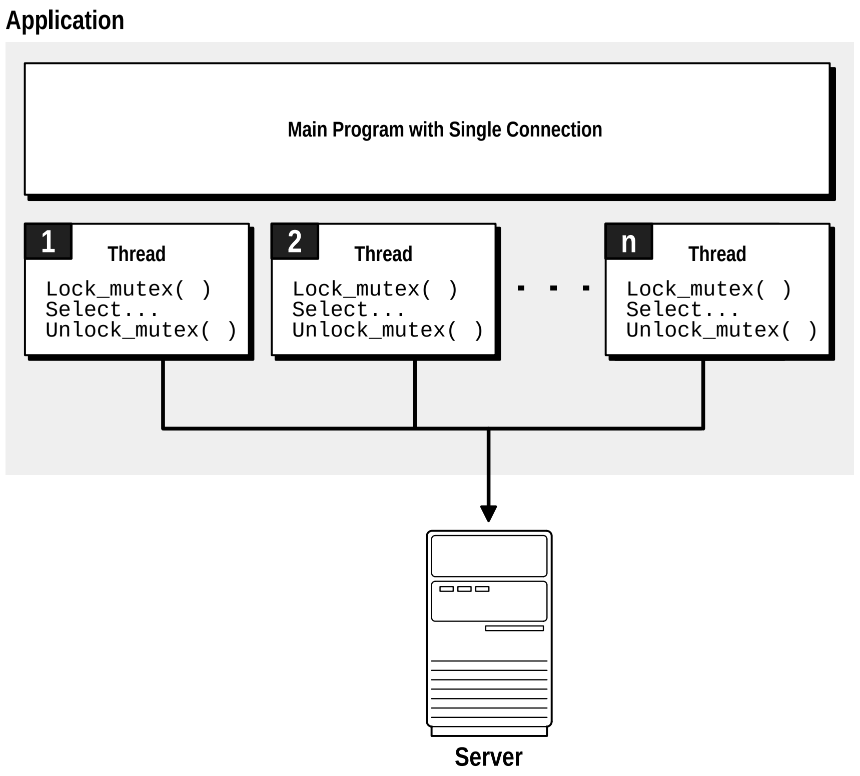

Figure 2 – 6 Connection Sharing Among Threads

In this environment, the various threads of execution must take turns accessing a single database connection to process a SQL statement, and this access is managed by the main program. This figure also demonstrates the use of mutual exclusivity (mutex) locking, as described earlier.

In contrast to Figure 2 – 6, Figure 2 – 7 shows an application running multiple threads of execution across multiple database connections.

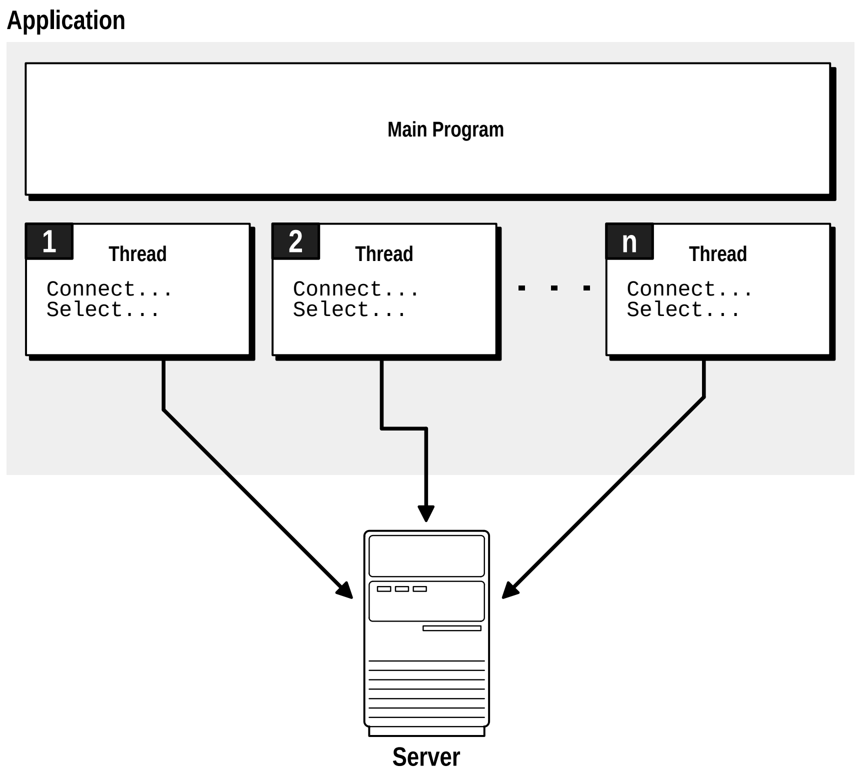

Figure 2 – 7 No Connection Sharing Among Threads

In this situation it is not necessary for the application to limit thread access to the connections, since each thread has a dedicated connection.

Although it is not shown in these two figures, it is also possible to develop applications in which a particular thread accesses multiple database connections.

Programming Multi–threaded OCI Applications

Before issuing any other OCI calls in an application, you must tell the OCI layer whether your environment is single–threaded or multi–threaded. This is accomplished with the OCI process initialization call, *opinit()* which takes one parameter, *mode*. The *mode* parameter, which is defined in *ocidfn.h*, takes one of two values:

OCI_EV_DEF	OCI Environment Default, for single–threaded environments.
OCI_EV_TSF	OCI Environment Thread–Safe, for thread–safe environments.

To maintain backward compatibility, if the *opinit()* call is skipped a single–threaded environment is assumed.

Even in a single–threaded application it is advisable to make the call to *opinit()* with *mode* set to OCI_EV_DEF, rather than skipping it. In addition to setting the proper environment, the call to *opinit()* also provides explicit documentation that the application is not thread–safe.

Warning: Skipping the call to *opinit()* in a multi–threaded environment will result in undefined behavior of OCI calls.

To benefit from the thread–safe OCI libraries, OCI programs must connect to an Oracle database using the *olog()* call, rather than the older *olon()* or *orlon()* calls. Use of the older calls implies that the application is running in a single–threaded environment, and subsequent OCI calls will not be thread–safe. Users who make a call to *olog()* may still run single–threaded programs if they so choose. A single–threaded environment can be specified with the *opinit()* call, described above.

The *olog()* function uses local data structures (LDA and HDA) which contain host and connection information. Only one logon can be active at any time on a single connection with a given set of these data structures. Therefore, it is the user's responsibility to make sure that only one logon exists at any time for a given LDA and HDA. Multiple threads should not issue logon calls with the same host and connection data structures. After a logoff, however, the same structures can be reused by another thread for another logon.

For more discussion of HDAs and LDAs, see the sections "Host Data Area" and "Logon Data Area" earlier in this chapter.

Similarly, the *ologof()* call must be made only once for a connection. Only one thread should issue an *ologof()* call for a given set of host and connection data structures.

See the description of the *opinit()* call in Chapter 4 for sample code showing the use of thread–safe SQL statement processing.

Piecewise Insert, Update and Fetch

Prior to Oracle Server release 7.3, OCI applications had to allocate memory for an entire column before it could be inserted or updated. This could cause serious memory problems in the case of LONG columns, which have a maximum size of 2 gigabytes.

Prior to release 7.3, piecewise fetches were possible using OFLNG, which is still available. The OFLNG call may still be useful when it is necessary to perform a piecewise fetch from a certain offset within a column.

With release 7.3, users have the option of using new OCI calls to perform piecewise inserts and updates, and more flexible piecewise fetches. A very large column may now be inserted or retrieved as a series of chunks of smaller size, minimizing client–side memory requirements.

Piecewise fetches are now more efficient. Unlike OFLNG or non–piecewise fetches, the new piecewise fetch operations are buffered locally and individual fetches access the local buffer, rather than accessing the main database across a network. This can improve application performance.

Piecewise operations are now more flexible. The size of individual pieces is determined at runtime by the application. Each piece may be of the same size as other pieces, or it may be of a different size.

This new piecewise functionality may be particularly useful when performing operations on extremely large blocks of string or binary data. An example of this would be operations involving database columns which store LONG or LONG RAW data. See the description of the *ftype* parameter on page 4 – 11 for information about which datatypes are valid for piecewise operations.

In addition to SQL statements, piecewise operations are also valid for PL/SQL blocks. PL/SQL is subject to the same limitations on datatypes which are mentioned as part of the *ftype* parameter description on page 4 – 11.

Figure 2 – 8 shows a single long column being inserted piecewise into a database table through a series of insert operations (i_1, i_2, i_3...i_n). In this example the inserted pieces are of varying sizes.

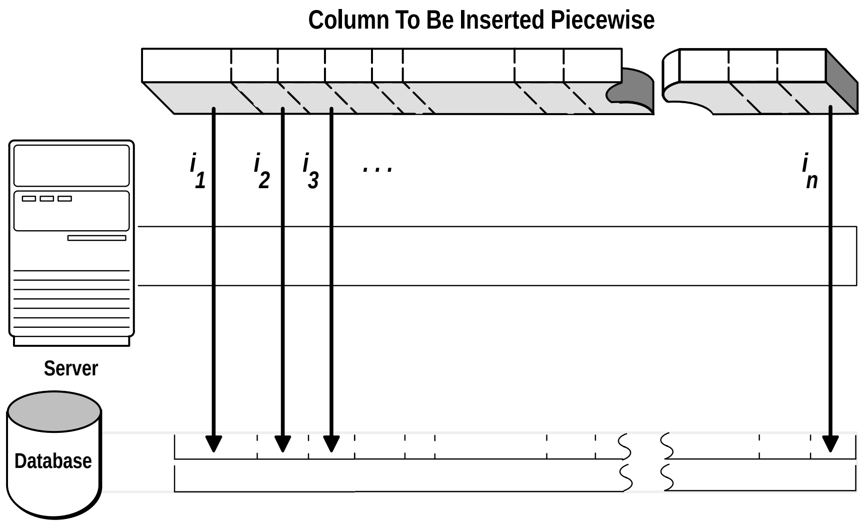

Figure 2 – 8 Piecewise Insert of a LONG Column

Four new calls have been added to the OCI to handle piecewise operations: OBINDPS (OCI Bind Piecewise), ODEFINPS (OCI Define Piecewise), OGETPI (OCI Get Piece Information) and OSETPI (OCI Set Piece Information). These calls are used in conjunction with new Oracle error codes to coordinate a piecewise operation.

The next two sections explain the steps that are involved in performing a piecewise insert and fetch. These are followed by additional comments about piecewise operations. For information about logging on, opening cursors, parsing statements and logging off, see the section "Developing an OCI Program" on page 2 – 19.

Performing a Piecewise Insert

Once a database connection is established and a cursor is opened, a piecewise insert begins with calls to parse a SQL or PL/SQL statement and to bind input values. The bind call for columns to be inserted piecewise must use the new OBINDPS routine. Bind calls for other placeholders may use any of the supported bind routines (OBNDRV, OBNDRN, OBNDRA or OBINDPS).

Following the parse and bind, the application performs a series of calls to OEXEC, OGETPI and OSETPI. Each time OEXEC is called it returns a value which is used in determining what action should be performed next. In general, the application retrieves a value indicating that the next piece needs to be inserted, populates a buffer with that piece and then executes an insert. When the last piece has been inserted, the operation is complete.

It is important to keep in mind that the insert buffer can be of arbitrary size and is allocated at runtime. In addition, each inserted piece does not need to be of the same size. The size of each piece to be inserted is established by each OSETPI call.

Note that if the same piece size is used for all inserts and the size of the data being inserted is not evenly divisible by the piece size, the final inserted piece will be smaller than the pieces which preceded it. For example, it a data value 18,536 bytes long is inserted in chunks of 50 bytes each, the last remaining piece will be only 36 bytes. The programmer must account for this by indicating the smaller size in the final OSETPI call.

The following steps outline the procedure involved in performing a piecewise insert.

Step 1. Log on to the database (OLOG), open a cursor (OOPEN) and parse a SQL statement (OPARSE).

Step 2. Bind a placeholder using OBINDPS. At this point you specify the maximum column length to be inserted, but you need not specify the actual size of the pieces you will use. There is an optional context pointer parameter which may be used by your application. The pointer is returned to the application in the OGETPI call.

Step 3. Call OEXEC for the first time. At this point no data is actually inserted, and error code ORA–03129 ('the next piece to be inserted is required') is returned to the application. If any other value is returned, it indicates that an error occurred.

Step 4. Call OGETPI to retrieve information about the piece which needs to be inserted. The parameters of OGETPI include a pointer which returns a value indicating whether the required piece is the first piece (OCI_FIRST_PIECE) or a subsequent piece (OCI_NEXT_PIECE). The possible parameter values are defined in *ocidfn.h*.

Step 5. The application populates a buffer with the piece of data to be inserted and calls OSETPI. The parameters passed to OSETPI include a pointer to the piece, a pointer to the length of the piece and a value indicating whether this is the first piece (OCI_FIRST_PIECE), an intermediate piece (OCI_NEXT_PIECE) or the last piece (OCI_LAST_PIECE).

Step 6. Call OEXEC again. If OCI_LAST_PIECE was indicated in Step 6 and OEXEC returns zero, all pieces were inserted successfully. If OEXEC returns ORA–03129, go back to Step 4 for the next insert. If OEXEC returns any other value, it indicates that an error occurred.

The piecewise operation is complete when the final piece has been successfully inserted. This is indicated by the zero return value from the final OEXEC call.

Piecewise updates are performed in a similar manner. For a piecewise update operation the insert buffer is populated with the data which is being updated and OEXEC is called to execute the update.

Performing a Piecewise Fetch

Once a database connection is established and a cursor is opened, a piecewise fetch begins with calls to parse a SQL or PL/SQL statement and define output variables. The define step must use the new ODEFINPS call for output variables which will be used in piecewise operations.

Following the parse, define and execute, the application performs a series of calls to OFETCH, OGETPI and OSETPI. Each time OFETCH is called it returns a value which is used in determining what action should be performed next. In general, the application retrieves a value indicating that the next piece needs to be fetched and then fetches the piece into a buffer.

It is important to keep in mind that the fetch buffer can be of arbitrary size and is allocated at runtime. In addition, each fetched piece does not need to be of the same size. The only requirement is that size of the final fetch must be exactly the size of the last remaining piece. The size of the piece to be fetched is established by each OSETPI call.

The following steps outline the method for fetching a row piecewise.

Step 1. Log on to the database (OLOG), open a cursor (OOPEN), and parse (OPARSE) and execute (OEXEC) a SQL statement.

Step 2. Call ODEFINPS. The OPCODE parameter for this call specifies that the operation is going to be performed piecewise. There is an optional context pointer parameter which may be used by your application. The pointer is returned to the application in the OGETPI call.

Step 3. Call OFETCH for the first time. At this point no data is actually retrieved, and error code ORA–03130 ('the buffer for the next piece to be fetched is required') is returned to the application. If any other value is returned, it indicates that an error occurred.

Step 4. Call OGETPI to obtain information about the piece to be fetched. The *piecep* parameter indicates whether it is the first piece (OCI_FIRST_PIECE) or a subsequent piece (OCI_NEXT_PIECE).

Step 5. Call OSETPI to specify the buffer into which you wish to fetch the piece.

Step 6. Call OFETCH again to retrieve the actual piece. If OFETCH returns zero, all the pieces have been fetched successfully. If OFETCH returns ORA–03130 then return to Step 4 to process the next piece. If any other value is returned, it indicates that an error occurred.

The piecewise fetch is complete when the final OFETCH call returns a value of zero.

Using Piecewise Operations in OCI Programs

In both the piecewise fetch and insert, it is important to understand the sequence of calls which are necessary for the operation to complete successfully. In particular, the programmer must keep in mind that for a piecewise insert it is necessary to call OEXEC one time more than the number of pieces to be inserted. This is because the first time OEXEC is called it merely returns a value indicating that the first piece to be inserted is required. As a result, if *n* pieces are being inserted, OEXEC ends up being called *n+1* times.

Similarly, when a piecewise fetch is being performed, OFETCH ends up being called once more than the number of pieces to be fetched.

Users who are working in an NLS (National Language Support) environment which uses multibyte characters must take special care when performing piecewise operations. The new OCI calls assume that multibyte character strings will be provided in pieces by the application such that each piece is a complete multibyte string by itself.

Users who are binding to PL/SQL tables can retrieve a pointer to the current index of the table during the OGETPI calls.

See the descriptions of the *osetpi()* and *ogetpi()* calls in Chapter 4 for C language code examples showing how these calls can be used in an OCI application. See the descriptions of the OBINDPS and ODEFINPS calls in Chapter 4 (for C), Chapter 5 (for COBOL) and Chapter 6 (for FORTRAN) for parameter descriptions and further information about those calls.

Arrays of Structures

Prior to release 7.3 of the Oracle7 Server, applications performing multi–row, multi–column operations were required to allocate a set of parallel arrays for the operation, one for each column being inserted, updated or fetched. For example, when fetching multiple rows of data from three columns, NAME, AGE and SALARY, the data would be fetched into a NAME array, an AGE array and a SALARY array, each of which would contain the data for several rows.

This method complicates the task of the application programmer, because related data which should be part of a single array of structures or records ends up being split across several parallel arrays of scalars.

With Oracle7 Server release 7.3, the OCI application developer can place related scalars in a single structure. Database operations are performed using an array of these structures. This new functionality provides increased flexibility for developers. In the above example, a single structure could contain separate fields to hold the NAME, AGE and SALARY data from one row in the database table. Data would then be fetched into an array of these structures.

In order to perform a multi–row, multi–column operation using an array of structures, the developer associates each column involved in the operation with a field in a structure. This association, which is part of the new OBINDPS and ODEFINPS calls, specifies where fetched data will be stored, or where inserted or updated data will be found.

Figure 2 – 9 is a graphical representation of this process. In the figure, the various fields in a database row are fetched into a single structure in an array of structures. Each column being fetched corresponds to one of the fields in the structure.

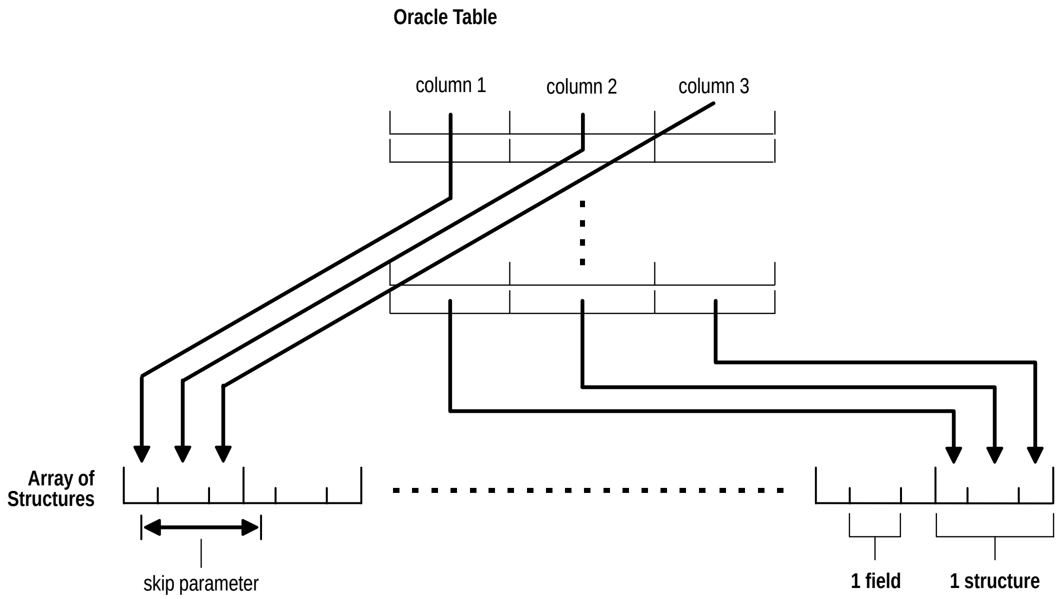

Figure 2 – 9 Database Fetch Into an Array of Structures

Skip Parameters

When column data is split across an array of structures it is no longer contiguous. The single array of structures stores data as though it were composed of several interleaved arrays of scalars. Because of this fact, developers must specify a "skip parameter" for each field being bound or defined. This skip parameter specifies the number of bytes that need to be skipped in the array of structures before the same field is encountered again. In general this will be equivalent to the byte size of one structure.

Figure 2 – 10 demonstrates how a skip parameter is determined. In this case the skip parameter is the sum of the sizes of the fields *f1*, *f2* and *f3*, which is 8 bytes. This equals the size of one structure.

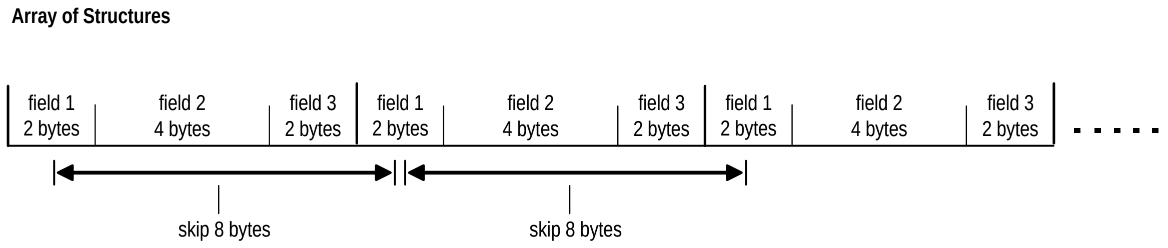

Figure 2 – 10 Determining Skip Parameters

On some systems it may be necessary to set the skip parameter to be *sizeof*(one array element) rather than *sizeof*(struct). This is because in some cases some compilers may insert padding into a structure. For example, consider an array of C structures consisting of two fields, a **ub4** and a **ub1**.

```
struct demo {
    ub4 field1;
    ub1 field2;
};
struct demo demo_array[MAXSIZE];
```

Some compilers insert three bytes of padding after the **ub1** so that the **ub4** which begins the next structure in the array is properly aligned. In this case, the following statement may return an incorrect value:

```
skip_parameter = sizeof(struct demo);
```

On some systems this will produce a proper skip parameter of eight. On other systems, the skip_parameter will be set to five bytes by this statement. In this case, use the following to get the correct value for the skip parameter:

```
skip_parameter = sizeof(demo_array[0]);
```

The ability to work with arrays of structures is an extension of the existing functionality for binding and defining arrays of program variables. It is still possible for programmers to work with standard arrays (as opposed to arrays of structures) in release 7.3 applications. When specifying a standard array operation the related skip will be equal to the size of the datatype of the array under consideration. For example, for an array declared as

```
text emp_names[4][20]
```

the skip parameter for the bind or define operation will be 20. Each data element in the array is then recognized as a separate unit, rather than being part of a structure.

OCI Calls Used With Arrays of Structures

Two new OCI calls must be used when performing operations involving arrays of structures: OBINDPS (for binding fields in arrays of structures for input variables) and ODEFINPS (for defining arrays of structures for output variables).

> **Note:** These calls are supported only when deferred mode linking is used. Using OBINDPS or ODEFINPS in non–deferred mode will result in an error being generated. If it is necessary to link in non–deferred mode, other bind and define calls must be used, and operations involving arrays of structures are not supported.

The implementation of arrays of structures also supports the use of indicator variables and return codes. OCI application developers can declare parallel arrays of column–level indicator variables and return codes, corresponding to the arrays of information being fetched, inserted or updated. These arrays can have their own skip parameters, which are specified during a call to OBINDPS or ODEFINPS.

There are many ways in which arrays of structures of program values and indicator variables could be set up. As one possible example, consider an application which fetches data from three database columns into an array of structures containing three fields. There can be a corresponding array of indicator variable structures of three fields, each of which is a column–level indicator variable for one of the columns being fetched from the database.

See the section "Indicator Variables" on page 2 – 29 for more information about indicator variables.

See the description of the *obindps()* and *odefinps()* calls in Chapter 4 for C language code examples showing how to use these calls in an OCI program.

Using PL/SQL in an OCI Program

PL/SQL is Oracle's procedural language extension to the SQL language. PL/SQL processes tasks that are more complicated than simple queries and SQL Data Manipulation Language statements. PL/SQL allows a number of constructs to be grouped into a single block and executed as a unit. Among these constructs are the following:

- one or more SQL statements
- variable declarations
- assignment statements
- procedural control statements such as IF–THEN–ELSE statements and loops
- exception handling

You can use PL/SQL blocks in your OCI program to perform the following operations:

- call Oracle stored procedures and stored functions
- combine procedural control statements with several SQL statements, to be executed as a single unit
- access special PL/SQL features such as records, tables, CURSOR FOR loops, and exception handling
- use cursor variables

See the *PL/SQL User's Guide and Reference* for information about coding PL/SQL blocks.

Binding Placeholders in a PL/SQL Block

You process a PL/SQL block by placing the block in a string variable, then parsing the string, binding any variables, and executing the statement containing the block, just as you would with a single SQL statement.

When you bind placeholders in a PL/SQL block to program variables, you must use either OBNDRA, OBINDPS or OBNDRV to perform the binds. OBNDRN cannot be used. You can use OBNDRA and OBINDPS to bind host variables that are either scalars or arrays. You can only bind scalar variables using OBNDRV.

For example, in a PL/SQL block such as

```
BEGIN
    SELECT ename,sal,comm INTO :emp_name, :salary, :commission
        WHERE ename = :emp_number;
END;
```

you would use OBNDRV or OBINDPS to bind variables in place of the :EMP_NAME, :SALARY, and :COMMISSION placeholders, and in place of the input placeholder :EMP_NUMBER.

> **Note:** You cannot use ODEFIN or ODEFINPS to bind host variables in a PL/SQL block. You must use OBNDRV, OBINDPS or OBNDRA.

If a PL/SQL block raises an unhandled exception, the values of bind variables are *not* returned.

A Program Example

Perhaps the most common use for PL/SQL blocks in an OCI program is to call stored procedures or stored functions. For example, assume that there is a procedure called RAISE_SALARY stored in the database, and you want to call this procedure from an OCI program. You do this by embedding a call to that procedure in an anonymous PL/SQL block, then processing the PL/SQL block in the OCI program.

The following program fragment, written in C, shows how to embed a stored procedure call in an OCI application. The program is heavily commented for the benefit of those who are not familiar with the C language. Comments occur between the beginning (/*) and ending (*/) comment delimiters. For brevity, this example does not check for errors.

The program fragment asks the user for an employee ID number and the employee's new salary. Then, by calling the OEXEC routine, the PL/SQL block that calls the stored procedure RAISE_SALARY is performed.

```
/* Define a string and initialize it with the text of
   the PL/SQL block. The '\' character continues lines
   inside a string literal. */

char plsql_statement[] = "BEGIN\
                           RAISE_SALARY(:EMP_NUMBER, :NEW_SAL);\
                          END;" ;

/* Declare an integer and a real variable. */

int    empnum;
float  salary;
char   empnum_stg[10];
char   salary_stg[10];

/* After connecting to Oracle and opening a cursor,
   parse the statement. Because the text of the block is
   null terminated, -1 is passed for the length parameter. */

    if (oparse(&cda, plsql_statement, -1, 1, 2))
        oci_error(&cda);

/* Bind the host variables. In C, an ampersand (&) before
  a variable means take the address of the variable. This is
  required here because C normally passes scalar parameters by
  value. 3 and 4 are the datatype codes for INTEGER and FLOAT. */

   if (obndrv(&cda, ":EMP_NUMBER", -1, &empnum, sizeof (int), 3,
          -1, 0, 0, -1, -1))
         oci_error (&cda);
   if (obndrv(&cda, ":NEW_SAL", -1, &salary, sizeof (float), 4,
          -1, 0, 0, -1, -1))
         oci_error(&cda);

/*  Query the user for an employee number, */
    printf("Enter the employee number: ");
    gets(empnum_stg);
    empnum = atoi(empnum_stg);

/*  and the new salary. */
    printf(”Enter the new salary: ”);
    gets(salary_stg);
    salary = atoi(salary_stg);

/*  Execute the PL/SQL block, which executes the
    called stored procedure. */
    if (oexec(&cda))
        oci_error(&cda);

/*  Commit the transaction. */
    if (ocom(&lda))
        oci_error(&cda);
```

Cursor Variables

Starting with Oracle7, Release 7.2, you can bind PL/SQL cursor variables to cursors (CDAs) in your OCI applications. A cursor variable is a reference to a cursor that is defined and opened on a Oracle 7.2 Server.

Some of the advantages of cursor variables are

- Encapsulation: queries are centralized in the stored procedure that opens the cursor variable.
- Improved maintenance: only the stored procedure needs to be changed if a table changes.
- Improved security: client applications with execute–only privileges may access rows through stored procedures.

See the *PL/SQL User's Guide and Reference* for complete information about cursor variables.

Using a Cursor Variable

The basic steps to using a cursor variable are:

Step 1. Declare at least two CDAs.

Step 2. Open one of the cursors with OOPEN (*cursor* in the example below).

Step 3. Using OPARSE, parse a PL/SQL block that contains a cursor variable. (The PL/SQL cursor variable can either be in a PL/SQL anonymous block or in a PL/SQL stored function or procedure.)

Step 4. In general, you must bind variables in a SQL statement or a PL/SQL block to variables in your OCI application. For cursor variables, you must bind (using OBNDRA or OBNDRV) the PL/SQL cursor variable to a second cursor in your application. You must indicate that the cursor's type is SQLT_CUR.

Step 5. Execute (using OEXEC or OEXN) the PL/SQL block. This associates the select statement with the cursor variable. From this point on, you can treat the newly bound cursor (*cursor_emp* in the example below) just like any other opened OCI cursor.

Step 6. Use ODESCR or ODEFIN to associate the select–list items of the previously bound and executed cursor (*cursor_emp*) to variables in your OCI program.

Step 7. Now every fetch (OFEN or OFETCH) will store its results in the OCI variables that you defined in the preceding step.

A Program Example

The following code fragment illustrates how to use cursor variables in an OCI application written in C. For the complete program see the sample program *cdemo5.c* on page A – 51.

```
/* excerpt from cdemo5.c */
/* Define a string and initialize it with the text of
   the PL/SQL block. The ’\’ character is used to continue lines
   inside a string literal. */
static text plsql_block[] =
   ”BEGIN \
      OPEN :cursor1 FOR select empno from emp; \
    END;” ;
static Cda_Def cursor, cursor_emp; /* the two cursors */
static Lda_Def lda;                        /* the LDA */
ub4 empno;
int rv;
...
/* After connecting to Oracle and opening a cursor,
   parse the statement. Since the text of the block is
   null terminated, –1 is passed for the length parameter. */
if (oparse(&cursor, plsql_block, (sb4) –1, (sword) TRUE, (ub4) 2))
        oci_error(&cursor);

/* Bind the host variables. */
if (obndra(&cursor, (text *) ”:cursor1”, –1, (ub1 *) &cursor_emp,
       –1, SQLT_CUR, –1, (sb2 *) 0, (ub2 *) 0, (ub2 *) 0,
       (ub4) 0, (ub4 *) 0, (text *) 0, 0, 0))
        oci_error(&cursor_emp);
```

```
/* Execute the PL/SQL block. */
if (oexec(&cursor))
        oci_error(&cursor);

/* Close the cursor. */
if (oclose(&cursor))
        oci_error(&cursor);
...

/* Define the output variable for empno. */
if (odefin(&cursor_emp, 1, (ub1 *) empno, (sword) sizeof(ub4),
       SQLT_INT, –1, (sb2 *) –1, (text *) 0, (sword) 0,
       (sword) 0, (ub2 *) 0, (ub2 *) 0))
       oci_error(&cursor_emp);

while (!(ofetch(&cursor)))             /* until an error occurs */
{
      printf(”%d\n”, empno);
}
if cursor_emp.rc <> 1403   /* ORA–01403 means fetch is complete */
      oci_error(&cursor);
```

Note: If a cursor variable has OPEN FOR applied to it in the PL/SQL block, then the equivalent OCI calls are OBNDRA (or OBNDRV), OPARSE, and OEXN (or OEXEC). If any of these operations are attempted after PL/SQL returns, an error occurs. You may not rebind or re–execute a cursor variable before it has been reparsed.

Obtaining PL/SQL Error Numbers and Messages

When an error occurs in a PL/SQL block, an Oracle error number in the 6500 PL/SQL errors is returned in the return code field of the CDA. However, if the error was generated by a user exception, the error number is in the –20000 to –20999 range. You obtain the PL/SQL–specific error codes and error messages by calling the OERHMS routine and passing it the error code. More than one PL/SQL error message can be returned on a single call to OERHMS, so you should allocate a large buffer for the message. A buffer size of 2000 bytes is usually sufficient.

For more information about PL/SQL error codes and messages, see the *PL/SQL User's Guide and Reference.*

Restrictions on Arrays

When you use OBNDRA to bind arrays in your OCI program to PL/SQL tables, there are limitations on the datatypes that can be bound. The full set of conversions that are described in Table 3 – 5 in Chapter 3 do not apply when binding arrays to PL/SQL tables.

Table 2 – 3 shows checkmarks for the conversions which can be performed between PL / SQL tables and host arrays.

	PL/SQL Table							
Host Array	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	✓							
CHARZ	✓							
DATE		✓						
DECIMAL					✓			
DISPLAY					✓			
FLOAT					✓			
INTEGER					✓			
LONG	✓		✓					
LONG VARCHAR			✓	✓		✓		✓
LONG VARRAW				✓		✓		
NUMBER					✓			
RAW				✓		✓		
ROWID							✓	
STRING			✓	✓		✓		✓
UNSIGNED					✓			
VARCHAR			✓	✓		✓		✓
VARCHAR2			✓	✓		✓		✓
VARNUM					✓			
VARRAW				✓		✓		

Table 2 – 3 Supported PL/SQL Array Element Conversions

Developing X/Open DTP Applications

An X / Open application is an application that operates in a distributed transaction processing (DTP) environment. In an abstract model, applications call on *resource managers* to provide many kinds of services. A database resource manager offers access to data in a database for an application. Transaction processing monitors that use the standard X / Open interface to resource managers incorporate a *transaction manager* for transaction coordination.

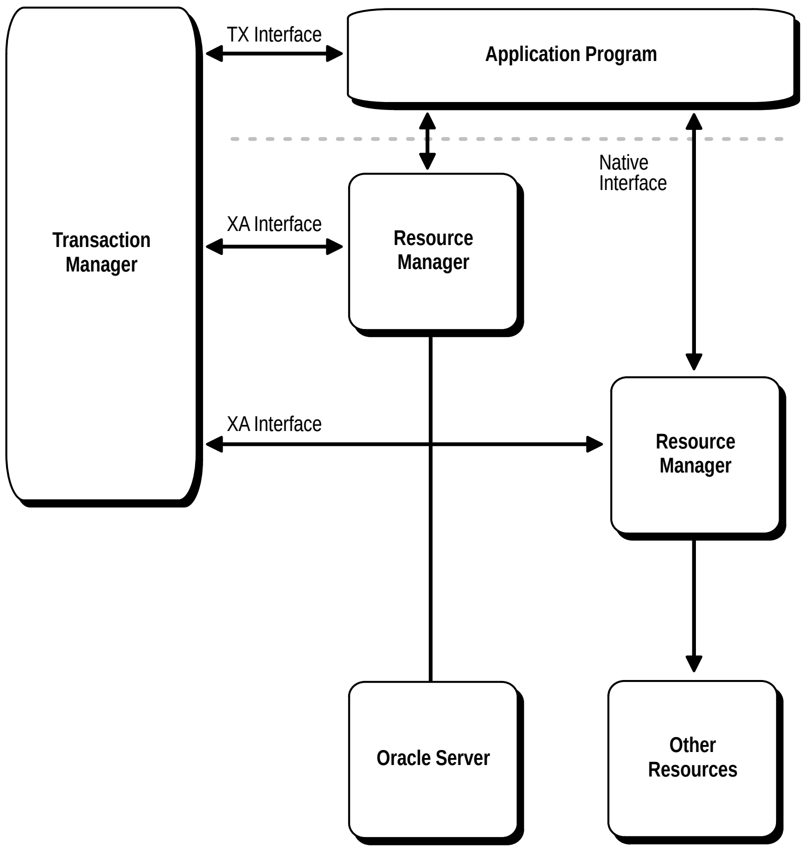

Figure 2 – 11 One Possible DTP Model

Figure 2 – 11 shows one way that the components of the DTP model can interact to provide consistent and efficient access to data in an Oracle database. The DTP model specifies the XA interface. Oracle provides an XA–compliant library to be linked into your applications. The native interface is the OCI API.

The DTP model that specifies how transaction and resource managers interact with application code is described in the *X/Open Guide Distributed Transaction Processing Reference Model* and related publications. These publications are available from

X/Open Company, Ltd.
1010 El Camino Real, Suite 380
Menlo Park CA 94025 USA

For more information on the XA interface, see the documentation provided with your transaction process (TP) monitor.

Oracle–Specific Information

You can use the Oracle Call Interfaces to develop applications that adhere to the X/Open standards. To do this, you must follow certain restrictions, as described in the next sections.

Connections

The application does not create and maintain the connections to a database. The transaction manager and the XA interface handle database connections and disconnections transparently. (The XA interface is supplied by Oracle.) This means that you normally do not use OLOG calls in an X/Open–compliant application, nor do you call OLOGOF to disconnect.

An OCI application requires a valid LDA (for example in the OOPEN call). Use the SQLLD2 routine to obtain a valid LDA for a specified connection, where the connection was established through the XA interface. See pages 4 – 115 (for C), 5 – 93 (for COBOL), or 6 – 86 (for FORTRAN) for a complete description of the SQLLD2 call.

Transaction Control Statements

Application code must not issue SQL statements or make OCI calls that affect the state of global transactions. You cannot issue a COMMIT for a global transaction in the application, because the transaction manager code must handle COMMITs. Also, you cannot issue a SQL DDL command, because DDL commands perform implicit COMMITs. And, of course, you do not use the OCI call OCOM for global transactions. The section “Illegal Operations” on page 2 – 56 lists all of the SQL commands and OCI calls that are not permitted for global transactions in the code you write to implement an application server.

Rollback

Applications can perform an internal ROLLBACK if they detect an error that prevents further SQL operations.

Note: This is subject to change in later versions of the XA interface.

Linking Resource Manager Applications

You must link in the XA library with your application’s object modules to obtain the XA interface functionality. Also, SQLLD2 is supplied with the Oracle Precompiler runtime library, SQLLIB. You must link SQLLIB with your application’s modules. See your Oracle system–specific documentation for complete instructions on linking XOPEN applications.

Illegal Operations

You cannot use the following OCI calls in an application server:

- OCOM, OCON, OCOF
- OLOG, ORLON, OLON, OLOGOF

In addition, you cannot issue SQL DDL commands, because they cause implicit commits. See Chapter 5 of the *Oracle7 Server SQL Reference* for a complete list of these commands.

You cannot use the following Transaction Control SQL commands:

- COMMIT
- ROLLBACK
- SET TRANSACTION
- SAVEPOINT

CHAPTER

3

Datatypes

This chapter is your principal reference for external datatype codes. These are required for the ODEFIN, OBNDRA, OBNDRV, and OBNDRN routines, and for the data conversions that are performed when you transfer data between your program and Oracle.

This chapter discusses the following topics:

- Oracle datatypes
- internal datatypes
- character strings and byte arrays
- external datatypes
- data conversions

Oracle Datatypes

When you change or insert data in an Oracle table, and when you receive data from Oracle on a query, the data passes between variables in your program and columns in the Oracle table. Oracle represents data internally in several formats. Among them are NUMBER, CHAR, DATE, and RAW.

Your OCI program stores data in variables, whose types are predefined by the language you are using. When you transfer data between Oracle and your program, you need to specify the format of the data in your program. For example, if you are processing the SQL statement

```
SELECT sal FROM emp WHERE empno = :employee_number
```

and you want the salary to come back as character data, rather than in a binary floating–point format, specify an Oracle external string datatype, such as VARCHAR2 (code = 1) or CHAR (code = 96) for the FTYPE parameter in the ODEFIN call. You also need to declare a string variable in your program and specify its address in the BUF parameter.

If you want the salary information to be returned as a binary floating–point value, however, specify the FLOAT (code = 4) external datatype. You also need to define a variable of the appropriate type for the BUF parameter.

Oracle performs most data conversions transparently. The ability to specify almost any external datatype provides a lot of power for performing specialized tasks. For example, you can input and output DATE values in pure binary format, with no character conversion involved, by using the DATE external datatype (code = 12). See the description of the DATE external datatype on page 3 – 15 for more information.

To control data conversion, you must use the appropriate external datatype codes in the bind and define routines, such as OBNDRA or ODEFIN. You must tell Oracle where the input or output variables are in your OCI program and their datatypes and lengths.

Internal Datatype Codes

The way Oracle represents data internally is conveyed to your program in the form of an *internal datatype code*. For example, if you do not know the datatypes of items to be returned by a query, you can call the ODESCR (describe) routine, which return an internal datatype code for each select–list item that is described. The internal datatypes are important for queries, because they let the program decide how to convert and format the output data.

External Datatype Codes

An *external datatype code* indicates to Oracle how the host variable represents data in your program. This determines how the data is converted when returned to output variables in your program, or how it is converted from input (bind) variables to Oracle column values. For example, if you want to convert a NUMBER in an Oracle column to a variable–length character array, you specify the VARCHAR2 external datatype code (1) in the ODEFIN call that defines the output variable.

To convert a bind variable to a value in an Oracle column, specify the external datatype code that corresponds to the type of the bind variable. For example, if you want to input a character string such as '25–JAN–64' to a DATE column, specify the datatype as a character string (1) and set the length parameter to nine.

It is always the programmer's responsibility to make sure that values are convertible. If you try to INSERT the string 'MY BIRTHDAY' into a DATE column, you will get an error when you execute the statement.

For a complete list of the external datatypes and datatype codes, see Table 3 – 2 on page 3 – 8.

Internal Datatypes

Table 3 – 1 represents the internal datatypes and datatype codes. This section, from the point of view of the Oracle user, provides the information needed to create or modify a database table.

Internal Oracle Datatype	Maximum Internal Length	Datatype Code
VARCHAR2	2000 bytes	1
NUMBER	21 bytes	2
LONG	2^31–1 bytes	8
ROWID	6 bytes	11
DATE	7 bytes	12
RAW	255 bytes	23
LONG RAW	2^31–1 bytes	24
CHAR	255 bytes	96
MLSLABEL	255 bytes	105

Table 3 – 1 Oracle Internal Datatypes

VARCHAR2

The VARCHAR2 datatype stores variable–length character strings. When a VARCHAR2 column is created, a maximum length between 1 and 2000 must be specified. Trailing blanks are never appended to a VARCHAR2 value on output.

See "Character Strings and Byte Arrays" on page 3 – 6 for more information about the VARCHAR2 internal datatype.

NUMBER

The NUMBER datatype stores fixed or floating–point numbers of virtually any size. You can specify *precision*, which is the total number of digits, and *scale*, which determines where rounding will occur.

The maximum precision of a NUMBER value is 38 decimal digits; the magnitude range is 1.0E–129 to 9.99E125. Scale can range from –84 to 127. For example, a scale of –3 means the number is rounded to the nearest thousand (3456 becomes 3000). A scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46).

When you specify precision and scale, Oracle does extra integrity checks before storing data in the column.

LONG

The LONG datatype stores variable–length character strings. LONG columns can store text, arrays of characters, or even complete documents. The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG column is 2^31–1 bytes.

You can use the OFLNG routine to retrieve a portion of a LONG (or LONG RAW) column, starting at any offset in the column.

You can also use the piecewise capabilities provided by ODEFINPS, OBINDPS, OGETPI and OSETPI to perform inserts, updates or fetches involving LONG columns.

> **Note:** The maximum length of LONG and LONG RAW columns is 2^31–1 bytes in an Oracle7 database. In Oracle Version 6 and earlier it is 65535 bytes.

Restrictions: You can use LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, function calls, or certain SQL clauses, such as WHERE, GROUP BY, and CONNECT BY. Only one LONG (or LONG RAW) column is allowed per database table, and that column cannot be indexed.

ROWID

The ROWID datatype returns the address of a row in an Oracle database. ROWIDs are the fastest way to access a particular row.

The internal size of an Oracle ROWID is six bytes. However, ROWIDs are not accessible in their internal format; the OCI program handles them in a binary or character external format. The size of the binary representation is operating–system dependent and is returned by a describe operation on a ROWID select–list item. The size of the character representation also varies. For additional information, see the discussion of VARCHAR2 and ROWID in the section "External Datatypes" on page 3 – 8.

DATE

The DATE datatype stores date and time information. Inside Oracle, DATEs are stored in a binary format. See the description of the DATE datatype on page 3 – 15 for a description of this format.

When a DATE column is converted to a character string in your program, it is returned using the default format mask for your session, or as specified in the INIT.ORA file.

If you need additional date information on a query, such as the time, or a date in Julian days, apply the TO_CHAR function to the date and use a format mask. Be sure to set the length of the output parameter in the ODEFIN routine to accommodate the additional characters. See *Oracle7 Server SQL Reference* for more information about TO_CHAR and format masks. Unless you want date information in the Oracle internal format (see the "External Datatypes" section on page 3 – 8), always convert DATE columns and expressions to and from character strings using external character datatypes, such as VARCHAR2 or STRING.

RAW

The RAW datatype represents binary data (for example, graphical data) that must not be interpreted as a character string. Gateways between networks might convert CHAR or LONG data from one character format to another (for example, ASCII to EBCDIC). RAW data is never converted in this way.

The maximum length of a RAW column is 255 bytes.

When RAW data in an Oracle table is converted to a character string in a program, the data is represented in hexadecimal character code. Each byte of the RAW data is returned as two characters that indicate the value of the byte, from '00' to 'FF'. If you want to input a character string in your program to a RAW column in an Oracle table, you must code the data in the character string using this hexadecimal code.

You can use the piecewise capabilities provided by ODEFINPS, OBINDPS, OGETPI and OSETPI to perform inserts, updates or fetches involving RAW (or LONG RAW) columns.

LONG RAW

The LONG RAW datatype is just like RAW, except that the maximum length of a LONG RAW column is 2^31–1 bytes. The restrictions that apply to LONG data also apply to LONG RAW data.

CHAR

The CHAR datatype stores fixed–length character strings. When a column is created of type CHAR, a maximum length between 1 and 255 is specified. When a CHAR value is output, trailing blanks are appended to the string, up to the column length specified.

See the section "Character Arrays and Strings" on page 3 – 7 for more information about the CHAR internal datatype.

MLSLABEL

You use the MLSLABEL datatype to store the binary format of an operating–system label. Trusted Oracle uses a label to control access to information. See the *Trusted Oracle7 Server Administrator's Guide* for more information about labels. The maximum width of an MLSLABEL column is 255 bytes.

In standard Oracle, you can define a column using the MLSLABEL datatype. However, the only valid value for the column is NULL. In Trusted Oracle7, you can insert any valid operating system label, in any valid format, into a column having the MLSLABEL datatype. Trusted Oracle implicitly converts the data to the binary format of a label.

Character Strings and Byte Arrays

There are five Oracle internal datatypes that you can use to specify columns that contain characters or arrays of bytes: CHAR, VARCHAR2, RAW, LONG, and LONG RAW.

CHAR, VARCHAR2, and LONG columns normally hold character data. RAW and LONG RAW hold bytes that are not interpreted as characters, for example, pixel values in a bit–mapped graphics image. Character data can be transformed when passed through a gateway between networks. For example, character data passed between machines using different languages (where single characters may be represented by differing numbers of bytes) can be significantly changed in length. Raw data is never converted in this way.

It is the responsibility of the database designer to choose the appropriate Oracle internal datatype for each column in the table. The OCI programmer must be aware of the many possible ways that character and byte–array data can be represented and converted between variables in the OCI program and Oracle tables.

When an array holds characters, the length parameter for the array in an OCI call is always passed in and returned in bytes, not characters.

CHAR and VARCHAR2

The VARCHAR2 internal datatype is equivalent to the CHAR datatype in Oracle V6 or earlier. The CHAR datatype in Oracle7 is a fixed–length string type. String comparisons between CHAR strings behave differently from string comparisons between VARCHAR2 strings.

String Comparison

Strings are compared using either *blank–padded* or *non–blank–padded* comparison semantics. In a blank–padded comparison, blanks are added to the shorter of the two strings to make the lengths equal. The strings are then compared, character by character, until a difference occurs or the end of the strings is reached. If a difference occurs, the string having the character with the greater value is considered greater.

In a non–blank–padded comparison, the strings are compared until a difference occurs or the end of the shorter string is reached. If two strings are equivalent up to the end of the shorter string, the longer string is considered greater. If two strings of equal length have no differing characters, they are equal.

Blank–padded comparisons are used whenever both of the strings are fixed length. See Chapter 3 of the *Oracle7 Server SQL Reference* for additional information about string comparison.

External Datatypes

Table 3 – 2 lists datatype codes for external datatypes. For each datatype, the table lists the program variable types for C, COBOL and FORTRAN from or to which Oracle internal data is normally converted.

EXTERNAL DATATYPE		**TYPE OF PROGRAM VARIABLE**		
NAME	**CODE**	**C**	**COBOL**	**FORTRAN**
VARCHAR2	1	char[n]	PIC X(n)	CHARACTER*n
NUMBER	2	unsigned char[21]	PIC X(21)	LOGICAL*1(21)
8–bit signed INTEGER	3	signed char	PIC S9(3) COMP	LOGICAL*1
16–bit signed INTEGER	3	signed short, signed int	PIC S9(4) COMP	INTEGER*2
32–bit signed INTEGER	3	signed int, signed long	PIC S9(9) COMP	INTEGER*4
FLOAT	4	float, double	PIC S9(n)V9(n) COMP–1,2	REAL*4, REAL*8
Null–terminated STRING	5	char[n+1]	n/a	n/a
VARNUM	6	char[22]	PIC X(22)	LOGICAL*1 (22)
PACKED DECIMAL	7	n/a	PIC S9(n)V9(n) COMP–3	n/a
LONG	8	char[n]	PIC X(n)	CHARACTER*n
VARCHAR	9	char[n+slen]	PIC X(n+slen) VARYING	LOGICAL*1(n+slen)
ROWID	11	char[n]	PIC X(n)	LOGICAL*1(n)
DATE	12	char[7]	PIC X(7)	LOGICAL*1(7)
VARRAW	15	unsigned char[n+slen]	PIC X(n+slen)	LOGICAL*1(n+slen)
RAW	23	unsigned char[n]	PIC X(n)	LOGICAL*1(n)
LONG RAW	24	unsigned char[n]	PIC X(n)	LOGICAL*1(n)

Notes:
Where the length is shown as *n*, it is a variable, and depends on the requirements of the program (or of the operating system, in the case of ROWID).
slen is the size in bytes of a short integer, *sizeof(***sb2***)* in C.
ilen is the size in bytes of an integer, *sizeof(***ub4***)* in C.

Table 3 – 2 External Datatypes and Codes

EXTERNAL DATATYPE		TYPE OF PROGRAM VARIABLE		
NAME	**CODE**	**C**	**COBOL**	**FORTRAN**
UNSIGNED INT	68	unsigned	n/a	n/a
DISPLAY	91	n/a	PIC S9(n) PIC S9(n)V9(n)	n/a
LONG VARCHAR	94	char[n+ilen]	PIC X(n+ilen)	LOGICAL*1(n+ilen)
LONG VARRAW	95	unsigned char[n+ilen]	PIC X(n+ilen)	LOGICAL*1(n+ilen)
CHAR	96	char[n]	PIC X(n)	CHARACTER*n
CHARZ	97	char[n+1]	n/a	n/a
CURSOR VARIABLE	102	struct cda_def		
MLSLABEL	105	char(n)	PIC X(n)	LOGICAL*1(n)

Notes:
Where the length is shown as *n*, it is a variable, and depends on the requirements of the program (or of the operating system, in the case of ROWID).
slen is the size in bytes of a short integer, *sizeof(***sb2***)* in C.
ilen is the size in bytes of an integer, *sizeof(***ub4***)* in C.

Table 3 – 2 External Datatypes and Codes

Each of the external datatypes is described below.

VARCHAR2

The VARCHAR2 datatype is a variable–length string of characters with a maximum length of 2000 bytes.

Input

The PROGVL parameter determines the length in the OBNDRA, OBNDRN, OBNDRV or OBINDPS call.

If the PROGVL parameter is greater than zero, Oracle obtains the bind variable value by reading exactly that many bytes, starting at the buffer address in your program. Trailing blanks are stripped, and the resulting value is used in the SQL statement or PL/SQL block. If, in the case of an INSERT statement, the resulting value is longer than the defined length of the database column, the INSERT fails, and an error is returned.

> **Note:** A trailing null is *not* stripped. Variables should be blank–padded but not null–terminated.

If the PROGVL parameter is –1, the character array is scanned for a null terminator character. If a null terminator is found in the first 2001 bytes of the array, the array elements up to, but not including the null terminator, are used as the bind variable value. If a null terminator is not found, Oracle returns an error. When the BUFL parameter is –1, trailing blanks (blanks immediately preceding the null terminator) are not stripped.

If the PROGVL parameter is zero, Oracle treats the bind variable as a null, regardless of its actual content. Of course, a null must be allowed for the bind variable value in the SQL statement. If you try to insert a null into a column that has a NOT NULL integrity constraint, Oracle issues an error, and the row is not inserted.

When the Oracle internal (column) datatype is NUMBER, input from a character string that contains the character representation of a number is legal. Input character strings are converted to internal numeric format. If the VARCHAR2 string contains an illegal conversion character, Oracle returns an error and the value is not input.

Output

Specify the desired length for the return value in the BUFL parameter of the ODEFIN or ODEFINPS call, or the PROGVL parameter of OBNDRA or OBNDRV for PL/SQL blocks. If zero is specified for the length, no data is returned.

If you omit the RLEN parameter of ODEFIN, returned values are blank–padded to the buffer length, and nulls are returned as a string of blank characters. If RLEN is included, returned values are not blank–padded. Instead, their actual lengths are returned in the RLEN parameter.

To check if a null is returned or if character truncation has occurred, include an indicator parameter in the ODEFIN call. The indicator parameter is set to –1 when a null is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to zero. If you do not specify an indicator parameter and a null is selected, the fetch call returns the error 1405.

Warning: If you are connected to an Oracle7 database, and you set the LNGFLG parameter of the OPARSE call to zero, then selecting a null with no indicator parameter defined does not cause an error to be returned. This is for Oracle Version 6 compatability.

Output to a character string from an internal NUMBER datatype can also be made. Number conversion follows the conventions established by National Language Support for your system. For example, your system might be configured to recognize a comma rather than period as the decimal point.

Special Considerations for ROWID

When an OCI program is connected to a non–Oracle data manager via an Oracle Open Gateway, the Oracle ROWID field in the CDA is not valid. In this case, ROWIDs must be explicitly SELECTed from the table into a character array.

The maximum length of the array is 255 bytes. To do dynamic memory allocation of the output buffer for a ROWID, you can call the ODESCR routine for a ROWID select–list item. When ODESCR returns datatype code 1 in the DBTYPE parameter, the DBSIZE parameter contains the correct size of the non–Oracle ROWID.

When you are connected to a non–Oracle data manager, you must use the VARCHAR2 external datatype code (1) in the FTYPE parameter of the bind or define call for all ROWIDs.

For maximum portability of your OCI application, Oracle recommends that you use explicit ROWIDs, rather than depending on the Oracle ROWID field of the CDA. In this case, always bind and define ROWIDs using the VARCHAR2 external datatype.

NUMBER

You should not need to use NUMBER as an external datatype. If you do use it, Oracle will output numeric values in its internal 21–byte binary format and will expect this format on input. The following discussion is included for completeness only.

Oracle stores values of the NUMBER datatype in a variable–length format. The first byte is the exponent and is followed by 1 to 20 mantissa bytes. The high–order bit of the exponent byte is the sign bit; it is set for positive numbers. The lower 7 bits represent the exponent, which is a base 100 digit with an offset of 65.

Each mantissa byte is a base 100 digit, in the range 1..100. For positive numbers, the digit has 1 added to it. So, the mantissa digit for the value 5 is 6. For negative numbers, instead of adding 1, the digit is subtracted from 101. So, the mantissa digit for the number –5 is 96 (101–5). Negative numbers have a byte containing 102 appended to the data bytes. However, negative numbers that have 20 mantissa bytes do not have the trailing 102 byte. Because the mantissa digits are stored in base 100, each byte can represent 2 decimal digits. The mantissa is normalized; leading zeroes are not stored.

Up to 20 data bytes can represent the mantissa. However, only 19 are guaranteed to be accurate. The 19 data bytes, each representing a base 100 digit, yield a maximum precision of 38 digits for an Oracle NUMBER.

If you specify the datatype code 2 in the FTYPE parameter of an ODEFIN or ODEFINPS call, your program receives numeric data in this Oracle internal format. The output variable should be a 21–byte array to accommodate the largest possible number. Note that only the bytes that represent the number are returned. There is no blank padding or null termination. If you need to know the number of bytes returned, use the VARNUM external datatype instead of NUMBER. See the description of VARNUM on page 3 – 13 for examples of the Oracle internal number format.

INTEGER

The INTEGER datatype converts numbers. An external integer is a signed binary number; the size in bytes is system dependent. The host system architecture determines the order of the bytes in the variable. A length specification is required for input and output. If the number being output from Oracle is not an integer, the fractional part is discarded, and no error or other indication is returned. If the number to be returned exceeds the capacity of a signed integer for the system, an “overflow on conversion” error is returned.

FLOAT

The FLOAT datatype processes numbers that have fractional parts or that exceed the capacity of an integer. The number is represented in the host system’s floating–point format. Normally the length is either four or eight bytes. The length specification is required for both input and output.

Because the internal format of an Oracle number is decimal and most floating–point implementations are binary, Oracle can represent numbers with greater precision than floating–point representations.

> **Note:** You may get a round–off error when converting between FLOAT and NUMBER. Thus, using a FLOAT as a bind variable in a query may return an ORA–1403 error. You can avoid this situation by converting the FLOAT into a STRING and then using datatype code 1 or 5.

STRING

The null–terminated STRING format behaves like the VARCHAR2 format (datatype code 1), except that the string must contain a null terminator character. This datatype is most useful for C programs.

Input

The string length supplied in the OBINDPS, OBNDRA, OBNDRN, or OBNDRV call limits the scan for the null terminator. If the null terminator is not found within the length specified, Oracle issues the error

```
ORA–01480:  trailing null missing from STR bind value
```

If the length is not specified in the bind call, an implied maximum string length of 2000 is used.

The minimum string length is two bytes. If the first character is a null terminator and the length is specified as two, a null is inserted in the column, if permitted. Unlike types 1 and 96, a string containing all blanks is not treated as a null on input; it is inserted as is.

Output

A null terminator is placed after the last character returned. If the string exceeds the field length specified, it is truncated and the last character position of the output variable contains the null terminator.

A null select–list item returns a null terminator character in the first character position. An ORA–01405 error is possible, as well.

VARNUM

The VARNUM datatype is like the external NUMBER datatype, except that the first byte contains the length of the number representation. This length does not include the length byte itself. Reserve 22 bytes to receive the longest possible VARNUM. Set the length byte when you input a VARNUM value to Oracle.

Table 3 – 3 shows several examples of the VARNUM values returned for numbers in an Oracle table.

Decimal Value	Length Byte	Exponent Byte	Mantissa Bytes	Terminator Byte
0	1	128	n/a	n/a
5	2	193	6	n/a
–5	3	62	96	102
2767	3	194	28, 68	n/a
–2767	4	61	74, 34	102
100000	2	195	11	n/a
1234567	5	196	2, 24, 46, 68	n/a

Table 3 – 3 VARNUM Examples

PACKED DECIMAL

The PACKED DECIMAL datatype converts between non–integral numbers in Oracle and a datatype that is suitable for calculation. In COBOL, the data area must be a signed COMP–3 field with an implied decimal point. The number of digits to the right of the decimal point is specified in the FMT parameter of the ODEFIN or ODEFINPS routine and in the SCALE parameter. For more information about defining conversion format strings, see the descriptions of ODEFIN in Chapter 5. The returned value can be used as is for COBOL calculations or can be moved to a computational field before calculations. The number will never be converted to scientific notation. If the number to be returned loses significant digits during the conversion, Oracle fills the buffer with asterisk (*) characters.

LONG

The LONG datatype stores character strings longer than 2000 bytes. You can store up to 2^31–1 bytes in a LONG column. Columns of this type are used only for storage and retrieval of long strings. They cannot be used in functions, expressions, or WHERE clauses. LONG column values are generally converted to and from character strings.

VARCHAR

The VARCHAR datatype stores character strings of varying length. The first two bytes contain the length of the character string, and the remaining bytes contain the string. The specified length of the string in a bind or a define call must include the two length bytes, so the largest VARCHAR string that can be received or sent is 65533 bytes long, not 65535. For converting longer strings, use the LONG VARCHAR external datatype.

ROWID

The ROWID datatype identifies a particular row in a database table. ROWID can be a select–list item in a query; for example:

```
SELECT rowid, ename, sal FROM emp FOR UPDATE OF sal
```

In this case, you use the returned ROWID in further INSERT, UPDATE, or DELETE statements.

Also, after INSERT, DELETE, UPDATE, and SELECT FOR UPDATE statements are executed, the Oracle ROWID field in the CDA can contain a binary representation of the ROWID for the row that was just changed or selected. However, this field is valid only if the OCI program is connected to an Oracle database. See the “Special Considerations” section on page 3 – 11, in the description of the external datatype VARCHAR2, for more information.

The size of the binary representation of a ROWID is system dependent. One way to determine the binary ROWID size on your system is to describe, using ODESCR, a SQL statement such as

```
SELECT rowid FROM dual
```

Whenever ODESCR returns the datatype code 11 in the DBTYPE parameter, the DBSIZE parameter contains the size in bytes of the binary ROWID.

Never attempt to construct a ROWID and use it in subsequent DML statements. Use only ROWIDs returned by Oracle.

DATE

The DATE datatype can update, insert, or retrieve a date value using the Oracle internal date binary format. A date in binary format contains seven bytes, as shown in Table 3 – 4.

BYTE	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example (for 30–NOV–1992, 3:17 PM)	119	192	11	30	16	18	1

Table 3 – 4 Format of the DATE Datatype

The century and year bytes are in an excess–100 notation. Dates Before Common Era (BCE) are less than 100. The era begins on 01–JAN–4712 BCE, which is Julian day 1. For this date, the century byte is 53, and the year byte is 88. The hour, minute, and second bytes are in excess–1 notation. The hour byte ranges from 1 to 24, the minute and second bytes from 1 to 60. If no time was specified when the date was created, the time defaults to midnight (1, 1, 1).

When you input a date in binary format using the DATE external datatype, the database does not do consistency or range checking. All data in this format must be carefully validated before input.

Note: There is little need to use the Oracle external DATE datatype in ordinary database operations. It is much more convenient to convert DATEs in character format, because the program usually displays (on a query) or inputs in a character format, such as ‘DD–MON–YY’.

VARRAW

The VARRAW datatype is similar to the RAW datatype. However, the first two bytes contain the length of the data. The specified length of the string in a bind or a define call must include the two length bytes. So the largest VARRAW string that can be received or sent is 65533 bytes long, not 65535. For converting longer strings, use the LONG VARRAW external datatype.

RAW	The RAW datatype is used for data that is not to be interpreted by Oracle. The maximum length of a RAW column is 255 bytes. The raw datatypes are intended for binary data or byte strings, for example, to store graphics character sequences. For more information, see to the *Oracle7 Server SQL Reference.*
LONG RAW	The LONG RAW datatype is similar to the RAW datatype, except that it stores raw data with a length up to 2^31–1 bytes.
UNSIGNED	The UNSIGNED datatype is used for unsigned binary integers. The size in bytes is system dependent. The host system architecture determines the order of the bytes in a word. A length specification is required for input and output. If the number being output from Oracle is not an integer, the fractional part is discarded, and no error or other indication is returned. If the number to be returned exceeds the capacity of an unsigned integer for the system, an “overflow on conversion” error is returned.
DISPLAY	The DISPLAY datatype stores numeric character data. The DISPLAY datatype refers to a COBOL “DISPLAY SIGN LEADING SEPARATE” number, which typically requires *n* + 1 bytes of storage for PIC S9(n) and *n* + *d* + 1 bytes for PIC S9(n)V9(d).
LONG VARCHAR	The LONG VARCHAR datatype stores data from and into an Oracle LONG column. The first four bytes of a LONG VARCHAR contain the length of the item. So, the maximum length of a stored item is 2^31–5 bytes. On a bind or define call, if the length of the data area is greater than 65533 bytes, pass a –1 in the length parameter and make sure the length is in the first four bytes of the data area *before* the bind or define.
LONG VARRAW	The LONG VARRAW datatype is used to store data from and into an Oracle LONG RAW column. The length is contained in the first four bytes. The maximum length is 2^31–5 bytes. On a bind or define call, if the length of the data area is greater than 65533 bytes, pass a –1 in the length parameter, and make sure the length is in the first four bytes of the data area *before* the bind or define.
CHAR	The CHAR datatype is a string of characters, with a maximum length of 255. CHAR strings are compared using blank–padded comparison semantics (see Chapter 3 in the *Oracle7 Server SQL Reference*).
Input	The length is determined by the PROGVL parameter in the OBNDRA, OBNDRN, or OBNDRV call.

Note: The entire contents of the buffer (PROGVL chars) is passed to the database, *including* any trailing blanks or nulls.

If the PROGVL parameter is zero, Oracle treats the bind variable as a null, regardless of its actual content. Of course, a null must be allowed for the bind variable value in the SQL statement. If you try to insert a null into a column that has a NOT NULL integrity constraint, Oracle issues an error and does not insert the row.

Negative values (especially –1) for the PROGVL parameter are *not* allowed for CHARs.

When the Oracle internal (column) datatype is NUMBER, input from a character string that contains the character representation of a number is legal. Input character strings are converted to internal numeric format. If the CHAR string contains an illegal conversion character, Oracle returns an error and does not input the value. Number conversion follows the conventions established by National Language Support settings for your system. For example, your system might be configured to recognize a comma (,) rather than a period (.) as the decimal point.

Output

Specify the desired length for the return value in the BUFL parameter of the ODEFIN or ODEFINPS call. If zero is specified for the length, no data is returned.

If you omit the RLEN parameter of ODEFIN, returned values are blank padded to the buffer length, and nulls are returned as a string of blank characters. If RLEN is included, returned values are not blank padded. Instead, their actual lengths are returned in the RLEN parameter.

To check if a null is returned or if character truncation has occurred, include an indicator parameter or array of indicator parameters in the ODEFIN or ODEFINPS call. An indicator parameter is set to –1 when a null is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to zero. If you do not specify an indicator parameter and a null is selected, the fetch call returns an ORA–01405 error.

Note: If you are connected to an Oracle7 database, but you set the LNGFLG parameter of the OPARSE call to zero, selecting a null with no indicator parameter defined returns no error.

Output to a character string from an internal NUMBER datatype can also be made. Number conversion follows the conventions established by the National Language Support settings for your system. For example, your system might use a comma (,) rather than a period (.) as the decimal point.

CHARZ

The CHARZ external datatype is similar to the CHAR datatype, except that the string must be null terminated on input, and Oracle places a null–terminator character at the end of the string on output. The null terminator serves only to delimit the string on input or output; it is not part of the data in the table.

On input, the length parameter must indicate the exact length including the null terminator. For example, if an array in C is declared as

```
char my_num[] = ”123.45”;
```

then the length parameter when you bind *my_num* must be seven. *Any* other value would return an error for this example.

CURSOR VARIABLE

The CURSOR VARIABLE datatype covers *all* cursor type definitions in PL/SQL, though you can use the older PL/SQL *static* cursors in a PL/SQL block. To bind a PL/SQL cursor variable to an OCI program variable in one of the host languages supported, you must use the regular cursor datatype for declaring CDAs and bind your variable using OBNDRA or OBINDPS.

After binding the program cursor to a PL/SQL cursor variable, you can execute the PL/SQL block. Then you can use the variable cursor as a regular program cursor. For example, you may then describe the select–list items, bind them to program variables (using ODEFIN or ODEFINPS), fetch rows, and close the cursor.

For more information see the section “Cursor Variables” on page 2 – 50.

MLSLABEL

Use the MLSLABEL datatype to store an operating–system label in binary form. A Trusted Oracle7 label controls access to information. See the *Trusted Oracle7 Server Administrator’s Guide* for more information about labels. In standard Oracle, you can define a column using the MLSLABEL datatype. However, the only valid value for the column is null.

The internal length of MLSLABEL is between two and five bytes.

Input — Trusted Oracle7 translates the input text string (up to 255 bytes long) into a binary label and ensures the label is valid within the operating system. If it is not, Trusted Oracle returns an error. If it is valid, the binary form of the label is stored in the target database column.

Output — Trusted Oracle7 translates the binary label to a character string.

Data Conversions

Table 3 – 5 shows the supported conversions from internal Oracle datatypes to external datatypes, and from external datatypes into internal column representations.

EXTERNAL		INTERNAL								
		1 VARCHAR2	2 NUMBER	8 LONG	11 ROWID	12 DATE	23 RAW	24 LONG RAW	96 CHAR	105 MLSLABEL
1	VARCHAR2	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3)	I/O	I/O(7)
2	NUMBER	I/O(4)	I/O	I					I/O(4)	
3	INTEGER	I/O(4)	I/O	I					I/O(4)	
4	FLOAT	I/O(4)	I/O	I					I/O(4)	
5	STRING	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3,5)	I/O	I/O(7)
6	VARNUM	I/O(4)	I/O	I					I/O(4)	
7	DECIMAL	I/O(4)	I/O	I					I/O(4)	
8	LONG	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3,5)	I/O	I/O(7)
9	VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3,5)	I/O	I/O(7)
11	ROWID	I		I	I/O				I	
12	DATE	I/O		I		I/O			I/O	
15	VARRAW	I/O(6)		I(5,6)			I/O	I/O	I/O(6)	
23	RAW	I/O(6)		I(5,6)			I/O	I/O	I/O(6)	
24	LONG RAW	O(6)		I(5,6)			I/O	I/O	O(6)	
68	UNSIGNED	I/O(4)	I/O	I					I/O(4)	
91	DISPLAY	I/O(4)	I/O	I					I/O(4)	
94	LONG VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3,5)	I/O	I/O(7)
95	LONG VARRAW	I/O(6)		I(5,6)			I/O	I/O	I/O(6)	
96	CHAR	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3)	I/O	I/O(7)
97	CHARZ	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3)	I/O	I/O(7)
105	MLSLABEL	I/O(8)		I/O(8)					I/O(8)	I/O

Notes:
(1) For input, host string must be in Oracle 'BBBBBBBB.RRRR.FFFF' format.
On output, column value is returned in same format.
(2) For input, host string must be the default DATE character format.
On output, column value is returned in same format
(3) For input, host string must be in hex format.
On output, column value is returned in same format.
(4) For output, column value must represent a valid number.
(5) Length must be less than or equal to 2000.
(6) On input, column value is stored in hex format.
For output, column value must be in hex format.
(7) For input, host string must be a valid OS label in text format.
On output, column value is returned in same format.
(8) For input, host string must be a valid OS label in raw format.
On output, column value is returned in same format.

Legend:
I = input only
O = output only
I/O = input or output

Table 3 – 5 Data Conversion

CHAPTER

4

The OCI Functions for C

This chapter describes each function in the OCI library for the OCI C programmer. The description of many of the functions includes an example that shows how an OCI program uses the function. Examples are not provided for the simpler functions. The description of each function has the following parts:

Purpose	What the function does.
Syntax	The function call with its parameter list.
Comments	A detailed description of the function, including examples.
Parameters	A detailed description of each parameter.
See Also	A list of other functions that affect or are used with this function. Not included if not applicable.

Be sure to read "Calling OCI Routines" on page 4 – 2. It contains important information about data structures, datatypes, parameter passing conventions, and other important information about the OCI functions.

Calling OCI Routines

This section describes data structures and coding rules that are specific to applications written in the C language. Refer to this section for information about data structures, datatypes, and parameter passing conventions in OCI C programs.

Datatypes

The datatypes used in the C examples in this guide are defined in the file *oratypes.h*. This file is port specific. The types defined, such as **sb2** and **ub4**, can take different C types on different systems. An example *oratypes.h* file for UNIX C compilers is listed in Appendix A.

Different OCI platforms may have different datatype definitions. The online location of the *oratypes.h* file can also be system specific. On Unix systems, it can be found at $ORACLE_HOME/rdbms/demo/oratypes.h. See your Oracle system–specific documentation for the location of *oratypes.h* on your system.

Data Structures

To use the OCI functions, you must define data structures for one or more LDAs and CDAs. The internal structure of these data areas is discussed in the section "OCI Data Structures" on page 2 – 2. The LDA structure is the same size as the CDA structure, and the same structure declaration can be used for both structures.

The only field an OCI application normally accesses in the LDA is the *return code* field. In the example code in this section, the datatypes *Lda_Def* and *Cda_Def*, as defined in the header file *ocidfn.h*, are used to define the LDAs and CDAs. This file is listed in Appendix A and is also available online; see the Oracle system–specific documentation for the online location of this file.

Parameter Names

The prototype parameter names in the function descriptions are six or less characters in length and do not contain non–alphanumeric characters. This maintains common names across all languages supported by the OCI. In your OCI C program, you can of course use longer, more descriptive names for the parameters.

Parameter Types

The OCI functions take three types of parameters:

- integers (**sword, eword**, **sb4**, **ub4**)
- short integers (**sb2**, **ub2**)
- character variables (**ub1, sb1**)
- addresses of program variables (pointers)
- memory pointers (**dvoid**)

The following two sections discuss special considerations to remember when passing parameters to the OCI functions.

Integers

When passing integer literals to an OCI function, you should cast the literal to the type of the parameter. For example, the *oparse()* function has the following prototype, using ANSI C notation:

```
oparse(Cda_Def *cursor, text *sqlstm, sb4 sqllen,
       sword defflg, ub4 lngflg);
```

If you call *oparse()* as

```
oparse(&cda, (text *) ″select sysdate from dual″, –1, 1, 2);
```

it will usually work on most 32–bit systems, although the C compiler might issue warning messages to the effect that the type conversions are non–portable. So, you should call *oparse()* as

```
oparse(&cda, (text *) ″select sysdate from dual″, (sb4) –1,
       (sword) 0, (ub4) 2);
```

Always be careful to distinguish signed and unsigned short integers (**sb2** and **ub2**) from integers (**sword**), and signed and unsigned long integers (**sb4** and **ub4**).

Addresses

Be careful to pass all pointer parameters as valid addresses. When passing the null pointer (0), Oracle recommends that you cast it to the appropriate type. When you pass a pointer as a parameter, your OCI program *must* allocate the storage for the object to which it points. OCI routines never allocate storage for program objects. String literals can be passed to an OCI function where the parameter type is **text *** as the *oparse()* example in the previous section demonstrates.

Parameter Classification

There are three kinds of parameters:

- required parameters
- optional parameters
- unused parameters

Required Parameters

Required parameters are used by Oracle, and the OCI program must supply valid values for them.

Optional Parameters

The use of optional parameters depends on the requirements of your program. The Syntax section for each routine in this chapter indicates optional parameters using square brackets ([ ]).

In most cases, an unused optional parameter is passed as –1 if it is an integer. It is passed as the null pointer (0) if it is an address parameter. For example, your program might not need to supply an indicator variable on a bind call, in which case all input values in the program could be non–null. The *indp* parameter in the bind functions *obindps(), obndra(), obndrv()*, and *obndrn()* is optional. This parameter is a pointer, so it is passed as a null pointer (**(sb2 *)** 0) when it is not used.

> **Note:** A value of –1 should not be passed for unused optional parameters in the new *obindps()* and *odefinps()* calls. Unused parameters in these calls must be passed a zero or NULL. See the descriptions of individual calls for more details about specific parameters.

Unused Parameters

Unused parameters are not used by Oracle, at least for the language being described. For example, for cross–language compatibility, some OCI functions have the parameters *fmt*, *fmtl*, and *fmtt*. These are the format string specifier for the packed decimal external datatype, and the string length and type parameters when this type is being bound. COBOL uses the packed decimal type, so these parameters are unused in C.

However, you always pass unused parameters. In C, pass these in the same way as omitted optional parameters. In most cases, this means passing –1 if it is an integer parameter, or 0 if it is a pointer. See the syntax and examples for the *odefin()* function(on page 4 – 35) for examples of how to pass omitted optional and unused parameters.

> **Note:** As with optional parameters, a value of –1 should not be passed for unused parameters in the new *obindps()* and *odefinps()* calls. Unused parameters in these calls must be passed a zero or NULL. See the descriptions of individual calls for more details about specific parameters.

The Syntax section (in the description of each function) uses angle brackets (< >) to indicate unused parameters.

Parameter Descriptions

Parameters for the OCI functions are described in terms of their type and their mode. When a parameter is a CDA or an LDA, the type is a *Cda_Def* * or a *Lda_Def* *. That is, a pointer to a *Cda_Def* or *Lda_Def* structure as defined in *ocidfn.h* (see page A – 8).

> **Note:** The OCI program must allocate these structures, not just the pointers.

Parameter Modes

When a parameter is a generalized pointer (that is, it can be a pointer to any variable or array of variables, depending on the requirements of your program), its type is listed as a ub1 pointer. The mode of a parameter has three possible values:

IN	A parameter that passes data to Oracle.
OUT	A parameter that receives data from Oracle on this or a subsequent call.
IN / OUT	A parameter that passes data on the call and receives data on the return from this or a subsequent call.

Function Return Values

When called from a C program, OCI functions return an integer value. The return value is 0 if the function completed without error. If a non–zero value is returned, an error occurred. In that event, you should check the *return code* field in the CDA to get the error number. The example programs in Appendix A demonstrate this.

The shorter code fragments in this chapter do not always check for errors.

Note: *oerhms()*, *sqlld2()*, and *sqllda()* are exceptions to this rule. *oerhms()* returns the length of the message. *sqlld2()* and *sqllda()* are void functions that return error indications in the LDA parameter.

Variable Location

When you bind and define program variables using *obindps()*, *obndra()*, *obndrv()*, *obndrn()*, *odefinps()* and *odefin()*, they are known to Oracle by their addresses. The address when bound must remain valid when the statement is executed.

If you pass LDA and CDA structures to other non–OCI functions in your program, always pass them as pointers, not by value. Oracle updates the fields in these structures after OCI calls. You also lose important information if your program uses copies of these structures, which will not be updated by OCI calls.

Caution: A change in the location of local variables may also cause errors in an OCI program. When the address of a local variable used in a subsequent call is passed to Oracle as a parameter in a bind or define call, you must be certain that the addressed variable is actually at the specified location when it is used in the subsequent execute or fetch call.

For more information about variable locations, see the section "Optimizing Compilers" on page 2 – 31.

obindps

Purpose *obindps()* associates the address of a program variable with a placeholder in a SQL or PL/SQL statement. Unlike older OCI bind calls, *obindps()* can be used to bind placeholders to be used in piecewise operations, or operations involving arrays of structures.

Syntax

```
obindps(Cda_Def *cursor, ub1 opcode, text *sqlvar,
        [sb4 sqlvl], ub1 *pvctx, sb4 progvl,
        sword ftype, <sword scale>, [sb2 *indp],
        [ub2 *alenp], [ub2 *rcodep], sb4 pv_skip,
        sb4 ind_skip, sb4 alen_skip, sb4 rc_skip,
        [ub4 maxsiz], [ub4 *cursiz], <text *fmt>,
        <sb4 fmtl>, <sword fmtt>);
```

Comments *obindps()* is used to associate the address of a program variable with a placeholder in a SQL or PL/SQL statement. Additionally, it can indicate that an application will be providing inserted or updated data incrementally at runtime. This piecewise insert is designated in the *opcode* parameter. *obindps()* is also used when an application will be inserting data stored in an array of structures.

> **Note:** This function is only compatible with Oracle Server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

With the introduction of *obindps()* there are now four fully–supported calls for binding input parameters, the other three being the older *obndra()*, *obndrn()* and *obndrv()*. Application developers should consider the following points when determining which bind call to use:

- *obindps()* is supported only when a program is linked in deferred mode. If it is necessary to link in non–deferred mode, another bind routine must be used. In this case, the ability to handle piecewise operations and arrays of structures is not supported.
- *obindps()* is more complex than the older bind calls. Users who are not performing piecewise operations and are not using arrays of structures may choose to use one of the older routines.
- *obindps()* does not support the ability to do a positional bind. If this functionality is needed, the bind should be performed using *obndrn()*.

Unlike older OCI calls, *obindps()* does not accept –1 for any optional or unused parameters. When it is necessary to pass a value to these parameters NULL or 0 should be used instead. The only exception to this rule is that a –1 length is acceptable for s*qlvl* if *sqlvar* is null–terminated.

See the sections “Piecewise Insert, Update and Fetch,” and “Arrays of Structures” in Chapter 2 for more information about piecewise operations, arrays of structures, skip parameters and the *obindps()* call.

The following sample code demonstrates the use of *obindps()* in an OCI program which performs an insert from an array of structures. This code is provided for demonstration purposes only, and does not constitute a complete program. Most of the work in the program is done within the *insert_records()* function.

For sample code demonstrating an array fetch, see the description of the *odefinps()* routine later in this chapter. For sample code demonstrating the use of *obindps()* for a piecewise insert, see the description of the *ogetpi()* routine later in this chapter.

```
...                                  /* OCI #include statements */

#define  DEFER_PARSE        1                     /* oparse flags */
#define  NATIVE             1
#define  VERSION_7          2
#define  ARRAY_SIZE         10

#define OCI_EXIT_FAILURE 1                        /*   exit flags */
#define OCI_EXIT_SUCCESS 0

void   insert_records();

struct emp_record                       /* employee data record */
{  int   empno;
   char  ename[11];
   char  job[11];
   int   mgr;
   char  hiredate[10];
   float sal;
   float comm;
   int   deptno;
};
typedef struct emp_record emp_record;

struct emp_record_indicators
{  short empno;                    /* indicator variable record */
   short ename;
   short job;
   short mgr;
   short hiredate;
   short sal;
   short comm;
   short deptno;
};
typedef struct emp_record_indicators emp_record_indicators;

Lda_Def   lda;                                /*  login area */
ub1       hda[256];                           /*   host area */
Cda_Def   cda;                                /* cursor area */

main()
{
  emp_record    emp_records[ARRAY_SIZE];
  emp_record_indicators emp_rec_inds[ARRAY_SIZE];
  int i=0;
  char yn[4];

 ...                                 /* log on to the database */

  for (i=0;i<ARRAY_SIZE;i++)
{
...                           /* prompt user for data necessary */
...                           /* to fill emp_records and        */
...                           /* emp_records_inds arrays        */
}
  insert_records(i,&emp_records, &emp_records_inds);

...                                /* log off from the database */

}

/* Function insert_records(): This function inserts the array   */
/*                            of records passed to it.          */

void      insert_records(n, emp_records, emp_rec_inds)
int n;
emp_record emp_records[];
emp_record_indicators  emp_rec_inds[];
{
  text *sqlstmt =(text *) ”INSERT INTO EMP (empno,ename, deptno) \
                         VALUES (:empno, :ename, :deptno)”;

  if (oopen(&cda, &lda, (text *)0, –1, –1, (text *)0, –1))
    exit(OCI_EXIT_FAILURE);

  if (oparse(&cda, sqlstmt, (sb4)–1, 0, (ub4)VERSION_7))
    exit(OCI_EXIT_FAILURE);

  if (obindps(&cda, 1, (text *)”:empno”,
              strlen(”:empno”), (ub1 *)&emp_records[0].empno,
              sizeof(emp_records[0].empno),
              SQLT_INT, (sword)0, (sb2 *) &emp_rec_inds[0].empno,
              (ub2 *)0, (ub2 *)0, (sb4) sizeof(emp_record),
              (sb4) sizeof(emp_record_indicators), 0, 0,
              0, (ub4 *)0, (text *)0, 0, 0))
    exit(OCI_EXIT_FAILURE);

  if (obindps(&cda, 1, (text *)”:ename”,
              strlen(”:ename”), (ub1 *)emp_records[0].ename,
              sizeof(emp_records[0].ename),
              SQLT_STR, (sword)0, (sb2 *) &emp_rec_inds[0].ename,
              (ub2 *)0, (ub2 *)0, (sb4) sizeof(emp_record),
              (sb4) sizeof(emp_record_indicators), 0, 0,
              0, (ub4 *)0, (text *)0, 0, 0))
    exit(OCI_EXIT_FAILURE);

  if (obindps(&cda, 1, (text *)”:deptno”,
              strlen(”:deptno”), (ub1 *)&emp_records[0].deptno,
              sizeof(emp_records[0].deptno),
              SQLT_INT, (sword)0, (sb2 *) &emp_rec_inds[0].deptno,
              (ub2 *)0, (ub2 *)0, (sb4) sizeof(emp_record),
              (sb4) sizeof(emp_record_indicators),
              0, 0, 0, (ub4 *)0, (text *)0, 0, 0))
    exit(OCI_EXIT_FAILURE);

  if (oexn(&cda,n,0))
    exit(OCI_EXIT_FAILURE);

  ocom(&lda);                              /* commit the insert */

 if (oclose(&cda))                              /* close cursor */
    exit(OCI_EXIT_FAILURE);
}
```

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
opcode	ub1	IN
sqlvar	text *	IN
sqlvl	sb4	IN
pvctx	ub1*	IN
progvl	sb4	IN
ftype	sword	IN
scale	sword	IN

Parameter Name	*Type*	*Mode*
indp	sb2 *	IN / OUT
alenp	ub2 *	IN
rcodep	ub2 *	OUT
pv_skip	sb4	IN
ind_skip	sb4	IN
alen_skip	sb4	IN
rc_skip	sb4	IN
maxsiz	ub4	IN
cursiz	ub4 *	IN / OUT
fmt	text *	IN
fmtl	sb4	IN
fmtt	sword	IN

Note: Since the *obindps()* call can be used in a variety of different circumstances, some items in the following list of parameter descriptions may include different explanations for how the parameter is used for piecewise operations, arrays of structures and standard scalar or array binds.

Standard scalar and array binds are those binds which were previously possible using other OCI bind calls (*obndra(), obndrv(),* and *obndrn()*).

cursor
A pointer to the CDA associated with the SQL statement or PL / SQL block being processed.

opcode
Piecewise bind: pass as 0.

Arrays of structures or standard bind: pass as 1.

sqlvar
Specifies the address of a character string holding the name of a placeholder (including the preceding colon, e.g., ":varname") in the SQL statement being processed.

sqlvl
The length of the character string in *sqlvar*, including the preceding colon. For example, the placeholder ":employee" has a length of nine. If the string is null terminated, this parameter can be specified as –1.

pvctx
Piecewise bind: A pointer to a context block entirely private to the application. This should be used by the application to store any information about the column being bound. One possible use would be to store a pointer to a file which will be referenced later. Each bind variable can then have its own separate file pointer. This pointer can be retrieved during a call to *ogetpi()*.

Arrays of structures or standard bind: A pointer to a program variable or array of program variables from which input data will be retrieved when the SQL statement is executed. For arrays of structures this should point to the first scalar element in the array of structures being bound. This parameter is equivalent to the *progv* parameter from the older OCI bind calls.

progvl
Piecewise bind: This should be passed in as the maximum possible size of the data element of type *ftype*.

Arrays of structures or standard bind: This should be passed as the length in bytes of the datatype of the program variable, array element or the field in a structure which is being bound.

ftype
The external datatype code of the program variable being bound. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. See the section "External Datatypes" in Chapter 3 for a list of datatype codes, and the listings of *ocidem.h* and *ocidfn.h* in Appendix A for lists of constant definitions corresponding to datatype codes.

For piecewise operations, the valid datatype codes are 1 (VARCHAR2), 5 (STRING), 8 (LONG) and 24 (LONG RAW).

scale
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

indp
Pointer to an indicator variable or array of indicator variables. For arrays of structures this may be an interleaved array of column–level indicator variables. See page 2 – 29 for more information about indicator variables.

alenp
Piecewise bind: pass as (**ub2** *)0.

Arrays of structures or standard bind: A pointer to a variable or array containing the length of data elements being bound. For arrays of structures, this may be an interleaved array of column–level length variables. The maximum usable size of the array is determined by the *maxsiz* parameter.

rcodep
Pointer to a variable or array of variables where column–level error codes are returned after a SQL statement is executed. For arrays of structures, this may be an interleaved array of column–level return code variables.

Typical error codes would indicate that data in *progv* has been truncated (ORA–01406) or that a null occurred on a SELECT or PL / SQL FETCH (ORA–01405).

pv_skip
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of structures holding program variables being bound. In general, this value will be *sizeof*(structure). If a standard array bind is being performed, this value should equal the size of one element of the array being bound.

ind_skip
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of indicator variables associated with an array holding program data to be inserted. This parameter will either equal the size of one indicator parameter structure (for arrays of structures) or the size of one indicator variable (for standard array bind).

alen_skip
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of data lengths associated with an array holding program data to be inserted. This parameter will either equal the size of one length variable structure (for arrays of structures) or the size of one length variable (for standard array bind).

rc_skip
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array used to store returned column–level error codes associated with the execution of a SQL statement. This parameter will either equal the size of one return code structure (for arrays of structures) or the size of one return code variable (for standard array bind).

maxsiz
The maximum size of an array being bound to a PL/SQL table. Values range from 1 to 32512, but the maximum size of the array depends on the datatype. The maximum array size is 32512 divided by the internal size of the datatype.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to ((**ub4**)0) for SQL scalar or array binds.

cursiz
A pointer to the actual number of elements in the array being bound to a PL/SQL table.

If *progv* is an IN parameter, set the *cursiz* parameter to the size of the array being bound. If *progv* is an OUT parameter, the number of valid elements being returned in the *progv* array is returned after PL/SQL block is executed.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to ((**ub4 ***) 0) for SQL scalar or array binds.

fmt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtl
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

See Also *obndra(), obndrn(), obndrv(), odefinps(), ogetpi(), osetpi().*

obndra

Purpose *obndra()* associates the address of a program variable or array with a placeholder in a SQL statement or PL/SQL block.

Syntax

```
obndra(Cda_Def *cursor, text *sqlvar, [sword sqlvl],
        ub1 *progv, sword progvl, sword ftype,
        <sword scale>, [sb2 *indp], [ub2 *alen],
        [ub2 *arcode], [ub4 maxsiz], [ub4 *cursiz],
        <text *fmt>, <sword fmtl>, <sword fmtt>);
```

Comments You can use *obndra()* to bind scalar variables or arrays in your program to placeholders in a SQL statement or a PL/SQL block. The *alen* parameter of the *obndra()* function allows you to change the size of the bound variable without actually rebinding the variable.

> **Note:** If *cursor* is a cursor variable that has been OPENed FOR in a PL/SQL block, then *obndra()* returns an error, unless a new SQL statement or PL/SQL block has been parsed on it.

When you bind arrays in your program to PL/SQL tables, you must use *obndra()*, because this function provides additional parameters that allow you to control the maximum size of the table and to retrieve the current table size after the block executes.

> **Note:** Applications running against a release 7.3 or later server that need to perform piecewise operations or utilize arrays of structures must use the newer *obindps()* routine instead of *obndra()*.

The *obndra()* function must be called after you call *oparse()* to parse the statement containing the PL/SQL block and before calling *oexn()* or *oexec()* to execute it.

Once you have bound a program variable, you can change the value in the variable (*progv)* and length of the variable (*progvl*) and re–execute the block without rebinding.

However, if you must change the type of the variable, you must reparse the statement or block and rebind the variable before re–executing.

The following short, but complete, example program shows how to use *obndra()* to bind arrays in a C program to tables in PL/SQL procedures.

```
#include <stdio.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>
```

```
Cda_Def cda;
Lda_Def lda;

/*  set up the table */
text *dt = (text *) ”DROP TABLE part_nos”;
text *ct = (text *) ”CREATE TABLE part_nos (partno NUMBER,
description\
            VARCHAR2(20))”;

text *cp = (text *) ”\
  CREATE OR REPLACE PACKAGE update_parts AS\n\
    TYPE part_number IS TABLE OF part_nos.partno%TYPE\n\
         INDEX BY BINARY_INTEGER;\n\
    TYPE part_description IS TABLE OF part_nos.description%TYPE\n\
         INDEX BY BINARY_INTEGER;\n\
    PROCEDURE add_parts (n            IN   INTEGER,\n\
                         descrip      IN   part_description,\n\
                         partno       IN   part_number);\n\
    END update_parts;”;

text *cb = (text *) ”\
  CREATE OR REPLACE PACKAGE BODY update_parts AS\n\
      PROCEDURE add_parts (n            IN   INTEGER,\n\
                           descrip      IN   part_description,\n\
                           partno       IN   part_number) is\n\
      BEGIN\n\
          FOR i IN 1..n LOOP\n\
              INSERT INTO part_nos\n\
                  VALUES (partno(i), descrip(i));\n\
          END LOOP;\n\
      END add_parts;\n\
  END update_parts;”;


#define DESC_LEN             20
#define MAX_TABLE_SIZE     1200

text *pl_sql_block = (text *) ”\
    BEGIN\n\
        update_parts.add_parts(3, :description, :partno);\n\
    END;”;

text  descrip[3][20] = {”Frammis”, ”Widget”, ”Thingie”};
sword numbers[]      = {12125,     23169,    12126};

ub2  descrip_alen[3] = {DESC_LEN, DESC_LEN, DESC_LEN};
ub2  descrip_rc[3];
ub4  descrip_cs = (ub4) 3;
ub2  descrip_indp[3];
```

```
ub2 num_alen[3] = {
    (ub2) sizeof (sword),
    (ub2) sizeof (sword),
    (ub2) sizeof (sword) };
ub2  num_rc[3];
ub4  num_cs = (ub4) 3;
ub2  num_indp[3];
ub1  hda[256];

main()
{
  printf(”Connecting to Oracle...”);
  if (olog(&lda, hda, ”scott/tiger”, –1, 0, –1, 0, –1,
           OCI_LM_DEF)) {
    printf(”Cannot logon as scott/tiger. Exiting...\n”);
    exit(1);
  }

  if (oopen(&cda, &lda, NULL, –1, –1, NULL, –1)) {
    printf(”Cannot open cursor, exiting...\n”);
    exit(1);
  }

  /*  Drop the table. */
  printf(”\nDropping table...”);
  if (oparse(&cda, dt, –1, 0, 2))
    if (cda.rc != 942)
      oci_error();

  printf(”\nCreating table...”);
  if (oparse(&cda, ct, –1, 0, 2))
    oci_error();

  /*  Parse and execute the create package statement. */
  printf(”\nCreating package...”);
  if (oparse(&cda, cp, –1, 0, 2))
    oci_error();
  if (oexec(&cda))
    oci_error();

  /*  Parse and execute the create package body statement. */
  printf(”\nCreating package body...”);
  if (oparse(&cda, cb, –1, 0, 2))
    oci_error();
  if (oexec(&cda))
    oci_error();
  /*  Parse the anonymous PL/SQL block that calls the
      stored procedure. */
  printf(”\nParsing PL/SQL block...”);
  if (oparse(&cda, pl_sql_block, –1, 0, 2))
    oci_error();

  /*  Bind the C arrays to the PL/SQL tables. */
  printf(”\nBinding arrays...”);
  if (obndra(&cda, (text *) ”:description”, –1, (ub1 *) descrip,
      DESC_LEN, VARCHAR2_TYPE, –1, descrip_indp, descrip_alen,
      descrip_rc, (ub4) MAX_TABLE_SIZE, &descrip_cs, (text *) 0,
      –1, –1))
    oci_error();

  if (obndra(&cda, (text *) ”:partno”, –1, (ub1 *) numbers,
      (sword) sizeof (sword), INT_TYPE, –1, num_indp,
      num_alen, num_rc, (ub4) MAX_TABLE_SIZE, &num_cs,
      (text *) 0, –1, –1))
    oci_error();

  printf(”\nExecuting block...”);
  if (oexec(&cda)) oci_error();
  printf(”\n”);
  if (oclose(&cda)) {
    printf(”Error closing cursor!\n”);
    return –1;
  }

  if (ologof(&lda)) {
    printf(”Error logging off!\n”);
    return –1;
  }
  exit(1);
}

oci_error()
{
  text msg[600];
  sword rv;

  rv = oerhms(&lda, cda.rc, msg, 600);
  printf(”\n\n%.*s”, rv, msg);
  printf(”Processing OCI function %s\n”, oci_func_tab[cda.fc]);
  if (oclose(&cda))
    printf(”Error closing cursor!\n”);
  if (ologof(&lda))
    printf(”Error logging off!\n”);
  exit(1);
}
```

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
sqlvar	text *	IN
sqlvl	sword	IN
progv (2)	ub1 * (1)	IN / OUT (3)
progvl	sword	IN
ftype	sword	IN
scale	sword	IN
indp (2)	sb2 *	IN / OUT (3)
alen (2)	ub2 *	IN / OUT
arcode (2)	ub2 *	OUT (4)
maxsiz	ub4	IN
cursiz	ub4 *	IN / OUT (3)
fmt	text *	IN
fmtl	sword	IN
fmtt	sword	IN

Note 1. *progv* is a pointer to the data buffer.

Note 2. If *maxsiz* > 1, must be an array with cardinality at least as great as *maxsiz*.

Note 3. IN / OUT parameter used or returned on the execute or fetch call.

Note 4. OUT parameter returned on the fetch call.

cursor
A pointer to the CDA associated with the SQL statement by the *oparse()* call.

sqlvar
Specifies the address of a character string containing the name of a placeholder (including the preceding colon) in the SQL statement.

sqlvl
The length of the character string *sqlvar*, including the preceding colon. For example, the placeholder :EMPLOYEE has a length of nine. If the placeholder name is a null–terminated character string (as in the example in this section), this parameter can be omitted (passed as –1).

progv
A pointer to a program variable or array of program variables from which input data will be retrieved or into which output data will be placed when *oexec(), oexn()*, or *oexfet()* is executed.

progvl
The length in bytes of the program variable or array element. Because *obndra()* might be called only once for many different *progv* values on successive execute calls, *progvl* must contain the maximum length of *progv.*

> **Note:** The datatype of *progvl* is **sword**. On some systems, this type might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this limits the maximum length of the buffer to 64K bytes. So, to bind a longer buffer for these datatypes, set *progvl* to –1, and pass the actual data area length (total buffer length – **sizeof (sb4)**) in the first four bytes of *progv.* Set this value before calling *obndra()***.**

ftype
The external datatype of the program variable in the user program. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. There is a list of external datatypes and type codes in the section "External Datatypes" on page 3 – 8.

scale
Only used for PACKED DECIMAL variables, which are not normally used in C. Set this parameter to –1. See the description of the OBNDRV routine on page 5 – 21 for information about this parameter.

indp
A pointer to an indicator variable, or array of indicator variables if *progv* is an array. As an array, *indp* must contain at least the same number of elements as *progv.*

See page 2 – 29 for more information about indicator variables.

alen
A pointer to an array of elements containing the length of the data. This is the effective length of the bind variable element, not the size of the array. For example, if the *progv* parameter is an array declared as

```
text  arr[5][20];
```

then *alen* should point to an array of at least five elements. The maximum usable size of the array is determined by the *maxsiz* parameter.

If *arr* in the above example is an IN parameter, each element in the array pointed to by *alen* should be set to the length of the data in the corresponding element in the *arr* array (<=20 in this example) before the execute call.

If *arr* in the above example is an OUT parameter, the length of the returned data appears in the array pointed to by *alen* after the PL/SQL block is executed.

Once the bind is done using *obndra()*, you can change the data length of the bind variable without rebinding. However, the length cannot be greater than that specified in *alen*.

arcode
An array containing the column–level error return codes. This parameter points to an array that will contain the error code for the bind variable after the execute call. The error codes that can be returned in *arcode* are those that indicate that data in *progv* has been truncated or that a null occurred on a SELECT or PL/SQL FETCH, for example, ORA–01405 or ORA–01406.

If *obndra()* binds an array of elements (that is, *maxsiz* is greater than one), then *arcode* must also point to an array of at least equal size.

maxsiz
The maximum size of an array being bound to a PL/SQL table. Values range from 1 to 32512, but the maximum size of the array depends on the datatype. The maximum array size is 32512 divided by the internal size of the datatype.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to ((**ub4**)0) for SQL scalar or array binds.

cursiz
A pointer to the actual number of elements in the array being bound to a PL/SQL table.

If *progv* is an IN parameter, set the *cursiz* parameter to the size of the array being bound. If *progv* is an OUT parameter, the number of valid elements being returned in the *progv* array is returned after PL/SQL block is executed.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to ((**ub4 ***) 0) for SQL scalar or array binds.

fmt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtl
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

See Also *obindps(), obndrv()*, *oexec()*, *oexn(), oparse()*.

obndrn obndrv

Purpose *obndrn()* and *obndrv()* associate the address of a program variable with the specified placeholder in the SQL statement. The placeholder is identified by name for the *obndrv()* function, and by number for *obndrn()*.

Syntax

```
obndrn(Cda_Def *cursor, sword sqlvn,
       ub1 *progv, sword progvl, sword ftype,
       <sword scale>, [sb2 *indp], <text *fmt>,
       <sword fmtl>, <sword fmtt>);
obndrv(Cda_Def *cursor, text *sqlvar,
       [sword sqlvl], ub1 *progv, sword progvl,
       sword ftype, <sword scale>, [sb2 *indp],
       <text *fmt>, <sword fmtl>, <sword fmtt>);
```

Comments You can call either *obndrv()* or *obndrn()* to bind the address of a variable in your program to a placeholder in the SQL statement being processed. If your application needs to perform piecewise operations or utilize arrays of structures, you must bind your variables using *obindps()* instead.

> **Note:** If *cursor* is a cursor variable that has been OPENed FOR in a PL/SQL block, then *obndrn()* or *obndra()* return an error, unless a new SQL statement or PL/SQL block has been parsed on it.

If you use *obndrv()*, the placeholder in the SQL statement consists of a colon (:) followed by a SQL identifier. The placeholder is *not* a program variable. For example, the SQL statement

```
SELECT ename,sal,comm FROM emp WHERE deptno = :Dept AND
    comm > :Min_com
```

has two placeholders, :Dept and :Min_com.

If you use *obndrn()*, the placeholders in the SQL statement consist of a colon followed by a literal integer in the range 1 to 255. The SQL statement

```
SELECT ename,sal,comm FROM emp WHERE deptno = :2 AND comm > :1
```

has two placeholders, :1 and :2.

An *obndrv()* call that binds the :Dept placeholder in the first SQL statement above to the program variable *dept_num* is

```
#define INT    3    /* external datatype code for integer */
Cda_Def cursor;
sword dept_num, minimum_comm;
...
obndrv(&cursor, ":Dept", –1, (ub1 *) &dept_num,
    (sword) sizeof(sword), INT, –1, (sb2*) 0, (text *) 0, –1, –1);
```

Because the literal ":Dept" is a null–terminated string, the *sqlvl* parameter is not needed; you pass it as –1. Some of the remaining parameters are optional. For example, *indp*, the pointer to an indicator variable, is optional and not used in this example. It is passed as 0 cast to an **sb2** pointer. *fmt* is not used, because the datatype is not packed decimal or display signed leading separate. Its absence is indicated by passing a null pointer.

If you use *obndrn()*, the parameter *sqlvn* identifies the placeholder by number. If *sqlvn* is set to 1, the program variable is bound to the placeholder :1. For example, *obndrn()* is called to bind the program variable *minimum_comm* to the placeholder :2 in the second SQL statement above as follows:

```
obndrn(&cursor, 2, (ub1 *) &dept_num, (sword) sizeof(sword),
       INT, –1, (sb2 *) 0, (text *) 0, –1, –1);
```

where the placeholder :2 is indicated in the *sqlvn* parameter by passing the value 2. The *sqlvn* parameter can be a variable and a literal.

You cannot use *obndrn()* in a PL/SQL block to bind program variables to placeholders, because PL/SQL does not recognize numbered placeholders. Always use *obndra()* (or *obndrv()*) and named placeholders within PL/SQL blocks.

The *obndrv()* or *obndrn()* function must be called after you call *oparse()* to parse the SQL statement and before calling *oexn()*, *oexec()*, or *oexfet()* to execute it. Once you have bound a program variable, you can change the value in the variable and re–execute the SQL statement without rebinding.

For example, if you have bound the address of *dept_num* to the placeholder ":Dept", and you now want to use *new_dept_num* (of the same datatype) when executing the SQL statement on page 4 – 22, you must call *obndrv()* again to bind the new program variable to the placeholder.

However, if you need to change the type or length of the variable, you must reparse and rebind before re–executing.

You should not use *obndrv()* and *obndrn() after* an *odescr()* call. If you do, you must first reparse and then rebind *all* variables.

At the time of the bind, Oracle stores the address of the program variable. If the same placeholder occurs more than once in the SQL statement, a single call to *obndrv()* or *obndrn()* binds all occurrences of the placeholder to the bind variable.

> **Note:** You can bind an array using *obndrv()* or *obndrn()*, but you must then specify the number of rows with either *oexn()*, *oexfet()*, or *ofen()*. This is the Oracle array interface.

The completion status of the bind is returned in the *return code* field of the CDA. A return code of zero indicates successful completion.

If your program is linked using the deferred mode option, bind errors that would be returned immediately in non–deferred mode are not detected until the bind operation is actually performed. This happens on the first describe (*odescr()*) or execute (*oexec()*, *oexn()*, or *oexfet()*) call after the bind.

Parameters

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN / OUT
sqlvar	text *	IN
sqlvl	sword	IN
sqlvn	sword	IN
progv	ub1 *	IN / OUT (1)
progvl	sword	IN
ftype	sword	IN
scale	sword	IN
indp	sb2 *	IN / OUT (1,2)
fmt	text *	IN
fmtl	sword	IN
fmtt	sword	IN

Note 1. Values are IN or IN / OUT parameters for *oexec(), oexn(),* or *oexfet()*.

Note 2. Can have the mode OUT when bound in a PL / SQL statement.

cursor
A pointer to the CDA associated with the SQL statement by the *oparse()* call.

sqlvar
Used only with *obndrv(),* this parameter specifies the address of a character string containing the name of a placeholder (including the preceding colon) in the SQL statement.

sqlvl
Used only with *obndrv(),* the *sqlvl* parameter is the length of the character string *sqlvar,* including the preceding colon. For example, the placeholder :Employee has a length of nine. If the placeholder name is a null–terminated character string, this parameter can be omitted (passed as –1).

sqlvn
Used only with *obndrn()***,** this parameter specifies a placeholder in the SQL statement referenced by the cursor by number. For example, if *sqlvn* is an integer literal or a variable equal to 2, it refers to all placeholders identified by :2 within the SQL statement.

progv
A pointer to a program variable or array variables. Values are input to Oracle when either *oexec()* or *oexn()* is executed. Data are retrieved when either *oexfet(), ofen(),* or *ofetch()* is performed.

progvl
The length in bytes of the program variable or array element. Since *obndrv()* or *obndrn()* might be called only once for many different *progv* values on successive execute or fetch calls, *progvl* must contain the maximum length of *progv.*

> **Note:** The datatype of *progvl* is **sword**. On some systems, this type might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this limits the maximum length of the buffer to 64K bytes. To bind a longer buffer for these datatypes, set *progvl* to –1 and pass the actual data area length (total buffer length – **sizeof (sb4)**) in the first four bytes of *progv.* Set this value before calling *obndrn()* or *obndrv().*

ftype
The Oracle external datatype of the program variable. Oracle converts the program variable between external and internal formats when the data is input to or retrieved from Oracle. See page 3 – 8 for a list of external datatypes.

scale
The scale parameter is valid only for PACKED DECIMAL variables, which are not normally used in C applications. Set this parameter to –1 to indicate that it is unused. See the description of the OBNDRV routineon page 5 – 21 for information about this parameter.

indp
A pointer to a short integer (or array of short integers) that serves as indicator variables.

On Input	If the indicator variable contains a negative value when the statement is executed, the corresponding column is set to null; otherwise, it is set to the value pointed to by *progv*.
On output	If the indicator variable contains a negative value after the fetch, the corresponding column contained a null.

fmt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtl
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

See Also *obindps(), obndra(), odescr(), oexec(), oexfet(), oexn(), oparse().*

obreak

Purpose *obreak()* performs an immediate (asynchronous) abort of any currently executing OCI function that is associated with the specified LDA. It is normally used to stop a long–running execute or fetch that has not completed.

Syntax

```
obreak(Lda_Def *lda);
```

Comments If no OCI function is active when *obreak()* is called, *obreak()* will be ignored unless the next OCI function called is a fetch. In this case, the subsequent fetch call will be aborted.

obreak() is the only OCI function that you can call when another OCI function is in progress. It should not be used when a connect operation (*olog()*) is in progress, because the LDA is in an indeterminate state. *obreak()* cannot return a reliable error status to the LDA, because it might be called when the Oracle internal status structures are in an inconsistent state.

> **Note:** *obreak()* aborts the currently executing OCI function not the connection.

obreak() is not guaranteed to work on all operating systems and does not work on all protocols. In some cases, *obreak()* may work with one protocol on an operating system, but may not work with other protocols on the same operating system.

Working with the OCI in non–blocking mode can provide a more consistent way of interrupting a SQL statement. See the section "Non–Blocking Mode" on page 2 – 32 for more information.

The following example shows how to use *obreak()* in an OCI program to interrupt a query if it does not complete in six seconds. This example works under many UNIX operating systems. The example must be linked two–task to work correctly.

```
#include <stdio.h>
#include <signal.h>
#include <ocidfn.h>
#include <ocidem.h>

Lda_Def lda;
Cda_Def cda;
ub1     hda[256];
```

```
/*  Define a new alarm function, to replace the standard
    alarm handler. */

sighandler()
{
  sword rv;

  fprintf(stderr, ”Alarm signal has been caught\n”);

  /* Call obreak() to interrupt the SQL statement in progress. */
  if (rv = obreak(&lda))
    fprintf(stderr, ”Error %d on obreak\n”, rv);
  else
    fprintf(stderr, ”obreak performed\n”);
}

err()
{
  text errmsg[512];
  sword n;

  n = oerhms(&lda, cda.rc, errmsg, sizeof (errmsg));
  fprintf(stderr, ”\n–Oracle error–\n%.*s”, n, errmsg);
  fprintf(stderr, ”while processing OCI function %s\n”,
          oci_func_tab[cda.fc]);
  oclose(&cda);
  ologof(&lda);
  exit(1);
}

main(argc, argv)
int argc;
char *argv[];
{
  void *old_sig;
  text name[10];

  /* Connect to Oracle. Program must be linked two–task,
     so connect using SQL*Net. */

  if (olog(&lda, hda, argv[1], –1, argv[2], –1,
           (text *) 0, –1, OCI_LM_DEF)) {
    printf(”cannot connect as %s\n”, argv[1]);
    exit(1);
  }
  if (oopen(&cda, &lda, 0, –1, –1, 0, –1)) {
    printf(”cannot open cursor data area\n”);
    exit(1);
  }
  signal(SIGALRM, sighandler);

  /* Parse a query statement. */
  if (oparse(&cda, ”select ename from emp”, –1, 0, 2))
    err();

  if (odefin(&cda, 1, name, sizeof (name), 1,
             –1, (sb2 *) 0, (text *) 0, 0, –1,
             (ub2 *) 0, (ub2 *) 0))
    err();

  if (oexec(&cda))
    err();

  /* Set the timeout */
  alarm(1);
  /* Begin the query. */
  for (;;) {
    if (ofetch(&cda)) {
      /* Break if no data found (should never happen,
         unless the alarm fails, or the emp table has
         less than 6 or so rows). */
      if (cda.rc == 1403) break;
      /* When the alarm is caught and obreak is performed,
         a 1013 error should be detected at this point. */
      err();
    }
    printf(”%10.10s\n”, name);
    /* Slow the query for the timeout. */
    sigpause();
  }
  fprintf(stderr, ”Unexpected termination.\n”);
  err();
}
```

Parameter

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	IN

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *olog()*.

ocan

Purpose

ocan() cancels a query after the desired number of rows have been fetched.

Syntax

```
ocan(Cda_Def *cursor);
```

Comments

ocan() informs Oracle that the operation in progress for the specified cursor is complete. The *ocan()* function thus frees any resources associated with the specified cursor, but keeps the cursor associated with its parsed representation in the shared SQL area.

For example, if you require only the first row of a multi–row query, you can call *ocan()* after the first *ofetch()* operation to inform Oracle that your program will not perform additional fetches.

If you use the *oexfet()* function to fetch your data, specifying a non–zero value for the *oexfet() cancel* parameter has the same effect as calling *ocan()* after the fetch completes.

Parameter

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN / OUT

cursor
A pointer to the cursor data area specified in the *oparse()* call associated with the query.

See Also

oexfet(), ofen(), ofetch(), oparse().

oclose

Purpose *oclose()* disconnects a cursor from the data areas in the Oracle Server with which it is associated.

Syntax

```
oclose(Cda_Def *cursor);
```

Comments The *oclose()* function frees all resources obtained by the *oopen()*, parse, execute, and fetch operations using the cursor. If *oclose()* fails, the *return code* field of the CDA contains the error code.

Parameter

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN/OUT

cursor
A pointer to the CDA specified in the associated *oopen()* call.

See Also *oopen()*, *oparse()*.

ocof

Purpose *ocof()* disables autocommit, that is, automatic commit of every SQL data manipulation statement.

Syntax

```
ocof(Lda_Def *lda);
```

Comments By default, autocommit is already disabled at the start of an OCI program. Turning on autocommit can have a serious impact on performance. So, if the *ocon()* (autocommit on) function enables autocommit for some special circumstance, use *ocof()* to disable autocommit as soon as it is practical.

If *ocof()* fails, the *return code* field of the LDA indicates the reason.

Parameter

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	IN / OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *ocom()*, *ocon()*, *olog()*.

ocom

Purpose *ocom()* commits the current transaction.

Syntax

```
ocom(Lda_Def *lda);
```

Comments The current transaction starts from the *olog()* call or the last *orol()* or *ocom()* call, and lasts until an *ocom()*, *orol()*, or *ologof()* call is issued.

If *ocom()* fails, the *return code* field of the LDA indicates the reason.

Do not confuse the *ocom()* call (COMMIT) with the *ocon()* call (turn autocommit on).

Parameter

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	IN/OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *ocon()*, *olog()*, *ologof()*, *orol()*.

ocon

Purpose

ocon() enables autocommit, that is, automatic commit of every SQL data manipulation statement.

Syntax

```
ocon(Lda_Def *lda);
```

Comments

By default, autocommit is disabled at the start of an OCI program. This is because it is more expensive and less flexible than placing *ocom()* calls after each logical transaction. When autocommit is on, a zero in the *return code* field after executing the SQL statement indicates that the transaction has been committed.

If *ocon()* fails, the *return code* field of the LDA indicates the reason

If it becomes necessary to turn autocommit on for some special circumstance, it is advisable to follow that with a call to *ocof()* to disable autocommit as soon as it is practical in order to maximize performance.

Do not confuse the *ocon()* function with the *ocom()* (COMMIT) function.

Parameter

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	IN / OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also

ocof(), *ocom()*, *olog()*.

odefin

Purpose *odefin()* defines an output variable for a specified select–list item of a SQL query.

Syntax

```
odefin(Cda_Def *cursor, sword pos, ub1 *buf,
       sword bufl, sword ftype, <sword scale>,
       [sb2 *indp], <text *fmt>, <sword fmtl>,
       <sword fmtt>, [ub2 *rlen],
       [ub2 *rcode]);
```

Comments An OCI program must call *odefin()* once for each select–list item in a SQL statement. Each call to *odefin()* associates an output variable in your program with a select–list item of the query. *odefin()* can define scalar or string program variables which are compatible with the external datatype (*ftype*). See Table 3–2 for a list of datatypes and compatible variables. The output variable may also be the address of an array of scalars or strings for use with the *oexfet()* and *ofen()* functions.

> **Note:** Applications running against a release 7.3 or later server that need to perform piecewise operations or utilize arrays of structures must use the newer *odefinps()* routine instead of *odefin()*.

Oracle places data in the output variables when the program calls *ofetch()*, *ofen()*, or *oexfet()*.

If you do not know the number of select–list items in the SQL statement, or the lengths and internal datatypes of the items, you can obtain this information at runtime using the *odescr()* function.

You can call *odefin()* only after you call *oparse()* to parse the SQL statement. You must also call *odefin()* before fetching the data.

odefin() associates output variables with select–list items using the position index of the select–list item in the SQL statement. Position indices start at 1 for the first (or leftmost) select–list item. For example, in the SQL statement

```
SELECT ename, empno, sal FROM emp WHERE sal > :min_sal
```

the select–list item SAL is in position 3, EMPNO is in position 2, and ENAME is in position 1.

If the type or length of bound variables changes between queries, you must reparse and rebind before re–executing.

You call *odefin()* to associate output buffers with the select–list items in the above statement as follows:

```
#define ENAME_LEN  20

Cda_Def cursor;       /* allocate a cursor  */
text employee_name[ENAME_LEN];
sword employee_number;
float salary;
sb2 ind_ename, ind_empno, ind_sal;
ub2 retc_ename, retc_empno, retc_sal;
ub2 retl_ename, retl_empno, retl_sal;
...
odefin(&cursor, 1, employee_name, ENAME_LEN, SQLT_STR,
    –1, &ind_ename, 0, –1, –1, &retl_ename, &retc_ename);
odefin(&cursor, 2, &employee_number, (int) sizeof(int), SQLT_INT,
    –1, &ind_empno, 0, –1, –1, &retl_empno, &retc_empno);
odefin(&cursor, 3, &salary, (int) sizeof(float), SQLT_FLT,
    –1, &ind_sal, 0, –1, –1, &retl_sal, &retc_sal);
```

Oracle provides return code information at the row level using the *return code* field in the CDA. If you require return code information at the column level, you must include the optional *rcode* parameter, as in the examples above. During each fetch, Oracle sets *rcode* for the select–list item processed. This return parameter contains Oracle error codes, and indicates either successful completion (zero) or an exceptional condition, such as "null item fetched", "item fetched was truncated", or other non–fatal column errors. The following codes are some of those that can be returned in the *rcode* parameter:

Code	***Meaning***
0	Success.
1405	A null was fetched.
1406	ASCII or string buffer data was truncated. The converted data from the database did not fit into the buffer. Check the value in *indp*, if specified, or *rlen* to determine the original length of the data.
1454	Invalid conversion specified: integers not of length 1, 2, or 4; reals not of length 4 or 8; invalid packed decimal conversions; packed decimal with more than 38 digits specified.
1456	Real overflow. Conversion of a database column or expression would overflow a floating–point number on this machine.
3115	Unsupported datatype.

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
pos	sword	IN
buf	ub1 *	IN (1)
bufl	sword	IN
ftype	sword	IN
scale	sword	IN
indp	sb2 *	IN (1)
fmt	text *	IN
fmtl	sword	IN
fmtt	sword	IN
rlen	ub2 *	IN (1)
rcode	ub2 *	IN (1)

Note 1. The *buffer*, *indp*, *retl*, and *rcode* parameters are OUT parameters for the *ofetch()*, *ofen()*, and *oexfet()* functions.

cursor
A pointer to the CDA specified in the associated *oparse()* call. This may be either a regular cursor or a cursor variable.

pos
An index for a select–list item in the query. Position indices start at 1 for the first (or leftmost) select–list item. The *odefin()* function uses the position index to associate output variables with a given select–list item. If you specify a position index greater than the number of items in the select–list, or less than 1, the behavior of *odefin()* is undefined.

If you do not know the number of items in the select–list, use the *odescr()* routine to determine it. See the second sample program in Appendix A for an example that does this.

buf
A pointer to the variable in the user program that receives the data when *ofetch()*, *ofen()*, or *oexfet()* executes. The variable can be of any type into which an Oracle column or expression result can be converted. See Chapter 3 for more information on datatype conversions.

> **Note:** If *odefin()* is being called to set up an array fetch operation using the *ofen()* or *oexfet()* functions, then the *buf* parameter must be the address of an array large enough to hold the set of items to be fetched.

bufl
The length in bytes of the variable being defined. If *buf* is an array, this is the size in bytes of one element of the array.

> **Note:** The datatype of *bufl* is **sword**. On some systems, this type might be only two bytes. When defining LONG VARCHAR and LONG VARRAW buffers, this appears to limit the maximum length of the buffer to 64K bytes. To define a longer buffer for these datatypes, set *bufl* to –1 and pass the actual data area length (total buffer length – **sizeof (sb4)**) in the first four bytes of *buf*. Set this value before calling *odefin()*.

ftype
The external datatype to which the select–list item is to be converted before it is moved to the output variable. A list of the external datatypes and datatype codes can be found in the "External Datatypes" section in Chapter 3.

scale
The scale of a packed decimal number. Not normally used in C.

indp
The *indp* value, after the fetch, indicates whether the select–list item fetched was null, truncated, or returned intact. See "Indicator Values" on page 2 – 29 for additional details.

If the output buffer size was too small to hold all of the data, the output was truncated. You can obtain the length of the data in the column using the expression

```
*(ub2 *) indp
```

If *oparse()* parses the SQL statement, and you do not define an indicator parameter for a column, a "fetched column value was truncated" error is returned for truncated select–list items.

> **Note:** If *odefin()* is being called to set up an array fetch operation using the *ofen()* or *oexfet()* functions, then the *indp* parameter must be the address of an array large enough to hold indicator variables for all the items that will be fetched.

The *indp* parameter offers only a subset of the functionality provided by the *rlen* and *rcode* parameters.

fmt
Not normally used in C. See the description of the ODEFIN routine in Chapter 5 for more information about packed decimal format specifiers.

fmtl
Not normally used in C. See the description of the ODEFIN routine in Chapter 5 for more information about packed decimal format specifiers.

fmtt
Not normally used in C. See the description of the ODEFIN routine in Chapter 5 for more information about packed decimal format specifiers.

rlen
A pointer to a **ub2** into which Oracle places the length of the data (plus length bytes, in the case of variable–length datatypes) after the fetch operation completes. If *odefin()* is being used to associate an array with a select–list item, the *rlen* parameter must also be an array of **ub2**s of the same size. Return lengths are valid after the *ofetch()*, *ofen()*, or *oexfet()* operation.

rcode
A pointer to an unsigned short integer that receives the column return code after the fetch. The error codes that can be returned in *rcode* are those that indicate that data in the column has been truncated or that a null occurred, for example, ORA–01405 or ORA–01406.

If *odefin()* is being used to associate an array with a select–list item, the *rcode* parameter must also be an array of **ub2**s of the same size**.**

See Also *odefinps()*, *odescr()***,** *oexfet()***,** *ofen()***,** *ofetch()***,** *oparse()*.

odefinps

Purpose *odefinps()* defines an output variable for a specified select–list item in a SQL query. This call can also specify if an operation will be performed piecewise or with arrays of structures.

Syntax

```
odefinps(Cda_Def *cursor, ub1 opcode, sword pos,
      ub1 *bufctx, sb4 bufl, sword ftype, <sword scale>,
     [sb2 *indp],<text *fmt>, <sb4 fmtl>, <sword fmtt>,
     [ub2 *rlenp], [ub2 *rcodep], sb4 buf_skip,
     sb4 ind_skip, sb4 len_skip, sb4 rc_skip);
```

Comments *odefinps()* is used to define an output variable for a specified select–list item in a SQL query. Additionally, it can indicate that an application will be fetching data incrementally at runtime. This piecewise fetch is designated in the *opcode* parameter. *odefinps()* is also used when an application will be fetching data into an array of structures.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

With the introduction of *odefinps()* there are now two fully–supported calls for binding input parameters, the other being the older *odefin()*. Application developers should consider the following points when determining which define call to use:

- *odefinps()* is supported only when a program is linked in deferred mode. If it is necessary to link in non–deferred mode, *odefin()* must be used. In this case, the ability to handle piecewise operations and arrays of structures is not supported.
- *odefinps()* is more complex than the older bind call. Users who are not performing piecewise operations and are not using arrays of structures may choose to use *odefin()*.

Unlike older OCI calls, *odefinps()* does not accept –1 for any optional or unused parameters. When it is necessary to pass a value to these parameters NULL or 0 should be used instead.

See the sections “Piecewise Insert, Update and Fetch,” and “Arrays of Structures” in Chapter 2 for more information about piecewise operations, arrays of structures, skip parameters and the *odefinps()* call.

The following sample code demonstrates the use of *odefinps()* in an OCI program which performs an insert from an array of structures. This code is provided for demonstration purposes only, and does not constitute a complete program. Most of the work in this program is done in the *array_fetch()* routine.

For sample code demonstrating an array insert, see the description of the *obindps()* routine earlier in this chapter. For sample code demonstrating the use of *odefinps()* for a piecewise fetch, see the description of the *osetpi()* routine later in this chapter.

```
...                                   /* OCI #include statements */

#define  DEFER_PARSE              1            /* oparse flags */
#define  NATIVE                   1
#define  VERSION_7                2
#define  NO_MORE_DATA             1403
#define  ARRAY_SIZE               10

#define OCI_EXIT_FAILURE 1                     /*   exit flags */
#define OCI_EXIT_SUCCESS 0

void   array_fetch();
void   print_results();

struct emp_record
{  int   empno;
   char  ename[11];
   char  job[11];
   int   mgr;
   char  hiredate[10];
   float sal;
   float comm;
   int   deptno;
};
typedef struct emp_record emp_record;

struct emp_record_indicators
{  short empno;
   short ename;
   short job;
   short mgr;
   short hiredate;
   short sal;
   short comm;
   short deptno;
};
typedef struct emp_record_indicators emp_record_indicators;
```

```
  Lda_Def   lda;                             /*       login area */
  ub1       hda[256];                        /*        host area */
  Cda_Def   cda;                             /*      cursor area */

main()
{
  ...                                        /* log on to oracle */
  array_fetch();
  ...                                       /* log off of oracle */
}

/* Function array_fetch(): This function retrieves EMP data     */
/*                    into an array of structs and prints them. */

void      array_fetch()
{
   emp_record            emp_records[20];
   emp_record_indicators emp_records_inds[20];
   int printed=0;
   int cont=1;
   int ret_val;
  text *sqlstmt = (text *) ”SELECT empno,ename,deptno \
                          FROM emp”;

  if (oopen(&cda, &lda, (text *)0, –1, –1, (text *)0, –1))
    exit(OCI_EXIT_FAILURE);

  if (oparse(&cda, sqlstmt, (sb4)–1, 0, (ub4)VERSION_7))
    exit(OCI_EXIT_FAILURE);

  if (odefinps(&cda, 1, 1, (ub1 *) &emp_records[0].empno,
               (ub4) sizeof(emp_records[0].empno), SQLT_INT, 0,
               (sb2 *) &emp_records_inds[0].empno, (text *)0, 0,
               0,(ub2 *) 0, (ub2 *) 0,
               (sb4) sizeof(emp_record), (sb4)
               sizeof(emp_record_indicators), 0, 0))
    exit(OCI_EXIT_FAILURE);

  if (odefinps(&cda, 1, 2, (ub1 *) emp_records[0].ename,
               (ub4) sizeof(emp_records[0].ename), SQLT_STR, 0,
               (sb2 *) &emp_records_inds[0].ename, (text *)0, 0,
               0,(ub2 *) 0, (ub2 *) 0,
               (sb4) sizeof(emp_record), (sb4)
               sizeof(emp_record_indicators), 0, 0))
    exit(OCI_EXIT_FAILURE);
  if (odefinps(&cda, 1, 3, (ub1 *) &emp_records[0].deptno,
               (ub4) sizeof(emp_records[0].deptno), SQLT_INT, 0,
               (sb2 *) &emp_records_inds[0].deptno, (text *)0, 0,
               0, (ub2 *) 0, (ub2 *) 0,
               (sb4) sizeof(emp_record), (sb4)
               sizeof(emp_record_indicators), 0, 0))
    exit(OCI_EXIT_FAILURE);

  oexec(&cda)
  while (cont)
  {
    printf("     Empno\tEname     \t    Deptno\n");
    printf("----------\t----------\t----------\n");
    ret_val=ofen(&cda,(sword) ARRAY_SIZE);
    switch (cda->rc)                  /* switch on return value */
    {
    case 0:
      print_results(emp_records,emp_records_inds,cda->rpc –
                    printed);
      printed=cda->rpc;
      break;
    case NO_MORE_DATA:
                                           /* print last batch? */
      if (cda->rpc > printed)
      {
         print_results(emp_records,emp_records_inds,cda->rpc –
                       printed);
         printed=cda->rpc;
      }
      cont=0;
      break;
    default:
      exit(OCI_EXIT_FAILURE);
    }
  }
  if (oclose(&cda))
    exit(OCI_EXIT_FAILURE);
}

void print_results(emp_records,emp_records_inds,n)
emp_record emp_records[];
emp_record_indicators emp_records_inds[];
int n;
{
   int i;
   for (i=0;i<n;i++)
   {
     if (emp_records_inds[i].empno == –1)
```

```
      printf(”%10.s\t”,””);
    else
      printf(”%10.d\t”, emp_records[i].empno);
    printf(”%–10.10s\t”, (emp_records_inds[i].ename ==–1 ? ”” :
        emp_records[i].ename));
    if (emp_records_inds[i].deptno== –1)
      printf(”%10.s\n”,””);
    else
      printf(”%10.d\n”, emp_records[i].deptno);
  }
}
```

Parameters

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN / OUT
opcode	ub1	IN
pos	sword	IN
bufctx	ub1 *	IN
bufl	sb4	IN
ftype	sword	IN
scale	sword	IN
indp	sb2 *	IN
fmt	text *	IN
fmtl	sb4	IN
fmtt	sword	IN
rlenp	ub2 *	OUT
rcodep	ub2 *	IN
buf_skip	sb4	IN
ind_skip	sb4	IN
len_skip	sb4	IN
rc_skip	sb4	IN

Note: Since the *odefinps()* call can be used in a variety of different circumstances, some items in the following list of parameter descriptions include different explanations for how the parameter is used for piecewise operations, arrays of structures and standard scalar or array binds.

Standard scalar and array defines are those defines which were previously possible using *odefin()*.

cursor
A pointer to the CDA associated with the SELECT statement being processed.

opcode
Piecewise define: pass as 0.

Arrays of structures or standard define: pass as 1.

pos
An index for the select–list column which needs to be defined. Position indices start from 1 for the first, or left–most, item of the query. The *odefinps()* function uses the position index to associate output variables with a given select–list item. If you specify a position index greater than the number of items in the select–list, or less than 1, the behavior of *odefinps()* is undefined.

If you do not know the number of items in the select list, use the *odescr()* routine to determine it. See the second sample program in Appendix A for an example that does this.

bufctx
Piecewise define: A pointer to a context block entirely private to the application. This should be used by the application to store any information about the column being defined. One possible use would be to store a pointer to a file which will be referenced later. Each output variable can then have its own separate file pointer. The pointer can be retrieved by the application during a call to *ogetpi()*.

Array of structures or standard define: This specifies a pointer to the program variable or the beginning of an array of program variables or structures into which the column being defined will be placed when the fetch is performed. This parameter is equivalent to the *buf* parameter of the *odefin()* call.

bufl
Piecewise define: The maximum possible size of the column being defined.

Array of structures or standard define: The length (in bytes) of the variable pointed to by *bufctx* into which the column being defined will be placed when a fetch is performed. For an array define, this should be the length of the first scalar element of the array of variables or structures pointed to by *bufctx*.

ftype
The external datatype to which the select–list item is to be converted before it is moved to the output variable. A list of the external datatypes and datatype codes can be found in the "External Datatypes" section in Chapter 3.

For piecewise operations, the valid datatype codes are 1 (VARCHAR2), 5 (STRING), 8 (LONG) and 24 (LONG RAW).

scale
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

indp
A pointer to an indicator variable or an array of indicator variables. If arrays of structures are used, this points to a possibly interleaved array of indicator variable structures.

fmt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtl
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

fmtt
Not normally used in C. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

rlenp
A pointer to an element or array of elements which will hold the length of a column or columns after a fetch is done. If arrays of structures are used, this points to a possibly interleaved array of length variable structures.

rcodep
A pointer to an element or array of elements which will hold column–level error codes which are returned by a fetch. If arrays of structures are used, this points to a possibly interleaved array of return code variable structures.

buf_skip
Piecewise define or standard scalar define: pass as 0.

Array of structures or standard array define: this is the skip parameter which specifies the number of bytes to be skipped in order to get to the next program variable element in the array being defined. In general, this will be the size of one program variable for a standard array define, or the size of one structure for an array of structures.

ind_skip
Piecewise define or standard scalar define: pass as 0.

Array of structures or standard array define: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next indicator variable in the possibly interleaved array of indicator variables pointed to by *indp*. In general, this will be the size of one indicator variable for a standard array define, and the size of one indicator variable structure for arrays of structures.

len_skip
Piecewise define or standard define: pass as 0.

Array of structures: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next column length in the possibly interleaved array of column lengths pointed to by *rlenp*. In general, this will be the size of one length variable for a standard array define, and the size of one length variable structure for arrays of structures.

rc_skip
Piecewise define or standard define: pass as 0.

Array of structures: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next return code structure in the possibly interleaved array of return codes pointed to by *rcodep*. In general, this will be the size of one return code variable for a standard array define, and the size of one length variable structure for arrays of structures.

See Also *obindps(), odefin(), ogetpi(), osetpi()*.

odescr

Purpose *odescr()* describes select–list items for SQL queries. The *odescr()* function returns internal datatype and size information for a specified select–list item.

Syntax

```
odescr(Cda_Def *cursor, sword pos,
       sb4 *dbsize, [sb2 *dbtype],
       [sb1 *cbuf], [sb4 *cbufl], [sb4 *dsize],
       [sb2 *prec], [sb2 *scale],
       [sb2 *nullok]);
```

Comments The *odescr()* function replaces the older *odsc()***.** You call *odescr()* after you have parsed the SQL statement (using *oparse()*) and after binding all input variables. *odescr()* obtains the following information about select–list items in a query:

- maximum size (*dbsize*)
- internal datatype code (*dbtype*)
- column name (*cbuf*)
- length of the column name (*cbufl*)
- maximum display size (*dsize*)
- precision of numeric items (*prec*)
- scale of numerics (*scale*)
- whether null values are permitted in the column (*nullok*)

A dependency exists between the results returned by a describe operation (*odescr()*) and a bind operation (*obindps(), obndra(), obndrn()* or *obndrv()*). Because a select–list item might contain bind variables, the type returned by *odescr()* can vary depending on the results of bind operations.

So, if you have placeholders for bind variables in a SELECT statement and you will use *odescr()* to obtain the size or datatype of select–list items, you should do the bind operation before the describe. If you need to rebind any input variables after performing a describe, you must reparse the SQL statement before rebinding.

> **Note:** Note that the rebind operation might change the results returned for a select–list item.

The *odescr()* function is particularly useful for dynamic SQL queries. That is, queries in which the number of select–list items, and their datatypes and sizes might not be known until runtime.

The *return code* field of the CDA indicates success (zero) or failure (non–zero) of the *odescr()* call.

The *odescr()* function uses a position index to refer to select–list items in the SQL query statement. For example, the SQL statement

```
SELECT ename, sal FROM emp WHERE sal > :Min_sal
```

contains two select–list items: ENAME and SAL. The position index of SAL is 2, and ENAME's index is 1.

The example program below is a complete C program that shows how you can describe select–list items. The program allows the user to enter SQL query statements at runtime, and prints out the name of each select–list item, the length of the name, and the datatype. See also the sample program *cdemo2.c* on page A – 27 for additional information on describing select lists.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>

#define  NPOS    13

Cda_Def cda;
Lda_Def lda;
ub1     hda[256];

main()
{
  text sql_statement[256];
  sword i, pos;
  text cbuf[NPOS][20];
  sb4 dbsize[NPOS], cbufl[NPOS], dsize[NPOS];
  sb2 dbtype[NPOS], prec[NPOS], scale[NPOS], nullok[NPOS];

  if (olog(&lda, hda, ”scott”, –1, ”tiger”, –1, 0, –1,
           OCI_LM_DEF)) {
    printf(”Cannot connect as scott. Exiting...\n”);
    exit(1);
  }
  if (oopen(&cda, &lda, 0, –1, –1, 0, –1)) {
    oci_error();
    exit(1);
  }
  for (;;) {
    printf(”\nEnter a query or \”exit\”> ”);
    gets(sql_statement);
    if (strncmp(sql_statement, ”exit”, 4) == 0) break;

    /*  parse the statement */
    if (oparse(&cda, sql_statement, –1, 0, 0)) {
      oci_error();
      continue;
    }
    for (pos = 1; pos <= NPOS; pos++) {
      cbufl[pos] = sizeof cbuf[pos];
      if (odescr(&cda, pos, &dbsize[pos], &dbtype[pos],
                 &cbuf[pos], &cbufl[pos], &dsize[pos],
                 &prec[pos], &scale[pos], &nullok[pos])) {
        if (cda.rc == 1007)
          break;
        oci_error();
        continue;
      }
    }

 /*  print out the total count and the names
     of the select–list items, column sizes, and datatype codes */
    pos––;
    printf(”\nThere were %d select–list items.\n”, pos);
    printf(”Item name                Length   Datatype\n”);
    printf(”\n”);
    for (i = 1; i <= pos; i++) {
      printf(”%*.*s”, cbufl[i], cbufl[i], cbuf[i]);
      printf(”%*c”, 25 – cbufl[i], ’ ’);
      printf(”%6d   %8d\n”, cbufl[i], dbtype[i]);
    }
  }
  oclose(&cda);
  ologof(&lda);
  exit(0);
}
oci_error()
{
  text msg[512];

  printf(”\nOracle ERROR\n”);
  oerhms(&lda, cda.rc, msg, (int) sizeof msg);
  printf(”%s”, msg);
  if (cda.fc != 0)
    printf(”processing OCI function %s\n”,
           oci_func_tab[cda.fc]);
}
```

Parameters

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN / OUT
pos	sword	IN
dbsize	sb4 *	OUT
dbtype	sb2 *	OUT
cbuf	sb1 *	OUT
cbufl	sb4 *	IN / OUT
dsize	sb4 *	OUT
prec	sb2 *	OUT
scale	sb2 *	OUT
nullok	sb2 *	OUT

cursor
A pointer to a CDA in the program. The *odescr()* function uses the cursor address to reference a specific SQL query statement that has been passed to Oracle by a prior *oparse()* call. This may be either a regular cursor or a cursor variable.

pos
The position index of the select–list item in the SQL query. Each item is referenced by position index, starting at one for the first (or leftmost) item. If you specify a position index greater than the number of items in the select–list or less than one, *odescr()* returns a "variable not in select–list" error in the *return code* field of the CDA.

dbsize
A pointer to a **signed long** that receives the maximum size of the column, as stored in the Oracle data dictionary. Values returned in *dbsize* are

Oracle Column Type	***Value***
CHAR, VARCHAR2, RAW	length of the column in the table
NUMBER	22 (the internal length)
DATE	7 (the internal length)
LONG, LONG RAW	0
ROWID	(system dependent)
Functions returning dataype 1 (such as TO_CHAR())	same as the *dsize* parameter

dbtype
Receives the internal datatype code of the select–list item. See Table 3 – 1 for a list of Oracle internal datatype codes. The datatype code returned for CHAR items (including literal strings in a select–list) can depend on how you parsed the SQL statement. If you used *oparse()* with the *lngflg* parameter set to 0, or *oparse()* with the *lngflg* parameter set to 1 when connected to a Version 6 database, CHAR items return the datatype code 1. Otherwise, *dbtype* returns 96.

The USER function in a select–list always returns the datatype code 1.

cbuf
Receives the name of the select–list item, that is, the name of the column or wording of the expression. The program must allocate a string long enough to receive the item name.

cbufl
Contains the length in bytes of *cbuf*. This parameter must be set before calling *odescr()*. If *cbufl* is not specified (that is, passed as 0), then the select–list item name is not returned. The name is truncated if it is longer than *cbufl*.

On return from *odescr()*, *cbufl* contains the length of the returned string in bytes.

dsize
Receives the maximum display size of the select–list item if the select–list item is returned as a character string. The *dsize* parameter is especially useful when functions, such as SUBSTR or TO_CHAR, are used to modify the representation of a column.

prec
Returns the precision of numeric select–list items. Precision is the total number of digits of a number. See "Internal Datatypes" on page 3 – 3 for additional information about precision and scale.

Pass this parameter as zero if you do not require the precision value.

scale
A pointer to a **short** that returns the scale of numeric select–list items. Pass this parameter as zero if you do not require the scale value.

For Version 6 of the RDBMS, *odescr()* returns the correct scale and precision of fixed–point numbers and returns precision and scale of zero for floating–point, as shown below:

SQL Datatype	***Precision***	***Scale***
NUMBER(P)	P	0
NUMBER(P,S)	P	S
NUMBER	0	0
FLOAT(N)	0	0

For Oracle7, the SQL types REAL, DOUBLE PRECISION, FLOAT, and FLOAT(N) return the correct precision and a scale of –127.

nullok
A pointer to a **short** that returns zero if null values are not permitted for the column, and non–zero if nulls are permitted.

Pass this parameter as zero if you do require the null status of the select–list item.

See Also *obindps(), obndra(), obndrn(), obndrv(), odefin()***,** *odefinps()***,** *oparse()*.

odessp

Purpose *odessp()* is used to describe the parameters of a PL/SQL procedure or function stored in an Oracle database.

Syntax

```
odessp(Lda_Def *lda, text *objnam,
       size_t onlen, ub1 *rsv1, size_t rsv1ln,
       ub1 *rsv2, size_t rsv2ln, ub2 *ovrld, ub2 *pos,
       ub2 *level, text **argnm, ub2 *arnlen,
       ub2 *dtype, ub1 *defsup, ub1 *mode, ub4 *dtsiz,
       sb2 *prec, sb2 *scale, ub1 *radix, ub4 *spare,
       ub4 *arrsiz)
```

Comments You call *odessp()* to get the properties of a stored procedure (or function) and the properties of its parameters. When you call *odessp()*, pass to it:

- A valid LDA for a connection that has execute privileges on the procedure.
- The name of the procedure, optionally including the package name. The package body does not have to exist, as long as the procedure is specified in the package.
- The total length of the procedure name, or –1 if it is null terminated.

If the procedure exists and the connection specified in the *lda* parameter has permission to execute the procedure, *odessp()* returns information about each parameter of the procedure in a set of array parameters. It also returns information about the return type if it is a function.

odessp() returns the same information for a parameter of type cursor variable as for a regular cursor.

Your OCI program must allocate the arrays for all parameters of *odessp()*, and you must pass a parameter (*arrsiz*) that indicates the size of the arrays (or the size of the smallest array if they are not equal). The *arrsiz* parameter returns the number of elements of each array that was returned by *odessp()*.

odessp() returns a non–zero value if an error occurred. The error number is in the *return code* field of the LDA. The following errors can be returned there:

–20000	The object named in the *objnam* parameter is a package, not a procedure or function.
–20001	The procedure or function named in *objnam* does not exist in the named package.

–20002	A database link was specified in *objnam*, either explicitly or by means of a synonym.
ORA–0xxxx	An Oracle code, usually indicating a syntax error in the procedure specification in *objnam*.

When *odessp()* returns successfully, the OUT array parameters contain the descriptive information about the procedure or function parameters, and the return type for a function. As an example, consider a package EMP_RECS in the SCOTT schema. The package contains two stored procedures and a stored function, all named GET_SAL_INFO. Here is the package specification:

```
create or replace package EMP_RECS as
procedure get_sal_info (
    name    in   emp.ename%type,
    salary  out  emp.sal%type);
procedure get_sal_info (
     ID_num  in   emp.empno%type,
     salary  out  emp.sal%type);
function get_sal_info (
    name    in   emp.ename%type) return emp.sal%type;
end EMP_RECS;
```

A code fragment to describe these procedures and functions follows:

```
#include <stdio.h>
#include <ocidfn.h>
#include <ocidem.h>
#define ASIZE   50
Lda_Def lda;
Cda_Def cda;
ub1 hda[256];

text *objnam = (text *) ”scott.emp_recs.get_sal_info”;

ub2   ovrld[ASIZE];
ub2   pos[ASIZE];
ub2   level[ASIZE];
text  argnm[ASIZE][30];
ub2   arnlen[ASIZE];
ub2   dtype[ASIZE];
ub1   defsup[ASIZE];
ub1   mode[ASIZE];
ub4   dtsize[ASIZE];
sb2   prec[ASIZE];
sb2   scale[ASIZE];
ub1   radix[ASIZE];
ub4   spare[ASIZE];
ub4   arrsiz = (ub4) ASIZE;
```

```
main() {
  int i, rv;
  if (olog(&lda, hda, (text *) ”scott”, –1,  (text *) ”tiger”, –1,
           0, –1, OCI_LM_DEF)) {
    printf(”cannot connect as scott\n”);
    exit(1);
  }
  printf(”connected\n”);
  /* call the describe function */
  rv = odessp(&lda, objnam, –1, (ub1 *) 0, 0, (ub1 *) 0, 0,
              ovrld, pos, level, argnm, arnlen, dtype,
              defsup, mode, dtsize, prec, scale, radix,
              spare, &arrsiz);
  if (rv != 0)
  {
    printf(”error in odessp %d\n”, lda.rc);
  }
/* print out the returned values */
  printf(”\nArrsiz = %ld\n”, arrsiz);
  if (arrsiz > ASIZE)
    arrsiz = ASIZE;
  printf(”ProcName O’load Level Pos Datatype”);
  printf(” Mode Dtsize Prec Scale Radix\n”);
  printf(”––––––––––––––––––––––––––––––––––––”);
  printf(”––––––––––––––––––––––––––––––\n”);
  for (i = 0; i < arrsiz; i++)
  {
    printf(”%8.8s %6d %5d %3d %8d %4d %6d %4d %5d %5d\n”,
           argnm[i], ovrld[i], level[i], pos[i],
           dtype[i], mode[i], dtsize[i], prec[i], scale[i],
           radix[i]);
  }
  exit(0);
}
```

When this call to *odessp()* completes, the return parameter arrays are filled in as shown in Table 4 – 1. The *arrsiz* parameter returns 6, as there were a total of 5 parameters and one function return type described.

	ARRAY ELEMENT					
PARAMETER	**0**	**1**	**2**	**3**	**4**	**5**
ovrld	1	1	2	2	3	3
pos	1	2	1	2	0	1
level	1	1	1	1	1	1
argnm	name	salary	ID_num	salary	NULL	name
arnlen	4	6	6	6	0	4
dtype	1	2	2	2	2	1
defsup	0	0	0	0	0	0
mode	0	1	0	1	1	0
dtsize	10	22	22	22	22	10
prec		7	4	7	7	
scale		2	0	2	2	
radix		10	10	10	10	
spare (1)	n/a	n/a	n/a	n/a	n/a	n/a
Note 1: Reserved by Oracle for future use.						

Table 4 – 1 Return Values from *odessp()* **Call**

Parameters

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	IN / OUT
objnam	text *	IN
onlen	size_t	IN
rsv1	ub1 *	IN
rsv1ln	size_t	IN
rsv2	ub1 *	IN
rsv2ln	size_t	IN
ovrld	ub2 *	OUT
pos	ub2 *	OUT
level	ub2 *	OUT
argnm	text **	OUT
arnlen	ub2 *	OUT
dtype	ub2 *	OUT
defsup	ub1 *	OUT

Parameter Name	*Type*	*Mode*
mode	ub1 *	OUT
dtsiz	ub4 *	OUT
prec	sb2 *	OUT
scale	sb2 *	OUT
radix	ub1 *	OUT
spare	ub4 *	OUT
arrsiz	ub4 *	IN / OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

objnam
The name of the procedure or function, including optional schema and package name. Quoted names are accepted. Synonyms are also accepted and are translated. Multi–byte characters can be used. The string can be null terminated. If it is not, the actual length in bytes must be passed in the *onlen* parameter.

onlen
The length in bytes of the *objnam* parameter. If *objnam* is a null–terminated string, pass *onlen* as –1;. Otherwise, pass the exact length.

rsv1
Reserved by Oracle for future use.

rsv1ln
Reserved by Oracle for future use.

rsv2
Reserved by Oracle for future use.

rsv2ln
Reserved by Oracle for future use.

ovrld
An array indicating whether the procedure is overloaded. If the procedure (or function) is not overloaded, 0 is returned. Overloaded procedures return *1...n* for *n* overloadings of the name.

pos
An array returning the parameter positions in the parameter list of the procedure. The first, or left–most, parameter in the list is position 1. When *pos* returns a 0, this indicates that a function return type is being described.

level
For scalar parameters, *level* returns 0. For a record parameter, 0 is returned for the record itself, then for each parameter in the record the parameter's level in the record is indicated, starting from 1, in successive elements of the returned value of *level*.

For array parameters, 0 is returned for the array itself. The next element in the return array is at level 1 and describes the element type of the array.

For example, for a procedure that contains three scalar parameters, an array of ten elements, and one record containing three scalar parameters at the same level, you need to pass *odessp()* arrays with a minimum dimension of nine: three elements for the scalars, two for the array, and four for the record parameter.

argnm
A pointer to an array of strings that returns the name of each parameter in the procedure or function. The strings are not null terminated. Each string in the array must be exactly 30 characters long.

arnlen
The length in bytes of each corresponding parameter name in *argnm*.

dtype
The Oracle datatype code for each parameter. See the *PL/SQL User's Guide and Reference* for a list of the PL/SQL datatypes. Numeric types, such as FLOAT, INTEGER, and REAL return a code of 2. VARCHAR2 returns 1. CHAR returns 96. Other datatype codes are shown in Table 3 – 5 in Chapter 3.

> **Note:** A *dtype* value of 0 indicates that the procedure being described has no parameters.

defsup
This parameter indicates whether the corresponding parameter has a default value. Zero returned indicates no default. One indicates that a default value was supplied in the procedure or function specification.

mode
This parameter indicates the mode of the corresponding parameter. Zero indicates an IN parameter, one an OUT parameter, and two an IN/OUT parameter.

dtsiz
The size of the datatype in bytes. Character datatypes return the size of the parameter. For example, the EMP table contains a column ENAME. If a parameter in a procedure is of the type EMP.ENAME%TYPE, the value 10 is returned for this parameter, because that is the length of the ENAME column in a single–byte character set.

For number types, 22 is returned. See the description of the *dbsize* parameter under *odescr()* on page 4 – 48 for more information.

prec
This parameter indicates the precision of the corresponding parameter if the parameter is numeric; otherwise, it returns zero.

scale
This parameter indicates the scale of the corresponding parameter if the parameter is numeric.

radix
This parameter indicates the radix of the corresponding parameter if it is numeric.

spare
Reserved by Oracle for future use.

arrsiz
When you call *odessp()*, pass the length of the arrays of the OUT parameters. If the arrays are not of equal length, you must pass the length of the shortest array. When *odessp()* returns, *arrsiz* returns the number of array elements filled in.

See Also *odescr()*.

oerhms

Purpose *oerhms()* returns the text of an Oracle error message, given the error code *rcode*.

Syntax

```
oerhms(Lda_Def *lda, sb2 rcode,
       text *buf, sword bufsiz);
```

Comments When you call *oerhms()*, pass the address of the LDA for the active connection as the first parameter. This is required to retrieve error messages that are correct for the database version being used on that connection.

The *oerhms()* function does not return zero when it completes successfully. It returns the number of characters in *buf*. The error message text in *buf* is null terminated.

When using *oerhms()* to return error messages from PL/SQL blocks (where the error code is between 6550 and 6599), be sure to allocate a large *buf*, because several messages can be returned. The maximum length of an Oracle error message is 512 characters.

For more information about the causes of Oracle errors and possible solutions, see the *Oracle7 Server Messages* manual.

The following example shows how to obtain an error message from a specific Oracle instance:

```
Lda_Def lda[2];  /* two separate connections in effect */
Cda_Def cda;
sword n_chars;
text msgbuf[512];
...
/*  when an error occurs on the second connection */
n_chars = oerhms(&lda[1], cda.rc, msgbuf, (int) sizeof(msgbuf));
```

Parameters

Parameter Name	***Type***	***Mode***
lda	Lda_Def *	IN/OUT
rcode	sb2	IN
buf	text *	OUT
bufsiz	sword	IN

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

rcode
The LDA or CDA return code containing an Oracle error number.

buf
A pointer to a buffer that receives the error message text. The message text is null terminated.

bufsiz
The size of the buffer in bytes. The maximum size of the buffer is essentially unlimited. However, values larger than 1000 bytes are not normally needed.

See Also *oermsg()*, *olog()*.

oexec

Purpose *oexec()* executes the SQL statement associated with a cursor.

Syntax

```
oexec(Cda_Def *cursor);
```

Comments Before calling *oexec()***,** you must call *oparse()* to parse the SQL statement, and this call must complete successfully. If the SQL statement contains placeholders for bind variables, you must call *obndrv(), obindps(), obndra()* or *obndrn()* to bind each placeholder to the address of a program variable before calling *oexec()*.

For queries, after *oexec()* is called, the program must explicitly request rows of the result set using *ofen()* or *ofetch()*.

For UPDATE, DELETE, and INSERT statements*, oexec()* executes the entire SQL statement and sets the *return code* field and the *rows processed count* field in the CDA. Note that an UPDATE or DELETE that does not affect any rows (no rows match the WHERE clause) returns success in the *return code* field and zero in the *rows processed count* field.

> **Note:** If *cursor* is a cursor variable that has been OPENed FOR in a PL/SQL block, then *oexec()* returns an error unless *oparse()* has been called for *cursor* with another SQL statement or PL/SQL block.

DML statements (e.g., UPDATE, INSERT) are executed when a call is made to *oexec(), oexn()* or *oexfet()*. DDL statements (e.g., CREATE TABLE, REVOKE) are executed on the parse if you have linked in non–deferred mode or if you have liked with the deferred option and the *defflg* parameter of *oparse()* is zero. If you have linked in deferred mode and the *defflg* parameter is non–zero, you must call *oexn()* or *oexec()* to execute the statement.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when *oparse()* is called in non–deferred mode are not detected until the first non–deferred call is made (usually an execute or describe call).

> **Note:** It is possible to use the *oexn()* function in place of *oexec()* by binding scalar variables, not arrays, and setting the *count* parameter to 1. For queries, use *oexfet()* in preference to *oexec()* followed by *ofen()*.

See the description of the *ofetch()* routine on page 4 – 74 for an example that shows how to use *oexec()*.

Parameter

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT

cursor
A pointer to the CDA specified in the associated *oparse()* call.

See Also *obindps(), obndra(), obndrn(), obndrv(), oexfet(), oexn(), oparse().*

oexfet

Purpose

oexfet() executes the SQL statement associated with a cursor, then fetches one or more rows. *oexfet()* can also perform a cancel of the cursor (the same as an *ocan()* call).

Syntax

```
oexfet(Cda_Def *cursor, ub4 nrows,
       sword cancel, sword exact);
```

Comments

Before calling *oexfet()***,** the OCI program must first call *oparse()* to parse the SQL statement, call *obndra()*, *obndrn()*, *obndrv()* or *obindps()* (if necessary) to bind input variables, then call *odefin()* or *odefinps()* to define output variables.

If the OCI program was linked using the deferred mode link option, the bind and define steps are deferred until *oexfet()* is called. If *oparse()* was called with the deferred parse flag (*defflg*) parameter non–zero, the parse step is also delayed until *oexfet()* is called. This means that your program can complete the processing of a SQL statement using a minimum of message round–trips between the client running the OCI program and the database server.

If you call *oexfet()* for a DML statement that is not a query, Oracle issues the error

```
ORA–01002:  fetch out of sequence
```

and the execute operation fails.

> **Note:** Using the deferred parse, bind, and define capabilities when processing a SQL statement requires more memory on the client system than the non–deferred sequence. So, you gain execution speed at the cost of some additional space.

When running against an Oracle7 database, where the SQL statement was parsed using *oparse()* with the *lngflg* parameter set to 1 or 2, a character string that is too large for its associated buffer is truncated. The column return code (*rcode*) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the *oexfet()* call does not return an error indication. If a null is encountered for a select–list item, the associated column return code (*rcode*) for that column is set to the error

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

and the indicator parameter is set to –1. The *oexfet()* call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, *oexfet()* does return an ORA–01405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

oexfet() both executes the statement and fetches the row or rows that satisfy the query. If you need to fetch additional rows after *oexfet()* completes, use the *ofen()* function. The following example shows how you can use deferred parse, bind, and define operations together with *oexfet()* to process a SQL statement:

```
Cda_Def cda;
Lda_Def lda;
text *sql_statement =
    “SELECT ename, sal FROM emp WHERE deptno = :1”;
float salaries[12000];
text names[12000][20];
sb2 sal_ind[12000], name_ind[12000];
char* dept_number_stg[10];
sword dept_number;
...
/* after connecting to Oracle ... */
oopen(&cda, &lda, 0, –1, –1, 0, –1);
oparse(&cda, sql_statement, –1, 1, 1);   /* deferred parse*/
printf(“Enter department number: ”);
gets(dept_number_stg);
dept_number = (sword) atoi(dept_number_stg);
...
obndrn(&cda, 1, &dept_number, (int) sizeof (int), 3, –1,
       0, 0, –1, –1);
odefin(&cda, 2, salaries, (int) sizeof (float),
       4, –1, sal_ind, 0, –1, –1); /* datatype FLOAT is 4 */
odefin(&cda, 1, names, 20, 1, –1, name_ind, 0, –1, –1);
/* retrieve 12000 or fewer salaries */
oexfet(&cda, 12000, 0, 0);  /* cancel and exact not set */
....
```

The number of rows that were fetched is returned in the *rows processed count* field of the CDA.

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
nrows	ub4	IN
cancel	sword	IN
exact	sword	IN

cursor
A pointer to a CDA specified in the associated *oparse()* call.

nrows
The number of rows to fetch. If *nrows* is greater than 1, you must define arrays to receive select–list values, as well as any indicator variables. See the description of *odefin()* on page 4 – 35 for more information.

If *nrows* is greater than the number of rows that satisfy the query, the *rows processed count* field in the CDA is set to the number of rows returned, and Oracle returns the error

```
ORA–01403:  no data found
```

Note: That the data is actually fetched.

cancel
If this parameter is non–zero when *oexfet()* is called, the cursor is canceled after the fetch completes. This has exactly the effect of issuing an *ocan()* call, but does not require the additional call overhead.

exact
If this parameter is non–zero when *oexfet()* is called, *oexfet()* returns an error if the number of rows that satisfy the query is not exactly the same as the number specified in the *nrows* parameter. Nevertheless, the rows are returned.

If the number of rows returned by the query is less than the number specified in the *nrows* parameter, Oracle returns the error

```
ORA–01403:  no data found
```

If the number of rows returned by the query is greater than the number specified in the *nrows* parameter, Oracle returns the error

```
ORA–01422:  Exact fetch returns more than requested
            number of rows
```

Note: If *exact* is non–zero, a cancel of the cursor is always performed, regardless of the setting of the *cancel* parameter.

See Also *obindps(), obndra(), obndrn(), obndrv(), odefin(), odefinps(), ofen(), oparse().*

Purpose *oexn()* executes a SQL statement. Array variables can be used to input multiple rows of data in one call.

Syntax `oexn(Cda_Def *cursor, sword iters, sword rowoff);`

Comments *oexn()* is similar to the older *oexec()*, but it allows you to take advantage of the Oracle array interface. *oexn()* allows operations using arrays of bind variables. *oexn()* is generally much faster than successive calls to *oexec()*, especially in a networked client–server environment.

> **Note:** If *cursor* is a cursor variable that has been OPENed FOR in a PL / SQL block, then *oexn()* returns an error, unless *oparse()* has been called on *cursor* with another SQL statement or PL / SQL block.

Variables are bound to placeholders in the SQL statement using *obndra()*, *obndrv()*, *obndrn()* or *obindps()***.** A pointer to the scalar or array is passed to the binding function. Data must be present in bind variables before you call *oexn()***.** For example:

```
Cda_Def cursor;
text names[10][20];        /* 2–dimensional array of char */
sword emp_nos[10];             /* an array of 10 integers */
sb2 ind_params[10];      /* array of indicator parameters */
text *sql_stmt = “INSERT INTO emp(ename, empno) VALUES \
                    (:N, :E)”;
...
/* parse the statement */
oparse(&cursor, sql_stmt, –1, 1, 1);    /* deferred parse */
/* bind the arrays to the placeholders */
obndrv(&cursor, “:N”, –1, names, 20, SQLT_CHR,
       –1, ind_params, 0, –1, –1);
/* empno is non–null, so indicator parameters are not used */
obndrv(&cursor, “:E”, –1, emp_nos, (int) sizeof(int),
       SQLT_INT, –1, 0, 0, –1, –1);
/* fill in the data and indicator parameters, then
   execute the statement, inserting the array values */
oexn(&cursor, 10, 0);
```

This example declares three arrays, one of ten integers, one of ten indicators, and one of ten 20–character strings. It also defines a SQL statement that inserts multiple rows into the database. After binding the arrays, the program must place data for the first INSERT in *names[0]* and e*mp_nos[0]*, for the second INSERT in *names[1]* and emp_nos[1], and so forth. (This step is not shown in the example.) Then *oexn()* is called to insert the data in the arrays into the EMP table.

The completion status of *oexn()* is indicated in the *return code* field of the CDA. The *rows processed count* in the CDA indicates the number of rows successfully processed. If the *rows processed count* is not equal to *iters*, the operation failed on array element *rows processed count* + 1.

You can continue to process the rest of the array even after a failure on one of the array elements as long as a rollback did not occur (obtained from the *flags1* field in the CDA). You do this by using *rowoff* to start operations at an array element other than the first.

In the above example, if the *rows processed count* was 5 at completion of *oexn()*, then row six was rejected. In this event, to continue the operation at row seven, call *oexn()* again as follows:

```
oexn(&cursor, 10, 6);
```

Note: The maximum number of elements in an array is 32767.

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
iters	sword	IN
rowoff	sword	IN

cursor
A pointer to the CDA specified in the associated *oparse()* call.

iters
The total size of the array of bind variables to be inserted. The size cannot be greater than 32767 items.

rowoff
The zero–based offset within the bind variable array at which to begin operations. *oexn()* processes (*iters – rowoff*) array elements if no error occurs.

See Also *oexec()*, *oexfet()*.

ofen

Purpose *ofen()* fetches one or multiple rows into arrays of variables, taking advantage of the Oracle array interface.

Syntax

```
ofen(Cda_Def *cursor, sword nrows);
```

Comments *ofen()* is similar to *ofetch()*; however, *ofen()* can fetch multiple rows into an array of variables with a single call. A pointer to the array is bound to a select–list item in the SQL query statement using *odefin()*.

When running against an Oracle7 database, where the SQL statement was parsed using *oparse()* with the *lngflg* parameter set to 1 or 2, a character string that is too large for its associated buffer is truncated, the column return code (*rcode*) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the *ofen()* call does not return an error indication. If a null is encountered for a select–list item, the associated column return code (*rcode*) for that column is set to the error

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

and the indicator parameter is set to –1. The *ofen()* call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, *ofen()* does return the 1405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

Even when fetching a single row, Oracle recommends that Oracle7 OCI programs use *oexfet()***,** with the *nrows* parameter set to 1, instead of the combination of *oexec()* and *ofen()*. Use *ofen()* after *oexfet()* to fetch additional rows when you do not know in advance the exact number of rows that a query returns.

The following example is a complete OCI program that shows how *ofen()* can be used to extract multiple rows using the array interface.

```
#include <stdio.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>
#define MAX_NAME_LENGTH 30

Lda_Def lda;
ub1     hda[256];
Cda_Def  cda;

main()
{
  static sb2      ind_a[10];
  static sword    empno[10];
  static text     names[10][MAX_NAME_LENGTH];
         ub2      rl[10], rc[10];
         sword    i, n, rows_done;

  /* connect to Oracle */
  if (olog(&lda, hda, ”scott/tiger”,
           –1, 0, –1, 0, –1, 0, OCI_LM_DEF)) {
    printf(”cannot connect to Oracle as scott/tiger\n”);
    exit(1);
  }

  /* open one cursor */
  if (oopen(&cda, &lda, 0, –1, –1, 0, –1)) {
    printf(”cannot open the cursor\n”);
    ologof(&lda);
    exit(1);
  }

  /* parse a query */
  if (oparse(&cda, ”select ename, empno from emp”, –1, 1, 2)) {
    oci_error(&cda);
    exit(1);
  }

  /* define the output variables */
  if (odefin(&cda, 1, names, MAX_NAME_LENGTH, 5, –1,
             ind_a, 0, –1, –1, rl, rc)) {
    oci_error(&cda);
    exit(1);
  }
  if (odefin(&cda, 2, empno, (int) sizeof (int), 3, –1,
             0, 0, –1, –1, 0, 0)) {
    oci_error(&cda);
    exit(1);
  }
  /* execute the SQL statement */
  if (oexec(&cda)) {
    oci_error(&cda);
    exit(1);
  }
  /* use ofen to fetch the rows, 10 at a time,
     and then display the results */
  for (rows_done = 0;;) {
    if (ofen(&cda, 10))
      if (cda.rc != 1403) {
        oci_error(&cda);    /* some error */
        exit(1);
      }
    /* the rpc is cumulative, so find out how many
       rows to display this time (always <= 10) */
    n = cda.rpc – rows_done;
    rows_done += n;
    for (i = 0; i < n; i++) {
      if (ind_a[i])
        printf(”%s    ”, ”(null)”);
      else
        printf(”%s%*c”, names[i],
               MAX_NAME_LENGTH – rl[i], ’ ’);
      printf(”%10d\n”, empno[i]);
    }
    if (cda.rc == 1403) break;  /* no more rows */
  }
  printf(”%d rows returned\n”, cda.rpc);
  if (oclose(&cda))
    exit(1);
  if (ologof(&lda))
    exit(1);
  exit(0);
}

oci_error(cda)
Cda_Def *cda;
{
  static text msg[512];
  sword len;

  len = oerhms(&lda, cda–>rc, msg, (int) sizeof (msg));
  printf(”\nOracle ERROR\n”);
  printf(”%.*s\n”, len, msg);
  printf(”Processing OCI function %s\n”,
         oci_func_tab[cda–>rc]);
  return 0;
}
```

The *return code* field of the CDA indicates the completion status of *ofen()*. The *rows processed count* field in the CDA indicates the cumulative number of rows successfully fetched. If the rows processed count increases by *nrows*, *ofen()* may be called again to get the next batch of rows. If the rows processed count does not increase by *nrows*, then an error, such as "no data found", has occurred.

Parameters

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN / OUT
nrows	sword	IN

cursor
A pointer to the CDA associated with the SQL statement by the *oparse()* call used to parse it.

nrows
The size of the defined variable array on which to operate. The size cannot be greater than 32767 items. If the size is one, then *ofen()* acts effectively just like *ofetch()*.

See Also *odefin()*, *odefinps()*, *oexfet()*, *ofetch()*, *oparse()*.

ofetch

Purpose *ofetch()* returns rows of a query to the program, one row at a time.

Syntax

```
ofetch(Cda_Def *cursor);
```

Comments Each select–list item of the query is placed into a buffer identified by a previous *odefin()* call. When running against Oracle7, where the SQL statement was parsed using *oparse()* with the *lngflg* parameter set to 1 or 2, a character string that is too large for its associated buffer is truncated, the column return code (rcode) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the *ofetch()* call does not return an error indication. If a null is encountered for a select–list item, the associated column return code (*rcode*) for that column is set to the error

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

and the indicator parameter is set to –1. The *ofetch()* call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, *ofetch()* does return the 1405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

Even when fetching a single row, Oracle recommends that Oracle7 OCI programs use *oexfet()*, with the *nrows* parameter set to 1, instead of the combination of *oexec()* and *ofetch()*.

The following example shows how you can obtain data from Oracle using *ofetch()* on a query statement. This example continues the one given in the description of the *odefin()* function earlier in this chapter. In that example, the select–list items in the SQL statement

```
SELECT ENAME, EMPNO, SAL FROM EMP WHERE
    SAL > :MIN_SAL
```

were associated with output buffers, and the addresses of column return lengths and return codes were bound. The example continues:

```
int rv;

oexec(&cursor);                  /* execute the statement */
/* fetch each row of the query */
for (;;) {
  if (rv = ofetch(&cursor))   /* break on row–level error */
    break;
  if (ret_codes[0] == 0)
    printf("%*.*s\t", retl[0], retl[0], employee_name);
  else if (ret_codes[0] == 1405)
    printf("%*.*s\t", retl[0], retl[0], "Null");
  else
    break;

if (ret_codes[1] == 0)
  printf("%d\t", employee_number);

  /* process remaining items */
  ..
.  printf("\n");
}
/* check rv for abnormal termination or just end–of–fetch */
if (rv != 1403)
  errrpt(&cursor);
...
```

Each *ofetch()* call returns the next row from the set of rows that satisfies a query. After each *ofetch()* call, the *rows processed count* in the CDA is incremented.

You cannot refetch rows previously fetched except by re–executing the *oexec()* call and moving forward through the active set again. After the last row has been returned, the next fetch returns a "no data found" return code. When this happens, the *rows processed count* contains the total number of rows returned by the query.

Parameter

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN/OUT

cursor
A pointer to the CDA associated with the SQL statement.

See Also *odefin()*, *odefinps()*, *odescr()*, *oexec()*, *oexfet()*, *ofen()*.

oflng

Purpose *oflng()* fetches a portion of a LONG or LONG RAW column.

Syntax

```
oflng(Cda_Def *cursor, sword pos,
      ub1 *buf, sb4 bufl, sword dtype,
      ub4 *retl, sb4 offset);
```

Comments LONG and LONG RAW columns can hold up to 2 gigabytes of data. The *oflng()* function allows you to fetch up to 64K bytes, starting at any offset, from the LONG or LONG RAW column of the current row. There can be only one LONG or LONG RAW column in a table; however, a query that includes a join operation can include in its select list several LONG–type items. The *pos* parameter specifies the LONG–type column that the *oflng()* call uses.

> **Note:** Although the datatype of *bufl* is **sb4**, *oflng()* can only retrieve up to 64K at a time. If an attempt is made to retrieve more than 64K, the returned data will not be complete. The use of **sb4** in the interface is for future enhancements.

Before calling *oflng()* to retrieve the portion of the LONG–type column, you must do one or more fetches to position the cursor at the desired row.

oflng() is useful in cases where unstructured LONG or LONG RAW column data cannot be manipulated as a solid block.; for example, a voicemail application that uses sampled speech, stored as one byte per sample, at perhaps 10000 samples per second. If the voice message is to be played out using a buffered digital–to–analog converter, and the buffer takes 64 Kbyte samples at a time, you can use *oflng()* to extract the message in chunks of this size, sending them to the converter buffer. See the *cdemo3.c* program in Appendix A for an example that demonstrates this technique.

When calling *oflng()* to retrieve multiple segments from a LONG–type column, it is much more efficient to retrieve sequentially from low to high offsets, rather than from high to low, or randomly.

> **Note:** With release 7.3, it may be possible to perform piecewise operations more efficiently using the new *obindps()*, *odefinps()*, *ogetpi()*, and *osetpi()* calls. See the section "Piecewise Insert, Update and Fetch" on page 2 – 39 for more information.

The program fragment below shows how to retrieve 64 Kbytes, starting at offset 70000, from a LONG column. There are two columns in the table; the LONG data is in column two.

```
#define DB_SIZE  65536
#define FALSE    0
#define TRUE     1

ub1 *data_area;
sb4 offset;
sb2 da_indp, id_no;
ub4 ret_len;
Cda_Def cda;

...

data_area = (ub1 *) malloc(DB_SIZE);

...

oparse(&cda, "SELECT id_no, data FROM data_table
             WHERE id_no = 100", –1, TRUE, 1);/* deferred parse */

/* define the first column – id_no, with no indicator parameter */
odefin(&cda, 1, &id_no, (int) sizeof (int), 3, –1, 0, 0, 0,
       –1, 0, 0);

/* define the 2nd column – data, with indicator parameter */
odefin(&cda, 2, data_area, DB_SIZE, 1, –1, &da_indp,
       0, 0, –1, 0, 0);

oexfet(&cda, 1, FALSE, FALSE);    /* cursor is now at the row */
oflng(&cda, 2, data_area, DB_SIZE, 1, &ret_len, (sb4) 70000);
```

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
pos	sword	IN
buf	ub1 *	OUT
bufl	sb4	IN
dtype	sword	IN
retl	ub4 *	OUT
offset	sb4	IN

cursor
A pointer to the CDA specified in the associated *oparse()* call.

pos
The index position of the LONG–type column in the row. The first column is position one. If the column at the index position is not a LONG type, a "column does not have LONG datatype" error is returned. If you do not know the position, you can use *odescr()* to index through the select–list. When a LONG datatype code (8 or 24) is returned in the *dtype* parameter, the value of the loop index variable (that started at 1) is the position of the LONG–type column.

buf
A pointer to the buffer that receives the portion of the LONG–type column data.

bufl
The length of *buf* in bytes.

dtype
The datatype code corresponding to the type of *buf*. See the "External Datatypes" section in Chapter 3 for a list of the datatype codes.

retl
The number of bytes returned. If more than 65535 bytes were requested and returned, the maximum value returned in this parameter is still 65535.

offset
The zero–based offset of the first byte in the LONG–type column to be fetched.

See Also *odescr()*, *oexfet()*, *ofen()*, *ofetch()*.

ogetpi

Purpose *ogetpi()* returns information about the next chunk of data to be processed as part of a piecewise insert, update or fetch.

Syntax

```
ogetpi (Cda_Def *cursor, sb1 *piecep,dvoid **ctxpp,
         eword* iterp, eword *indexp);
```

Comments *ogetpi()* is used (in conjunction with *osetpi()*) in an OCI application to determine whether more pieces exist to be either inserted, updated, or fetched as part of a piecewise operation.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

See the section “Piecewise Insert, Update and Fetch” in Chatper 2 for more information about piecewise operations and the *ogetpi()* call.

The following sample code demonstrates the use of *ogetpi()* in an OCI program which performs an piecewise insert. This code is provided for demonstration purposes only, and does not constitute a complete program.

For sample code demonstrating the use of *ogetpi()* for a piecewise fetch, see the description of the *osetpi()* routine later in this chapter.

This sample program performs a piecewise insert into a LONG RAW column in an Oracle table. The program is invoked with arguments specifying the name of the file to be inserted and the size of the piece to be used for the insert. It then inserts that file and its name into the database. Most of the data processing is done in the *insert_file()* routine.

```
/* The table FILES is used for this example:
 *
 * SQL> describe FILES
 * Name         Null?       Type
 * FILENAME     NOT NULL    VARCHAR2(255)
 * FILECONTENT              LONG RAW
 */
...                                  /* OCI #include statements */

#define  DEFER_PARSE        1                  /* oparse flags */
#define  NATIVE             1
#define  VERSION_7          2
```

```
#define  OCI_MORE_INSERT_PIECES –3129

#define OCI_EXIT_FAILURE 1                      /*   exit flags */
#define OCI_EXIT_SUCCESS 0

void   insert_file();

                 /* Usage : piecewise_insert filename piecesize */
main(argc, argv)
int    argc;
char  *argv[];
{
  Lda_Def   lda;                            /*       login area */
  ub1       hda[256];                       /*        host area */
  Cda_Def   cda;                            /*      cursor area */

...                                         /* log on to Oracle */

  insert_file(&lda, &cda, argv[1], atol(argv[2]));

...                                         /*log off of Oracle */
}

/* Function insert_file(): This function loads long raw data    */
/*          into a memory buffer from a source file and then    */
/*          inserts it piecewise into the database              */
/*                                                              */
/* Note: If necessary, the context pointer could be used to     */
/*          point to a file being used in a piecewise operation.*/
/*          It is not necesssay in this example program, so a   */
/*          dummy value is passed instead.                      */

void      insert_file(lda, cda, filename, piecesize)
Lda_Def  *lda;
Cda_Def  *cda;
text     *filename;
ub4       piecesize;
{
  text  *longbuf;       /* buffer to hold long column on insert */
  ub4    len_longbuf;                     /*  length of longbuf */
  ub2    col_rcode;                       /* Column return code */
  text   errmsg[2000];
  int    fd;
  char  *context = ”context pointer”;
  ub1    piece;
  ub4    iteration;
  ub4    plsqltable;
  ub1    cont = (ub1)1;
```

```
text   *sqlstmt = (text *)
                 ”INSERT INTO FILES (filename, filecontent) \
                  VALUES (:filename, :filecontent)”;
 if (oopen(cda, lda, (text *)0, –1, –1, (text *)0, –1))
  exit(OCI_EXIT_FAILURE);

printf(”\nOpening source file %s\n”, filename);
if (!(fd = open((char *)filename, O_RDONLY)))
  exit(1);
                  /* Allocate memory for storage of one piece */
len_longbuf = piecesize;
longbuf     = (text *)malloc(len_longbuf);
if (longbuf == (text *)NULL)
  exit(1);

if (oparse(cda, sqlstmt, (sb4)–1, 0, (ub4)VERSION_7))
  exit(OCI_EXIT_FAILURE);

if (obndrv(cda, (text *)”:filename”, –1, filename, –1,
           SQLT_STR, –1, (sb2 *)0, (ub1 *)0, –1, –1))
  exit(OCI_EXIT_FAILURE);

if (obindps(cda, 0, (text *)”:filecontent”,
            strlen(”:filecontent”), (ub1 *)context, len_longbuf,
            SQLT_LBI, (sword)0, (sb2 *)0,
            (ub2 *)0, &col_rcode, 0, 0, 0, 0,
            0, (ub4 *)0, (text *)0, 0, 0))
  exit(OCI_EXIT_FAILURE);

while (cont)
{
  oexec(cda);
  switch (cda.rc)
  {
  case 0:                           /*  operation is finished */
    cont = 0;
    break;
  case OCI_MORE_INSERT_PIECES:      /* ORA–03129 was returned */
    if ((len_longbuf = read(fd, longbuf, len_longbuf)) == –1)
      exit(OCI_EXIT_FAILURE);

    ogetpi(cda, &piece, (dvoid **)&context, &iteration,
           &plsqltable);

    if (len_longbuf < piecesize)               /* last piece? */
      piece = OCI_LAST_PIECE;
    osetpi(cda, piece, longbuf, &len_longbuf);
    break;
  default:
        err_report(lda, cda);
        exit(OCI_EXIT_FAILURE);
      }
    }
    ocom(lda);                              /* Commit the insert */
    if (close(fd))                          /*        close file */
      exit(OCI_EXIT_FAILURE);
    if (oclose(cda))                        /*      close cursor */
      exit(OCI_EXIT_FAILURE);
  }
```

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
piecep	sb1 *	OUT
ctxpp	dvoid **	OUT
iterp	eword *	OUT
indexp	eword *	OUT

cursor
A pointer to the CDA associated with the SQL or PL / SQL statement being processed.

piecep
Specifies whether the next piece to be fetched or inserted is the first piece, an intermediate piece or the last piece. Possible values are OCI_FIRST_PIECE (one) or OCI_NEXT_PIECE (two).

ctxpp
A pointer to the user–defined context pointer, which is optionally passed as part of an *obindps()* or *odefinps()* call. This pointer is returned to the application during the *ogetpi()* call. If *ctxpp* is passed as NULL, the parameter is ignored. The application may already know which buffer it needs to pass in *osetpi()* at run time.

iterp
Pointer to the current iteration. During an array insert it will tell you which row you are working with. Starts from 0.

indexp
Pointer to the current index of an array mapped to a PL / SQL table, if an array is bound for an insert. The value of *indexp* varies between zero and the value set in the *cursiz* parameter of the *obindps()* call.

See Also *obindps(), odefinps(), osetpi().*

olog

Purpose *olog()* establishes a connection between an OCI program and an Oracle database.

Syntax

```
olog(Lda_Def *lda, ub1 *hda, text *uid, sword uidl,
     text *pswd, sword pswdl, text *conn, sword connl,
     ub4 mode)
```

Comments An OCI program can connect to one or more Oracle instances multiple times. Communication takes place using the LDA and the HDA defined within the program. It is the *olog()* function that connects the LDA to Oracle.

The HDA is a program–allocated data area associated with each *olog()* logon call. Its contents are entirely private to Oracle, but the HDA must be allocated by the OCI program. Each concurrent connection requires one LDA–HDA pair.

> **Note:** The HDA must be initialized to all zeros (binary zeros, not the "0" character) before the call to *olog()*, or runtime errors will occur. In C, this means that the HDA must be declared as global or static, rather than as a local or automatic character array. See the sample code below for typical declarations and initializations for a call to *olog()*.

The HDA has a size of 256 bytes on 32–bit systems, and 512 bytes on 64–bit systems. If memory permits, it is possible to allocate a 512–byte HDA on a 32–bit system to increase portability of aplications.

Refer to the section "Host Data Area" in Chapter 2 for more information about HDAs.

After the *olog()* call, the HDA and the LDA must remain at the same program address they occupied at the time *olog()* was called.

For example:

```
#include <stdlib.h>

             /* global variable declarations */

Lda_Def lda[2];        /* establish two LDAs */
ub1     hda[2][512];        /* and two HDA's */

text *uid1 = "SCOTT/TIGER";
text *uid2 = "SYSTEM";
text *pwd  = "MANAGER";
...
```

```
/* first connect as scott */
if (olog(&lda[0], &hda[0], uid1, –1, (text *) 0, –1,
         (text *) 0, –1, OCI_LM_DEF))
{
  error_handler(&lda[0]);
  exit(EXIT_FAILURE);
}
...
/* and later as the system manager */
if (olog(&lda[1], &hda[1], uid2, –1, pwd, –1,
         (text *) 0, –1, OCI_LM_DEF))
{
  error_handler(&lda[1]);
  exit(EXIT_FAILURE);
}
```

When an OCI program has issued an *olog()* call, a subsequent *ologof()* call using the same LDA commits all outstanding transactions for that connection. If a program fails to disconnect or terminates abnormally, then all outstanding transactions are rolled back.

The LDA *return code* field indicates the result of the *olog()* call. A zero return code indicates a successful connection.

The *mode* parameter specifies whether the connection is in blocking or non–blocking mode. For more information on connection modes, see "Non–Blocking Mode" on page 2 – 32. For a short example program, see the *onbset()* description on page 4 – 89.

You should also refer to the section on SQL*Net in your Oracle system–specific documentation for any particular notes or restrictions that apply to your operating system.

Parameters

Parameter Name	***Type***	***Mode***
lda	Lda_Def *	IN / OUT
hda	ub1 *	OUT
uid	text *	IN
uidl	sword	IN
pswd	text *	IN
pswdl	sword	IN
conn	text *	IN
connl	sword	IN
mode	ub4	IN

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

hda
A pointer to a host data area struct. See Chapter 2 for more information on host data areas.

uid
Specifies a string containing the username, an optional password, and an optional host machine identifier. If you include the password as part of the *uid* parameter, put it immediately after the username and separate it from the username with a '/'. Put the host machine identifier after the username or the password, preceded by the '@' sign.

If the password is not included in this parameter, it must be in the *pswd* parameter. Examples of valid *uid* parameters are

```
name
name/password
name@service_name
name/password@service_name
```

The following example is not a correct example of a *uid*:

```
name@service_name/password
```

uidl
The length of the string pointed to by *uid*. If the string pointed to by *uid* is null terminated, this parameter should be passed as –1.

pswd
A pointer to a string containing the password. If the password is specified as part of the string pointed to by *uid*, this parameter should be passed as **(text *) 0**.

pswdl
The length of the string pointed to by *pswd*. If the string pointed to by *pswd* is null terminated, this parameter should be passed as –1.

conn
Specifies a string containing a SQL*Net V2 connect descriptor to connect to a database. If the connect string is passed in as part of the *uid*, this parameter should be passed as **(text *) 0**.

connl
The length of the string pointed to by *conn*. If the string pointed to by *conn* is null terminated, this parameter can be passed as –1.

mode
Specifies whether the connection is in blocking or non–blocking mode. Possible values are OCI_LM_DEF (for blocking) or OCI_LM_NBL (for non–blocking).

See Also *ologof()***,** *onbset(), sqllda()*.

ologof

Purpose *ologof()* disconnects an LDA from the Oracle program global area and frees all Oracle resources owned by the Oracle user process.

Syntax

```
ologof(Lda_Def *lda);
```

Comments A COMMIT is automatically issued on a successful *ologof()* call; all currently open cursors are closed. If a program logs off unsuccessfully or terminates abnormally, all outstanding transactions are rolled back.

If the program has multiple active connections, a separate *ologof()* must be performed for each active LDA.

If *ologof()* fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
lda	Lda_Def *	IN / OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *olog()*.

onbclr

Purpose *onbclr()* places a database connection in blocking mode.

Syntax `onbclr(Lda_Def *lda);`

Comments If there is a pending call on a non–blocking connection and *onbclr()* is called, the pending call (*onbclr()*), when resumed, will block.

Parameters

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	IN

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *olog(), onbset(), onbtst()*.

onbset

Purpose *onbset()* places a database connection in non–blocking mode for all subsequent OCI calls on this connection.

Syntax

```
onbset(Lda_Def *lda);
```

Comments *onbset()* will succeed only if the library is linked in deferred mode and if the network driver supports non–blocking operations.

> **Note:** *onbset()* requires SQL*Net Release 2.1 or higher. It cannot be used with a single–task driver.

The following example code demonstrates the use of the non–blocking calls. Before running the OCI program, you must execute this SQL script.

```
set echo on;
connect system/manager;
drop user ocitest cascade;
create user ocitest identified by ocitest;
grant connect,resource to ocitest;
connect ocitest/ocitest;
create table oci21tab (col1 varchar2(30));
insert into oci21tab values (’A’);
insert into oci21tab values (’AB’);
insert into oci21tab values (’ABC’);
insert into oci21tab values (’ABCD’);
insert into oci21tab values (’ABCDE’);
insert into oci21tab values (’ABCDEF’);
insert into oci21tab values (’ABCDEFG’);
insert into oci21tab values (’ABCDEFGH’);
insert into oci21tab values (’ABCDEFGHI’);
insert into oci21tab values (’ABCDEFGHIJ’);
insert into oci21tab values (’ABCDEFGHIJK’);
insert into oci21tab values (’ABCDEFGHIJKL’);
commit;
```

This program performs a long–running, hardcoded insert and demonstrates the use of non–blocking calls. This example is included online as *oci21.c* and *oci21.sql*. See your Oracle system–specific documentation for the location of these files.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <oratypes.h>
/* LDA and CDA struct declarations */
#include <ocidfn.h>
```

```
#ifdef __STDC__
#include <ociapr.h>
#else
#include <ocikpr.h>
#endif
/* demo constants and structs */
#include <ocidem.h>
/* oparse flags */
#define  DEFER_PARSE         1
#define  NATIVE              1
#define  VERSION_7           2
/* exit flags */
#define OCI_EXIT_FAILURE     1
#define OCI_EXIT_SUCCESS     0
#define BLOCKED          –3123   /* ORA–03123 */
#define SUCCESS              0

Lda_Def lda;            /* login area */
ub1     hda[HDA_SIZE];  /* host area */
Cda_Def cda;            /* cursor area */

/* Function prototypes */
void log_on ();
void log_off ();
void setup();
void err_report();
void insert_data();
void do_exit();

/* SQL statement used in this program */
text *sqlstmt = (text *)”INSERT INTO oci21tab (col1)\
                         SELECT a.col1 \
                         FROM oci21tab a, oci21tab b”;
main(argc, argv)
eword argc;
text **argv;
{
  log_on();       /* logon to Oracle database */
  setup();       /* prepare sql statement */
  insert_data();
  log_off();      /* logoff Oracle database */
  do_exit(OCI_EXIT_SUCCESS);
}
/* Function:    setup
 * Description: This routine does the necessary setup
 * to execute the SQL statement. Specifically, it does
 * the open, parse, bind and define phases as needed. */
void setup()
{
  if (onbset(&lda))     /* make the connection non–blocking */
  {
    err_report((Cda_Def *)&lda);
    do_exit(OCI_EXIT_FAILURE);
  }
  if (onbtst(&lda))     /* verify that it is non–blocking */
  {
    printf(”connection is still blocking!!!\n”);
    do_exit(OCI_EXIT_FAILURE);
  }
  if (oopen(&cda, &lda, (text *) 0, –1, –1,
      (text *) 0, –1))                        /* open */
  {
    err_report(&cda);
    do_exit(OCI_EXIT_FAILURE);
  }
  if (oparse(&cda, sqlstmt, (sb4) –1, DEFER_PARSE,
      (ub4) VERSION_7))
  {
    err_report(&cda);
    do_exit(OCI_EXIT_FAILURE);
  }
}
/* Function:    insert_data
 * Description: This routine inserts the data into the table */
void insert_data()
{
   ub1 done = 0;
   /* number of times statement blocked */
   static ub1 blocked_cnt = 0;
   while (!done)
   {
     switch(oexec(&cda))
     {
       case BLOCKED:     /* will come through here multiple
                          * times, but print msg once */
            blocked_cnt++;
            break;
       case SUCCESS:
            done = 1;
            break;
       default:
            err_report(&cda);
            /* get out of application */
            do_exit(OCI_EXIT_FAILURE);
     }
   }
   printf(”\n Execute call blocked %ld times\n”, blocked_cnt);
   if (onbclr(&lda))  /* clear the non–blocking status of the
                       * connection */
   {
     err_report((Cda_Def *)&lda);
     do_exit(OCI_EXIT_FAILURE);
   }
}
/* Function:    err_report
 * Description: This routine prints out the most recent
 *              OCI error */
void err_report(cursor)
Cda_Def *cursor;
{
    sword n;
    text msg[512];     /* message buffer to hold error text */
    if (cursor–>fc > 0)
      printf(”\n–– ORACLE error when processing \
             OCI function %s \n\n”,
             oci_func_tab[cursor–>fc]);
    else
      printf(”\n–– ORACLE error\n”);

    n = (sword)oerhms(&lda, cursor–>rc, msg, (sword) sizeof msg);
    printf(”%s\n”, msg);
}
/* Function:    do_exit
 * Description: This routine exits with a status */
void do_exit(status)
eword status;
{
  if (status == OCI_EXIT_FAILURE)
     printf(”\n Exiting with FAILURE status %d\n”, status);
  else
     printf(”\n Exiting with SUCCESS status %d\n”, status);

  exit(status);
}
/* Function: log_on
 * Description: This routine logs onto the database as
 *              OCITEST/OCITEST. */
void log_on()
{

  if (olog(&lda, hda, (text *)”OCITEST”, –1, (text *)”OCITEST”,
           –1, (text*)”inst1_nonblock” , –1, OCI_LM_DEF))
  {
    err_report((Cda_Def *)&lda);
    exit(OCI_EXIT_FAILURE);
  }
  printf("\n Connected to Oracle as ocitest\n");
}
/* Function: log_off
 * Description: This routine closes out any cursors and logs
 *              off the database */
void log_off()
{
  if (oclose(&cda))            /* close cursor */
  {
    printf("Error closing cursor 1.\n");
    do_exit(OCI_EXIT_FAILURE);
  }
  if (ologof(&lda))            /* log off the database */
  {
    printf("Error on disconnect.\n");
    do_exit(OCI_EXIT_FAILURE);
  }
}
```

Parameters

Parameter Name	***Type***	***Mode***
lda	Lda_Def *	IN / OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *olog(), onbclr(), onbtst()*.

onbtst

Purpose *onbtst()* tests whether a database connection is in non–blocking mode.

Syntax

```
onbtst(Lda_Def *lda);
```

Comments If the connection is in the non–blocking mode, *onbtst()* returns 0. Otherwise, it returns ORA–03128 in the *return code* field.

> **Note:** If the connection is in blocking mode, the user may call *onbset()* to place the channel in non–blocking mode, if allowed by the network driver.

Parameters

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	IN/OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *olog()*, *onbclr()*, *onbset()*.

oopen

Purpose *oopen()* opens the specified cursor.

Syntax

```
oopen(Cda_Def *cursor, Lda_Def *lda,
      <text *dbn>, <sword dbnl>, <sword arsize>,
      <text *uid>, <sword uidl>);
```

Comments *oopen()* associates a cursor data area in the program with a data area in the Oracle Server. Oracle uses these data areas to maintain state information about the processing of a SQL statement. Status concerning error and warning conditions, and other information, such as function codes, is returned to the CDA in your program, as Oracle processes the SQL statement.

A program can have many cursors active at the same time.

The *oparse()* function parses a SQL statement and associates it with a cursor. In the OCI functions, SQL statements are always referenced using a cursor as the handle.

It is possible to issue an *oopen()* call on a cursor that is already open. This has no effect on the cursor, but it does affect the value in the Oracle OPEN_CURSORS counter. Repeatedly reopening an open cursor may result in an ORA–01000 error ('maximum open cursors exceeded'). Refer to the *Oracle7 Server Messages* manual for information about what to do if this happens.

The *return code* field of the CDA indicates the result of the *oopen()*. A return code value of zero indicates a successful *oopen()* call.

See the description of the *obndra()* function earlier in this chapter for an example program demonstrating the use of *oopen()*.

Parameters

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	OUT
lda	Lda_Def *	IN / OUT
dbn	text *	IN
dbnl	sword	IN
arsize	sword	IN
uid	text *	IN
uidl	sword	IN

cursor
A pointer to a cursor data area associated with the program.

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

dbn
This parameter is included only for Oracle Version 2 compatibility. It should be passed as 0 in later versions.

dbnl
This parameter is included only for Oracle Version 2 compatibility. It should be passed as –1 in later versions.

arsize
Oracle7 does not use the *areasize* parameter. The data areas in the Oracle Server used by cursors are automatically resized as required.

uid
A pointer to a character string containing the userid and the password. The password must be separated from the userid by a '/'.

uidl
The length of the string pointed to by *uid*. If *uid* points to a null–terminated string, this parameter can be omitted.

See Also *olog()*, *oparse()*.

oopt

Purpose *oopt()* sets rollback options for non–fatal Oracle errors involving multi–row INSERT and UPDATE SQL statements. It also sets wait options in cases where requested resources are not available; for example, whether to wait for locks.

Syntax

```
oopt(Cda_Def *cursor, sword rbopt, sword waitopt);
```

Comments The *rbopt* parameter is not supported in Oracle Server Version 6 or later.

Parameters

Parameter Name	*Type*	*Mode*
cursor	Cda_Def *	IN / OUT
rbopt	sword	IN
waitopt	sword	IN

cursor
A pointer to the CDA used in the associated *oopen()* call.

rbopt
The action to be taken when a non–fatal Oracle error occurs. If this option is set to zero, all errors, even non–fatal errors, cause the current transaction to be rolled back. If this option is set to 2, only the failing row will be rolled back during a non–fatal row–level error. This is the default setting.

waitopt
Specifies whether to wait for resources or return with an error if they are currently not available. If this option is set to zero, the program waits indefinitely if resources are not available. This is the default action. If this option is set to 4, the program will receive an error return code whenever a resource is requested but is unavailable. Use of *waitopt* set to 4 can cause many error return codes while waiting for internal resources that are locked for short durations. The only resource errors received are for resources requested by the calling process.

See Also *oopen()***.**

oparse

Purpose *oparse()* parses a SQL statement or a PL/SQL block and associates it with a cursor. The parse can optionally be deferred.

Syntax

```
oparse(Cda_Def *cursor, text *sqlstm,
       [sb4 sqll], sword defflg,
       ub4 lngflg);
```

Comments *oparse()* passes the SQL statement to Oracle for parsing. If the *defflg* parameter is non–zero, the parse is deferred until the statement is executed, or until *odescr()* is called to describe the statement. Once the parse is performed, the parsed representation of the SQL statement is stored in the Oracle shared SQL cache. Subsequent OCI calls reference the SQL statement using the cursor name.

An open cursor can be reused by subsequent *oparse()* calls within a program, or the program can define multiple concurrent cursors when it is necessary to maintain multiple active SQL statements.

> **Note:** When *oparse()* is called with the *defflg* parameter set to a non–zero value, you cannot receive most error indications until the parse is actually performed. The parse is performed at the first call to *odescr()*, *oexec()*, *oexn()*, or *oexfet()*. However, the SQL statement string is scanned on the client system, and some errors, such as "missing double quote in identifier", can be returned immediately.

The statement can be any valid SQL statement or PL/SQL anonymous block. Oracle parses the statement and selects an optimal access path to perform the requested function.

Data Definition Language statements are executed on the parse if you have linked in non–deferred mode or if you have liked with the deferred option and the *defflg* parameter of *oparse()* is zero. If you have linked in deferred mode and the *defflg* parameter is non–zero, you must call *oexn()* or *oexec()* to execute the statement.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when *oparse()* is called in non–deferred mode are not detected until the first non–deferred call is made (usually an execute or describe call).

The example below opens a cursor and parses a SQL statement. The *oparse()* call associates the SQL statement with the cursor.

```
Lda_Def lda;
Cda_Def cursor;
text *sql_stmt =
    “DELETE FROM emp WHERE empno = :Employee_number”;
...
oopen(&cursor, &lda, (text *)0, –1, –1, (text *)0, –1);
oparse(&cursor, sql_stmt, –1, 1, 2);
```

SQL syntax error codes are returned in the CDA’s *return code* field. If the statement cannot be parsed, the *parse error offset* field indicates the location of the error in the SQL statement text. See the section “Cursor Data Area” on page 2 – 4 for a list of the information fields available in the CDA after an *oparse()* call.

Parameters

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN / OUT
sqlstm	text *	IN
sqll	sb4	IN
defflg	sword	IN
lngflg	ub4	IN

cursor
A pointer to the CDA specified in the *oopen()* call.

sqlstm
A pointer to a string containing the SQL statement.

sqll
Specifies the length of the SQL statement. If the SQL statement string pointed to by *sqlstm* is null terminated, this parameter can be omitted.

defflg
If non–zero and the application was linked in deferred mode, the parse of the SQL statement is deferred until an *odescr()*, *oexec()*, *oexn()*, or *oexfet()* call is made.

> **Note:** Bind and define operations are also deferred until the execute or describe step if the program was linked using the deferred mode link option.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when *oparse()* is called in non–deferred mode are not detected until the first non–deferred call is made (usually an execute or describe call).

lngflg
The *lngflg* parameter determines how Oracle handles the SQL statement or PL/SQL anonymous block. To ensure strict ANSI conformance, Oracle7 defines several datatypes and operations in a slightly different way than Oracle Version 6. The table below shows the differences between Version 6 and Oracle7 that are affected by the *lngflg* parameter.

Behavior	***V6***	***V7***
CHAR columns are fixed length (including those created by a CREATE TABLE statement).	NO	YES
An error is issued if an attempt is made to fetch a null into an output variable that has no associated indicator variable.	NO	YES
An error is issued if a fetched value is truncated and there is no associated indicator variable.	YES	NO
Describe (*odescr()*) returns internal datatype 1 for fixed–length strings	YES	NO
Describe (*odescr()*) returns datatype 96 for fixed–length strings.	n/a	YES

The *lngflg* parameter has three possible settings:

0 Specifies version 6 behavior (the database you are connected to can be any version 6 or later database).

1 Specifies the normal behavior for the database version to which the program is connected (either version 6 or Oracle7).

2 Specifies Oracle7 behavior. If you use this value for the parameter, and you are not connected to an Oracle7 database, Oracle issues the error

```
ORA–01011: Cannot use this language type when
         talking to V6 database
```

See Also *odescr()***,** *oexec()***,** *oexfet()***,** *oexn()***,** *oopen()***.**

opinit

Purpose *opinit()* initializes the OCI process environment. This includes specifying whether the application is single– or multi–threaded.

Syntax

```
opinit (ub4 mode);
```

Comments The *mode* parameter of the opinit() call indicates whether the application making the call is running a single– or multi–threaded environment. See the section "Thread Safety" in Chapter 2 for more information about using thread–safe calls in an OCI program.

If *mode* is set to OCI_EV_DEF or a call to *opinit()* is skipped altogether, for backward compatibility a single–threaded environment is assumed. Using thread safety adds a very small amount of overhead to the program, and this can be avoided by running in single–threaded mode.

> **Note:** Even when running a single–threaded application it is advisable to make the call to *opinit()*, with *mode* set to OCI_EV_DEF, rather than skipping it. In addition to setting the environment, the call to *opinit()* provides documentation that the application is not thread–safe.

If *mode* is set to OCI_EV_TSF, then the OCI application can make OCI calls from multiple threads of execution within a single program.

The following examples demonstrate how the same task might be accomplished in a multi–connection environment and a single–connection, multi–threaded environment. The task is to process a series of bank account transactions.

This code is provided for demonstration purposes only, and does not constitute a complete program.

The first example demonstrates how the transactions could be processed in a multi–threaded environment with a single connection. It is assumed that a user's program could call its own functions for the OS–specific thread package. Function calls could include calls to thread–safe packages from DCE or Posix for thread and semaphore management.

This program creates one session and multiple threads. Each thread executes zero or more transactions. The transactions are specified in a transient structure called "records." The transactions consist of moving a specified amount of money from one account to another. The example assumes that accounts 10001 through 10007 are set up in the database.

```
/* The table ACCOUNTS is used for this example:
 *
 * SQL> describe ACCOUNTS
 * Name                    Null?    Type
 * ACCOUNT                          NUMBER(36)
 * BALANCE                          NUMBER(36,2)
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h> /* dce thread/semaphore management package */
#include <ocidfn.h>
#include <ocikpr.h>

void   do_transaction();
void   get_transaction();

#define THREADS  3
#define DEFERRED 1
#define ORACLE7_BEHAVIOR 2

struct parameters
{ Lda_Def * lda;
  int thread_id;
};

typedef struct parameters parameters;
struct record_log
{  char action;
   unsigned int from_account;
   unsigned int to_account;
   float  amount;
};

typedef struct record_log record_log;
record_log records[]= { { 'M', 10001, 10002, 12.50 },
                        { 'M', 10001, 10003, 25.00 },
                        { 'M', 10001, 10003, 123.00 },
                        { 'M', 10001, 10003, 125.00 },
                        { 'M', 10002, 10006, 12.23 },
                        { 'M', 10007, 10008, 225.23 },
                        { 'M', 10002, 10008, 0.70 },
                        { 'M', 10001, 10003, 11.30 },
                        { 'M', 10003, 10002, 47.50 },
                        { 'M', 10002, 10006, 125.00 },
                        { 'M', 10007, 10008, 225.00 }};
```

```
static unsigned int trx_nr=0;
pthread_mutex_t mutex;

ub1     hda[256];

main()
{
  Lda_Def lda;
  pthread_t thread[THREADS];
  parameters params[THREADS];
  int i;
  pthread_addr_t status;

  opinit(OCI_EV_TSF);

                       /* log on to Oracle w/a single session */
  if(olog(lda, hda, (text *) ”SCOTT/TIGER”, –1, (text *) 0, –1,
               (text *) 0, –1, OCI_LM_DEF))
     exit(OCI_EXIT_FAILURE);

                     /*Create mutex for transaction retrieval */
  if (pthread_mutex_init(&mutex,pthread_mutexattr_default))
  {
     printf(”Can’t initialize mutex\n”);
     exit(OCI_EXIT_FAILURE);
  }
                                                 /*Spawn threads*/
  for(i=0;i<THREADS;i++)
  {
    params[i].lda=&lda;
    params[i].thread_id=i;
    printf(”Thread %d... ”,i);
    if (pthread_create(&thread[i],pthread_attr_default,
            (pthread_startroutine_t)do_transaction,
            (pthread_addr_t) &params[i]))
      printf(”Cant create thread %d\n”,i);
    else
      printf(”Created\n”);
  }
                                      /* Logoff session....    */
  for(i=0;i<THREADS;i++)
  {                                   /*wait for thread to end */
       printf(”Thread %d ....”,i);
     if (pthread_join(thread[i],&status))
       printf(”Error when wating for thread % to terminate\n”, i);
     else
      printf(”stopped\n”);
     printf(”Detach thread...”);
     if (pthread_detach(&thread[i]))
```

```
        printf(”Error detaching thread! \n”);
      else
        printf(”Detached!\n”);
  }
  printf(”Stop Session....”);
  ologof(&lda);
                   /*Destroys mutex for transaction retrieval */
  if (pthread_mutex_destroy(&mutex))
  {
    printf(”Can’t destroy mutex\n”);
    exit(1);
  }
}

/* Function do_transaction():  This functions executes one      */
/*             transaction out of the record array. The record  */
/*             array is ’managed’ by get_transaction().         */

void do_transaction(params)
parameters *params;
{
  Lda_Def * lda=params–>lda;
  Cda_Def cda;
  record_log *trx;
  text * pls_trans=(text *) \
                   ”BEGIN \
                    UPDATE ACCOUNTS \
                    SET    BALANCE=BALANCE+:amount \
                    WHERE  ACCOUNT=:to_account; \
                    UPDATE ACCOUNTS  \
                    SET    BALANCE=BALANCE–:amount \
                    WHERE  ACCOUNT=:from_account;  \
                    COMMIT;  \
                  END;”;

                           /* NOTE use of mutex for OCI calls */
   if (pthread_mutex_lock(&mutex))
     printf(”Can’t lock mutex\n”);
   if (oopen(&cda,lda, 0,–1,0,(text *)0,–1))
     err_report(&cda);
   if (oparse(&cda, pls_trans , –1, DEFERRED, ORACLE7_BEHAVIOR))
     err_report(&cda);
   if (pthread_mutex_unlock(&mutex))
     printf(”Can’t unlock mutex\n”);
                                   /* Done all transactions ? */
  while (trx_nr < (sizeof(records)/sizeof(record_log)))
  {
    get_transaction(&trx);
    printf(”Thread %d executing transaction\n”,params–>thread_id);
    switch(trx–>action)
    {
      case ’M’:
                              /* NOTE use of mutex for OCI calls */
                if (pthread_mutex_lock(&mutex))
                    printf(”Can’t lock mutex\n”);
                obndrv(&cda, ”:amount”, –1, (ub1 *)
                    &trx–>amount, sizeof(float), SQLT_FLT, –1,
                    (sb2 *) 0, (text *) 0, –1, –1);
                obndrv(&cda, ”:to_account”, –1, (ub1 *)
                    &trx–>to_account, sizeof(int), SQLT_INT, –1,
                    (sb2 *) 0, (text *) 0, –1, –1);
                obndrv(&cda,”:from_account”, –1, (ub1 *)
                    &trx–>from_account, sizeof(int), SQLT_INT, –1,
                    (sb2 *) 0, (text *) 0, –1, –1);
                oexec(&cda);
                if (pthread_mutex_unlock(&mutex))
                   printf(”Can’t unlock mutex\n”);
                 break;
       default:  break;
    }
                                /* Give other threads a chance.. */
    pthread_yield();
  }
                              /* NOTE use of mutex for OCI calls */
  if (pthread_mutex_lock(&mutex))
    printf(”Can’t lock mutex\n”);
  if (oclose(&cda))
    err_report(&cda);
  if (pthread_mutex_unlock(&mutex))
    printf(”Can’t unlock mutex\n”);
}

/* Function get_transaction: This routine returns the next      */
/*                            transaction to process             */

void get_transaction(trx)
record_log ** trx;
{
  if (pthread_mutex_lock(&mutex))
    printf(”Can’t lock mutex\n”);
  *trx=&records[trx_nr];
  trx_nr++;
  if (pthread_mutex_unlock(&mutex))
    printf(”Can’t unlock mutex\n”);

}
```

The second example demonstrates how the transactions could be processed in an environment with multiple connections.

This program creates as many sessions as there are threads. Each thread executes zero or more transactions. The transactions are specified in a transient structure called “records.” The transactions consist of moving a specified amount of money from one account to another. The example assumes that accounts 10001 through 10007 are set up in the database.

```
/* The table ACCOUNTS is used for this example:
 *
 * SQL> describe ACCOUNTS
 * Name                   Null?    Type
 * ACCOUNT                         NUMBER(36)
 * BALANCE                         NUMBER(36,2)
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>   /* thread & semaphore management package */
#include <ocidfn.h>
#include <ocikpr.h>

void   do_transaction();
void   get_transaction();

#define CONNINFO ”scott/tiger”
#define THREADS  3
#define DEFERRED 1
#define ORACLE7_BEHAVIOR 2

struct parameters
{ Lda_Def * lda;
  int thread_id;
};

typedef struct parameters parameters;
struct record_log
{  char action;
   unsigned int from_account;
   unsigned int to_account;
   float  amount;
};


typedef struct record_log record_log;
```

```
record_log records[]= { { ’M’, 10001, 10002, 12.50 },
                        { ’M’, 10001, 10003, 25.00 },
                        { ’M’, 10001, 10003, 123.00 },
                        { ’M’, 10001, 10003, 125.00 },
                        { ’M’, 10002, 10006, 12.23 },
                        { ’M’, 10007, 10008, 225.23 },
                        { ’M’, 10002, 10008, 0.70 },
                        { ’M’, 10001, 10003, 11.30 },
                        { ’M’, 10003, 10002, 47.50 },
                        { ’M’, 10002, 10006, 125.00 },
                        { ’M’, 10007, 10008, 225.00 }};

static unsigned int trx_nr=0;
pthread_mutex_t mutex;
ub1     hda[THREADS][256];

main()
{
  Lda_Def lda[THREADS];
  pthread_t thread[THREADS];
  parameters params[THREADS];
  int i;
  pthread_addr_t status;

  opinit(OCI_EV_TSF);

                     /* log on to Oracle w/multiple connections */
  for(i=0;i<THREADS;i++)
  {
  if(olog(lda[i], hda[i], (text *) ”SCOTT/TIGER”, –1, (text *) 0,
          –1, (text *) 0, –1, OCI_LM_DEF))
     exit(OCI_EXIT_FAILURE);
  }
                      /* create mutex for transaction retrieval */
  if (pthread_mutex_init(&mutex,pthread_mutexattr_default))
  {
     printf(”Can’t initialize mutex\n”);
     exit(1);
  }
                                                /* spawn threads */
  for(i=0;i<THREADS;i++)
  {
    params[i].lda=&lda[i];
    params[i].thread_id=i;
    printf(”Thread %d... ”,i);
    if (pthread_create(&thread[i],pthread_attr_default,
            (pthread_startroutine_t)do_transaction,
            (pthread_addr_t) &params[i]))
      printf(”Cant create thread %d\n”,i);
    else
      printf(”Created\n”);
  }

...                                         /* logoff sessions....*/

...                   /*destroy mutex for transaction retrieval */

}

/* Function do_transaction(): executes one transaction out of   */
/*               the records array. The records array is        */
/*               ’managed’ by the get_transaction function.     */

void do_transaction(params)
parameters *params;
{
  Lda_Def * lda=params–>lda;
  Cda_Def cda;
  record_log *trx;
  text * pls_trans=(text *) \
                     ”BEGIN \
                      UPDATE ACCOUNTS \
                      SET    BALANCE=BALANCE+:amount \
                      WHERE  ACCOUNT=:to_account; \
                      UPDATE ACCOUNTS  \
                      SET    BALANCE=BALANCE–:amount \
                      WHERE  ACCOUNT=:from_account;  \
                      COMMIT;  \
                      END;”;

                            /* NOTE lack of mutex for OCI calls */
   oopen(&cda,lda, 0,–1,0,(text *)0,–1);
   ioparse(&cda, pls_trans , –1, DEFERRED, ORACLE7_BEHAVIOR);

                                     /* Done all transactions ? */
  while (trx_nr < (sizeof(records)/sizeof(record_log)))
  {
    get_transaction(&trx);
    printf(”Thread %d executing transaction\n”,params–>thread_id);
    switch(trx–>action)
    {
      case ’M’:
                            /* NOTE lack of mutex for OCI calls */
          obndrv(&cda, ”:amount”, –1, (ub1 *) &trx–>amount,
             sizeof(float), SQLT_FLT, –1, (sb2 *) 0,
             (text *) 0, –1, –1)
```

```
         obndrv(&cda, ”:to_account”, –1, (ub1 *) &trx–>to_account,
            sizeof(int), SQLT_INT, –1, (sb2 *) 0,
            (text *) 0, –1, –1)

         obndrv(&cda,”:from_account”, –1,
            (ub1 *) &trx–>from_account, sizeof(int),
            SQLT_INT, –1, (sb2 *) 0, (text *) 0, –1, –1)
         oexec(&cda)
         break;

default: break;
    }
  }
                            /* NOTE lack of mutex for OCI calls */
  if (oclose(&cda))
      err_report(&cda);
}

/* Function get_transaction(): gets next transaction to process */

void get_transaction(trx)
record_log ** trx;
{
  if (pthread_mutex_lock(&mutex))
    printf(”Can’t lock mutex\n”);
  *trx=&records[trx_nr];
  trx_nr++;
  if (pthread_mutex_unlock(&mutex))
    printf(”Can’t unlock mutex\n”);

}
```

Parameter

Parameter Name	*Type*	*Mode*
mode	ub4	IN

mode
There are two values for the *mode* parameter: OCI_EV_DEF (zero), for single–threaded environments, and OCI_EV_TSF (one), for multi–threaded environments.

See Also *olog()*.

orol

Purpose *orol()* rolls back the current transaction.

Syntax

```
orol(Lda_Def *lda);
```

Comments The current transaction is defined as the set of SQL statements executed since the *olog()* call or the last *ocom()* or *orol()* call. If *orol()* fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
lda	Lda_Def *	IN / OUT

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

See Also *ocom()*, *olog()*.

osetpi

Purpose *osetpi()* sets information about the next chunk of data to be processed as part of a piecewise insert, update or fetch.

Syntax

```
osetpi (Cda_Def *cursor, sb1 piece,
        dvoid *bufp, ub4 *lenp);
```

Comments An OCI application uses *osetpi()* to set the information about the next piecewise insert, update, or fetch. The *bufp* parameter is a pointer to either the buffer containing the next piece to be inserted, or to the buffer where the next fetched piece will be stored.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

See the section “Piecewise Insert, Update and Fetch” in Chatper 2 for more information about piecewise operations and the *osetpi()* call.

The following sample code demonstrates the use of *osetpi()* to perform a piecewise fetch. This code is provided for demonstration purposes only, and does not represent a complete program.

For sample code demonstrating the use of *osetpi()* for a piecewise insert, see the description of the *osetpi()* routine later in this chapter.

This sample program performs a piecewise fetch from a LONG RAW column in an Oracle table. The program extracts the data from FILECONTENT and reconstitutes it in a file called FILENAME. The program is invoked with arguments specifying the file name and the piece size to be used for the fetch. Most of the data processing is done in the *fetch_file()* routine.

```
/* The table FILES is used for this example:
 *
 * SQL> describe FILES
 * Name          Null?       Type
 * FILENAME      NOT NULL    VARCHAR2(255)
 * FILECONTENT               LONG RAW
 */
...                                    /* OCI #include statements */

#define  DEFER_PARSE              1           /*  oparse flags */
#define  NATIVE                   1
#define  VERSION_7                2
#define  OCI_MORE_FETCH_PIECES    -3130
#define  MAX_COL_SIZE             2147483648    /* 2 gigabytes */
```

```
#define  OCI_EXIT_FAILURE 1                    /*    exit flags */
#define  OCI_EXIT_SUCCESS 0

void   fetch_file();

Lda_Def   lda;                            /*        login area */
ub1       hda[256];                       /*         host area */
Cda_Def   cda;                            /*       cursor area */

                   /* Usage: piecewise_fetch filename piecesize */
main(argc, argv)
int    argc;
char  *argv[];
{
...                                         /*  log on to Oracle */

  fetch_file(argv[1], atol(argv[2]));

...                                         /* log off of Oracle */
}

/* Function fetch_file(): retrieves contents of ’filename’      */
/*         from database and (re)stores it back to disk         */
/*                                                              */
/* Note: If necessary, the context pointer could be used to     */
/*          point to a file being used in a piecewise operation.*/
/*          It is not necesssay in this example program, so a   */
/*          dummy value is passed instead.                      */

void      fetch_file(filename, piecesize)
text     *filename;
ub4       piecesize;
{
  text   *longbuf;      /* buffer to hold long column on insert */
  ub4     len_longbuf;                      /* length of buffer */
  text    errmsg[2000];
  int     fd;
  char   *context = ”context pointer”;
  ub1     piece;
  eword   iteration = 0;
  eword   plsqltable;
  ub1     cont = 1;
  ub2     col_rcode;
  ub2     col_len;
  text    *sqlstmt = (text *) ”SELECT filecontent \
                               FROM FILES      \
                               WHERE filename=:filename”;
```

```
if (oopen(&cda, &lda, (text *)0, –1, –1, (text *)0, –1))
  exit(OCI_EXIT_FAILURE);

if (!(fd = open((char *)filename, O_WRONLY | O_CREAT, 511)))
  exit(OCI_EXIT_FAILURE);
                  /* Allocate memory for storage of one piece */
len_longbuf = piecesize;
longbuf     = (text *)malloc(len_longbuf);
if (longbuf == (text *)NULL)
  exit(OCI_EXIT_FAILURE);

if (oparse(&cda, sqlstmt, (sb4)–1, 0, (ub4)VERSION_7))
  exit(OCI_EXIT_FAILURE);

if (obndrv(&cda, (text *)":filename", –1, filename, –1,
           SQLT_STR, –1, (sb2 *)0, (text *)0, –1, –1))
  exit(OCI_EXIT_FAILURE);

if (odefinps(&cda, 0, 1, (ub1 *)&context, (ub4) MAX_COL_SIZE,
             SQLT_LBI, 0, (sb2 *)0, (text *)0, 0, 0,
             (ub2 *)&col_len, (ub2 *)&col_rcode, 0, 0, 0, 0))
  exit(OCI_EXIT_FAILURE);

if (oexec(&cda))         /*             execute SQL statement */
   exit(OCI_EXIT_FAILURE);
while (cont)             /* while pieces remain to be fetched */
{
  ofetch(&cda)
  switch (cda.rc)        /* do fetch & switch on return value */
  {
  case 0:                /*        write last piece to buffer */
    if (len_longbuf != write(fd, longbuf, len_longbuf))
      exit(OCI_EXIT_FAILURE);
    cont = 0;
    break;
  case OCI_MORE_FETCH_PIECES:   /*     ORA–03130 was returned */
    ogetpi(&cda, &piece, &context, &iteration, &plsqltable);

    if (piece!=OCI_FIRST_PIECE) /* can't write on first fetch */
      if (len_longbuf != write(fd, longbuf, len_longbuf))
        exit(OCI_EXIT_FAILURE);

    osetpi(&cda, piece, longbuf, &len_longbuf);
    break;
  default:
    exit(OCI_EXIT_FAILURE);    /* other value indicates error */
  }
}
```

```
  if (close(fd))                                      /*   close file */
    exit(OCI_EXIT_FAILURE);
  if (oclose(&cda))                                   /* close cursor */
    exit(OCI_EXIT_FAILURE);
}
```

Parameters

Parameter Name	***Type***	***Mode***
cursor	Cda_Def *	IN
piece	sb1	IN
bufp	dvoid *	IN
lenp	ub4 *	IN / OUT

cursor
A pointer to the cursor data area associated with the SQL or PL / SQL statement.

piece
Specifies the piece being provided or fetched. Possible values are OCI_FIRST_PIECE (one), OCI_NEXT_PIECE (two) and OCI_LAST_PIECE (three). Relevant when the buffer is being set after error ORA–03129 was returned by a call to *oexec()*.

bufp
A pointer to a data buffer. If *osetpi()* is called as part of a piecewise insert, this pointer must point to the next piece of the data to be transmitted. If *osetpi()* is called as part of a piecewise fetch, this is a pointer to a buffer to hold the next piece to be retrieved.

lenp
A pointer to the length in bytes of the current piece. If a piece is provided, the value is unchanged on return. If the buffer is filled up and part of the data is truncated, *lenp* is modified to reflect the length of the piece in the buffer.

See Also *obindps()*, *odefinps()*, *ogetpi()*.

sqlld2

Purpose The *sqlld2()* routine is provided for OCI programs that operate as application servers in an X/Open distributed transaction processing environment. *sqlld2()* fills in fields in the LDA parameter according to the connection information passed to it.

Syntax

```
dvoid sqlld2(Lda_Def *lda, text *cname, sb4 *cnlen);
```

Comments OCI programs that operate in conjunction with a transaction manager do not manage their own connections. However, all OCI programs require a valid LDA. You use *sqlld2()* to obtain the LDA.

sqlld2() fills in the LDA using the connection name passed in the *cname* parameter. The *cname* parameter must match the *db_name* alias parameter of the XA info string of the *xa_open()* call. If this parameter is a null pointer or if the *cnlen* parameter is set to zero, an LDA for the default connection is returned. Your program must allocate the LDA, then pass the pointer to it in the *lda* parameter.

sqlld2() does not return a value directly. If you call *sqlld2()* and there is no valid connection, the error

```
ORA–01012:  not logged on
```

is returned in the *return code* field of the *lda* parameter.

sqlld2() must be invoked whenever there is an active XA transaction. This means it must be invoked after *xa_open()* and *xa_start()*, and before x*a_end*(). Otherwise an ORA–01012 error will result.

sqlld2() is part of SQLLIB, the Oracle Precompiler library. SQLLIB must be linked into all programs that call *sqlld2()*. See your Oracle system–specific documentation for information about linking SQLLIB.

The example code on the following page demonstrates how you can use *sqlld2()* to obtain a valid LDA for a specific connection.

```
#include “ocidfn.h”
#include “ociapr.h”

/* define two LDAs */
Lda_Def lda1;
Lda_Def lda2;
sb4 clen1 = –1L;
sb4 clen2 = –1L;
...
/* get the first LDA for OCI use */
sqlld2(&lda1, “NYdbname”, &clen1);
if (lda1.rc != 0)
  handle_error();
/* get the second LDA for OCI use */
sqlld2(&lda2, “LAdbname”, &clen2);
if (lda2.rc != 0)
  handle_error();
```

Parameters

Parameter Name	*Type*	*Mode*
lda	Lda_Def *	OUT
cname	text *	IN
cnlen	sb4 *	IN

lda
A pointer to a local data area struct. You must allocate this data area before calling *sqlld2()***.**

cname
A pointer to the name of the database connection. If the name is a null–terminated string, you can pass the *cnlen* parameter as –1L. If the name is not null terminated, pass the exact length of the string in *cnlen*.

If the name consists of all blanks, *sqlld2()* returns the LDA for the default connection.

The *cname* parameter must match the *db_name* alias parameter of the XA info string of the *xa_open()* call.

cnlen
A pointer to the length of the *cname* parameter. You can pass this parameter as –1 if *cname* is a null–terminated string. If *cnlen* is passed as zero, *sqlld2()* returns the LDA for the default connection, regardless of the contents of *cname*.

sqllda

Purpose The *sqllda()* routine is for programs that mix precompiler code and OCI calls. A pointer to an LDA is passed to *sqllda()*. On return from *sqllda()*, the required fields in the LDA are filled in.

Syntax

```
dvoid sqllda(Lda_Def *lda);
```

Comments If your program contains both precompiler statements and calls to OCI functions, you cannot use *olog()*, (or *orlon()* or *olon()*) to log on to Oracle. You must use the embedded SQL command

```
EXEC SQL CONNECT ...
```

to log on. However, many OCI functions require a valid LDA. The *sqllda()* function obtains the LDA. *sqllda()* is part of SQLLIB, the precompiler library.

sqllda() fills in the LDA using the connect information from the most recently executed SQL statement. So, the safest practice is to call *sqllda()* immediately after doing the EXEC SQL CONNECT ... statement.

sqllda() does not return a value directly. If you call *sqllda()* and there is no valid connection, the error

```
ORA–01012:  not logged on
```

is returned in the *return code* field of the *lda* parameter.

The example below demonstrates how you can do multiple remote connections in a mixed Precompiler–OCI program. See Chapter 3 in the *Programmer's Guide to the Oracle Precompilers* for additional information about multiple remote connections.

```
EXEC SQL BEGIN DECLARE SECTION;
    text user_id[20], passwd[20], db_string1[20], db_string2[20];
    text dbn1[20], dbn2[20];
EXEC SQL END DECLARE SECTION;
...
/* host program declarations */
Lda_Def lda1;     /* declare two LDAs */
Lda_Def lda2;
dvoid sqllda(Lda_Def *);  /* declare the sqllda function */
...
/* set up strings */
strcpy(user_id, ”scott”);
strcpy(passwd, ”tiger”);
strcpy(db_string1, ”newyork”);
strcpy(db_string2, ”losangeles”);
strcpy(dbn1, ”NY”);
strcpy(dbn2, ”LA”);
```

```
...
/* do the connections */
EXEC SQL CONNECT :user_id IDENTIFIED BY :passwd
   AT :dbn1 USING :db_string1;

/* get the first LDA for OCI use */
sqllda(&lda1);
EXEC SQL CONNECT :user_id IDENTIFIED BY :passwd
    AT :dbn2 USING :db_string2;

/* get the second LDA for OCI use */
sqllda(&lda2);
```

Parameter

Parameter Name	***Type***	***Mode***
lda	Lda_Def *	OUT

lda
A pointer to a local data area struct. You must allocate this data area before calling *sqllda()*.

CHAPTER

5

The OCI Routines for COBOL

This chapter describes each subroutine in the OCI library for the COBOL OCI programmer. For all but the most simple routines, an example shows how a COBOL OCI program uses the routine. The description of each *routine* has five parts:

Purpose	What the routine does.
Syntax	The routine call with its parameter list.
Comments	A detailed description of the routine, including examples.
Parameters	A detailed description of each parameter.
See Also	Other routines that affect or are used with this routine.

Be sure to read the introductory section, "Calling OCI Routines" on page 5 – 2. It contains important information about data structures, datatypes, parameter passing conventions, and other things you need to know in calling COBOL OCI routines.

Calling OCI Routines

This section describes data structures and coding rules that are specific to applications written in COBOL. Refer to this section for information about data structures, datatypes, and parameter passing conventions in the COBOL call interface.

COBOL Data Areas

To use the OCI routines, you must declare data structures for one or more LDAs and CDAs.

The following declaration of the LDA and CDA is VMS specific:

```
DATA DIVISION.
WORKING–STORAGE SECTION.
01  LDA.
02    LDA–V2RC  PIC S9(4) COMP.
02    FILLER    PIC X(10).
02    LDA–RC    PIC S9(4) COMP.
02    FILLER    PIC X(50).
01  CURSOR–1.
02    C–V2RC    PIC S9(4) COMP.
02    C–TYPE    PIC S9(4) COMP.
02    C–ROWS    PIC S9(9) COMP.
02    C–OFFS    PIC S9(4) COMP.
02    C–FNC     PIC X.
02    FILLER    PIC X.
02    C–RC      PIC S9(4) COMP.
02    FILLER    PIC X(50).
01  CURSOR–2.
02    C–V2RC    PIC S9(4) COMP.
...
* declare remaining cursor data areas
```

The LDA and CDA are always 64 bytes. However, the size of fields in these areas are system dependent. See the Oracle system–specific documentation for the sizes on your system.

In the parameter descriptions and code examples in this chapter, the CDA is listed as CURSOR and is as defined in CURSOR–1 above. Also, the LDA is listed as LDA.

These parameters must always be passed by reference (the address of the area is passed). This is the default parameter passing mechanism for COBOL compilers.

COBOL Parameter Types

Parameters for the OCI routines are of three types:

- Address
- Binary Integer (PIC S9(9) COMP, PIC S9(4) COMP, or PIC S9(2) COMP)
- Character string (PIC X(n))

Address parameters pass the address of a variable in your program to Oracle. In COBOL, all parameters are normally passed to a subprogram by reference, so you simply pass all address parameters as you would normally pass any other parameter.

Integer parameters in OCI COBOL are in the format PIC S9(2) COMP, PIC S9(4) COMP, referred to in the other chapters of this guide as a "short integer," or in the format PIC S9(9) COMP, referred to elsewhere as "integer." All parameters should be passed by reference, which is the default parameter passing mechanism in COBOL. The OCI libraries will correctly dereference these parameters.

⚠ **Warning:** Even if your COBOL compiler supports call by value, do not pass integer parameters by value.

On some systems, binary integers must be declared as COMP–5, rather than COMP.

Character strings are a special type of parameter. A *length parameter* must be specified for character strings. Length parameters for strings are PIC S9(9) variables specifying the size in bytes of the character string.

COBOL Parameter Classification

There are three kinds of parameters in the USING phrase of an OCI subroutine:

- required parameters
- optional parameters
- unused parameters

Required Parameters

Required parameters are used by Oracle and must be supplied by the program. You must pass a valid value for each required parameter. If you do not supply a required parameter, your program will behave unpredictably.

Optional Parameters

Optional parameters are those that may or may not be used by Oracle, depending on the requirements of your program. Depending on your compiler, you can omit an optional parameter if your program does not need it. For example, you might decide that you do not want to supply an indicator variable on a bind call, because all your input values must be non–null. The INDP indicator parameter in the bind routines

OBNDRV and OBNDRN is optional. In the Syntax section for each routine in this chapter, optional parameters are surrounded by square brackets ([ ]).

There are several ways that you can indicate to Oracle that an optional parameter is being omitted. If your COBOL compiler permits missing parameters using a keyword such as “OMITTED”, you can use this convention to indicate that you are not using the optional parameter.

Many COBOL compilers permit trailing parameters to be omitted. If the optional parameters that you want to omit are the last parameters in the list and your compiler permits this, simply do not pass them.

If the parameter is of the type PIC S9(4) or PIC S9(9) *and is not an address parameter*, you can declare a variable for the parameter, move a **–1** value to it, and pass it normally. If the parameter is an *address* parameter, you cannot indicate that it is being omitted by passing a **–1** as the value in the parameter. For an address parameter, you can indicate that it is not being used only if your compiler supports a mechanism for omitting parameters or for passing parameters *by value*. In the latter case, you pass a 0 by value.

> **Note:** A value of –1 should not be passed for unused optional parameters in the new OBINDPS and ODEFINPS calls. Unused parameters in these calls must be passed a zero. See the descriptions of individual calls for more details about specific parameters.

In summary, if your compiler does not support omitted parameters or passing parameters by value, you cannot omit an optional address parameter. Make sure that when you pass an optional parameter it does not contain values that can affect the SQL statement. For example, the INDP parameter in the OBNDRA, OBNDRN, or OBNDRV routine is optional. If you are binding a DML statement, make sure that the value in the INDP parameter is zero when the statement is executed. Otherwise, the statement might fail.

In the code examples in this chapter, all optional parameters are passed. Mechanisms that are compiler specific, such as passing by value or using the OMITTED keyword, are not used.

Unused Parameters

Unused parameters are not used by Oracle, at least for the language being discussed. For example, the OCI logon routines specify an unused parameter called AUDIT. Unused parameters are passed in the same way as omitted optional parameters. In the syntax descriptions in this chapter, unused parameters are surrounded by angle brackets (<>).

Note: As with optional parameters, a value of –1 should not be passed for unused parameters in the new OBINDPS and ODEFINPS calls. Unused parameters in these calls must be passed a zerO. See the descriptions of individual calls for more details about specific parameters.

See page 5 – 21 for the description of the OBNDRN routine for examples of how to pass omitted optional and unused parameters.

Parameter Descriptions

In this chapter, parameters for the OCI routines are described in terms of their type and their mode. The type is either “Address”, PIC X(n), PIC S9(4) COMP or PIC S9(4) COMP–5, PIC S9(9) COMP, or PIC S9(9) COMP–5. The mode of a parameter has three possible values:

IN	A parameter that passes data to Oracle.
OUT	A parameter that receives data from Oracle on this call or a subsequent call.
IN / OUT	A parameter that passes data on the call, and that receives data on the return from this call or a subsequent call.

Linking COBOL OCI Programs

Check your Oracle system–specific documentation for additional information about linking COBOL OCI programs. It may be necessary to include extra libraries for linking on some platforms.

OBINDPS

Purpose OBINDPS associates the address of a program variable with a placeholder in a SQL or PL/SQL statement. Unlike older OCI bind calls, OBINDPS can be used to bind placeholders to be used in piecewise operations, or operations involving arrays of structures.

Syntax

```
CALL ”OBINDPS” USING CURSOR, OPCODE, SQLVAR, [SQLVL],
         PVCTX, PROGVL, [SCALE], [INDP], [ALENP],
        [RCODEP], PV–SKIP, IND–SKIP, ALEN–SKIP, RC–SKIP,
        MAXSIZ], [CURSIZ], [FMT], [FMTL], [FMTT].
```

Comments OBINDPS is used to associate the address of a program variable with a placeholder in a SQL or PL/SQL statement. Additionally, it can indicate that an application will be providing inserted or updated data incrementally at runtime. This piecewise insert is designated in the OPCODE parameter. OBINDPS is also used when an application will be inserting data stored in an array of structures.

> **Note:** This function is only compatible with Oracle Server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

With the introduction of OBINDPS there are now four fully–supported calls for binding input parameters, the other three being the older OBNDRA, OBNDRN and OBNDRV. Application developers should consider the following points when determining which bind call to use:

- OBINDPS is supported only when a program is linked in deferred mode. If it is necessary to link in non–deferred mode, another bind routine must be used. In this case, the ability to handle piecewise operations and arrays of structures is not supported.
- OBINDPS is more complex than the older bind calls. Users who are not performing piecewise operations and are not using arrays of structures may choose to use one of the older routines.
- OBINDPS does not support the ability to do a positional bind. If this functionality is needed, the bind should be performed using OBNDRN.

Unlike older OCI calls, OBINDPS does not accept –1 for any optional or unused parameters. When it is necessary to pass a value to these parameters NULL or 0 should be used instead.

See the sections “Piecewise Insert, Update and Fetch,” and “Arrays of Structures” in Chapter 2 for more information about piecewise operations, arrays of structures, skip parameters and the OBINDPS call.

For a C language example which uses OBINDPS to perform an insert from an array of structures, see the description of the *obindps()* call on page 4 – 6.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
OPCODE	PIC S9(2) COMP	IN
SQLVAR	PIC X(n)	IN
SQLVL	PIC S9(9) COMP	IN
PVCTX	PIC S9(2) COMP	IN
PROGVL	PIC S9(9) COMP	IN
FTYPE	PIC S9(9) COMP	IN
SCALE	PIC S9(9) COMP	IN
INDP	PIC S9(4) COMP	IN / OUT
ALENP	PIC S9(4) COMP	IN
RCODEP	PIC S9(4) COMP	OUT
PV–SKIP	PIC S9(9) COMP	IN
IND–SKIP	PIC S9(9) COMP	IN
ALEN–SKIP	PIC S9(9) COMP	IN
RC–SKIP	PIC S9(9) COMP	IN
MAXSIZ	PIC S9(9) COMP	IN
CURSIZ	PIC S9(9) COMP	IN / OUT
FMT	PIC X(6)	IN
FMTL	PIC S9(9) COMP	IN
FMTT	PIC S9(9) COMP	IN

Note: Since the OBINDPS call can be used in a variety of different circumstances, some items in the following list of parameter descriptions include different explanations for how the parameter is used for piecewise operations, arrays of structures and standard scalar or array binds.

Standard scalar and array binds are those binds which were previously possible using other OCI bind calls (OBNDRA, OBNDRN, and OBNDRV).

CURSOR
The CDA associated with the SQL statement or PL/SQL block being processed.

OPCODE
Piecewise bind: pass as 0.

Arrays of structures or standard bind: pass as 1.

SQLVAR
A character string holding the name of a placeholder (including the preceding colon, e.g., ":VARNAME") in the SQL statement being processed.

SQLVL
The length of the character string in SQLVAR, including the preceding colon. For example, the placeholder ":EMPLOYEE" has a length of nine.

PVCTX
Piecewise bind: A pointer to a context block entirely private to the application. This should be used by the application to store any information about the column being bound. One possible use would be to store a pointer to a file which will be referenced later. Each bind variable can then have its own separate file pointer. This pointer can be retrieved during a call to OGETPI.

Arrays of structures or standard bind: A pointer to a program variable or array of program variables from which input data will be retrieved when the SQL statement is executed. For arrays of structures this should point to the first scalar element in the array of structures being bound. This parameter is equivalent to the PROGV parameter from the older OCI bind calls.

PROGVL
Piecewise bind: This should be passed in as the maximum possible size of the data element of type FTYPE.

Arrays of structures or standard bind: This should be passed as the length in bytes of the datatype of the program variable, array element or the field in a structure which is being bound.

FTYPE
The external datatype code of the program variable being bound. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. See the section "External Datatypes" in Chapter 3 for a list of datatype codes, and the listings of *ocidem.h* and *ocidfn.h* in Appendix A for lists of constant definitions corresponding to datatype codes.

For piecewise operations, the valid datatype codes are 1 (VARCHAR2), 5 (STRING), 8 (LONG) and 24 (LONG RAW).

SCALE
Specifies the number of digits to the right of the decimal point for fields where FTYPE is 7 (PACKED DECIMAL). SCALE is ignored for all other types.

INDP
Pointer to an indicator variable or array of indicator variables. For arrays of structures this may be an interleaved array of column–level indicator variables. See page 2 – 29 for more information about indicator variables.

ALENP
Piecewise bind: pass as 0.

Arrays of structures or standard bind: A pointer to a variable or array containing the length of data elements being bound. For arrays of structures, this may be an interleaved array of column–level length variables. The maximum usable size of the array is determined by the *maxsiz* parameter.

RCODEP
Pointer to a variable or array of variables where column–level error codes are returned after a SQL statement is executed. For arrays of structures, this may be an interleaved array of column–level return code variables.

Typical error codes would indicate that data in PROGV has been truncated (ORA–01406) or that a null occurred on a SELECT or PL/SQL FETCH (ORA–01405).

PV–SKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of structures holding program variables being bound. In general, this value will be the size of one structure. If a standard array bind is being performed, this value should equal the size of one element of the array being bound.

IND–SKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of indicator variables associated with an array holding program data to be inserted. This parameter will either equal the size of one indicator parameter structure (for arrays of structures) or the size of one indicator variable (for standard array bind).

ALEN–SKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of data lengths associated with an array holding program data to be inserted. This parameter will either equal the size of one length variable structure (for arrays of structures) or the size of one length variable (for standard array bind).

rRC–SKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array used to store returned column–level error codes associated with the execution of a SQL statement. This parameter will either equal the size of one return code structure (for arrays of structures) or the size of one return code variable (for standard array bind).

MAXSIZ
The maximum size of an array being bound to a PL/SQL table. Values range from 1 to 32512, but the maximum size of the array depends on the datatype. The maximum array size is 32512 divided by the internal size of the datatype.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to zero for SQL scalar or array binds.

CURSIZ
A pointer to the actual number of elements in the array being bound to a PL/SQL table.

If PROGV is an IN parameter, set the CURSIZ parameter to the size of the array being bound. If PROGV is an OUT parameter, the number of valid elements being returned in the PROGV array is returned after PL/SQL block is executed.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to zero for SQL scalar or array binds.

FMT
A character string that contains the format specifier for a packed decimal variable. This optional parameter is only used when the type of the defined variable is PACKED DECIMAL (PIC S9(N)V9(N) COMP–3). The specifier has the form “mm.[+/–]nn”, where “mm” is the total number of digits, from 1 to 38, and “nn” is the number of decimal places, or scale. For example, “09.+02” would be the format specifier for an Oracle column of the internal type NUMBER(9,2). The plus or minus sign is required. If “+” is used, “nn” is the number of digits to the right of the decimal place. If “–” is specified, then “nn” is the power of ten by which the number is multiplied before it is placed in the output buffer.

If your compiler does not allow you to omit optional parameters, then pass the length (FMTL) parameter with a value of zero to indicate that there is no format specifier.

FMTL
The length of the format conversion specifier string. If zero, then FMT and FMTT are unused parameters.

FMTT
Specifies the format type of the conversion format string. The only value allowed is 7 (PACKED DECIMAL)

See Also OBNDRA, OBNDRN, OBNDRV, ODEFINPS, OGETPI, OSETPI.

OBNDRA

Purpose OBNDRA binds the address of a program variable or array to a placeholder in a SQL statement or PL/SQL block.

Syntax

```
CALL ”OBNDRA” USING CURSOR, SQLVAR, SQLVL,
      PROGV, PROGVL, FTYPE, [SCALE], [INDP],
      [ALEN], [ARCODE], [MAXSIZ], [CURSIZ],
      [FMT], [FMTL], [FMTT].
```

Comments You can use OBNDRA to bind scalar variables or arrays in your program to placeholders in a SQL statement or a PL/SQL block. The OBNDRA routine has a parameter, ALEN, that allows you to change the size of the bound variable without actually rebinding the variable.

Note: Applications running against a release 7.3 or later server that need to perform piecewise operations or utilize arrays of structures must use the OBINDPS routine instead of OBNDRA.

When you bind arrays in your program to PL/SQL tables, you must use OBNDRA, since this routine provides additional parameters that allow you to control the maximum size of the table, and to retrieve the current table size after the block has been executed.

Call OBNDRA after you call OPARSE to parse the statement containing the PL/SQL block and before calling OEXEC to execute it.

Once you have bound a program variable, you can change the value in the variable (PROGV) and the length of the variable (PROGVL) and re–execute the block without rebinding.

However, if you need to change the type of the variable, you must reparse and rebind before re–executing.

The following short but complete example program shows how you can use OBNDRA to bind tables in a COBOL program to tables in a PL/SQL block.

```
IDENTIFICATION DIVISION.
PROGRAM–ID.  OBNDRA–TEST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING–STORAGE SECTION.

01  LDA.
    03  LDA–V2RC             PIC S9(4) COMP.
    03  FILLER               PIC X(10).
    03  LDA–RC               PIC S9(4) COMP.
    03  FILLER               PIC X(50).
```

```
01  CURSOR.
    03  CURS–V2RC              PIC S9(4) COMP.
    03  CURS–TYPE              PIC S9(4) COMP.
    03  CURS–ROWS–PROCESSED    PIC S9(9) COMP.
    03  CURS–OFFS              PIC S9(4) COMP.
    03  CURS–FNC               PIC X.
    03  FILLER                 PIC X.
    03  CURS–RC                PIC S9(4) COMP.
    03  FILLER                 PIC X(50).

01  HOST–DATA–AREA             PIC X(512).
01  ERR–RC                     PIC S9(4) COMP.
01  MSGBUF                     PIC X(500).
01  MSGBUF–L                   PIC S9(9) COMP.

01  PART–UPDATE.
    03  DESCRIP                OCCURS 3 TIMES PIC X(20).
    03  PARTNOS                OCCURS 3 TIMES PIC S9(9) COMP.

01  BND–VARS.
    03  DESCRIP–ALEN           OCCURS 3 TIMES PIC S9(4) COMP.
    03  DESCRIP–RC             OCCURS 3 TIMES PIC S9(4) COMP.
    03  PARTNO–ALEN            OCCURS 3 TIMES PIC S9(4) COMP.
    03  PARTNO–RC              OCCURS 3 TIMES PIC S9(4) COMP.

01  DESCRIP–CS                 PIC S9(9) VALUE  3 COMP.
01  DESCRIP–MAX                PIC S9(9) VALUE  3 COMP.
01  DESCRIP–PH                 PIC X(5)  VALUE ":DESC".
01  DESCRIP–PH–L               PIC S9(9) VALUE  5 COMP.
01  DESCRIP–L                  PIC S9(9) VALUE 20 COMP.
01  PARTNO–CS                  PIC S9(9) VALUE  3 COMP.
01  PARTNO–MAX                 PIC S9(9) VALUE  3 COMP.
01  PARTNO–PH                  PIC X(6)  VALUE ":PARTS".
01  PARTNO–PH–L                PIC S9(9) VALUE  6 COMP.

01  USERNAME                   PIC X(11) VALUE "SCOTT".
01  USERNAME–L                 PIC S9(9) VALUE  5 COMP.
01  PASSWORD                   PIC X(5)  VALUE "TIGER".
01  PASSWORD–L                 PIC S9(9) VALUE  5 COMP.
01  CONN                       PIC S9(9) VALUE  0 COMP.
01  CONN–L                     PIC S9(9) VALUE  0 COMP.
01  CONN–MODE                  PIC S9(9) VALUE  0 COMP.
01  VARCHAR2–TYPE              PIC S9(9) VALUE  1 COMP.
01  INT–TYPE                   PIC S9(9) VALUE  3 COMP.
01  DESCRIP–LEN                PIC S9(9) VALUE 20 COMP.
01  INT–L                      PIC S9(9) VALUE  4 COMP.
01  MAX–TABLE                  PIC S9(9) VALUE 20 COMP.
01  ZERO–A                     PIC S9(9) VALUE  0 COMP.
01  VERSION–7                  PIC S9(9) VALUE  2 COMP.
```

```
01  MINUS–ONE                 PIC S9(9) VALUE –1 COMP.


01  DROP–TBL                  PIC X(20) VALUE
    ”DROP TABLE part_nos”.
01  DROP–TBL–L                PIC S9(9) VALUE 20 COMP.
01  CREATE–TBL                PIC X(100) VALUE
    ”CREATE TABLE part_nos
–         ”(partno NUMBER(8), description CHAR(20))”.
01  CREATE–TBL–L              PIC S9(9) VALUE 100 COMP.
01  CREATE–PKG                PIC X(256) VALUE
     ”CREATE PACKAGE update_parts AS
–    ”TYPE pnt IS TABLE OF NUMBER
–    ”INDEX BY BINARY_INTEGER;
–    ”TYPE pdt IS TABLE OF CHAR(20)
–    ”INDEX BY BINARY_INTEGER;
–    ”PROCEDURE add_parts (n IN INTEGER,
–    ”descrip IN pdt,
–    ”partno IN pnt);
–    ”END update_parts;”.

01  CREATE–PKG–L              PIC S9(9) VALUE 256 comp.
01  CREATE–PKG–BODY           PIC X(256) VALUE
     ”CREATE PACKAGE BODY update_parts AS
–    ”PROCEDURE add_parts (n IN INTEGER,
–    ”descrip IN pdt,
–    ”partno IN pnt);
–    ”BEGIN
–    ”FOR i IN 1..n LOOP
–    ”INSERT INTO part_nos
–    ”VALUES (partno(i), descrip(i));
–    ”END LOOP;
–    ”END;
–    ”END update_parts;”.

* PL/SQL anonymous block, calls update_parts
01  PLS–BLOCK                 PIC X(100) VALUE
    ”BEGIN add_parts(3, :DESC, :PARTS); END;”.
01  PLS–BLOCK–L               PIC S9(9) VALUE 100 COMP.

PROCEDURE DIVISION.
START–MAIN.
*  Connect to Oracle in non–blocking mode.
*  HDA must be initialized to zeros before call to OLOG.
    MOVE LOW–VALUES TO HOST–DATA–AREA.
    CALL ”OLOG” USING LDA, HOST–DATA–AREA, USERNAME,
          USERNAME–L, PASSWORD, PASSWORD–L, CONN,
          CONN–L, CONN–MODE.
    IF LDA–RC IN LDA NOT = 0
       PERFORM ORA–ERROR
       GO TO EXIT–STOP.

DISPLAY ” ”.
DISPLAY ”Connected to Oracle as user ”, USERNAME.
DISPLAY ” ”.

*  Open the cursor.
*  Use parameters PASSWORD, etc. for unused parameters.
    CALL ”OOPEN” USING CURSOR, LDA, PASSWORD, PASSWORD–L,
                 MINUS–ONE, PASSWORD, PASSWORD–L.
    IF CURS–RC IN CURSOR NOT = 0
       PERFORM ORA–ERROR
       GO TO EXIT–STOP.

*  Parse the drop table statement.
*  The statement is executed by OPARSE because
*  the DEFFLG parameter is zero.
CALL ”OPARSE” USING CURSOR, DROP–TBL, DROP–TBL–L,
                    ZERO–A, VERSION–7.
IF CURS–RC IN CURSOR NOT = 0 AND
    CURS–RC NOT = 942
    PERFORM ORA–ERROR
    GO TO EXIT–CLOSE.

*  Parse as well as execute the create table statement.
CALL ”OPARSE” USING CURSOR, CREATE–TBL, CREATE–TBL–L,
                    ZERO–A, VERSION–7.
IF CURS–RC IN CURSOR NOT = 0
    PERFORM ORA–ERROR
    GO TO EXIT–CLOSE.

*  Parse the PL/SQL block.
CALL ”OPARSE” USING CURSOR, PLS–BLOCK, PLS–BLOCK–L,
                    ZERO–A, VERSION–7.
IF CURS–RC IN CURSOR NOT = 0
    PERFORM ORA–ERROR
    GO TO EXIT–CLOSE.

*  Bind the two COBOL tables.
MOVE DESCRIP–L TO DESCRIP–ALEN(1).
MOVE DESCRIP–L TO DESCRIP–ALEN(2).
MOVE DESCRIP–L TO DESCRIP–ALEN(3).
CALL ”OBNDRA” USING CURSOR, DESCRIP–PH, DESCRIP–PH–L,
                    DESCRIP(1), DESCRIP–L, VARCHAR2–TYPE,
                    MINUS–ONE, ZERO–A, DESCRIP–ALEN(1),
                    DESCRIP–RC(1), MAX–TABLE, DESCRIP–CS,
                    ZERO–A, MINUS–ONE, MINUS–ONE.
```

```
IF CURS–RC IN CURSOR NOT = 0
    PERFORM ORA–ERROR
    GO TO EXIT–CLOSE.
MOVE INT–L TO PARTNO–ALEN(1).
MOVE INT–L TO PARTNO–ALEN(2).
MOVE INT–L TO PARTNO–ALEN(3).
CALL ”OBNDRA” USING CURSOR, PARTNO–PH, PARTNO–PH–L,
                    PARTNOS(1), INT–L, INT–TYPE,
                    MINUS–ONE, ZERO–A,
                    PARTNO–ALEN(1), PARTNO–RC(1),
                    MAX–TABLE, PARTNO–CS,
                    ZERO–A, MINUS–ONE, MINUS–ONE.
IF CURS–RC IN CURSOR NOT = 0
    PERFORM ORA–ERROR
    GO TO EXIT–CLOSE.

*  Execute the PL/SQL block, calling update_parts.
CALL ”OEXEC” USING CURSOR.
IF CURS–RC IN CURSOR NOT = 0
    PERFORM ORA–ERROR
    GO TO EXIT–CLOSE.

EXIT–CLOSE.
    CALL ”OCLOSE” USING CURSOR.
    CALL ”OLOGOF” USING LDA.
EXIT–STOP.
    STOP RUN.

ORA–ERROR.
IF LDA–RC IN LDA NOT = 0
    DISPLAY ”OLOGON error”
    MOVE LDA–RC TO ERR–RC
    MOVE ”0” TO CURS–FNC
ELSE IF CURS–RC IN CURSOR NOT = 0
    MOVE CURS–RC IN CURSOR TO ERR–RC
DISPLAY ”Oracle error. Code is ”, ERR–RC WITH CONVERSION,
        ” Function is ”, CURS–FNC WITH CONVERSION.
CALL ”OERHMS” USING LDA, ERR–RC, MSGBUF, MSGBUF–L.
DISPLAY MSGBUF.
```

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
SQLVAR	PIC X(n)	IN
SQLVL	PIC S9(9) COMP	IN
PROGV	(Address) (1)	IN / OUT(2)
PROGVL	PIC S9(9) COMP	IN
FTYPE	PIC S9(9) COMP	IN
SCALE	PIC S9(9) COMP	IN
INDP	PIC S9(4) COMP	IN / OUT(2)
ALEN	PIC S9(4) COMP	IN / OUT
ARCODE	PIC S9(4) COMP	OUT(3)
MAXSIZ	PIC S9(9) COMP	IN
CURSIZ	PIC S9(9) COMP	IN / OUT(2)
FMT	PIC X(6)	IN
FMTL	PIC S9(9) COMP	IN
FMTT	PIC S9(9) COMP	IN

Note 1. PROGV is the address of the program variable.

Note 2. IN / OUT parameter on the execute call.

Note 3. Value returned; OUT parameter on the execute call.

CURSOR
A cursor data area within the program.

SQLVAR
A character string containing the name of a placeholder (including the preceding colon) in the SQL statement.

SQLVL
The length of the character string SQLVAR (including the preceding colon). For example, the placeholder :DEPT has a length of 5.

PROGV
The address of a program variable or table of program variables from which data will be retrieved when OEXEC is issued.

PROGVL
The length in bytes of the program variable or array element. Since OBNDRA might be called only once for many different PROGV values on successive execute calls, PROGVL must contain the maximum length of PROGV.

> **Note:** The PROGVL parameter is a PIC S(9) COMP. However, on some systems the underlying parameter type in the OCI library might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this appears to limit the maximum length of the buffer to 64K bytes. To bind a longer buffer for these datatypes, move –1 to PROGVL and pass the actual data area length (total buffer length – 4) in the first four bytes of PROGV. Set this value before calling OBNDRA.

FTYPE
The external datatype of the program variable as defined within the user program. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. A list of external datatypes and type codes is in Chapter 3.

SCALE
Specifies the number of digits to the right of the decimal point for fields where FTYPE is 7 (PACKED DECIMAL). SCALE is ignored for all other types.

INDP
An indicator parameter, or a table of indicator parameters if PROGV is a table. As a table, INDP must contain at least the same number of elements as the PROGV table.

See Chapter 2 for more information about indicator variables.

ALEN
A table containing the length of the data. This is the effective length in bytes of the bind variable element, not the size of the table containing the elements. For example, if PROGV is a table declared as

```
03  DESCRIP           OCCURS 3 TIMES PIC X(20).
```

then ALEN must refer to a table of at least three elements.

If DESCRIP in the above example is an IN parameter, each element in the table indicated by ALEN should be set to the length of the data in the corresponding element of the DESCRIP table (20 in this example) before the execute call.

If DESCRIP in the above example is an OUT parameter, the length of the returned data appears in the table indicated by ALEN after the SQL statement or PL/SQL procedure is executed.

Once the bind is done using OBNDRA, you can change the length of the bind variable without rebinding. However, the length cannot be greater than that specified in ALEN.

ARCODE
The column–level error return code. This parameter will contain the error code for the bind variable after the execute call. The error codes that can be returned in ARCODE are those that indicate that data in the returned PROGV has been truncated or that a null occurred in the column, for example, ORA–01405 or ORA–01406.

If OBNDRA is being used to bind a table (that is, PROGV is a table), then ARCODE must also be a table of at least equal size.

MAXSIZ
The maximum size for the array being bound. Values range from 1 to 32767. If OBNDRA is being used to bind a scalar, set this parameter to zero.

CURSIZ
The current size of the array.

If PROGV is an IN parameter, set the value of CURSIZ to the size of the table being bound. If PROGV is an OUT parameter, then the number of valid elements being returned in the PROGV table is returned in CURSIZ after the SQL statement or PL/SQL block is executed.

To use OBNDRA to bind a scalar, you must be able to pass a zero *by value* in this parameter. If your COBOL compiler does not have a mechanism for passing parameters by value, you must use OBNDRV to bind scalars.

FMT
The address of a character string that contains the format specifier for a packed decimal variable. This optional parameter is only used when the type of the bind variable is PACKED DECIMAL (PIC S9(N)V9(N) COMP–3) (datatype 7). The specifier has the form “mm.[+/–]nn”, where “mm” is the total number of digits, from 1 to 38, and “nn” is the number of decimal places, or scale. For example, “09.+02” would be the format specifier for an Oracle column of the internal type NUMBER(9,2). The plus or minus sign is required. If “+” is used, “nn” is the number of digits to the right of the decimal place. If “–” is specified, then “nn” is the power of ten by which the number is multiplied before it is placed in the output buffer.

When this parameter is not used and your compiler does not allow you to omit optional parameters, then pass the length (FMTL) parameter with a value of zero to indicate that there is no format specifier.

FMTL
The length of the format conversion specifier string. If zero, then FMT and FMTT become unused parameters.

FMTT
Specifies the format type of the conversion format string. The only value allowed is 7 (PACKED DECIMAL).

See Also OBINDPS, OBNDRN, OBNDRV, OEXEC, OEXN, OPARSE.

OBNDRN OBNDRV

Purpose OBNDRN and OBNDRV associate the address of a program variable, PROGVAR, with the specified placeholder in the SQL statement. The placeholder is identified by name (SQLVAR) for the OBNDRV routine and by number for OBNDRN. OEXEC then uses these addresses to assign values to the placeholders when executing the SQL statement.

Syntax

```
CALL ”OBNDRN” USING CURSOR, SQLVN, PROGV, PROGVL,
      FTYPE, [SCALE], [INDP], [FMT], [FMTL], [FMTT].
```

```
CALL ”OBNDRV” USING CURSOR, SQLVAR, SQLVL,
      PROGV, PROGVL, FTYPE, [SCALE], [INDP],
      [FMT], [FMTL], [FMTT].
```

Comments You can call either OBNDRV or OBNDRN to bind the address of a variable in your program to a placeholder in the SQL statement being processed. If an application needs to perform piecewise operations or utilize arrays of structures, you must bind your variables using OBINDPS instead.

The placeholder in the SQL statement is a SQL identifier. It must not be an Oracle reserved word and must be preceded by a colon (:) in the SQL statement. For example, the following SQL statement

```
SELECT ename,sal,com FROM emp WHERE deptno = :DEPT AND
   comm > :MIN_COM
```

has two placeholders, :DEPT and :MIN_COM.

OBNDRV and OBNDRN differ only in the way they specify the placeholder. The OBNDRV routine specifies the placeholder in the SQL statement symbolically by name. For example, an OBNDRV call that binds the :DEPT placeholder in the SQL statement above to the program variable DEPT–NUM is

```
DATA DIVISION.
77   DEPT–NUM    PIC S9(9) COMP.
77   INT–TYPE    PIC S9(9) VALUE 3 COMP.
77   PH          PIC X(5)  VALUE ”:DEPT”.
77   PH–L        PIC S9(9) VALUE 5 COMP.
77   MIN–COM     PIC S9(9) COMP.
77   INT–LEN     PIC S9(9) VALUE 4 COMP.
77   MINUS–1S    PIC S9(4) VALUE –1 COMP.
77   MINUS–1L    PIC S9(9) VALUE –1 COMP.
...
PROCEDURE DIVISION.
...
CALL ”OBNDRV” USING CURSOR, PH, PH–L, DEPT–NUM, INT–LEN,
    INT–TYPE, MINUS–1L, MINUS–1S, MINUS–1L, MINUS–1L, MINUS–1L.
```

The OBNDRN routine is used to bind the program variables to placeholders that are entered in the SQL statement as numbers preceded by a colon; for example:

```
SELECT ename,sal,comm FROM emp WHERE deptno = :1 AND
    comm > :2
```

When OBNDRN is called, the parameter SQLVN identifies the placeholder by number. If SQLVN is set to 1, the program variable is bound to the placeholder :1. For example, OBNDRN is called to bind the program variable MIN–COM to the placeholder :2 in the SQL statement above as follows:

```
...
PROCEDURE DIVISION.
...
CALL ”OBNDRN” USING CURSOR, PH–2, MIN–COM, INT–LEN, INT–TYPE,
    MINUS–1L, MINUS–1S, MINUS–1L, MINUS–1L, MINUS–1L.
```

where the placeholder “:2” is indicated in the SQLVARNUM parameter by the value 2.

> **Note:** When using OBNDRN, the placeholder is :N, where N is greater than or equal to 1 and not greater than 255.

In a PL/SQL block, you cannot use OBNDRN to bind program variables to placeholders, since PL/SQL does not recognize numbered placeholders. Always use OBNDRV and named placeholders in PL/SQL blocks.

The OBNDRV or OBNDRN routine must be called *after* you call OPARSE to parse the SQL statement and *before* calling OEXEC to execute it.

If the values of the program variables change, you do not need to rebind using these routines before re–executing, since it is the address of the variables that is bound. However, if you change the actual program variable, you must rebind before re–executing.

For example, if you have bound the address of DEPT–NUM to the placeholder :DEPT, and you now want to use NEW–DEPT–NUM when executing the SQL statement above, you must call OBNDRV again to bind the new program variable address to the placeholder.

Also, you cannot in general rebind a placeholder to a variable of different type without reparsing the SQL statement. So, if you need to rebind with a different variable type, call OPARSE first to reparse the statement.

In general, OBNDRV and OBNDRN are not supported after an ODESCR call. You must issue an OPARSE call after an ODESCR call before binding any remaining placeholders.

At the time of the bind, Oracle stores the address of the program variable. If the same placeholder name occurs more than once in the SQL statement, a single call to OBNDRV or OBNDRN will bind all occurrences of the placeholder. The completion status of the bind is returned in the *return code* field of the cursor data area. A return code of zero indicates successful completion.

If your program is linked using the deferred mode option, bind errors that would normally be returned immediately are not detected until the bind operation is actually performed. This happens on the first describe (ODESCR) or execute (OEXEC, OEXN, or OEXFET) call after the bind.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
SQLVAR	PIC X(n)	IN
SQLVL	PIC S9(9) COMP	IN
SQLVN	PIC S9(9) COMP	IN
PROGV	(Address)	IN (1)
PROGVL	PIC S9(9) COMP	IN
FTYPE	PIC S9(9) COMP	IN
SCALE	PIC S9(9) COMP	IN
INDP	PIC S9(4) COMP	IN (1)
FMT	PIC X(6)	IN
FMTL	PIC S9(9) COMP	IN
FMTT	PIC S9(9) COMP	IN

Note 1. This is an IN parameter for the OEXEC routine.

CURSOR
A cursor data area within the program.

SQLVAR
Used only with OBNDRV, this parameter specifies the address of a character string containing the name of a placeholder (including the preceding colon) in the SQL statement.

SQLVL
Used only with OBNDRV, the SQLVL parameter is the length of the character string SQLVAR (including the preceding colon). For example, the placeholder :DEPT has a length of 5.

SQLVN
Used only with OBNDRN, this parameter specifies a placeholder by number in the SQL statement referenced by the cursor. For example, if SQLVN is 2, it refers to all placeholders identified by :2 within the SQL statement.

PROGV
The address of a program variable or table of program variables from which data will be retrieved when OEXEC or OEXN (for tables) is executed.

PROGVL
The length in bytes of the program variable. Since OBNDRV or OBNDRN might be called only once for many different PROGV values on successive OEXEC calls, PROGVL must contain the maximum length of PROGV.

> **Note:** The PROGVL parameter is a PIC S(9) COMP. However, on some systems the underlying parameter type in the OCI library might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this appears to limit the maximum length of the buffer to 64K bytes. To bind a longer buffer for these datatypes, move –1 to PROGVL, and pass the actual data area length (total buffer length – 4) in the first four bytes of PROGV. Set this value before calling OBNDRN or OBNDRV.

FTYPE
The external datatype of the program variable as defined within the user program. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. A list of external datatypes and type codes is in Chapter 3.

SCALE
Specifies the number of digits to the right of the decimal point for fields where FTYPE is 7 (PACKED DECIMAL). SCALE is ignored for all other types.

INDP
Indicates whether the column value should be set to null. If this parameter contains a negative value when OEXEC, OEXN, or OEXFET is called, the column is set to null; otherwise, it is set to the value in the PROGVAR parameter.

> **Note:** If OBNDRV or OBNDRN is being used to bind a table of elements (that is, if PROGV is a table), then the INDP parameter must also be a table of indicator parameters at least as large as the PROGV table.

FMT
The address of a character string that contains the format specifier for a packed decimal variable. This optional parameter is only used when the type of the bind variable is PACKED DECIMAL (PIC S9(N)V9(N) COMP–3) (datatype 7). The specifier has the form "mm.[+/–]nn", where "mm"is the total number of digits, from 1 to 38, and "nn" is the number of decimal places, or scale. For example, "09.+02" would be the format specifier for an Oracle column of the internal type NUMBER(9,2). The plus or minus sign is required. If "+" is used, "nn" is the number of digits to the right of the decimal place. If "–" is specified, then "nn" is the power of ten by which the number is multiplied before it is placed in the output buffer.

When this parameter is not used, and your compiler does not allow you to omit optional parameters, then pass the length (FMTL) parameter with a value of zero to indicate that there is no format specifier.

FMTL
The length of the format conversion specifier string. If zero, then FMT and FMTT become unused parameters.

FMTT
Specifies the format type of the conversion format string. The only value allowed is 7 (PACKED DECIMAL).

See Also OBINDPS, OBNDRA, ODESCR, OEXEC, OEXFET, OEXN, OPARSE.

OBREAK

Purpose

OBREAK performs an immediate (asynchronous) abort of any currently executing OCI routine that is associated with the specified LDA. It is normally used to stop a long–running execute or fetch call that has not yet completed.

Syntax

```
CALL ”OBREAK” USING LDA.
```

Comments

If no OCI routine is active when OBREAK is called, OBREAK will be ignored unless the next OCI routine called is OFETCH. In this case, the subsequent OFETCH call will be aborted.

OBREAK is the *only* OCI routine that you can call when another OCI routine is in progress. It should not be used when a logon (OLOG) is in progress, since the LDA is in an indeterminate state. The OBREAK routine cannot return a reliable error status to the LDA, since it might be called when the Oracle internal status structures are in an inconsistent state.

> **Note:** *obreak()* aborts the currently executing OCI function not the connection.

OBREAK is not guaranteed to work on all operating systems and does not work on all protocols. In some cases, OBREAK may work with one protocol on an operating system, but may not work with other protocols on the same operating system.

See the description of *obreak()* in Chapter 4 for a code example in C which runs under most UNIX operating systems..

Parameter

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also

OLOG.

OCAN

Purpose OCAN cancels a query after the desired number of rows have been fetched.

Syntax

```
CALL ″OCAN″ USING CURSOR.
```

Comments OCAN informs Oracle that the operation in progress for the specified cursor is complete. The OCAN function thus frees any resources associated with the specified cursor, but keeps the cursor associated with its parsed representation in the shared SQL area.

For example, if you require only the first row of a multi–row query, you can call OCAN after the first OFETCH operation to inform Oracle that your program will not perform additional fetches.

If you use the OEXFET function to fetch your data, specifying a non–zero value for the OEXFET CANCELparameter has the same effect as calling OCAN after the fetch completes.

Parameter

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT

CURSOR
The address of the cursor data area specified in the OPARSE call associated with the query.

See Also OEXFET, OFEN, OFETCH, OPARSE.

OCLOSE

Purpose OCLOSE disconnects a cursor from the data areas associated with it in the Oracle Server.

Syntax

```
CALL ”OCLOSE” USING CURSOR.
```

Comments The OCLOSE routine frees all resources obtained by the OOPEN, OPARSE, and OEXEC operations using this cursor. If OCLOSE fails, the *return code* field of the cursor data area contains the error code.

Parameter

Parameter Name	***Type***	***Mode***
CURSOR	(Address)	IN/OUT

CURSOR
The cursor data area specified in the OOPEN call.

See Also OOPEN, OPARSE.

OCOF

Purpose OCOF disables *autocommit*, that is, automatic commit of every SQL data manipulation statement.

Syntax

```
CALL ″OCOF″ USING LDA.
```

Comments By default, autocommit is already disabled at the start of an OCI program. Having autocommit ON can have a serious impact on performance. So, if the OCON (autocommit ON) routine is used to enable autocommit for some special circumstance, OCOF should be used to disable autocommit as soon as it is practical.

If OCOF fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCOM, OCON, OLOG.

OCOM

Purpose OCOM commits the current transaction.

Syntax

```
CALL ”OCOM” USING LDA.
```

Comments The current transaction starts from the OLOG call or the last OROL or OCOM call, and lasts until an OCOM, OROL, or OLOGOF call is issued.

If OCOM fails, the reason is indicated in the *return code* field of the LDA.

Do not confuse the OCOM call (COMMIT) with the OCON call (turn autocommit ON).

Parameter

Parameter Name	*Type*	*Mode*
LDA	(Address)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCON, OLOG, OLOGOF, OROL.

OCON

Purpose OCON enables *autocommit*, that is, automatic commit of every SQL data manipulation statement.

Syntax

```
CALL ″OCON″ USING LDA.
```

Comments By default, autocommit is disabled at the start of an OCI program. This is because it is more expensive and less flexible than placing OCOM calls after each logical transaction. When autocommit is on, a zero in the *return code* field after calling OEXEC indicates that the transaction has been committed.

If OCON fails, the reason is indicated in the *return code* field of the LDA.

if it becomes necessary to turn autocommit on for some special circumstance, it is advisable to follow that with a call to OCOF to disable autocommit as soon as it is practical in order to maximize performance.

Do not confuse the OCON routine with the OCOM (COMMIT) routine.

Parameter

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCOF, OCOM, OLOG.

ODEFIN

Purpose

ODEFIN defines an output buffer for a specified select–list item in a SQL query.

Syntax

```
CALL ”ODEFIN” USING CURSOR, POS, BUF, BUFL, FTYPE,
    [SCALE], [INDP], [FMT], [FMTL], [FMTT],
    [RLEN], [RCODE].
```

Comments

An OCI program must call ODEFIN once for each select–list item in the SQL statement. Each call to ODEFIN associates an output variable in the program with a select–list item of the query. The output variable must be a scalar or a character string and must be compatible with the external datatype specified in the FTYPE parameter. See Table 3 – 2 for a list of datatypes and compatible variables. For use with OEXFET or OFEN, the output variable can be a table of scalars, or strings.

Oracle places data in the output variables when the program calls OEXFET, OFEN, or OFETCH.

If you do not know the lengths and datatypes of the select–list items, you can obtain this information by calling ODESCR before calling ODEFIN.

Call ODEFIN parsing the SQL statement. Call ODEFIN before calling the fetch routine (OEXFET, OFEN, or OFETCH).

> **Note:** Applications running against a release 7.3 or later server that need to perform piecewise operations or utilize arrays of structures must use the newer ODEFINPS routine instead of ODEFIN.

ODEFIN associates output variables with select–list items using the position index of the select–list item in the SQL statement. Position indices start at 1 for the first (or left–most) select–list item. For example, in the SQL statement

```
SELECT ename, empno, sal FROM emp WHERE sal > :MIN_SAL
```

the select–list item SAL is in position 3, EMPNO is in position 2, and ENAME is in position 1.

If the type or length of bound variables changes between queries, you must reparse and rebind before re–executing.

You call ODEFIN to associate output buffers with the select–list items in the above statement as follows:

```
DATA DIVISION.
77   EMP–NAME    PIC X(10).
77   EMP–NAME–L  PIC S9(9) VALUE 10  COMP.
77   CHAR–TYPE   PIC S9(9) VALUE  1  COMP.
77   EMP–NUM     PIC S9(9) COMP.
77   EMP–NUM–L   PIC S9(9) VALUE  4  COMP.
77   INT–TYPE    PIC S9(9) VALUE  3  COMP.
77   SAL         PIC S9(4)V9(2)      COMP–3.
77   SAL–L       PIC S9(9) VALUE  4  COMP.
77   INDP        PIC S9(4) VALUE  0  COMP.
77   FMT         PIC X(6)  VALUE ”08.+02”.
77   FMT–L       PIC S9(9) VALUE  6  COMP.
77   FMT–T       PIC S9(9) VALUE  7  COMP.
77   RET–LEN1    PIC S9(4) COMP.
77   RET–CODE1   PIC S9(4) COMP.
77   RET–LEN2    PIC S9(4) COMP.
77   RET–CODE2   PIC S9(4) COMP.
77   RET–LEN3    PIC S9(4) COMP.
77   RET–CODE3   PIC S9(4) COMP.
77   ZERO–ARG    PIC S9(9) VALUE 0 COMP.
77   ONE         PIC S9(9) VALUE 1 COMP.
77   TWO         PIC S9(9) VALUE 2 COMP.
77   THREE       PIC S9(9) VALUE 3 COMP.
77   SCALE       PIC S9(9) VALUE 7 COMP.
...
PROCEDURE DIVISION.
...
CALL ”ODEFIN” USING CURSOR, ONE, EMP–NAME, EMP–NAME–L, CHAR–TYPE,
    ZERO, INDP, FMT, ZERO–ARG, FMT–T, RET–LEN1, RET–CODE1.
CALL ”ODEFIN” USING CURSOR, TWO, EMP–NUM, EMP–NUM–L, INT–TYPE,
    ZERO, INDP, FMT, ZERO–ARG, FMT–T, RET–LEN2, RET–CODE2.
CALL ”ODEFIN” USING CURSOR, THREE, SAL, SAL–L, FMT–T,
    TWO, INDP, FMT, FMT–L, FMT–T, RET–LEN3, RET–CODE3.
```

Oracle provides return code information at the row level using the *return code* field in the cursor data area. If you require return code information at the column level, you must define the optional RCODE parameter, as in the examples above.

During each fetch, Oracle sets the associated RCODE for each select–list item processed. This code contains actual error codes and indicates either successful completion (0) or an exceptional condition, such as a null item fetched, the item fetched was truncated, or other non–fatal column errors.

The following codes can be returned in the RCODE parameter:

Code	***Meaning***
0	Success.
1405	A NULL value was fetched.
1406	ASCII or string buffer data was truncated. The converted data from the database did not fit into the buffer. Check the value in INDP, if specified, or RLEN to determine the original length of the data.
1454	Invalid conversion specified: integers not of length 1,2, or 4; reals not of length 4 or 8; invalid packed decimal conversions; packed decimal with more than 38 digits specified.
1456	Real overflow. Conversion of a database column or expression would overflow a real number of this machine.
3115	Unsupported datatype.

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	(Address)	IN / OUT
POS	PIC S9(9) COMP	IN
BUF	(Address)	IN (1)
BUFL	PIC S9(9) COMP	IN
FTYPE	PIC S9(9) COMP	IN
SCALE	PIC S9(9) COMP	IN
INDP	PIC S9(4) COMP	IN (1)
FMT	PIC X(6)	IN
FMTL	PIC S9(9) COMP	IN
FMTT	PIC S9(9) COMP	IN
RLEN	PIC S9(4) COMP	IN (1)
RCODE	PIC S9(4) COMP	IN (1)

Note 1. Values for these parameters are valid only after the subsequent OEXFET, OFEN, or OFETCH routine returns. These are effectively OUT parameters for those routines.

CURSOR
The cursor data area specified in the OPARSE call for the SQL statement being defined.

POS
An index for a select–list item in the query. Position indices start at 1 for the first (or left–most) select–list item. The ODEFIN routine uses the position index to associate output buffers with a given select–list item. If you specify a position index greater than the number of items in the select–list or less than 1, ODEFIN returns a “variable not in select list” error in the *return code* field of the cursor data area.

BUF
The address of the variable in the user program that receives the data when OFETCH or OFEN is executed. The variable can be of any type into which an Oracle column or expression result can be converted. See Chapter 3 for more information on datatype conversions.

> **Note:** If ODEFIN is being called to define a table fetch operation using the OFEN routine, then the BUFFER parameter must be a table large enough to hold the set of items that will be fetched.

BUFL
The length in bytes of the buffer being defined. If BUF is a table, this is the size in bytes of one element of the table.

> **Note:** The BUFL parameter is a PIC S(9) COMP. However, on some systems the underlying parameter type in the OCI library might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this appears to limit the maximum length of the buffer to 64K bytes. To bind a longer buffer for these datatypes, move –1 to BUFL and pass the actual data area length (total buffer length – 4) in the first four bytes of BUF. Set this value before calling ODEFIN.

FTYPE
The code for the external datatype to which the select–list item is to be converted before it is moved to the output variable. A list of the external datatypes and type codes is in Chapter 3.

SCALE
Specifies the number of digits to the right of the decimal point to be returned for items of FTYPE = 7 (PACKED DECIMAL). Ignored for all other datatypes.

INDP
The value of INDP after OEXFET, OFETCH, or OFEN is executed indicates whether the select–list item fetched was null, truncated, or not altered. See the section "Indicator Variables" in Chapter 2 for more information.

> **Note:** The INDP parameter contains a subset of the functionality provided by the RLEN and RCODE parameters.

FMT
A character string that contains the format specifier for a packed decimal variable. This optional parameter is only used when the type of the defined variable is PACKED DECIMAL (PIC S9(N)V9(N) COMP–3). The specifier has the form "mm.[+/–]nn", where "mm" is the total number of digits, from 1 to 38, and "nn" is the number of decimal places, or scale. For example, "09.+02" would be the format specifier for an Oracle column of the internal type NUMBER(9,2). The plus or minus sign is required. If "+" is used, "nn" is the number of digits to the right of the decimal place. If "–" is specified, then "nn" is the power of ten by which the number is multiplied before it is placed in the output buffer.

If your compiler does not allow you to omit optional parameters, then pass the length (FMTL) parameter with a value of zero to indicate that there is no format specifier.

FMTL
The length of the format conversion specifier string. If zero, then FMT and FMTT are unused parameters.

FMTT
Specifies the format type of the conversion format string. The only value allowed is 7 (PACKED DECIMAL)

RLEN
A PIC S9(4) variable or table. Oracle places the actual length of the returned column in this variable after a fetch is performed. If ODEFIN is being used to associate a table with a select–list item, the RLEN parameter must also be a table of PIC S9(4) variables of the same size as the BUF table. Return lengths are valid after the fetch.

RCODE
A PIC S9(4) variable or table. Oracle places the column return code in this variable after the fetch is performed. If ODEFIN is being used to associate a table with a select–list item, the RCODE parameter must also be a table of PIC S9(4) variables of the same size as the BUF table.

See Also ODEFINPS, ODESCR, OEXFET, OFEN, OFETCH, OPARSE.

ODEFINPS

Purpose ODEFINPS defines an output variable for a specified select–list item in a SQL query. This call can also specify if an operation will be performed piecewise or with arrays of structures.

Syntax

```
CALL ”ODEFINPS” USING CURSOR, OPCODE, POS, BUFCTX, BUFL,
FTYPE [SCALE], [INDP], [FMT], [FMTL], [FMTT], [RLENP],
[RCODEP], BUF–SKIP, IND–SKIP, LEN–SKIP, RC–SKIP.
```

Comments ODEFINPS is used to define an output variable for a specified select–list item in a SQL query. Additionally, it can indicate that an application will be fetching data incrementally at runtime. This piecewise fetch is designated in the OPCODE parameter. ODEFINPS is also used when an application will be fetching data into an array of structures.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

With the introduction of ODEFINPS there are now two fully–supported calls for binding input parameters, the other being the older ODEFIN. Application developers should consider the following points when determining which define call to use:

- ODEFINPS is supported only when a program is linked in deferred mode. If it is necessary to link in non–deferred mode, ODEFIN must be used. In this case, the ability to handle piecewise operations and arrays of structures is not supported.
- ODEFINPS is more complex than the older bind call. Users who are not performing piecewise operations and are not using arrays of structures may choose to use ODEFIN.

Unlike older OCI calls, ODEFINPS does not accept –1 for any optional or unused parameters. When it is necessary to pass a value to these parameters NULL or 0 should be used instead.

See the sections “Piecewise Insert, Update and Fetch,” and “Arrays of Structures” in Chapter 2 for more information about piecewise operations, arrays of structures, skip parameters and the ODEFINPS call.

See the description of *odefinps()* on page 4 – 40 for a sample C language program demonstrating the use of ODEFINPS.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
OPCODE	PIC S9(2) COMP	IN
POS	PIC S9(9) COMP	IN
BUFCTX	PIC S9(2) COMP	IN
BUFL	PIC S9(9) COMP	IN
FTYPE	PIC S9(9) COMP	IN
SCALE	PIC S9(9) COMP	IN
INDP	PIC S9(4) COMP	IN
FMT	PIC X(6)	IN
FMTL	PIC S9(9) COMP	IN
FMTT	PIC S9(9) COMP	IN
RLENP	PIC S9(4) COMP	OUT
RCODEP	PIC S9(4) COMP	IN
BUF–SKIP	PIC S9(9) COMP	IN
IND–SKIP	PIC S9(9) COMP	IN
LEN–SKIP	PIC S9(9) COMP	IN
RC–SKIP	PIC S9(9) COMP	IN

Note: Since the ODEFINPS call can be used in a variety of different circumstances, some items in the following list of parameter descriptions include different explanations for how the parameter is used for piecewise operations, arrays of structures and standard scalar or array binds.

Standard scalar and array defines are those defines which were previously possible using ODEFIN.

CURSOR
The CDA associated with the SELECT statement being processed.

OPCODE
Piecewise define: pass as 0.

Arrays of structures or standard define: pass as 1.

POS
An index for the select–list column which needs to be defined. Position indices start from 1 for the first, or left–most, item of the query. The ODEFINPS function uses the position index to associate output

variables with a given select–list item. If you specify a position index greater than the number of items in the select–list, or less than 1, the behavior of ODEFINPS is undefined.

If you do not know the number of items in the select list, use the ODESCR routine to determine it. See the second sample program in Appendix B for an example that does this.

BUFCTX
Piecewise define: A pointer to a context block entirely private to the application. This should be used by the application to store any information about the column being defined. One possible use would be to store a pointer to a file which will be referenced later. Each output variable can then have its own separate file pointer. The pointer can be retrieved by the application during a call to OGETPI.

Array of structures or standard define: This specifies a pointer to the program variable or the beginning of an array of program variables or structures into which the column being defined will be placed when the fetch is performed. This parameter is equivalent to the BUF parameter of the ODEFIN call.

BUFL
Piecewise define: The maximum possible size of the column being defined.

Array of structures or standard define: The length (in bytes) of the variable pointed to by BUFCTX into which the column being defined will be placed when a fetch is performed. For an array define, this should be the length of the first scalar element of the array of variables or structures pointed to by BUFCTX.

FTYPE
The external datatype to which the select–list item is to be converted before it is moved to the output variable. A list of the external datatypes and datatype codes can be found in the "External Datatypes" section in Chapter 3.

For piecewise operations, the valid datatype codes are 1 (VARCHAR2), 5 (STRING), 8 (LONG) and 24 (LONG RAW).

SCALE
Specifies the number of digits to the right of the decimal point for fields where FTYPE is 7 (PACKED DECIMAL). SCALE is ignored for all other types.

INDP
A pointer to an indicator variable or an array of indicator variables. If arrays of structures are used, this points to a possibly interleaved array of indicator variables.

FMT
A character string that contains the format specifier for a packed decimal variable. This optional parameter is only used when the type of the defined variable is PACKED DECIMAL (PIC S9(N)V9(N) COMP–3). The specifier has the form “mm.[+/–]nn”, where “mm” is the total number of digits, from 1 to 38, and “nn” is the number of decimal places, or scale. For example, “09.+02” would be the format specifier for an Oracle column of the internal type NUMBER(9,2). The plus or minus sign is required. If “+” is used, “nn” is the number of digits to the right of the decimal place. If “–” is specified, then “nn” is the power of ten by which the number is multiplied before it is placed in the output buffer

.If your compiler does not allow you to omit optional parameters, then pass the length (FMTL) parameter with a value of zero to indicate that there is no format specifier.

FMTL
The length of the format conversion specifier string. If zero, then FMT and FMTT are unused parameters.

FMTT
Specifies the format type of the conversion format string. The only value allowed is 7 (PACKED DECIMAL)

RLENP
An element or array of elements which will hold the length of a column or columns after a fetch is done. If arrays of structures are used, this points to a possibly interleaved array of length elements.

RCODEP
An element or array of elements which will hold column–level error codes which are returned by a fetch. If arrays of structures are used, this points to a possibly interleaved array of return code elements.

BUF–SKIP
Piecewise define or standard scalar define: pass as 0.

Array of structures or standard array define: this is the skip parameter which specifies the number of bytes to be skipped in order to get to the next program variable element in the array being defined. In general, this will be the size of one program variable for a standard array define, or the size of one structure for an array of structures.

IND–SKIP
Piecewise define or standard scalar define: pass as 0.

Array of structures or standard array define: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next indicator variable in the possibly interleaved array of indicator variables pointed to by INDP. In general, this will be the size of one indicator variable for a standard array define, and the size of one indicator variable structure for arrays of structures.

LEN–SKIP
Piecewise define or standard define: pass as 0.

Array of structures: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next column length in the possibly interleaved array of column lengths pointed to by RLENP. In general, this will be the size of one length variable for a standard array define, and the size of one length variable structure for arrays of structures.

RC–SKIP
Piecewise define or standard define: pass as 0.

Array of structures: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next return code structure in the possibly interleaved array of return codes pointed to by RCODEP. In general, this will be the size of one return code variable for a standard array define, and the size of one length variable structure for arrays of structures.

See Also OBINDPS, ODEFIN, OGETPI, OSETPI.

ODESCR

Purpose

ODESCR describes select–list items for dynamic SQL queries. The ODESCR procedure returns internal datatype and size information for a specified select–list item.

Syntax

```
CALL ”ODESCR” USING CURSOR, POS, DBSIZE, [DBTYPE],
    [CBUF], [CBUFL], [DSIZE], [PREC], [SCALE],
    [NULLOK].
```

Comments

You can call ODESCR after an OPARSE call to obtain the following information for each select–list item:

- maximum size of a column name (DBSIZE)
- internal datatype code (DBTYPE)
- column name (CBUF)
- length in bytes of the column name (CBUFL)
- maximum display size (DSIZE)
- precision of a numeric (PREC)
- scale of a numeric (SCALE)
- null status of the column (NULLOK)

This routine is used for interactive or dynamic SQL queries, that is, queries in which the number of select–list items, as well as their datatypes and sizes, might not be known until runtime.

The *return code* field of the cursor data area indicates success (zero) or failure (non–zero) of the ODESCR call.

The ODESCR routine uses a position index to refer to select–list items in the SQL query statement. For example, the SQL statement

```
SELECT ename, sal FROM emp WHERE sal > :MIN_SAL
```

contains two select–list items: ENAME and SAL. The position index of SAL is 2, and ENAME’s index is 1. The example on the next page shows how you can call ODESCR repeatedly to describe the first 12 select–list items in an arbitrary SQL statement. See the second sample program in Appendix B for additional information.

> **Note:** A dependency exists between the results returned by a describe operation (ODESCR) and a bind operation (OBINDPS, OBNDRA, OBNDRN or OBNDRV). Because a select–list item might contain bind variables, the type returned by ODESCR can vary depending on the results of bind operations.

So, if you have placeholders for bind variables in a SELECT statement and you plan to use ODESCR to obtain the size or datatype of select–list items, you should do the bind operation before the describe. If you need to rebind any input variables after performing a describe, you must reparse the SQL statement before rebinding. Note that the rebind operation might change the results returned for a select–list item.

The following complete program shows how the ODESCR routine can be used to describe the select list of arbitrary dynamic SQL statements.

```
* ODESCR.COB
*
* Demo example for COBOL dynamic ODESCR

IDENTIFICATION DIVISION.
PROGRAM–ID.  ODESCR–TEST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING–STORAGE SECTION.

01  LDA.
    03  LDA–V2RC               PIC S9(4) COMP.
    03  FILLER                 PIC X(10).
    03  LDA–RC                 PIC S9(4) COMP.
    03  FILLER                 PIC X(50).

01  CURSOR.
    03  CURS–V2RC              PIC S9(4) COMP.
    03  CURS–TYPE              PIC S9(4) COMP.
    03  CURS–ROWS–PROCESSED    PIC S9(9) COMP.
    03  CURS–OFFS              PIC S9(4) COMP.
    03  CURS–FNC               PIC X.
    03  FILLER                 PIC X.
    03  CURS–RC                PIC S9(4) COMP.
    03  FILLER                 PIC X(50).

01  HOST–DATA–AREA             PIC X(512).
01  XCOL–NAMES.
    03  NAME                   OCCURS 12 TIMES PIC X(30).
01  XCOL–L.
    03  NAME–L                 OCCURS 12 TIMES PIC S9(9)
        COMP.
01  XDBSIZE.
    03  DBSIZE                 OCCURS 12 TIMES PIC S9(4)
        COMP.
01  XCOL–DTYPE.
    03  DBTYPE                 OCCURS 12 TIMES PIC S9(4)
        COMP.
```

```
01  XCOL–DSIZE.
    03  DSIZE                 OCCURS 12 TIMES PIC S9(9)
        COMP.
01  XCOL–PRECISION.
    03  PRECISION             OCCURS 12 TIMES PIC S9(4)
        COMP.
01  XCOL–SCALE.
    03  SCALE                 OCCURS 12 TIMES PIC S9(4)
        COMP.
01  XCOL–NULLS–ALLOWED.
    03  NULL–OK               OCCURS 12 TIMES PIC S9(4)
        COMP.

01  USER–ID                   PIC X(11) VALUE ”SCOTT”.
01  USER–ID–L                 PIC S9(9) VALUE 5  COMP.
01  PASSWORD                  PIC X(5)  VALUE ”TIGER”.
01  PASSWORD–L                PIC S9(9) VALUE 5  COMP.
01  CONN                      PIC S9(9) VALUE 0  COMP.
01  CONN–L                    PIC S9(9) VALUE 0  COMP.
01  CONN–MODE                 PIC S9(9) VALUE 0  COMP.
01  MINUS–ONE                 PIC S9(9) VALUE –1 COMP.
01  SQL–STATEMENT             PIC X(80).
01  SQL–STATEMENT–L           PIC S9(9) VALUE 80 COMP.
01  VERSION–7                 PIC S9(9) VALUE 2  COMP.
01  POS                       PIC S9(9) COMP.
01  INDX                      PIC 9(3)  COMP.
01  MSGBUF                    PIC X(80).
01  MSGBUF–L                  PIC S9(9) VALUE 80 COMP.
01  ERR–RC                    PIC S9(4) COMP.
01  IMD–PARSE                 PIC S9(9) VALUE 0  COMP.
01  SQLL                      PIC S9(9) COMP.

PROCEDURE DIVISION.
START–MAIN.

*  Connect to Oracle in non–blocking mode.
*  HDA must be initialized to zeros before call to OLOG.
    MOVE LOW–VALUES TO HOST–DATA–AREA.
    CALL ”OLOG” USING LDA, HOST–DATA–AREA, USERNAME,
          USERNAME–L, PASSWORD, PASSWORD–L, CONN,
          CONN–L, CONN–MODE.
  IF LDA–RC IN LDA NOT = 0
      PERFORM ORA–ERROR
      GO TO EXIT–STOP.
  DISPLAY ” ”.
  DISPLAY ”Connected to Oracle as user: ” USER–ID.
  DISPLAY ” ”.
```

```
*  Open the cursor.
*  Use parameters PASSWORD, etc. for unused parameters.
   CALL ”OOPEN” USING CURSOR, LDA, PASSWORD, PASSWORD–L,
                 MINUS–ONE, PASSWORD, PASSWORD–L.
   IF CURS–RC IN CURSOR NOT = 0
     PERFORM ORA–ERROR
     GO TO EXIT–STOP.

   DISPLAY ”Enter SQL statements with ’;’ terminator”.

   PERFORM PROCESS–STATEMENT
     THRU PROCESS–STATEMENT–EXIT 10000 TIMES.

EXIT–CLOSE.
   CALL ”OCLOSE” USING CURSOR.
EXIT–LOGOFF.
   CALL ”OLOGOF” USING LDA.
EXIT–STOP.
   STOP RUN.

*  Accept and describe SQL SELECT statements.

PROCESS–STATEMENT.
   DISPLAY ” ”.
   DISPLAY ”> ” WITH NO ADVANCING.

   ACCEPT SQL–STATEMENT.

   MOVE 0 TO SQLL.
   INSPECT SQL–STATEMENT TALLYING SQLL
     FOR CHARACTERS BEFORE INITIAL ’;’.
   IF SQLL = 80
     GO TO EXIT–CLOSE.

   CALL ”OSQL” USING CURSOR, SQL–STATEMENT, SQLL,
                      IMD–PARSE, VERSION–7.
   IF CURS–RC IN CURSOR NOT = 0
     PERFORM ORA–ERROR
     GO TO EXIT–CLOSE.
   PERFORM DESCRIBE–STATEMENT VARYING POS FROM 1 BY 1
       UNTIL (CURS–RC IN CURSOR = 1007 OR POS > 12).

   SUBTRACT 2 FROM POS.
   DISPLAY ”There were”, POS WITH CONVERSION,
           ” select–list items”.
   PERFORM
     VARYING INDX FROM 1 BY 1 UNTIL INDX > POS
     DISPLAY NAME(INDX), NAME–L(INDX) WITH CONVERSION,
       DBTYPE(INDX) WITH CONVERSION
```

```
  END-PERFORM.
PROCESS-STATEMENT-EXIT.

DESCRIBE-STATEMENT.

  MOVE 30 TO NAME-L(POS).
  CALL ”ODESCR” USING CURSOR, POS, DBSIZE(POS),
                      DBTYPE(POS), NAME(POS),
                      NAME-L(POS), DSIZE(POS),
                      PRECISION(POS), SCALE(POS),
                      NULL-OK(POS).
  IF (CURS-RC IN CURSOR NOT = 0 AND
      CURS-RC IN CURSOR NOT = 1007)
      PERFORM ORA-ERROR
      GO TO EXIT-CLOSE.

ORA-ERROR.
  IF LDA-RC IN LDA NOT = 0
    DISPLAY ”OLOGON error”
    MOVE LDA-RC TO ERR-RC
    MOVE 0 TO CURS-FNC
  ELSE IF CURS-RC IN CURSOR NOT = 0
    MOVE CURS-RC IN CURSOR TO ERR-RC
    DISPLAY ”Oracle error. Code is ”,
             ERR-RC WITH CONVERSION,
            ” Function is ”, CURS-FNC WITH CONVERSION.
  CALL ”OERHMS” USING LDA, ERR-RC, MSGBUF, MSGBUF-L.
  DISPLAY MSGBUF.
```

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
POS	PIC S9(9) COMP	IN
DBSIZE	PIC S9(9) COMP	OUT
DBTYPE	PIC S9(4) COMP	OUT
CBUF	PIC X(n)	OUT
CBUFL	PIC S9(9) COMP	IN / OUT
DSIZE	PIC S9(9) COMP	OUT
PREC	PIC S9(4) COMP	OUT
SCALE	PIC S9(4) COMP	OUT
NULLOK	PIC S9(4) COMP	OUT

CURSOR
The cursor data area in the program associated by OPARSE with the SQL statement being described.

POS
This is the position index of the select–list item in the SQL query. Each item is referenced by position as if they were numbered left to right (or first to last) consecutively beginning with 1. If you specify a position index greater than the number of items in the select–list or less than 1, ODESCR returns a “variable not in select–list” error in the *return code* field of the cursor data area.

DBSIZE
DBSIZE receives the *maximum* size of the column, as stored in the Oracle data dictionary. If the column is defined as VARCHAR2, CHAR, or NUMBER, the length returned is the maximum length specified for the column.

DBTYPE
DBTYPE receives the internal datatype code of the select–list item. A list of Oracle internal datatype codes and the possible external conversions for each of them is provided in Chapter 3.

CBUF
A character buffer in the program that receives the name of the select–list item (name of the column or wording of the expression).

CBUFL
CBUFL is set to the length of CBUF. CBUFL should be set before ODESCR is called. If CBUFL is not specified or if the value contained in CBUFL is 0, then the column name is not returned.

On return from ODESCR, CBUFL contains the length of the column or expression name that was returned in CBUF. The column name in CBUF is truncated if it is longer that the length specified in CBUFL on the call.

DSIZE
DSIZE receives the maximum display size of the select–list item if the select–list item is returned as a character string. The DSIZE parameter is especially useful when SQL routines, like SUBSTR or TO_CHAR, are used to modify the representation of a column. Values returned in DSIZE are listed in the following table.

Oracle Column Type	*Value*
CHAR, VARCHAR2, RAW	length of the column in the table
NUMBER	22 (the internal length)
DATE	7 (the internal length)
LONG, LONG RAW	0
ROWID	(system dependent)
Functions returning datatype 1 (such as TO_CHAR())	same as the *dsize* parameter

PREC
The PREC parameter receives the precision of select–list items.

SCALE
The SCALE parameter receives the scale of numeric select–list items.

For Version 6 of the RDBMS, ODESCR returns the correct scale and precision of fixed–point numbers and returns precision and scale of zero for floating–point numbers, as shown below:

SQL Datatype	*Precision*	*Scale*
NUMBER(P)	P	0
NUMBER(P,S)	P	S
NUMBER	0	0
FLOAT(N)	0	0

For Oracle7, the SQL types REAL, DOUBLE PRECISION, FLOAT, and FLOAT(N) return the correct precision and a scale of –127.

NULLOK
NULLOK is set to 1 if null values are allowed for that column and to 0 if they are not.

See Also OBINDPS, OBNDRA, OBNDRN, OBNDRV, ODEFIN, ODEFINPS, OPARSE.

ODESSP

Purpose ODESSP is used to describe the parameters of a PL/SQL procedure or function stored in an Oracle database.

Syntax

```
CALL "ODESSP" USING LDA, OBJNAM, ONLEN, RSV1, RSV1LN,
     RSV2, RSV2LN, OVRLD, POS, LEVEL, ARGNM,
     ARNLEN, DTYPE, DEFSUP, PMODE, DTSIZ,
     PREC, SCALE, RADIX, SPARE, ARRSIZ.
```

Comments You call ODESSP to get the properties of a stored procedure (or function) and the properties of its parameters. When you call ODESSP, pass to it:

- A valid LDA for a connection that has, at the least, execute privileges on the procedure.
- The name of the procedure, optionally including the package name. The package body does not have to exist, as long as the procedure is specified in the package.
- The total length of the procedure name.

If the procedure exists and the connection specified in the LDA parameter has permission to execute the procedure, ODESSP returns information about each parameter of the procedure in a set of array parameters. It also returns information about the return type if it is a function.

Your OCI program must allocate the arrays for all parameters of ODESSP, and you must pass a parameter (ARRSIZ) that indicates the size of the arrays (or the size of the smallest array if they are not equal). The ARRSIZ parameter returns the number of elements of each array that was returned by ODESSP.

ODESSP returns a non–zero value if an error occurred. The error number is in the *return code* field of the LDA. The following errors can be returned there:

–20000	The object named in the OBJNAM parameter is a package, not a procedure or function.
–20001	The procedure or function named in OBJNAM does not exist in the named package.
–20002	A database link was specified in OBJNAM, either explicitly or by means of a synonym.
ORA–0xxxx	An Oracle code, usually indicating a syntax error in the procedure specification in OBJNAM.

When ODESSP returns successfully, the OUT array parameters contain the descriptive information about the procedure or function parameters, and the return type for a function. As an example, consider a package EMP_RECS in the SCOTT schema. The package contains two stored procedures and a stored function, all named GET_SAL_INFO. Here is the package specification:

```
create or replace package EMP_RECS as

  procedure get_sal_info (
    name    in   emp.ename%type,
    salary  out  emp.sal%type);

  procedure get_sal_info (
    IDnum  in   emp.empno%type,
    salary  out  emp.sal%type);

  function get_sal_info (
    name      emp.ename%type) return emp.sal%type;

end EMP_RECS;
```

A code fragment to describe these procedures and functions follows:

```
WORKING–STORAGE SECTION.

01  LDA.
    03  LDA–V2RC        PIC S9(4) COMP.
    03  FILLER          PIC X(10).
    03  LDA–RC          PIC S9(4) COMP.
    03  FILLER          PIC X(50).

01  OBJNAM              PIC X(30) VALUE
    ”SCOTT.EMP_RECS.GET_SAL_INFO”..
01  ONLEN               PIC S9(9) VALUE 27 COMP.
01  RSV1                PIC X(10) VALUE ””.
01  RSV1LN              PIC S9(9) VALUE 0 COMP.
01  RSV2                PIC X(10) VALUE ””.
01  RSV2LN              PIC S9(9) VALUE 0 COMP.
01  ARRSIZ              PIC S9(9) VALUE 10 COMP.

01  PARM–OVRLD.
    03  OVRLD           OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–POS.
    03  POS             OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–LEVEL.
    03  LEVEL           OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–NAMES.
    03  ARGNAM          OCCURS 10 TIMES PIC X(30).
```

```
01  PARM–L.
    03  ARNLEN          OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–DTYPE.
    03  DTYPE           OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–DEFSUP.
    03  DEFSUP          OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–PMODE.
    03  PMODE            OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–DTSIZ
    03  DTSIZ           OCCURS 10 TIMES PIC S9(9) COMP.
01  PARM–PREC.
    03  PREC            OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–SCALE.
    03  SCALE           OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–RADIX.
    03  RADIX           OCCURS 10 TIMES PIC S9(4) COMP.
01  PARM–SPARE.
    03  SPARE           OCCURS 10 TIMES PIC S9(9) COMP.

...

  PROCEDURE DIVISION.
  START–MAIN.

...
  MOVE 10 TO ARRSIZ
  CALL ”ODESSP” USING LDA, OBJNAM, ONLEN, RSV1, RSV1LN,
        RSV2, RSV2LN, OVRLD, POS, LEVEL, ARGNM,
        ARNLEN, DTYPE, DEFSUP, PMODE, DTSIZ,
        PREC, SCALE, RADIX, SPARE, ARRSIZ.

  DISPLAY ” ”.
  DISPLAY ”PARM NAME   POS   OVERLOAD    DATATYPE”.

  PERFORM
    VARYING INDX FROM 1 BY 1 UNTIL INDX > ARRSIZ
    DISPLAY ARGNAMINDX), POS(INDX) WITH CONVERSION,
        OVRLD(INDX) WITH CONVERSION, DTYPE(INDX)
        WITH CONVERSION.
  END–PERFORM.
...
```

When this call to ODESSP completes, the return parameter arrays are filled in as shown in Table 5 – 1. 6 is returned in the ARRSIZ parameter, as there were a total of five parameters and one function return type described.

	ARRAY ELEMENT					
PARAMETER	0	1	2	3	4	5
OVRLD	1	1	2	2	3	3
POS	1	2	1	2	0	1
LEVEL	1	1	1	1	1	1
ARGNM	name	salary	ID_num	salary	NULL	name
ARNLEN	4	6	6	6	0	4
DTYPE	1	2	2	2	2	1
DEFSUP	0	0	0	0	0	0
PMODE	0	1	0	1	1	0
DTSIZE	10	22	22	22	22	10
PREC		7	4	7	7	
SCALE		2	0	2	2	
RADIX		10	10	10	10	
SPARE[1]	n/a	n/a	n/a	n/a	n/a	n/a
Note 1: Reserved by Oracle for future use.						

Table 5 – 1 ODESSP Return Values

Parameters

Parameter Name	*Type*	*Mode*
LDA	(Table)	IN / OUT
OBJNAM	PIC S9(9) COMP	IN
ONLEN	PIC S9(9) COMP	IN
RSV1	PIC X(N)	IN
RSV1LN	PIC S9(9) COMP	IN
RSV2	PIC X(N)	IN
RSV2LN	PIC S9(9)	IN
OVRLD	PIC S9(4)	OUT
POS	PIC S9(4)	OUT
LEVEL	PIC S9(4)	OUT
ARGNM	PIC X(30) OCCURS (M) TIMES	OUT
ARNLEN	PIC S9(4)	OUT
DTYPE	PIC S9(4)	OUT
DEFSUP	PIC S9(4)	OUT
PMODE	PIC S9(4)	OUT

Parameter Name	*Type*	*Mode*
DTSIZ	PIC S9(4)	OUT
PREC	PIC S9(4)	OUT
SCALE	PIC S9(4)	OUT
RADIX	PIC S9(4)	OUT
SPARE	PIC S9(9)	OUT
ARRSIZ	PIC S9(9)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

OBJNAM
The name of the procedure or function, including optional schema and package name. Quoted names are accepted. Synonyms are also accepted and are translated. Multi–byte characters can be used. The string can be null terminated. If it is not, the actual length must be passed in the ONLEN parameter.

ONLEN
The length in bytes of the OBJNAM parameter. If OBJNAM is a null–terminated string, pass ONLEN as –1; otherwise, pass the exact length.

RSV1
Reserved by Oracle for future use.

RSV1LN
Reserved by Oracle for future use.

RSV2
Reserved by Oracle for future use.

RSV2LN
Reserved by Oracle for future use.

OVRLD
An array indicating whether the procedure is overloaded. If the procedure (or function) is not overloaded, zero is returned. Overloaded procedures return *1...n* for *n* overloadings of the name.

POS
An array returning the parameter positions in the parameter list of the procedure. The first, or left–most, parameter in the list is position 1. When *pos* returns a zero, this indicates that a function return type is being described.

LEVEL
For scalar parameters, LEVEL returns zero. For a record parameter, zero is returned for the record itself, then for each parameter in the record the parameter's level in the record is indicated, starting from one, in successive elements of the returned value of LEVEL.

For array parameters, zero is returned for the array itself. The next element in the return array is at level one, and describes the element type of the array.

For example, for a procedure that contains three scalar parameters, an array of ten elements, and one record containing three scalar parameters at the same level, you need to pass ODESSP arrays with a minimum dimension of nine: three elements for the scalars, two for the array, and four for the record parameter.

ARGNM
An array of strings that returns the name of each parameter in the procedure or function.

ARNLEN
An array returning the length in bytes of each corresponding parameter name in ARGNM.

DTYPE
The Oracle datatype code for each parameter. See the *PL/SQL User's Guide and Reference* for a list of the PL / SQL datatypes. Numeric types, such as FLOAT, INTEGER, and REAL, return a code of 2. VARCHAR2 returns 1. CHAR returns 96. Other datatype codes are shown in Table 3 – 5 in Chapter 3.

> **Note:** A DTYPE value of zero indicates that the procedure being described has no parameters.

DEFSUP
This parameter indicates whether the corresponding parameter has a default value. Zero returned indicates no default; one indicates that a default value was supplied in the procedure or function specification.

PMODE
This parameter indicates the mode of the corresponding parameter. Zero indicates an IN parameter, one an OUT parameter, and two an IN / OUT parameter.

DTSIZ
The size of the datatype in bytes. Character datatypes return the size of the parameter. For example, the EMP table contains a column ENAME. If a parameter in a procedure being described is of the type EMP.ENAME%TYPE, the value 10 is returned for this parameter, since that is the length of the ENAME column.

For number types, 22 is returned. See the description of the DBSIZE parameter under ODESCR in this chapter for more information.

PREC
This parameter indicates the precision of the corresponding parameter if the parameter is numeric.

SCALE
This parameter indicates the scale of the corresponding parameter if the parameter is numeric.

RADIX
This parameter indicates the radix of the corresponding parameter if it is numeric.

SPARE
Reserved by Oracle for future use.

ARRSIZ
When you call ODESSP, pass the length of the arrays of the OUT parameters. If the arrays are not of equal length, you must pass the length of the shortest array.

When ODESSP returns, ARRSIZ returns the number of array elements filled in.

See Also ODESCR.

OERHMS

Purpose OERHMS returns the text of an Oracle error message, given the error code RCODE.

Syntax `CALL ”OERHMS” USING LDA, RCODE, BUF, BUFSIZ.`

Comments When you call OERHMS, pass the address of the LDA for the active connection as the first parameter. This is required to retrieve error messages that are correct for the database version being used on that connection.

When using OERHMS to return error messages from PL/SQL blocks (where the error code is between 6550 and 6599), be sure to allocate a large BUF, since several messages can be returned. 1000 bytes should be sufficient to handle most cases.

For more information about the causes of Oracle errors and possible solutions, see the *Oracle7 Server Messages* manual.

The following example shows how to obtain an error message from a specific Oracle instance:

```
DATA DIVISION.
01   LDA–1.
     03  LDA–V2RC                PIC S9(4) COMP.
     03  FILLER                  PIC X(10).
     03  LDA–RC                  PIC S9(4) COMP.
     03  FILLER                  PIC X(50).
01   LDA–2.
     03  LDA–V2RC                PIC S9(4) COMP.
     03  FILLER                  PIC X(10).
     03  LDA–RC                  PIC S9(4) COMP.
     03  FILLER                  PIC X(50).
77   MSG                         PIC X(512).
77   MSG–L                       PIC S9(9) VALUE 512 COMP.
...
PROCEDURE DIVISION.
...
*  after logging on, and getting an error in another call
*  on the connection using LDA–2...
CALL ”OERHMS” USING LDA–2, LDA–RC in LDA–2,
     MSG, MSG–L.
DISPLAY MSG.
```

Parameters

Parameter Name	*Type*	*Mode*
LDA	(Address)	IN / OUT
RCODE	PIC S9(4) COMP	IN
BUF	PIC X(n)	OUT
BUFSIZ	PIC S9(9) COMP	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

RCODE
The return code containing an Oracle error number.

BUF
A character buffer that receives the error message text.

BUFSIZ
The size of the buffer in bytes. The buffer size is essentially unlimited. However, a buffer of between 512 and 1024 bytes is usually sufficient.

See Also OERMSG, OLOG.

OEXEC

Purpose OEXEC executes the SQL statement associated with a cursor.

Syntax

```
CALL "OEXEC" USING CURSOR.
```

Comments Before calling OEXEC, you must call OPARSE to parse the SQL statement; this call must complete successfully. If the SQL statement is a query, you must call ODEFIN or ODEFINPS to associate each select–list item in the query with the address of a program output buffer. If the SQL statement contains placeholders for bind variables, you must also call OBINDPS, OBNDRA, OBNDRV or OBNDRN to bind each placeholder to the address of a program variable.

For queries, after OEXEC is called, your program must explicitly request each row of the result using OFEN or OFETCH.

For UPDATE, DELETE, and INSERT statements, OEXEC executes the SQL statement and sets the *return code* field and the *rows processed count* field in the cursor data area. Note that an UPDATE that does not affect any rows (no rows match the WHERE clause) returns success in the *return code* field and zero in the *rows processed count* field.

Data Definition Language statements are executed on the parse if you have linked in non–deferred mode or if you have liked with the deferred option and the DEFFLG parameter of OPARSE is zero. If you have linked in deferred mode and the DEFFLG parameter is non–zero, you must call OEXN or OEXEC to execute the statement.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when OPARSE is called in non–deferred mode are not detected until the first non–deferred call is made (usually an execute or describe call).

Refer to the description of the OFETCH routine in this chapter for an example showing how OEXEC is used.

Parameter

Parameter Name	***Type***	***Mode***
CURSOR	(Address)	IN / OUT

CURSOR
The cursor data area specified in the associated OPARSE call.

See Also OBINDPS, OBNDRA, OBNDRN, ODEFIN, ODEFINPS, OPARSE.

OEXFET

Purpose OEXFET executes the SQL statement associated with a cursor, then fetch data from one or more rows. A cancel (equivalent to a call to OCAN) of the cursor can also be performed by OEXFET.

Syntax

```
CALL "OEXFET" USING CURSOR, NROWS, CANCEL, EXACT.
```

Comments Before calling OEXFET, the OCI program must first call OPARSE to parse the SQL statement, call OBINDPS, OBNDRA, OBNDRN or OBNDRV to bind input variables (if any), then call ODEFIN or ODEFINPS to define the output variables.

If the OCI program was linked using the deferred mode option, the bind and define steps are deferred until OEXFET is called. If OPARSE was called with the deferred parse flag (DEFFLG) parameter non–zero, the parse step is also delayed until OEXFET is called. This means that your program can complete the processing of a SQL statement using a minimum of message round–trips between the client running the OCI program and the database server.

If you call OEXFET for a DML statement that is not a query, Oracle issues the error

```
ORA–01002:  fetch out of sequence
```

and the execute operation fails.

> **Note:** Using the deferred parse, bind, and define capabilities to process a SQL statement requires more memory on the client system than the non–deferred sequence. So, you gain execution speed at the expense of the additional space required.

When running against an Oracle7 database where the SQL statement was parsed using OPARSE with the LNGFLG Parameter set to 1 or 2, a character string that is too large for its associated buffer is truncated, the column return code (RCODE) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the OEXFET call does not return an error indication. If a null is encountered for a select–list item, the associated column return code (RCODE) for that column is set to the error

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

and the indicator parameter is set to –1. The OEXFET call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, OEXFET does return an ORA–01405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

OEXFET both executes the statement and fetches the row or rows that satisfy the query. If you need to fetch additional rows after OEXFET completes, use the OFEN routine.

The following example shows how to use deferred parse, bind, and define operations together with OEXFET to process a SQL statement:

```
DATA DIVISION.
WORKING STORAGE SECTION.
01 XTABLE1.
   02 ENAMES        PIC X(20) OCCURS 12000 TIMES.
   02 INDP          PIC S9(4) OCCURS 12000 TIMES COMP.
   02 RET–L         PIC S9(4) OCCURS 12000 TIMES COMP.
   02 RET_CODE      PIC S9(4) OCCURS 12000 TIMES COMP.

77    SQL–STMT      PIC X(50) VALUE ”SELECT ename
     –   FROM emp WHERE deptno = 20”.
77    ONE           PIC S9(9) VALUE 1     COMP.
77    MAX–EMPLOYEES PIC S9(9) VALUE 12000 COMP.
77    ENAME–LEN     PIC S9(9) VALUE 20 COMP.
77    SQL–STMT–L    PIC S9(9) VALUE 50 COMP.
...
PROCEDURE DIVISION.
...
*  Parse the statement.  Set the deferred flag.
CALL ”OPARSE” USING CURSOR, SQL–STMT, SQL–STMT–L, ONE.
...
*  Call ODEFIN to define the output variables.
CALL ”ODEFIN” USING CURSOR, ONE, NAMES, ENAME–LEN, CHAR–T,
     ZERO, INDP, FMT, ZERO, FMT–T, RET–L, RET–CODE.

CALL ”OEXFET” USING CURSOR, MAX–EMPLOYEES, ZERO, ZERO.
```

In this example, the EXACT parameter is set to zero, so a “no data found” error is not returned if the number of rows returned is less than MAX–EMPLOYEES.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
NROWS	PIC S9(9) COMP	IN
CANCEL	PIC S9(9) COMP	IN
EXACT	PIC S9(9) COMP	IN

CURSOR
The cursor data area specified in the associated OPARSE call.

NROWS
The number of rows to fetch. If NROWS is greater than 1, then you must define tables to receive the select–list values, as well as any indicator parameters. See the description of ODEFIN for more information.

If NROWS is greater than the number of rows that satisfy the query, the *rows processed count* field in the CDA is set to the number of rows returned, and Oracle returns the error

```
ORA–01403:  no data found
```

CANCEL
If this parameter is non–zero when OEXFET is called, the cursor is cancelled after the fetch completes. This has exactly the effect of issuing an OCAN call, but does not require the additional call overhead.

EXACT
If this parameter is non–zero when OEXFET is called, OEXFET returns an error if the number of rows that satisfy the query is not exactly the same as the number specified in the NROWS parameter. If the number of rows returned by the query is less than the number specified in the NROWS parameter, Oracle returns the error

```
ORA–01403:  no data found
```

If the number of rows returned by the query is greater than the number specified in the NROWS parameter, Oracle returns the error

```
ORA–01422:  Exact fetch returns more than requested
            number of rows
```

Note: If EXACT is non–zero, a cancel of the cursor is always performed, regardless of the setting of the CANCEL parameter.

See Also OBINDPS, OBNDRA, OBNDRN, OBNDRV, ODEFIN, ODEFINPS, OFEN, OPARSE.

OEXN

Purpose OEXN executes a SQL statement. Tables can be bound to placeholders in the statement, taking advantage of the Oracle array interface.

Syntax

```
CALL ”OEXN” USING CURSOR, ITERS, ROWOFF.
```

Comments OEXN is similar to OEXEC, but it allows you to take advantage of the Oracle array (table) interface. OEXN allows operations using a table containing multiple bind variables. OEXN is generally much faster than successive calls to OEXEC, especially in a networked environment.

The following example declares three tables, one of ten integers (PIC S9(9) COMP), one of ten indicator variables (PIC S9(4) COMP), and one of ten 20–character strings, and defines a SQL statement that inserts multiple rows into the database. After binding the tables, the program must place data for the first INSERT in ENAMES(1) and EMP_NOS(1), for the second INSERT in ENAMES(2) and EMP_NOS(2), and so forth. (This is not shown in the example.) Then OEXN is called to insert the data in the tables into the database.

```
DATA DIVISION.
WORKING STORAGE SECTION.
01 XTABLE1.
   02 IND–PARAMS    PIC S9(4) OCCURS 10 TIMES COMP.
   02 ENAMES        PIC X(20) OCCURS 10 TIMES.
   02 EMP–NOS       PIC S9(9) OCCURS 10 TIMES COMP.
77 SQL–STMT         PIC X(50) VALUE ”INSERT INTO EMP
     –(ENAME, EMPNO) VALUES (:N, :E)”.
77    ENAME–LEN     PIC S9(9) VALUE 20 COMP.
77    TABLE–LEN     PIC S9(9) VALUE 10 COMP.
77    PH1           PIC X(2) VALUE ”:N”.
77    PH2           PIC X(2) VALUE ”:E”.
77    SQL–STMT–LEN  PIC S9(9) VALUE 48 COMP.
...
PROCEDURE DIVISION.
...
*  Parse the statement.
CALL ”OPARSE” USING CURSOR, SQL–STMT, SQL–STMT–LEN, ZERO.
*  Bind the tables using OBNDRV.
*  First, make sure that IND–PARAMS is zeroed.
PERFORM VARYING J FROM 1 BY 1
        UNTIL J > 10
     MOVE ZERO TO IND–PARAMS(J)
END PERFORM.
CALL ”OBNDRV” USING CURSOR, PH1, TWO, ENAMES(1), ENAME–LEN,
     ONE, ZERO, IND–PARAMS(1).
CALL ”OBNDRV” USING CURSOR, PH2, TWO, EMP–NOS(1), INT–LEN,
     THREE, ZERO, IND–PARAMS(1).
```

```
...
*  After getting the data for the ENAMES and EMP–NOS tables,
*  execute the statement, inserting the values in the
*  tables into the Oracle emp table.
CALL ”OEXN” USING CURSOR, TABLE–LEN, ZERO.
```

The completion status of OEXN is indicated in the *return code* field of the cursor data area. The *rows processed count* in the cursor data area indicates the number of rows successfully processed.

The *rows processed count* also returns the number of rows successfully processed before an error. If the SQL statement is processing only one row per table element and if the *rows processed count* is not equal to ITERS, the operation failed on table element *rows processed count* + 1.

You can continue to process the rest of the table even after a failure on one of the table elements as long as a rollback did not occur (obtained from the *flags1* field in the cursor data area). You do this by using the zero–based ROWOFF parameter to start operations at table elements other than the first. In the above example, if the *rows processed count* was 5 at completion of OEXN, then row 6 was rejected. In this event, to continue the operation at row 7, call OEXN again as follows:

```
CALL ”OEXN” USING CURSOR, TABLE–LEN, SIX.
```

Note: The maximum number of elements in a table used by OEXN is 32767.

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	(Address)	IN/OUT
ITERS	PIC S9(9) COMP	IN
ROWOFF	PIC S9(9) COMP	IN

CURSOR
The cursor data area specified in the associated OPARSE call.

ITERS
The size of the table of bind variables to be used. The size cannot be greater than 32767 items.

ROWOFF
The zero–based offset within the bind variable table at which to begin operations. OEXN processes (ITERS – ROWOFF) elements if no error occurs.

See Also OEXFET, OEXEC.

OFEN

Purpose OFEN fetches multiple rows into tables, taking advantage of the Oracle array interface.

Syntax

```
CALL ”OFEN” USING CURSOR, NROWS.
```

Comments OFEN is similar to OFETCH; however, OFEN fetches multiple rows into a table with a single call. The address of the table is bound to a select–list item in the SQL query statement using ODEFIN or ODEFINPS.

When running against an Oracle7 database where the SQL statement was parsed using OPARSE with the LNGFLG parameter set to 1 or 2, a character string that is too large for its associated output variable is truncated, the column return code (RCODE) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the OFEN call does not return an error indication. If a null is encountered for a select–list item, the associated column return code (RCODE) for that column is set to the error

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

and the indicator parameter is set to –1. The OFEN call does not return an error.

However, if no indicator parameter has been defined and the program is running against an Oracle7 database, OFEN *does* return the 1405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

Even when fetching a single row, Oracle recommends that Oracle7 OCI programs use OEXFET, with the NROWS parameter set to 1, instead of the combination of OEXEC and OFEN. Use OFEN after OEXFET to fetch additional rows when you do not know in advance the exact number of rows that a query returns.

The complete program on the following pages shows how OFEN can be used to fetch rows from an Oracle database table, ten rows at a time.

```
IDENTIFICATION DIVISION.
PROGRAM–ID.  OFEN–TEST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING–STORAGE SECTION.
```

```
01  LDA.
    03  LDA–V2RC               PIC S9(4) COMP.
    03  FILLER                 PIC X(10).
    03  LDA–RC                 PIC S9(4) COMP.
    03  FILLER                 PIC X(50).
01  CURSOR.
    03  CURS–V2RC              PIC S9(4) COMP.
    03  CURS–TYPE              PIC S9(4) COMP.
    03  CURS–ROWS–PROCESSED    PIC S9(9) COMP.
    03  CURS–OFFS              PIC S9(4) COMP.
    03  CURS–FNC               PIC X.
    03  FILLER                 PIC X.
    03  CURS–RC                PIC S9(4) COMP.
    03  FILLER                 PIC X(50).
01  HOST–DATA–AREA             PIC X(512).

01  XNAMES.
    03  NAME                   OCCURS 10 TIMES PIC X(15).
01  XSAL.
    03  SALARY                 OCCURS 10 TIMES PIC S9(4)V99
        COMP–3.
01  XEMPNO.
    03  EMP–NO                 OCCURS 10 TIMES PIC S9(9)
        COMP.
01  XINDS–NAME.
    03  INDI–NAME              OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XRCODE–NAME.
    03  RET–CODE–NAME          OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XRLEN–NAME.
    03  RET–LEN–NAME           OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XINDS–EMPNO.
    03  INDI–EMPNO             OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XRCODE–EMPNO.
    03  RET–CODE–EMPNO         OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XRLEN–EMPNO.
    03  RET–LEN–EMPNO          OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XINDS–SAL.
    03  INDI–SAL               OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XRCODE–SAL.
    03  RET–CODE–SAL           OCCURS 10 TIMES PIC S9(4)
        COMP.
01  XRLEN–SAL.
```

```
    03  RET–LEN–SAL           OCCURS 10 TIMES PIC S9(4)
        COMP.

01  USER–ID                   PIC X(11) VALUE ”SCOTT”.
01  USER–ID–L                 PIC S9(9) VALUE 5  COMP.
01  PASSWORD                  PIC X(5)  VALUE ”TIGER”.
01  PASSWORD–L                PIC S9(9) VALUE 5  COMP.
01  CONN                      PIC S9(9) VALUE 0  COMP.
01  CONN–L                    PIC S9(9) VALUE 0  COMP.
01  CONN–MODE                 PIC S9(9) VALUE 0  COMP.
01  FMT–SAL                   PIC X(6)  VALUE ”07.+02”.
01  FMT–SAL–L                 PIC S9(9) VALUE 6  COMP.
01  SAL–T                     PIC S9(9) VALUE 7  COMP.
01  NAME–L                    PIC S9(9) VALUE 15 COMP.
01  NAME–T                    PIC S9(9) VALUE 96 COMP.
01  MINUS–ONE                 PIC S9(9) VALUE –1 COMP.
01  ZERO–A                    PIC S9(9) VALUE 0  COMP.
01  ONE                       PIC S9(9) VALUE 1  COMP.
01  TWO                       PIC S9(9) VALUE 2  COMP.
01  THREE                     PIC S9(9) VALUE 3  COMP.
01  FOUR                      PIC S9(9) VALUE 4  COMP.
01  TEN                       PIC S9(9) VALUE 10 COMP.
01  N–ROWS                    PIC 9(4)  COMP.
01  N–DONE                    PIC 9(4)  COMP.
01  INDX                      PIC 9(4)  COMP.
01  DUMMY                     PIC X(3)  VALUE ”   ”.

01  SQL–STATEMENT             PIC X(33) VALUE
    ”SELECT ENAME, EMPNO, SAL FROM EMP”.
01  SQL–STATEMENT–L           PIC S9(9) VALUE 33 COMP.
01  VERSION–7                 PIC S9(9) VALUE 2  COMP.
01  MSGBUF                    PIC X(80).
01  MSGBUF–L                  PIC S9(9) VALUE 80 COMP.
01  ERR–RC                    PIC S9(4) COMP.
01  DEL–PARSE                 PIC S9(9) VALUE 1  COMP.
PROCEDURE DIVISION.
START–MAIN.

*  Connect to Oracle in non–blocking mode.
*  HDA must be initialized to zeros before call to OLOG.
    MOVE LOW–VALUES TO HOST–DATA–AREA.
    CALL ”OLOG” USING LDA, HOST–DATA–AREA, USERNAME,
          USERNAME–L, PASSWORD, PASSWORD–L, CONN,
          CONN–L, CONN–MODE.
  IF LDA–RC IN LDA NOT = 0
     PERFORM ORA–ERROR
     GO TO EXIT–STOP.

  DISPLAY ” ”.
  DISPLAY ”Connected to Oracle as user ”, USER–ID.
  DISPLAY ” ”.

*  Open the cursor.
  CALL ”OOPEN” USING CURSOR, LDA, PASSWORD, PASSWORD–L,
                    MINUS–ONE, PASSWORD, PASSWORD–L.
  IF CURS–RC IN CURSOR NOT = 0
     PERFORM ORA–ERROR
     GO TO EXIT–STOP.
*  Parse the SQL select statement.
  CALL ”OPARSE” USING CURSOR, SQL–STATEMENT,
                    SQL–STATEMENT–L, DEL–PARSE,
                    VERSION–7.
  IF CURS–RC IN CURSOR NOT = 0
  PERFORM ORA–ERROR
  GO TO EXIT–CLOSE.

*  Define the three select–list items.
  CALL ”ODEFIN” USING CURSOR, ONE, NAME(1), NAME–L,
                      NAME–T, MINUS–ONE, INDI–NAME(1),
                      DUMMY, ZERO–A, ,RET–LEN–NAME(1),
                      RET–CODE–NAME(1).
  CALL ”ODEFIN” USING CURSOR, TWO, EMP–NO(1), FOUR,
                      THREE, MINUS–ONE, INDI–EMPNO(1),
                      DUMMY, ZERO–A, ZERO–A,
                      RET–LEN–EMPNO(1),RET–CODE–EMPNO(1).
  CALL ”ODEFIN” USING CURSOR, THREE, SALARY(1), FOUR,
                      SAL–T, MINUS–ONE, INDI–SAL(1),
                      FMT–SAL, FMT–SAL–L, SAL–T,
                      RET–LEN–SAL(1),
                      RET–CODE–SAL(1).
*  Execute the SQL statement.
  CALL ”OEXN” USING CURSOR, ONE, ZERO–A.

*  Perform the table fetches, 10 at a time.  End when
*  CURS–RC = 1403 (no more data found), or on an error.
  MOVE 0 TO N–ROWS.
FETCH–LOOP.
  CALL ”OFEN” USING CURSOR, TEN.
  IF (CURS–RC NOT = 1403 AND CURS–RC NOT = 0)
     PERFORM ORA–ERROR
     GO TO EXIT–CLOSE.
  MOVE CURS–ROWS–PROCESSED TO N–DONE.
  SUBTRACT N–ROWS FROM N–DONE.
  ADD N–DONE TO N–ROWS.
  DISPLAY ”Employee name      Number Salary  RC(NAME)
– ”  RL(NAME)”.
DISPLAY ”–––––––––––––––––––––––––––––––––––––––––
– ”  –––––––––”.
```

```
  PERFORM DISPLAY–LINE THRU DISPLAY–LINE–EXIT
    VARYING INDX FROM 1 BY 1 UNTIL INDX > N–DONE.
  IF CURS–RC IN CURSOR = 0
    GO TO FETCH–LOOP.

  DISPLAY ” ”.
  DISPLAY CURS–ROWS–PROCESSED WITH CONVERSION,
          ” rows processed.”.
EXIT–CLOSE.
  CALL ”OCLOSE” USING CURSOR.
  CALL ”OLOGOF” USING LDA.
EXIT–STOP.
  STOP RUN.
DISPLAY–LINE.
  IF (RET–CODE–NAME(INDX) = 1405)
     MOVE ”NULL” TO NAME(INDX)
     MOVE 0 TO SALARY(INDX).
  DISPLAY NAME(INDX), EMP–NO(INDX) WITH CONVERSION,
          SALARY(INDX) WITH CONVERSION,
          RET–CODE–NAME(INDX) WITH CONVERSION, ”      ”,
          RET–LEN–NAME(INDX) WITH CONVERSION.
DISPLAY–LINE–EXIT.

ORA–ERROR.
  IF LDA–RC IN LDA NOT = 0
     DISPLAY ”OLOGON error”
     MOVE LDA–RC TO ERR–RC
     MOVE 0 TO CURS–FNC
  ELSE IF CURS–RC IN CURSOR NOT = 0
     MOVE CURS–RC IN CURSOR TO ERR–RC
  DISPLAY ”Oracle error. Code is ”,
          ERR–RC WITH CONVERSION,
          ” Function is ”, CURS–FNC WITH CONVERSION.
  CALL ”OERHMS” USING LDA, ERR–RC, MSGBUF, MSGBUF–L.
  DISPLAY MSGBUF.
```

OFEN can be called repeatedly until there is no more data to fetch, indicated by a ”no more data” return message. The completion status of OFEN is indicated in the *return code* field of the cursor data area. The *rows processed count* field in the cursor data area indicates the cumulative number of rows successfully fetched. If the *rows processed count* increases by NROWS, OFEN may be called again to get the next batch of rows. If the *rows processed count* does not increase by NROWS, then an error, such as “no data found”, has occurred.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
NROWS	PIC S9(9) COMP	IN

CURSOR
The cursor data area specified in the OPARSE call.

NROWS
The size of the defined variable table on which to operate. The size cannot be greater than 32767 items. If NROWS is set to 1, OFEN acts effectively just like OFETCH.

See Also ODEFIN, ODEFINPS, OEXFET, OFETCH, OPARSE.

OFETCH

Purpose OFETCH returns rows of a query to the user program, one row at a time.

Syntax

```
CALL ”OFETCH” USING CURSOR.
```

Comments Each select–list item of the query is placed into a buffer identified by a previous ODEFIN call.

When running against an Oracle7 database where the SQL statement was parsed using OPARSE with the LNGFLG parameter set to 1 or 2, a character string that is too large for its associated output variable is truncated, the column return code (RCODE) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the OFETCH call does not return an error indication.

If a null is encountered for a select–list item, the associated column return code (RCODE) for that column is set to the error

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

and the indicator parameter is set to –1. The OFETCH call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, OFETCH *does* return an ORA–01405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

The following example shows how you can obtain data from Oracle using OFETCH on a query statement. This example continues the one shown in the description of the ODEFIN routine earlier in this chapter. In that example, the select–list items in the SQL statement

```
SELECT ename, empno, sal FROM emp WHERE
    sal > :MIN_SAL
```

were associated with output buffers, and the addresses of column return lengths and return codes were bound. The example continues:

```
PROCEDURE DIVISION.
...
*  Bind the output buffers to the select–list items.
CALL ”ODEFIN” USING CURSOR, ONE, EMP–NAME, EMP–NAME–L, CHAR–TYPE,
    ZERO, INDP, FMT, ZERO, FMT–T, RET–LEN1, RET–CODE1.
CALL ”ODEFIN” USING CURSOR, TWO, EMP–NUM, EMP–NUM–L, INT–TYPE,
    ZERO, INDP, FMT, ZERO, FMT–T, RET–LEN2, RET–CODE2.
```

```
CALL ”ODEFIN” USING CURSOR, THREE, SAL, SAL–L, FMT–T,
    TWO, INDP, FMT, FMT–L, FMT–T, RET–LEN3, RET–CODE3.

*  Execute the query.
CALL ”OEXEC” USING CURSOR.
*  Fetch each row of the query.
PERFORM UNTIL C–RC IN CURSOR NOT = 0
   CALL ”OFETCH” USING CURSOR.
*  Check the return code for the first column.
*  Was a NULL value returned?
   IF RET–CODE1 = 1405
      DISPLAY ”NULL      ” NO ADVANCING
   ELSE
      DISPLAY EMP–NAME NO ADVANCING.
*  Check the second column return code.
   IF RET–CODE2 = 1405
      DISPLAY ”NULL  ” NO ADVANCING
   ELSE
      DISPLAY EMP–NUM NO ADVANCING.
   IF RET–CODE3 = 1405
*  Check the third column code.
      DISPLAY ”NULL  ”.
   ELSE
      DISPLAY SAL.
END PERFORM.
```

Each OFETCH call returns the next row from the set of rows that satisfies a query. After each OFETCH call, the *rows processed count* in the cursor data area is incremented.

There is no way to refetch rows previously fetched except by re–executing the OEXEC call and moving forward through the active set again. After the last row has been returned, the next fetch will return a ”no data found” return code. When this happens, *rows processed count* contains the total number of rows recovered by the query.

Parameter

Parameter Name	***Type***	***Mode***
CURSOR	(Address)	IN / OUT

CURSOR
The cursor data area associated with the SQL statement by the OPARSE call.

See Also ODEFIN, ODEFINPS, ODESCR, OEXEC, OEXFET, OFEN.

OFLNG

Purpose OFLNG fetches a portion of a LONG or LONG RAW column.

Syntax

```
CALL ”OFLNG” USING CURSOR, POS, BUF, BUFL,
      DTYPE, RETL, OFFSET.
```

Comments In Oracle7, LONG and LONG RAW columns can hold up to 2 gigabytes of data. The OFLNG procedure allows you to fetch up to 64K bytes, starting at any offset, in the LONG or LONG RAW column of the current row. There can be only one LONG or LONG RAW column in a table; however, a query that includes a join operation can include in its select list several LONG–type items. The POS parameter specifies the LONG–type column that the OFLNG call uses.

> **Note:** Although the datatype of BUFL is PIC S9(9) COMP, OFLNG can only retrieve up to 64K at a time. If an attempt is made to retrieve more than 64K, the returned data will not be complete. The use of PIC S9(9) COMP in the interface is for future enhancements.

Before calling OFLNG to retrieve the portion of the LONG column, you must do one or more fetches to position the cursor at the desired row.

> **Note:** With release 7.3, it may be possible to perform piecewise operations more efficiently using the new OBINDPS, ODEFINPS, OGETPI and OSETPI calls. See the section “Piecewise Insert, Update and Fetch” on page 2 – 39 for more information.

The example below shows how to retrieve 64 Kbytes, starting at offset 70000, from a LONG RAW column. See also the third code example in Appendix B for a complete demonstration program that uses OFLNG.

```
DATA DIVISION.
WORKING STORAGE SECTION.
77    SQL–STMT      PIC X(50) VALUE ”SELECT id_no
     –FROM data_table1 WHERE id_no = :1”.
77    ONE           PIC S9(9) VALUE 1      COMP.
77    DATA–BUF      PIC X(65536).
77    CHAR–TYPE     PIC S9(9) VALUE 1      COMP.
77    RET–L         PIC S9(4)              COMP.
77    OFFSET        PIC S9(9) VALUE 70000  COMP.
77    DB–L          PIC S9(9) VALUE 65536  COMP.
77    SQL–STMT–L    PIC S9(9) VALUE 45     COMP.
77    LONG–POS      PIC S9(9) VALUE 2      COMP.
...
PROCEDURE DIVISION.
...
```

```
*  Parse the statement.  Define a variable for id_no
*  Then do a fetch of one row to position the cursor.
...
   CALL ”OEXFET” USING CURSOR, ONE, ZERO, ONE.
   DISPLAY ”ROW ID NUMBER IS”, ID_NO.
...
   *  Get the portion of the column.
   CALL ”OFLNG” USING CURSOR, LONG–POS, DATA–BUF, DB–L,
      CHAR–TYPE, RET–L, OFFSET.
```

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
POS	PIC S9(4) COMP	IN
BUF	(Address)	OUT
BUFL	PIC S9(9) COMP	IN
DTYPE	PIC S9(9) COMP	IN
RETL	PIC S9(9) COMP	OUT
OFFSET	PIC S9(9) COMP	IN

CURSOR
The cursor data area specified in the associated OPARSE call.

POS
The index position of the LONG–type column. The first position is position one. If the column at the index position is not a LONG–type, a “column does not have LONG datatype” error is returned.

BUF
The buffer that receives the portion of the LONG–type column data.

BUFL
The length of BUF in bytes.

DTYPE
The datatype code corresponding to the datatype of BUF. See “External Datatypes” in Chapter 3 for a list of datatype codes.

RETL
The number of bytes returned. If more than 65535 bytes were requested and returned, the value 65535 is returned in this parameter.

OFFSET
Zero–based offset of the first byte in the LONG column to be fetched.

See Also ODESCR, OEXFET, OFEN, OFETCH.

OGETPI

Purpose OGETPI returns information about the next chunk of data to be processed as part of a piecewise insert, update or fetch.

Syntax

```
CALL ”OGETPI” USING CURSOR, PIECEP, CTXPP, ITERP, INDEXP.
```

Comments OGETPI is used (in conjunction with OSETPI) in an OCI application to determine whether more pieces exist to be either inserted, updated, or fetched as part of a piecewise operation.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

See the section “Piecewise Insert, Update and Fetch” in Chatper 2 for more information about piecewise operations and the OGETPI call.

For a sample C language program illustrating the use of OGETPI in an OCI program which performs an piecewise insert, see the description of the *ogetpi()* call in Chapter 4.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
PIECEP	PIC S9(2) COMP	OUT
CTXPP	PIC S9(9) COMP	OUT
ITERP	PIC S9(9) COMP	OUT
INDEXP	PIC S9(9) COMP	OUT

CURSOR
The CDA associated with the SQL or PL / SQL statement being processed.

PIECEP
Specifies whether the next piece to be fetched or inserted is the first piece, an intermediate piece or the last piece. Possible values are one (for the first piece) and two (for a subsequent piece) for a fetch or insert, and three (for the last piece) for fetches only. These values are defined in *ocidfn.h* as OCI_FIRST_PIECE, OCI_NEXT_PIECE, and OCI_LAST_PIECE.

CTXPP
The user–defined context pointer, which is optionally passed as part of an OBINDPS or ODEFINPS call. This pointer is returned to the application during the OGETPI call. If CTXPP is passed as NULL, the parameter is ignored. The application may already know which buffer it needs to pass in OSETPI at run time.

ITERP
The current iteration. During an array insert it will tell you which row you are working with. Starts from 0.

INDEXP
Pointer to the current index of an array mapped to a PL/SQL table, if an array is bound for an insert. The value of INDEXP varies between zero and the value set in the *cursiz* parameter of the OBINDPS call.

See Also OBINDPS, ODEFINPS, OSETPI.

OLOG

Purpose OLOG establishes a connection between an OCI program and an Oracle database.

Syntax

```
CALL ”OLOG” USING LDA, HDA, UID, UIDL,
     [PASSWD], [PASSWDL], [CONN], [CONNL], CONN–MODE.
```

Comments An OCI program can connect to one or more Oracle instances multiple times. Communication takes place using the LDA and the HDA defined within the program. It is the OLOG function that connects the LDA to Oracle.

The HDA is a program–allocated data area associated with each OLOG logon call. Its contents are entirely private to Oracle, but the HDA must be allocated by the OCI program. Each concurrent connection requires one LDA–HDA pair.

> **Note:** The HDA must be initialized to all zeros (binary zeros, not the “0” character) before the call to OLOG, or runtime errors will occur. In COBOL this can be accomplished through the use of the compiler–generated figurative constant LOW–VALUES. See the sample code below for an example.

The HDA has a size of 256 bytes on 32–bit systems, and 512 bytes on 64–bit systems. If memory permits, it is possible to allocate a 512–byte HDA on a 32–bit system to increase portability of aplications.

Refer to the section “Host Data Area” in Chapter 2 for more information about HDAs.

After the OLOG call, the HDA and the LDA must remain at the same program address they occupied at the time OLOG was called.

When an OCI program issues an OLOG call, a subsequent OLOGOF call using the same LDA commits all outstanding transactions for that connection. If a program fails to disconnect or terminates abnormally, then all outstanding transactions are rolled back.

The LDA *return code* field indicates the result of the OLOG call. A zero return code indicates a successful connection.

The MODE parameter specifies whether the connection is in blocking or non–blocking mode. For more information on connection modes, see page 2 – 32. For a short example program in C, see the *onbset()* description on page 4–67.

You should also refer to the section on SQL*Net in your Oracle system–specific documentation for any particular notes or restrictions that apply to your operating system.

The following code fragment demonstrates a typical set of declarations and initializations for a call to OLOG.

```
DATA DIVISION
01  LDA.
    02   LDA–V2RC       PIC S9(4) COMP.
    02   FILLER         PIC X(10).
    02   LDA–RC         PIC S9(4) COMP.
    02   FILLER         PIC X(50).
01  HDA                 PIC X(512).

...

77  USER–ID             PIC X(5)  VALUE ”SCOTT”.
77  USER–ID–L           PIC S9(9) VALUE 5 COMP.
77  PSW                 PIC X(5)  VALUE ”tiger”.
77  PSW–L               PIC S9(9) VALUE 5 COMP.
77  CONN                PIC S9(9) VALUE 0 COMP.
77  CONN–L              PIC S9(9) VALUE 0 COMP.
77  CONN–MODE           PIC S9(9) VALUE 0 COMP.

...

PROCEDURE DIVISION

...

*  CONNECT TO ORACLE IN NON–BLOCKING MODE.
*  CONN–MODE = ZERO INDICATES NON–BLOCKING CONNECTION.
*  HDA MUST BE INITIALIZED TO ZEROS BEFORE CALL TO OLOG.

MOVE LOW–VALUES TO HDA.
CALL ”OLOG” USING LDA, HDA, USER–ID, USER–ID–L,
      PSW, PSW–L, CONN, CONN–L, CONN–MODE.

IF LDA–RC NOT = 0
   PERFORM ORA–ERROR
   GO TO EXIT–STOP

DISPLAY ”Connected to ORACLE as user ”, USER–ID.

...
```

Parameters

Parameter Name	*Type*	*Mode*
LDA	(Address)	IN / OUT
HDA	PIC X(256)	OUT
UID	PIC X(n)	IN
UIDL	PIC S9(9) COMP	IN
PSWD	PIC X(n)	IN
PSWDL	PIC S9(9) COMP	IN
CONN	PIC X(n)	IN
CONNL	PIC S9(9) COMP	IN
CONN–MODE	PIC S9(9) COMP	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

HDA
A host data area. See Chapter 2 for more information on host data areas.

UID
A string containing the user ID, an optional password, and an optional host machine identifier. If you include the password as part of the UID parameter, put it immediately after the user ID and separate it from the user ID with a '/'. Put the host system identifier after the password or user ID, separated by the '@' sign.

If you do not include the password in this parameter, it must be in the PSWD parameter. Examples of valid UID strings are

```
NAME
NAME/PASSWORD
NAME@SERVICENAME
NAME/PASSWORD@SERVICENAME
```

The following string is not a correct example of the USERID parameter:

```
NAME@SERVICENAME/PASSWORD
```

UIDL
The length of the UID string.

PSWD
A string containing the password. If the password is specified as part of the UID string, this parameter can be omitted.

PSWDL
The length of the PSWD parameter.

CONN
A string containing a SQL*Net V2 connect descriptor to connect to a database. If the connect descriptor is specified as part of the UID string, this parameter can be omitted.

CONNL
The length of the CONN parameter.

CONN–MODE
Specifies whether the connection is in blocking or non–blocking mode. Possible values are zero (for blocking) or one (for non–blocking).

See Also OLOGOF, ONBSET, SQLLDA.

OLOGOF

Purpose OLOGOF disconnects an LDA from the Oracle program global area and frees all Oracle resources owned by the Oracle user process.

Syntax

```
CALL ”OLOGOF” USING LDA.
```

Comments A COMMIT is automatically issued on a successful OLOGOF call; all currently opened cursors are closed. If a program logs off unsuccessfully or terminates abnormally, all outstanding transactions are rolled back.

If the program has multiple active logons, a separate call to OLOGOF must be performed for each active LDA. If OLOGOF fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OLOG.

ONBCLR

Purpose ONBCLR places a database connection in non–blocking mode.

Syntax

```
CALL ″ONBCLR″ USING LDA.
```

Comments If there is a pending call on a connection and ONBCLR is called, the pending call, when resumed, will block.

Parameters

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See also OLOG, ONBSET, ONBTST.

ONBSET

Purpose ONBSET places a database connection in non–blocking mode for all subsequent OCI calls on this connection.

Syntax

```
CALL ”ONBSET” USING LDA.
```

Comments ONBSET will succeed if the library is linked in deferred mode and if the network driver supports non–blocking operations.

> **Note:** This call also requires SQL*Net Release 2.1 or higher. It is not compatible with a single–task driver.

Parameters

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See also OLOG, ONBCLR, ONBTST.

ONBTST

Purpose ONBTST tests whether a database connection is in non–blocking mode.

Syntax

```
CALL ”ONBTST” USING LDA.
```

Comments If the connection is in blocking mode, the user may call ONBSET to place the channel in non–blocking mode, if allowed by the network driver.

Parameters

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See also OLOG, ONBCLR, ONBSET.

OOPEN

Purpose OOPEN opens the specified cursor.

Syntax

```
CALL ”OOPEN” USING CURSOR, LDA, <DBN>, <DBNL>,
     <ARSIZE>, <UID>, <UIDL>.
```

Comments OOPEN associates a cursor data area in the program with data areas in the Oracle Server. Oracle uses these areas to maintain state information about the processing of a SQL statement. Status concerning error and warning conditions, as well as other information, such as function codes, is returned to the cursor data area in your program as Oracle processes the SQL statement.

An OCI program can have many cursors active at the same time.

The OPARSE routine is used to parse a SQL statement and associate it with a cursor. In the OCI routines, SQL statements are always referenced using a cursor as the handle.

The *return code* field of the cursor data area indicates the result of the OOPEN. A return code value of zero indicates a successful OOPEN call.

It is possible to issue an OOPEN call on a cursor that is already open. This has no effect on the cursor, but it does affect the value in the Oracle OPEN_CURSORS counter. Repeatedly reopening an open cursor may result in an ORA–01000 error (’maximum open cursors exceeded’). Refer to the *Oracle7 Server Messages* manual for information about what to do if this happens.

See the description of the OPARSE routine in this chapter for an example that uses the OOPEN routine.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	OUT
LDA	(Address)	IN / OUT
DBN	PIC X(n)	IN
DBNL	PIC S9(9) COMP	IN
ARSIZE	PIC S9(9) COMP	IN
UID	PIC X(n)	IN
UIDL	PIC S9(9) COMP	IN

CURSOR
A cursor data area associated with the program.

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

DBN
This parameter is included only for Oracle Version 2 compatibility. It is not used in later versions.

DBNL
This parameter is included only for Oracle Version 2 compatibility. It is not used in later versions.

ARSIZE
The ARSIZE (areasize) parameter is no longer used in Oracle7, as the data areas used by cursors in the Oracle Server are resized automatically as required.

UID
A character string containing the user ID and the password. The password must be separated from the user ID by a '/'.

If the connection to Oracle was established using the Version 2 OLOGON call, then UID and UIDL are used in the OOPEN call. If the OLON routine was used, UID and UIDL are ignored in the OOPEN call.

UIDL
The length of the UID parameter.

See Also OLOG, OPARSE.

OOPT

Purpose OOPT is used to set rollback options for non–fatal Oracle errors involving multi–row INSERT and UPDATE SQL statements. It is also used to set wait options in cases where requested resources are not available; for example, whether to wait for locks.

Syntax

```
CALL ”OOPT” USING CURSOR, RBOPT, WAITOPT.
```

Comments The RBOPT parameter is not supported in Oracle Server Version 6 or later.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN / OUT
RBOPT	PIC S9(9) COMP	IN
WAITOPT	PIC S9(9) COMP	IN

CURSOR
The cursor data area specified in the OOPEN call.

RBOPT
The action to be taken when a non–fatal Oracle error occurs. If this option is set to 0, all errors, even non–fatal errors, cause the current transaction to be rolled back. If this option is set to 2, only the failing row will be rolled back during a non–fatal row–level error. This is the default setting.

WAITOPT
Specifies whether to wait for resources or continue without them if they are currently not available. If this option is set to 0, the program waits indefinitely if resources are not available. This is the default. If this option is set to 4, the program will receive an error return code whenever a resource is requested but is not available. Use of WAITOPT set to 4 can cause many error return codes while waiting for internal resources that are locked for short durations. The only resource errors received are for resources requested by the calling process.

See Also OOPEN.

OPARSE

Purpose OPARSE parses a SQL statement or a PL/SQL block and associates it with a cursor. The parse can optionally be deferred.

Syntax

```
CALL ”OPARSE” USING CURSOR, SQLSTM, SQLL,
      DEFFLG, LNGFLG.
```

Comments OPARSE passes the SQL statement to Oracle for parsing. If the DEFFLG parameter is non–zero, the parse is deferred until the statement is executed or until ODESCR is called to describe the statement. Once the parse is performed, the parsed representation of the SQL statement is stored in the Oracle shared SQL cache. Subsequent OCI calls reference the SQL statement using the cursor name.

An open cursor can be reused by subsequent OPARSE calls within a program, or the program can define multiple concurrent cursors when it is necessary to maintain multiple active SQL statements.

If OPARSE is used to parse a query, the fetch call returns a “fetched column value was truncated” if a column value was truncated and no indicator parameter was defined for that column using ODEFIN.

> **Note:** When OPARSE is called with the DEFFLG parameter set, you cannot receive most error indications until the parse is actually performed. The parse is performed at the first call to ODESCR, OEXEC, OEXN, or OEXFET. However, the SQL statement string is scanned on the client system, and some errors, such as “missing double quote in identifier”, can be returned immediately.

The statement can be any valid SQL statement, or PL/SQL anonymous block. Oracle parses the statement and selects an optimal access path to perform the requested function.

Data Definition Language statements are executed on the parse if you have linked in non–deferred mode or if you have liked with the deferred option and the DEFFLG parameter of OPARSE is zero. If you have linked in deferred mode and the DEFFLG parameter is non–zero, you must call OEXN or OEXEC to execute the statement.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when OPARSE is called in non–deferred mode are not detected until the first non–deferred call is made (usually an execute or describe call).

The example below opens a cursor and parses a SQL statement. The call to OPARSE associates the SQL statement with the cursor.

```
DATA DIVISION.
WORKING STORAGE SECTION.
77    SQL–STMT      PIC X(50) VALUE ”SELECT ename
–     ” FROM emp WHERE deptno = 20”.
PROCEDURE DIVISION.
...
*  Open the cursor.
CALL ”OOPEN” USING CURSOR, LDA, DBN, DBN–L,
     AREASIZE, USER–ID, USER–ID–L.
*  Parse the statement.  Set the deferred flag, and the version
*  flag to one, indicating ”native” database version (V6 or
Oracle7).
*  The deferred parse flag is ignored if the database is V6..
CALL ”OPARSE” USING CURSOR, SQL–STMT, SQL–STMT–L, ONE, ONE.
...
```

SQL syntax error codes are returned in the cursor data area's *return code* field and *parse error offset* field. *Parse error offset* indicates the location of the error in the SQL statement text. See "Cursor Data Area" on page 2 – 4 for a list of the information fields available in the cursor data area after an OPARSE call.

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	(Address)	IN / OUT
SQLSTM	PIC X(n)	IN
SQLL	PIC S9(9) COMP	IN
DEFFLG	PIC S9(9) COMP	IN
LNGFLG	PIC S9(9) COMP	IN

CURSOR
The cursor data area specified in the OOPEN call.

SQLSTM
A string containing a SQL statement.

SQLL
The length of the SQL statement in bytes.

DEFFLG
If non–zero, the parse of the SQL statement is deferred until an ODESCR, OEXEC, OEXN, or OEXFET call is made. Note that bind and define operations are also deferred until the execute or describe step if the program was linked using the deferred mode option. See "Deferred

Statement Execution" on page 2 – 14 for more information about the deferred mode link option.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment.

LNGFLG
The LNGFLG parameter determines the way that Oracle handles the SQL statement or PL/SQL anonymous block. To ensure strict ANSI conformance, Oracle7 defines several datatypes and operations in a slightly different way than Version 6. The table below shows the differences between Version 6 and Oracle7:

Behavior	***V6***	***V7***
CHAR columns are fixed length (including created by a CREATE TABLE statement).	NO	YES
An error is issued if an attempt is made to fetch a null value into an output variable that has no associated indicator variable.	NO	YES
An error is issued if a fetched value is truncated and there is no indicator variable.	YES	NO
Describe (ODESCR) returns internal datatype 1 for CHAR columns.	YES	NO
Describe (ODESCR) returns internal datatype 96 for CHAR columns.	n/a	YES

The LNGFLG parameter has three possible settings:

0	Specifies Version 6 behavior (the database to which you are connected can be either Version 6 or Oracle7).
1	Specifies the normal behavior for the database version to which the program is connected (either Version 6 or Oracle7).
2	Specifies Oracle7 behavior. If you use this value for the parameter and you are not connected to an Oracle7 database, Oracle issues the error `ORA–01011:  Cannot use this language type when talking to a V6 database`

See Also ODESCR, OEXEC, OEXFET, OEXN, OOPEN.

OROL

Purpose OROL rolls back the current transaction.

Syntax

```
CALL ”OROL” USING LDA.
```

Comments The current transaction is defined as the set of SQL statements executed since the OLOG call or the last OCOM or OROL call. If OROL fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
LDA	(Address)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCOM, OLOG.

OSETPI

Purpose OSETPI sets information about the next chunk of data to be processed as part of a piecewise insert, update or fetch.

Syntax `CALL ″OSETPI″ USING CURSOR, PIECE, BUFP, LENP.`

Comments An OCI application uses *osetpi()* to set the information about the next piecewise insert, update, or fetch. The *bufp* parameter is a pointer to either the buffer containing the next piece to be inserted, or to the buffer where the next fetched piece will be stored.

Comments An OCI application uses OSETPI to set the information about the next piecewise insert, update, or fetch. The BUFP parameter is either the buffer containing the next piece to be inserted, or to the buffer where the next fetched piece will be stored.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

See the section “Piecewise Insert, Update and Fetch” in Chatper 2 for more information about piecewise operations and the OSETPI call.

For a sample C language program illustrating the use of OSETPI to perform a piecewise fetch, see the description of the *osetpi()* routine in Chapter 4.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	(Address)	IN
PIECE	PIC S9(2) COMP	IN
BUF	PIC S9(9) COMP	IN
LENP	PIC S9(9) COMP	IN / OUT

CURSOR
The cursor data area associated with the SQL or PL / SQL statement.

PIECE
Specifies the piece being provided or fetched. Possible values are one (for the first piece), two (for a subsequent piece) or three (for the last piece). These values are defined in *ocidfn.h* as OCI_FIRST_PIECE, OCI_NEXT_PIECE, and OCI_LAST_PIECE. Relevant when the buffer is being set after error ORA–03129 was returned by a call to OEXEC.

BUFP
A data buffer. If OSETPI is called as part of a piecewise insert, this pointer must point to the next piece of the data to be transmitted. If OSETPI is called as part of a piecewise fetch, this is a buffer to hold the next piece to be retrieved.

LENP
The length in bytes of the current piece. If a piece is provided, the value is unchanged on return. If the buffer is filled up and part of the data is truncated, LENP is modified to reflect the length of the piece in the buffer.

See Also OBINDPS, ODEFINPS, OGETPI.

SQLLD2

Purpose The SQLLD2 routine is provided for OCI programs that operate as application servers in an X/Open distributed transaction processing environment. SQLLD2 fills in fields in an LDA, according to the connection information passed to it.

Syntax

```
CALL ”SQLLD2” USING LDA, CNAME, CNLEN.
```

Comments OCI programs that operate in conjunction with a transaction manager do not manage their own connections. However, all OCI programs require a valid LDA. You use SQLLD2 to obtain the LDA.

SQLLD2 fills in the LDA using the connection name passed in the CNAME parameter. The CNAME parameter must match the DB_NAME alias parameter of the XA info string of the XA_OPEN call. If the CNLEN parameter is set to zero, an LDA for the default connection is returned. Your program must declare the LDA, then pass it as a parameter.

If you call SQLLD2 and there is no valid connection, the error

```
ORA–01012:  not logged on
```

is returned in the *return code* field of the LDA parameter.

SQLLD2 must be invoked whenever there is an active XA transaction. This means that it must be invoked after XA_OPEN and XA_START, and before XA_END. Otherwise and ORA–01012 error will result.

SQLLD2 is part of SQLLIB, the Oracle Precompiler library. SQLLIB must be linked into all programs that call SQLLD2. See your Oracle system–specific documentation for information about linking SQLLIB.

The example below demonstrates how you can use SQLLD2 to obtain a valid LDA for a specific connection:

```
DATA DIVISION.
WORKING STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END–EXEC.
    01  USER–ID     PIC X(20).
    01  PASSWORD    PIC X(20).
    01  DB–STRING–1 PIC X(14) VALUE ”D:NEWYORK”.
    01  DB–STRING–2 PIC X(17) VALUE ”D:LOSANGELES”.
EXEC SQL END DECLARE SECTION END–EXEC.
...
01  LDA–1.
    02  LDA–RC      PIC S9(4) COMP.
    02  FILLER      PIC X(10).
    02  LDA–V4RC    PIC S9(4) COMP.
    02  FILLER      PIC X(50).
```

```
01  LDA–2.
    02  LDA–RC      PIC S9(4) COMP.
    02  FILLER      PIC X(10).
    02  LDA–V4RC    PIC S9(4) COMP.
    02  FILLER      PIC X(50).
...
PROCEDURE DIVISION.
...
*  Do the first connection.
MOVE ”SCOTT” TO USER–ID.
MOVE ”TIGER” TO PASSWORD.
EXEC SQL DECLARE DBN1 DATABASE END–EXEC.
EXEC SQL CONNECT :USER–ID IDENTIFIED BY :PASSWORD
    AT DBN1 USING :DB–STRING–1 END–EXEC.
*  Get the LDA for the first connection.
CALL ”SQLLDA” USING LDA–1.
*  Do the second connection.
EXEC SQL DECLARE DBN2 DATABASE END–EXEC.
EXEC SQL CONNECT :USER–ID IDENTIFIED BY :PASSWORD
    AT DBN2 USING :DB–STRING–2 END–EXEC.
*  Get the LDA for the second connection.
CALL ”SQLLDA” USING LDA–2.
```

Parameters

Parameter Name	*Type*	*Mode*
LDA	(Address)	OUT
CNAME	PIC X(n)	IN
CNLEN	PIC S9(9) COMP	IN

LDA
The address of a local data area. You must declare this data area before calling SQLLD2.

CNAME
The name of the database connection. If the name consists of all blanks, SQLLD2 returns the LDA for the default connection.

CNLEN
The length of the CNAME parameter. If CNLEN is passed as zero, SQLLD2 returns the LDA for the default connection, regardless of the contents of CNAME.

SQLLDA

Purpose SQLLDA is part of the SQLLIB library that is used with the Oracle Precompilers. This routine is provided for programs that mix both precompiler code and OCI calls. An address of an LDA is passed to SQLLDA; the precompiler fills in the required fields in the LDA.

Syntax

```
CALL ”SQLLDA” USING LDA.
```

Comments If your program contains both precompiler statements and calls to OCI routines, you cannot use OLOG to log on to Oracle. You must use the embedded SQL command

```
EXEC SQL CONNECT ...
```

to log on. However, many OCI routines require a valid LDA. The SQLLDA routine obtains the LDA. SQLLDA is part of SQLLIB, the precompiler library.

SQLLDA fills in the LDA using the connect information from the most recently executed SQL statement. So, you should call SQLLDA immediately after doing the connect with the EXEC SQL CONNECT ... statement.

The example below demonstrates how you can do multiple remote logons in a mixed Precompiler–OCI program. Refer to Chapter 3 in the *Programmer's Guide to the Oracle Precompilers,* R1.5 for additional information about multiple remote logons.

```
DATA DIVISION.
WORKING STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END–EXEC.
    01  USER–ID     PIC X(20).
    01  PASSWORD    PIC X(20).
    01  DB–STRING–1 PIC X(14) VALUE ”D:NEWYORK”.
    01  DB–STRING–2 PIC X(17) VALUE ”D:LOSANGELES”.
EXEC SQL END DECLARE SECTION END–EXEC.
...
01  LDA–1.
    02  LDA–RC      PIC S9(4) COMP.
    02  FILLER      PIC X(10).
    02  LDA–V4RC    PIC S9(4) COMP.
    02  FILLER      PIC X(50).
01  LDA–2.
    02  LDA–RC      PIC S9(4) COMP.
    02  FILLER      PIC X(10).
    02  LDA–V4RC    PIC S9(4) COMP.
    02  FILLER      PIC X(50).
...
PROCEDURE DIVISION.
```

```
...
*  Do the first connection.
MOVE ”SCOTT” TO USER–ID.
MOVE ”TIGER” TO PASSWORD.
EXEC SQL DECLARE DBN1 DATABASE END–EXEC.
EXEC SQL CONNECT :USER–ID IDENTIFIED BY :PASSWORD
    AT DBN1 USING :DB–STRING–1 END–EXEC.
*  Get the LDA for the first connection.
CALL ”SQLLDA” USING LDA–1.
*  Do the second connection.
EXEC SQL DECLARE DBN2 DATABASE END–EXEC.
EXEC SQL CONNECT :USER–ID IDENTIFIED BY :PASSWORD
    AT DBN2 USING :DB–STRING–2 END–EXEC.
*  Get the LDA for the second connection.
CALL ”SQLLDA” USING LDA–2.
```

Parameter

Parameter Name	***Type***	***Mode***
LDA	(Address)	OUT

LDA
The address of a local data area. You must declare this data area before calling SQLLDA.

CHAPTER

6 The OCI Routines for FORTRAN

This chapter describes each subroutine in the OCI library for the OCI FORTRAN programmer. The description of many of the functions includes an example that shows how the routine can be used in an OCI program. Examples are omitted for the simpler subroutines. The description of each routine has five parts:

Purpose	What the routine does.
Syntax	The routine call with its parameter list.
Comments	A detailed description of the routine, including examples.
Parameters	A specific description of each parameter.
See Also	Other routines that affect or are used with this routine.

Be sure to read “Calling OCI Routines” in this chapter. It contains important information about data structures, datatypes, parameter passing conventions, and other important information about the OCI routines.

Calling OCI Routines

This section describes data structures and coding rules that are specific to applications written in FORTRAN. Refer to this section for information about data structures, datatypes, and parameter passing conventions for FORTRAN OCI programs.

Data Structures

To use the OCI routines, you must declare data structures for one or more LDAs and CDAs. In the examples in this Guide, these data areas are declared as INTEGER*2 arrays of 32 elements (total of 64 bytes) as follows:

```
INTEGER*2    LDA(32)
INTEGER*2    CURSOR(32)
```

The offsets of elements in these structures are system dependent. See your Oracle system–specific documentation for the size and alignments of the CDA and LDA components on your system.

If your FORTRAN compiler supports STRUCTURE declarations, you might find it more convenient to define STRUCTUREs for the CDA and the LDA, following the listings in Chapter 2 and Appendix A. Note, however, that the size and offsets of the LDA and CDA still depend on how the C compiler on your system aligns structure elements, since the underlying library structures are defined in C.

FORTRAN Parameter Types

Parameters for the OCI routines are of three types:

- Address
- Integer (INTEGER*1, INTEGER*2, INTEGER*4)
- Character string (CHARACTER*n)

Address parameters pass the address of a variable in your program to Oracle. Although the concept of the address of a variable is alien in many FORTRAN contexts, it is important when using the OCIs. In FORTRAN, all parameters are normally passed to a subprogram by reference, so you simply pass all address parameters as you would normally pass any other parameter. Note that all OUT parameters are address parameters.

However, some FORTRAN compilers do not pass all parameters by reference. For example, Digital Equipment Corporation VAX/VMS compilers pass CHARACTER variables using descriptors. These compilers provide a mechanism (%REF() in the VAX/VMS case) to force passing by reference. You must use this override mechanism to make sure that all address parameters are passed as variable addresses.

Integer parameters are normally four bytes (INTEGER*4). Where two–byte or one–byte integers are required, this is noted in the parameters section. Pass all integer parameters as you would normally pass any other parameter. The OCI library routines will correctly dereference these parameters.

⚠ **Warning:** Even if your FORTRAN compiler supports call by value, do *not* pass integer parameters by value.

Character strings are a special type of parameter. A *length parameter* must be specified for character strings. Length parameters for strings are INTEGER*4 variables specifying the length in bytes of the character string. In the example code in the text, the function LEN_TRIM is used extensively to return the length of a character string, minus any trailing blanks. An example implementation of this function can be found in the sample programs in Appendix C.

FORTRAN Parameter Classification

There are three kinds of parameters in the parameter list of an OCI subroutine:

- required parameters
- optional parameters
- unused parameters

Required Parameters

Required parameters are used by Oracle, and the OCI program must supply a valid value for each required parameter.

Optional Parameters

The use of optional parameters depends on the requirements of your program. The Syntax section for each routine in this chapter indicates optional parameters using square brackets ([ ]).

Unused Parameters

Unused parameters are not used by Oracle, at least for the language being discussed. For example, for cross–language compatibility some OCI functions can take the parameters FMT, FMTL, and FMTT. These are the format string specifier for a packed decimal external datatype, and the string length and type parameters. The packed decimal type is used mainly by COBOL programs, so these parameters are unused in FORTRAN. In the Syntax sections in this chapter, unused parameters are surrounded by angle brackets (< >).

If the optional or unused parameter is an INTEGER*2 or an INTEGER*4, and *is not an address parameter,* you can declare a variable for the parameter, code a –1 value in it, and pass it normally. In this case, you *must* pass the parameter by reference.

Note: A value of –1 should not be passed for unused or optional parameters in the new OBINDPS and ODEFINPS calls. Unused parameters in these calls must be passed a zero. See the descriptions of individual calls for more details about specific parameters.

If a parameter is an *address* parameter, you cannot indicate that it is being omitted by passing a –1 as the value in the parameter. For an address parameter, you can indicate that it is not being used only if your compiler supports a mechanism for passing parameters *by value*, or if you can physically omit items in the parameter list.

For example, you can indicate to Oracle that the INDP parameter is being omitted using the VAX/VMS compiler as follows:

```
CALL OBNDRV(CURSOR, PHNAME, PHNAML, PRGVAR,
1PRGVL, FTYPE, SCALE, %VAL(-1), FMT, FMTL, FMTT)
```

where a –1 is passed by value in place of the INDP parameter.

In summary, if your compiler does not support missing parameters (, ,) or passing parameters by value, you cannot omit an address parameter. In this case, you should either make sure that the value in the parameter will not cause unforeseen actions (put a 0 in the INDP parameter on an OBNDRV or OBNDRN call) or ignore a returned value (if you do not need it, ignore the value in the INDP parameter of ODEFIN after an OFETCH call).

In the code examples in this chapter, optional parameters are always passed. Compiler–specific mechanisms, such as passing by value or omitting parameters, are not used.

Unused parameters are passed in the same way as omitted optional parameters.

Refer to the description of the OBNDRN routine for more examples of how to pass optional and unused parameters.

FORTRAN Parameter Descriptions

In this chapter, parameters for the OCI routines are described in terms of their type and their mode. The type is normally either INTEGER*2 , INTEGER*4, CHARACTER*n, or arrays of these types. In the few cases where an OUT parameter may be of any type, the type is listed as (ADDRESS). In that case, you simply pass a buffer of the appropriate size. The mode of a parameter has three possible values:

IN	A parameter that passes data to Oracle.
OUT	A parameter that receives data from Oracle on this or a subsequent call.
IN / OUT	A parameter that passes data on the call, and receives data on the return from this call *or from a subsequent call.*

Linking FORTRAN OCI Programs

Check your Oracle system–specific documentation for additional information about linking FORTRAN OCI programs. It may be necessary to include extra libraries for linking on some platforms.

OBINDPS

Purpose OBINDPS associates the address of a program variable with a placeholder in a SQL or PL/SQL statement. Unlike older OCI bind calls, OBINDPS can be used to bind placeholders to be used in piecewise operations, or operations involving arrays of structures.

Syntax

```
CALL OBINDPS(CURSOR, OPCODE, SQLVAR, [SQLVL], PVCTX,
        PROGVL, FTYPE, [SCALE], [INDP], [ALENP], [RCODEP],
        PVSKIP, INDSKIP, ALENSKIP, RCSKIP, [MAXSIZ],
        [CURSIZ], [FMT], [FMTL], [FMTT])
```

Comments OBINDPS is used to associate the address of a program variable with a placeholder in a SQL or PL/SQL statement. Additionally, it can indicate that an application will be providing inserted or updated data incrementally at runtime. This piecewise insert is designated in the OPCODE parameter. OBINDPS is also used when an application will be inserting data stored in an array of structures.

> **Note:** This function is only compatible with Oracle Server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

With the introduction of OBINDPS there are now four fully–supported calls for binding input parameters, the other three being the older OBNDRA, OBNDRN and OBNDRV. Application developers should consider the following points when determining which bind call to use:

- OBINDPS is supported only when a program is linked in deferred mode. If it is necessary to link in non–deferred mode, another bind routine must be used. In this case, the ability to handle piecewise operations and arrays of structures is not supported.
- OBINDPS is more complex than the older bind calls. Users who are not performing piecewise operations and are not using arrays of structures may choose to use one of the older routines.
- OBINDPS does not support the ability to do a positional bind. If this functionality is needed, the bind should be performed using OBNDRN.

Unlike older OCI calls, OBINDPS does not accept –1 for any optional or unused parameters. When it is necessary to pass a value to these parameters NULL or 0 should be used instead.

See the sections “Piecewise Insert, Update and Fetch,” and “Arrays of Structures” in Chapter 2 for more information about piecewise operations, arrays of structures, skip parameters and the OBINDPS call.

For a C language example which uses OBINDPS to perform an insert from an array of structures, see the description of the *obindps()* call on page 4 – 6.

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	INTEGER*2(32)	IN / OUT
OPCODE	INTEGER*1	IN
SQLVAR	CHARACTER*n	IN
SQLVL	INTEGER*4	IN
PVCTX	INTEGER*1	IN
PROGVL	INTEGER*4	IN
FTYPE	INTEGER*4	IN
SCALE	INTEGER*4	IN
INDP	INTEGER*2	IN / OUT
ALENP	INTEGER*2	IN
RCODEP	INTEGER*2	OUT
PVSKIP	INTEGER*4	IN
INDSKIP	INTEGER*4	IN
ALENSKIP	INTEGER*4	IN
RCSKIP	INTEGER*4	IN
MAXSIZ	INTEGER*4	IN
CURSIZ	INTEGER*4	IN / OUT
FMT	CHARACTER*6	IN
FMTL	INTEGER*4	IN
FMTT	INTEGER*4	IN

Note: Since the OBINDPS call can be used in a variety of different circumstances, some items in the following list of parameter descriptions include different explanations for how the parameter is used for piecewise operations, arrays of structures and standard scalar or array binds.

Standard scalar and array binds are those binds which were previously possible using other OCI bind calls (OBNDRA, OBNDRN, and OBNDRV).

CURSOR
The CDA associated with the SQL statement or PL/SQL block being processed.

OPCODE
Piecewise bind: pass as 0.

Arrays of structures or standard bind: pass as 1.

SQLVAR
A character string holding the name of a placeholder (including the preceding colon, e.g., ":VARNAME") in the SQL statement being processed.

SQLVL
The length of the character string in SQLVAR, including the preceding colon. For example, the placeholder ":EMPLOYEE" has a length of nine.

PVCTX
Piecewise bind: A pointer to a context block entirely private to the application. This should be used by the application to store any information about the column being bound. One possible use would be to store a pointer to a file which will be referenced later. Each bind variable can then have its own separate file pointer. This pointer can be retrieved during a call to OGETPI.

Arrays of structures or standard bind: A pointer to a program variable or array of program variables from which input data will be retrieved when the SQL statement is executed. For arrays of structures this should point to the first scalar element in the array of structures being bound. This parameter is equivalent to the PROGV parameter from the older OCI bind calls.

PROGVL
Piecewise bind: This should be passed in as the maximum possible size of the data element of type FTYPE.

Arrays of structures or standard bind: This should be passed as the length in bytes of the datatype of the program variable, array element or the field in a structure which is being bound.

FTYPE
The external datatype code of the program variable being bound. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. See the section "External

Datatypes” in Chapter 3 for a list of datatype codes, and the listings of *ocidem.h* and *ocidfn.h* in Appendix A for lists of constant definitions corresponding to datatype codes.

For piecewise operations, the valid datatype codes are 1 (VARCHAR2), 5 (STRING), 8 (LONG) and 24 (LONG RAW).

SCALE
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

INDP
Pointer to an indicator variable or array of indicator variables. For arrays of structures this may be an interleaved array of column–level indicator variables. See page 2 – 29 for more information about indicator variables.

ALENP
Piecewise bind: pass as 0.

Arrays of structures or standard bind: A pointer to a variable or array containing the length of data elements being bound. For arrays of structures, this may be an interleaved array of column–level length variables. The maximum usable size of the array is determined by the *maxsiz* parameter.

RCODEP
Pointer to a variable or array of variables where column–level error codes are returned after a SQL statement is executed. For arrays of structures, this may be an interleaved array of column–level return code variables.

Typical error codes would indicate that data in PROGV has been truncated (ORA–01406) or that a null occurred on a SELECT or PL/SQL FETCH (ORA–01405).

PVSKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of structures holding program variables being bound. In general, this value will be the size of one structure. If a standard array bind is being performed, this value should equal the size of one element of the array being bound.

INDSKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of indicator variables associated with an array holding program data to be inserted. This parameter will either equal the size of one indicator parameter structure (for arrays of structures) or the size of one indicator variable (for standard array bind).

ALENSKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array of data lengths associated with an array holding program data to be inserted. This parameter will either equal the size of one length variable structure (for arrays of structures) or the size of one length variable (for standard array bind).

RCSKIP
Piecewise bind or standard scalar bind: pass as zero or NULL.

Arrays of structures or standard array bind: This is the skip parameter for an array used to store returned column–level error codes associated with the execution of a SQL statement. This parameter will either equal the size of one return code structure (for arrays of structures) or the size of one return code variable (for standard array bind).

MAXSIZ
The maximum size of an array being bound to a PL/SQL table. Values range from 1 to 32512, but the maximum size of the array depends on the datatype. The maximum array size is 32512 divided by the internal size of the datatype.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to zero for SQL scalar or array binds.

CURSIZ
A pointer to the actual number of elements in the array being bound to a PL/SQL table.

If PROGV is an IN parameter, set the CURSIZ parameter to the size of the array being bound. If PROGV is an OUT parameter, the number of valid elements being returned in the PROGV array is returned after PL/SQL block is executed.

This parameter is only relevant when binding to PL/SQL tables. Set this parameter to zero for SQL scalar or array binds.

FMT
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

FMTL
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

FMTT
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

See Also OBNDRA, OBNDRN, OBNDRV, ODEFINPS, OGETPI, OSETPI.

OBNDRA

Purpose OBNDRA binds the address of a program variable or array to a placeholder in a SQL statement or PL/SQL block.

Syntax

```
CALL OBNDRA(CURSOR, SQLVAR, SQLVL,
     PROGV, PROGVL, FTYPE, [SCALE], [INDP],
     [ALEN], [ARCODE], [MAXSIZ], [CURSIZ]
     <FMT>, <FMTL>, <FMTT>)
```

Comments You can use OBNDRA to bind scalar variables or arrays in your program to placeholders in a SQL statement or a PL/SQL block. The OBNDRA routine has a parameter, ALEN, that allows you to change the size of the bound variable without actually rebinding the variable.

> **Note:** Applications running against a release 7.3 or later server that need to perform piecewise operations or utilize arrays of structures must use the newer OBINDPS routine instead of OBNDRA.

When you bind arrays in your program to PL/SQL tables, you must use OBNDRA, since this routine provides additional parameters that allow you to control the maximum size of the table and to retrieve the current table size after the block has been executed.

Call OBNDRA after you call OPARSE to parse the statement containing the PL/SQL block and before calling OEXEC to execute it.

Once you have bound a program variable, you can change the value in the variable (PROGV) and length of the variable (PROGVL) and re–execute the block without rebinding.

However, if you need to change the type of the variable, you must reparse and rebind before re–executing.

The following short but complete example program shows how you can use OBNDRA to bind arrays in a FORTRAN program to tables in a PL/SQL block.

```
    IMPLICIT INTEGER*4 (A–Z)

* Logon and CDA, plus connect strings
    INTEGER*2      CDA(32), LDA(32)
    INTEGER*2      HDA(256)
    CHARACTER*20   UID, PWD

* Variables for OBNDRA
    INTEGER*2      DSALEN(3), DSRC(3)
    INTEGER*4      DSCS
    INTEGER*2      PNALEN(3), PNRC(3)
    INTEGER*4      PNCS

* Input variables
    CHARACTER*20   DESCRP(3)
    INTEGER*4      PRTNOS(3)

* Placeholders
    CHARACTER*10   DSPH, PNPH

* Variables to hold SQL statements
    CHARACTER*20   DRPTBL, CRTTBL
    CHARACTER*500  CRTPKG, PKGBDY, PLSBLK

* Initialize all variables
    UID = ’scott’
    PWD = ’tiger’

    DESCRP(1) = ’Frammis’
    DESCRP(2) = ’Widget’
    DESCRP(3) = ’Thingie’
    PRTNOS(1) = 12125
    PRTNOS(2) = 23169
    PRTNOS(3) = 12126

    DSPH = ’:DESC’
    PNPH = ’:PARTS’

    DRPTBL = ’DROP TABLE part_nos’
    CRTTBL = ’CREATE TABLE part_nos
   +          (partno NUMBER(8), description CHAR(20))’
    CRTPKG = ’CREATE OR REPLACE PACKAGE update_parts AS
   +             TYPE part_number IS TABLE OF part_nos.partno%TYPE
   +                 INDEX BY BINARY_INTEGER;
   +             TYPE part_description IS TABLE OF
   +                 part_nos.description%TYPE
   +                 INDEX BY BINARY_INTEGER;
   +             PROCEDURE add_parts (n IN INTEGER,
   +                                  descrip IN part_description,
   +                                  partno INpart_number);
   +           END update_parts;’


    PKGBDY = ’CREATE OR REPLACE PACKAGE BODY update_parts AS
   +          PROCEDURE add_parts (n IN INTEGER,
   +            descrip IN part_description,
   +             partno IN part_number);
   +          BEGIN
   +             FOR i IN 1..n LOOP
   +                INSERT INTO part_nos
```

```
     +                    VALUES (partno(i), descrip(i));
     +              END LOOP;
     +          END;
     +        END update_parts;'

*  PL/SQL anonymous block, calls update_parts
      PLSBLK = 'BEGIN add_parts(3, :DESC, :PARTS); END;'

*  Connect to Oracle in non-blocking mode.
*  HDA must be initialized to zeros before call to OLOG.
      DATA HDA/256*0/
      CALL OLOG(LDA, HDA, UID, LEN_TRIM(UID),
     +          PWD, LEN_TRIM(PWD), 0, -1, 0)
      IF (LDA(7) .NE. 0) THEN
        PRINT *, 'Cannot connect with username scott.'
        GOTO 999
      END IF
      PRINT *, 'Connected to Oracle.'

*  Open the cursor
      CALL OOPEN(CDA, LDA, UID, -1, -1, UID, -1)
      IF (CDA(7) .NE. 0) THEN
        PRINT *, ' Error opening cursor. Exiting....'
        GOTO 999
      END IF

*  Parse drop table, also executes
      CALL OPARSE(CDA, DRPTBL, LEN_TRIM(DRPTBL), 1, 2)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 999
      ENDIF

*  Parse create table, also executes
      CALL OPARSE(CDA, CRTTBL, LEN_TRIM(CRTTBL), 1, 2)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 999
      ENDIF

*  Parse and execute CREATE PACKAGE
      CALL OPARSE(CDA, CRTPKG, LEN_TRIM(CRTPKG), 1, 2)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 999
      ENDIF
      CALL OEXEC(CDA)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(CDA, LDA)
        GOTO 999
      ENDIF
* Parse and execute CREATE PACKAGE BODY
      CALL OPARSE(CDA, PKGBDY, LEN_TRIM(PKGBDY), 1, 2)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 999
      ENDIF
      CALL OEXEC(CDA)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(CDA, LDA)
        GOTO 999
      ENDIF

* Bind the arrays to the placeholders
      DO 10 I = 1, 3
        DSALEN(I) = 20
10      PNALEN(I) = 4

      CALL OBNDRA(CDA, DSPH, LEN_TRIM(DSPH), DESCRP, 20, 1, –1,
     +             0, DSALEN, DSRC, 10, 3, 0, –1, –1)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 999
      ENDIF

      CALL OBNDRA(CDA, PNPH, LEN_TRIM(PNPH), PRTNOS, 4, 3, –1,
     +             0, PNALEN, PNRC, 10, 3, 0, –1, –1)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 999
      ENDIF

*  Execute the PL/SQL block
      CALL OEXEC(CDA)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 999
      ENDIF

      PRINT *, 'Parts table updated.'


999 CALL OCLOSE(CDA)
      CALL OLOGOF(LDA)

END
```

```
    SUBROUTINE ERRRPT(LDA, CDA)
    INTEGER*2 LDA(32), CDA(32)

    CHARACTER*80  MSG

    CALL OERHMS(LDA, CDA(7), MSG, 80)

    PRINT ’(/, 1X, A)’, MSG
    PRINT ’(1X, A, I3)’, ’processing OCI routine’, CDA(6)

    RETURN
    END


    INTEGER FUNCTION LEN_TRIM(STRING)
    CHARACTER*(*) STRING

    INTEGER NEXT

    DO 10 NEXT = LEN(STRING), 1, –1
      IF (STRING(NEXT : NEXT) .NE. ’ ’) THEN
        LEN_TRIM = NEXT
        RETURN
      ENDIF
10  CONTINUE

    LEN_TRIM = 0

    RETURN
    END
```

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	INTEGER*2(32)	IN / OUT
SQLVAR	CHARACTER*n	IN
SQLVL	INTEGER*4	IN
PROGV	(Address) (1)	IN / OUT(2)
PROGVL	INTEGER*4	IN
FTYPE	INTEGER*4	IN
SCALE	INTEGER*4	IN
INDP	INTEGER*2	IN / OUT(2)
ALEN	INTEGER*2	IN / OUT
ARCODE	INTEGER*2	OUT (3)

Parameter Name	*Type*	*Mode*
MAXSIZ	INTEGER*4	IN
CURSIZ	INTEGER*4	IN / OUT(2)
FMT	CHARACTER*6	IN
FMTL	INTEGER*4	IN
FMTT	INTEGER*4	IN

Note 1. PROGV is the address of the program variable.
Note 2. IN / OUT parameter on the execute call.
Note 3. Value returned; OUT parameter on the execute call.

CURSOR
This is a CDA within the program.

SQLVAR
A character string containing the name of a placeholder (including the preceding colon) in the SQL statement.

SQLVL
The length of the character string SQLVAR (including the preceding colon). For example, the placeholder :DEPT has a length of five.

PROGV
The address of a program variable or table of program variables from which data will be retrieved when OEXEC is issued.

PROGVL
The length in bytes of the program variable or array element. Since OBNDRA might be called only once for many different PROGV values on successive execute calls, PROGVL must contain the maximum length of PROGV.

> **Note:** The PROGVL parameter is an INTEGER. On some systems, however, the underlying parameter type in the OCI library might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this appears to limit the maximum length of the buffer to 64K bytes. To bind a longer buffer for these datatypes, set PROGVL to –1 and pass the actual data area length (total buffer length – 4) in the first four bytes of PROGV. Set this value before calling OBNDRA.

FTYPE
The external datatype of the program variable as defined within the user program. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. A list of external datatypes and type codes is in Chapter 3.

SCALE
Specifies the number of digits to the right of the decimal point for fields where FTYPE is 7 (PACKED DECIMAL). SCALE is ignored for all other types.

INDP
An indicator parameter, or array of indicator variables if *progv* is an array. As an array, INDP must contain at least the same number of elements as PROGV. See Chapter 2 for more information about indicator variables.

ALEN
A pointer to an array of elements containing the length of the data. This is the effective length of the bind variable element, not the size of the array. For example, if the PROGV parameter is an array declared as

```
CHARACTER*10  ARR(5)
```

then ALEN should also point to an array of at least five elements. The maximum usable size of the array is determined by the MAXSIZ parameter.

If ARR in the above example is an IN parameter, each element in the array pointed to by ALEN should be set to the length of the data in the corresponding element in the ARR array (ten in this example) before the execute call.

If ARR in the above example is an OUT parameter, the length of the returned data appears in the array pointed to by ALEN after the PL/SQL block is executed.

Once the bind is done using OBNDRA, you can change the data length of the bind variable without rebinding. However, the length cannot be greater than that specified in ALEN.

ARCODE
An array containing the column–level error return codes. This parameter is an array that will contain the error code for the bind variable after the execute call. The error codes that can be returned in ARCODE are those that indicate that data in PROGV has been truncated, or that a null occurred on a SELECT or PL/SQL FETCH; for example, ORA–01405 or ORA–01406.

If OBNDRA is being used to bind an array of elements (that is, MAXSIZ is greater than one), then ARCODE must also point to an array of at least equal size.

MAXSIZ
The maximum size of the array being bound. Values range from 1 to 32512, but the maximum size of the array depends on the datatype. The maximum array size is 32512 divided by the internal size of the datatype. If OBNDRA is being used to bind a scalar, set this parameter to zero. A value of one means an array one element long.

CURSIZ
The actual number of elements in the array.

If PROGV is an IN parameter, set the CURSIZ parameter to the size of the array being bound. If PROGV is an OUT parameter, the number of valid elements being returned in the PROGV array is returned after the SQL statement or PL/SQL block is executed.

To use OBNDRA to bind a scalar, you must be able to pass a zero *by value* in this parameter. If your FORTRAN compiler does not have a mechanism for passing parameters by value, you must use OBNDRV to bind scalars. It does not work to pass a zero by reference to indicate that a scalar is being bound.

FMT
Not normally used in FORTRAN. See the description of OBNDRN on page 6 – 20 for more information.

FMTL
Not normally used in FORTRAN. See the description of OBNDRN on page 6 – 20 for more information.

FMTT
Not normally used in FORTRAN. See the description of OBNDRN on page 6 – 20 for more information.

See Also OBINDPS, OBNDRV, OEXEC, OEXN, OPARSE.

OBNDRN OBNDRV

Purpose OBNDRN and OBNDRV associate the address of a program variable with the specified placeholder in the SQL statement. The placeholder is identified by name for the OBNDRV function and by number for OBNDRN. Values must be placed in the variables before the SQL statement is executed.

Syntax

```
CALL OBNDRN(CURSOR, SQLVN, PROGV, PROGVL, FTYPE,
            SCALE, [INDP], <FMT>, <FMTL>, <FMTT>)
CALL OBNDRV(CURSOR, SQLVAR, [SQLVL], PROGV, PROGVL,
            FTYPE, <SCALE>, [INDP], <FMT>,
            <FMTL>, <FMTT>)
```

Comments You can call either OBNDRV or OBNDRN to bind the address of a variable in your program to a placeholder in the SQL statement being processed. If an application needs to perform piecewise operations or utilize arrays of structures, you must bind your variables using OBINDPS instead.

If you use OBNDRV, the placeholder in the SQL statement is a colon (:) followed by a SQL identifier. For example, the SQL statement

```
SELECT ename,sal,comm FROM emp WHERE deptno = :DEPT AND
    comm > :MINCOM
```

has two placeholders, :DEPT and :MINCOM.

If you use OBNDRN, the placeholders in the SQL statement consist of a colon followed by a literal integer in the range 1 to 255. The SQL statement

```
SELECT ename,sal,comm FROM emp WHERE deptno = :2 AND comm > :1
```

has two placeholders, :1 and :2. An OBNDRV call that binds the :DEPT placeholder in the first SQL statement above to the program variable *DEPTNO* is

```
    INTEGER*2    CURSOR(32), INDP
    INTEGER      DEPTNO, PHDLEN
    CHARACTER*5  PHDEPT
    CHARACTER*6  FMT
*   ..
    PHDEPT = ':DEPT'
    PHDLEN = 5
    INDP = 0
    CALL OBNDRV(CURSOR, PHDEPT, PHDLEN, DEPTNO, 4, 3,
  1              –1, INDP, FMT, –1, –1)
```

Note that the INDP parameter is not really used; therefore, it is set to zero in the program so that a null will not be the effective bind variable. The FMT parameter is declared, but never set or used. The –1 length that is passed for the FMTL parameter, and the fact that the datatype is INTEGER (DTYPE = 3) and not PACKED DECIMAL (DTYPE = 7, ensures that the empty format string is never accessed.

If you use OBNDRN, the parameter SQLVN identifies the placeholder by number. If SQLVN is set to 1, the program variable is bound to the placeholder :1. For example, OBNDRN is called to bind the program variable MINCOM to the placeholder :2 in the second SQL statement above as follows:

```
CALL OBNDRN(CURSOR, 2, MINCOM, 4, 3, –1, INDP, FMT, –1, –1);
```

where the placeholder :2 is indicated in the SQLVN parameter by passing the value 2. The SQLVN parameter can be a literal or a variable.

In a PL/SQL block, you cannot use OBNDRN to bind program variables to placeholders, since PL/SQL does not recognize numbered placeholders. Always use OBNDRV and named placeholders in PL/SQL blocks.

The OBNDRV or OBNDRN routine must be called after you call OPARSE to parse the SQL statement and before calling OEXEC or OEXFET to execute it.

If the value of the program variable changes, you do not need to rebind before re–executing, since it is the address of the variable that is bound, not the value. However, if you change the actual program variable, you must rebind before re–executing.

For example, if you have bound the address of DEPTN to :DEPT and you now want to use NDEPTN when executing the SQL statement above, you must call OBNDRV again to bind the new program variable to the placeholder.

Also, you cannot in general rebind a placeholder to a variable of a different type without reparsing the SQL statement. So, if you need to rebind with a different variable type, call OPARSE first to reparse the statement.

You should avoid using OBNDRV and OBNDRN after an ODESCR call, since bind variables can occur in a select–list item, and if bound after the describe, the size or datatype may change.

At the time of the bind, Oracle stores the address of the program variable. If the same placeholder occurs more than once in the SQL statement, a single call to OBNDRN or OBNDRV binds all occurrences of the placeholder to the bind variable.

The completion status of the bind is returned in the *return code* field of the CDA. A return code of zero indicates successful completion.

If your program is linked using the deferred mode option, bind errors that would normally be returned immediately are not detected until the bind operation is actually performed. This happens on the first describe (ODESCR) or execute (OEXEC, OEXN, or OEXFET) call after the bind.

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	INTEGER*2(32)	IN / OUT
SQLVAR	CHARACTER*n	IN
SQLVL	INTEGER*4	IN
SQLVN	INTEGER*4	IN
PROGV	(Address)	IN (1)
PROGVL	INTEGER*4	IN
FTYPE	INTEGER*4	IN
SCALE	INTEGER*4	IN
INDP	INTEGER*2	IN (1)
FMT	CHARACTER*6	IN
FMTL	INTEGER*4	IN
FMTT	INTEGER*4	IN

Note 1. Values are IN parameters for OEXEC, OEXN, or OEXFET.

CURSOR
A CDA.

SQLVAR
Used only with OBNDRV, this parameter is a character string containing the name of a placeholder (including the preceding colon) in the SQL statement.

SQLVL
Used only with OBNDRV, the SQLVL parameter is the length of the character string SQLVAR (including the preceding colon). For example, the placeholder :EMPLOYEE has a length of nine.

SQLVN
Used only with OBNDRN**,** this parameter specifies a placeholder in the SQL statement referenced by the cursor by number. For example, if SQLVN is 2, it refers to all placeholders identified by :2 within the SQL statement.

PROGV
A program variable or array of program variables that provide input data when at execute time.

PROGVL
The length in bytes of the program variable or array element. Since OBNDRV or OBNDRN might be called only once for many different PROGV values on successive execute calls, PROGVL must contain the maximum length of PROGV.

> **Note:** The PROGVL parameter is an INTEGER. On some systems, however, the underlying parameter type in the OCI library might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this appears to limit the maximum length of the buffer to 64K bytes. To bind a longer buffer for these datatypes, set PROGVL to –1 and pass the actual data area length (total buffer length – 4) in the first four bytes of PROGV. Set this value before calling OBNDRN or OBNDRV.

FTYPE
The external datatype of the program variable as defined within the user program. Oracle converts the program variable from external to internal format before it is bound to the SQL statement. See "External Datatype Codes" on page 3 – 3 for a complete list of external datatypes.

SCALE
Not normally used in FORTRAN. See the description of OBNDRN on page 6 – 20 for more information.

INDP
INDP is an indicator parameter. If the value is negative when the statement is executed, the column is set to null; otherwise, it is set to the value in PROGV. If the array interface is being used, this parameter must be an array of two–byte integers.

FMT
Not normally used in FORTRAN. See the description of OBNDRN on page 6 – 20 for more information.

FMTL
Not normally used in FORTRAN. See the description of OBNDRN on page 6 – 20 for more information.

FMTT
Not normally used in FORTRAN. See the description of OBNDRN on page 6 – 20 for more information.

See Also OBINDPS, OBNDRA, ODESCR, OEXEC, OEXFET, OEXN, OPARSE.

OBREAK

Purpose OBREAK performs an immediate (asynchronous) abort of any currently executing OCI routine that is associated with the specified LDA. It is normally used to stop a long–running execute or fetch call that has not yet completed.

Syntax

```
CALL OBREAK(LDA)
```

Comments If no OCI routine is active when OBREAK is called, OBREAK will be ignored unless the next OCI routine called is OFETCH. In this case, the subsequent OFETCH call will be aborted.

OBREAK is the *only* OCI routine that you can call when another OCI routine is in progress. It should not be used when a logon (OLOG) is in progress, since the LDA is in an indeterminate state. The OBREAK routine cannot return a reliable error status to the LDA, since it might be called when the Oracle internal status structures are in an inconsistent state.

> **Note:** *obreak()* aborts the currently executing OCI function not the connection.

OBREAK is not guaranteed to work on all operating systems and does not work on all protocols. In some cases, OBREAK may work with one protocol on an operating system, but may not work with other protocols on the same operating system.

See the description of *obreak()* in Chapter 4 for a code example in C which runs under most UNIX operating systems..

Parameter

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OLOG.

OCAN

Purpose

OCAN informs Oracle that the operation in progress for the specified cursor is complete. The OCAN routine thus frees any resources associated with the specified cursor, but keeps the cursor associated with the associated data areas in the Oracle Server.

Syntax

```
CALL OCAN(CURSOR)
```

Comments

OCAN informs Oracle that the operation in progress for the specified cursor is complete. The OCAN function thus frees any resources associated with the specified cursor, but keeps the cursor associated with its parsed representation in the shared SQL area.

For example, if you require only the first row of a multi–row query, you can call OCAN after the first OFETCH operation to inform Oracle that your program will not perform additional fetches.

If you use the OEXFET function to fetch your data, specifying a non–zero value for the OEXFET CANCEL parameter has the same effect as calling OCAN after the fetch completes.

Parameter

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT

CURSOR
The address of the cursor data area specified in the OPARSE call associated with the query

See Also

OEXFET, OFEN, OFETCH, OPARSE.

OCLOSE

Purpose OCLOSE disconnects a cursor from the data areas associated with it in the Oracle Server.

Syntax

```
CALL OCLOSE(CURSOR)
```

Comments The OCLOSE routine frees all resources obtained by the OOPEN, OPARSE**,** and execute and fetch operations using this cursor. If OCLOSE fails, the *return code* field of the CDA contains the error code.

Parameter

Parameter Name	***Type***	***Mode***
CURSOR	INTEGER*2(32)	IN / OUT

CURSOR
The CDA specified in the OOPEN call.

See Also OOPEN, OPARSE.

OCOF

Purpose OCOF disables *autocommit*, that is, automatic commit of every SQL data manipulation statement.

Syntax

```
CALL OCOF(LDA)
```

Comments By default, autocommit is already disabled at the start of an OCI program. Having autocommit ON can have a serious impact on performance. So, if the OCON (autocommit ON) routine is used to enable autocommit for some special circumstance, OCOF should be used to disable autocommit as soon as it is practical.

If OCOF fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCOM, OCON, OLOG.

OCOM

Purpose OCOM commits the current transaction.

Syntax

```
CALL OCOM(LDA)
```

Comments The current transaction starts from the OLOG call or the last OROL or OCOM call, and lasts until an OCOM, OROL, or OLOGOF call is issued.

If OCOM fails, the reason is indicated in the *return code* field of the LDA.

Do not confuse the OCOM call (COMMIT) with the OCON call (turn autocommit ON).

Parameter

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCON, OLOG, OLOGOF, OROL.

OCON

Purpose OCON enables *autocommit*, that is, automatic commit of every SQL data manipulation statement.

Syntax

```
CALL OCON(LDA)
```

Comments By default, autocommit is disabled at the start of an OCI program. This is because it is more expensive and less flexible than placing OCOM calls after each logical transaction. When autocommit is on, a zero in the *return code* field after calling OEXEC indicates that the transaction has been committed.

If OCON fails, the reason is indicated in the *return code* field of the LDA.

If it becomes necessary to turn autocommit on for some special circumstance, it is advisable to follow that with a call to OCOF to disable autcommit as soon as it is practical in order to maximize performance.

Do not confuse the OCON routine with the OCOM (COMMIT) routine.

Parameter

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCOF, OCOM, OLOG.

ODEFIN

Purpose ODEFIN defines an output buffer for a specified select–list item of a SQL query.

Syntax

```
CALL ODEFIN(CURSOR, POS, BUF, BUFL, FTYPE, <SCALE>,
            [INDP], <FMT>, <FMTL>, <FMTT>,
            [RLEN], [RCODE])
```

Comments An OCI program must call ODEFIN once for each select–list item in the SQL statement. Each call to ODEFIN associates an output variable in the program with a select–list item of the query. The output variable must be a scalar or string or, for use with OEXFET or OFEN, an array of scalars or strings. It must be compatible with the external datatype specified in the FTYPE parameter. See Table 3 – 2 for a list of datatypes and compatible variables.

> **Note:** Applications running against a release 7.3 or later server that need to perform piecewise operations or utilize arrays of structures must use the newer ODEFINPS routine instead of ODEFIN.

Oracle places data in the output variables when the program calls OFETCH, OFEN, or OEXFET.

If you do not know the number, lengths, and datatypes of the select–list items, you obtain this information by calling ODESCR before calling ODEFIN.

Call ODEFIN after parsing the SQL statement. Call ODEFIN before calling the fetch routine (OFETCH, OFEN, or OEXFET).

ODEFIN associates output variables with select–list items using the position index of the select–list item in the SQL statement. Position indices start at 1 for the first (or left–most) select–list item. For example, in the SQL statement

```
SELECT ENAME, EMPNO, SAL FROM emp WHERE sal > :MINSAL
```

the select–list item SAL is in position 3, EMPNO is in position 2, and ENAME is in position 1.

If the type or length of bound variables changes between queries, you must reparse and rebind before re–executing.

You call ODEFIN to associate output variables with the select–list items in the above statement as demonstrated in the following sample code fragment:

```
      CHARACTER*10   ENAME
      INTEGER*2      ENAMEL, INDP
      INTEGER*2      RCODES(3), RETL(3)
      INTEGER        EMPNUM, SCALE
      REAL*4         SALARY

*    The following variables are declared and passed as
*    arguments to ODEFIN. They are either not used,
*    or if used (like INDP), the values can be ignored after the
*    fetch.

      INTEGER*2          INDP
      INTEGER            FMTL, FMTT, SCALE
      CHARACTER*2       FMT

      ...

      FMTL = 0

      CALL ODEFIN(CURSOR, 1, ENAME, ENAMEL, 1,
     1      SCALE, INDP, FMT, FMTL,
     2      FMTT, RETL(1), RCODES(1))

      CALL ODEFIN(CURSOR, 2, EMPNUM, 4, 3, SCALE,
     1      INDP, FMT, FMTL, FMTT,
     2      RETL(2), RCODES(2))

      CALL ODEFIN(CURSOR, 3, SALARY, 4, 4, SCALE,
     1      INDP, FMT, FMTL, FMTT,
     2      RETL(3), RCODES(3))
```

where ENAMEL contains a known length value (it can be obtained by calling ODESCR).

Oracle provides return code information at the row level using the *return code* field in the CDA. If you require return code information at the column level, you must include the optional RCODE parameter, as in the examples above. During each fetch, Oracle sets RCODE for each select–list item processed. This return parameter indicates either successful completion (zero) or an exception condition, such as a null item fetched, the item fetched was truncated, or other non–fatal column errors. The following codes are some of the error codes that can be returned in the RCODE parameter:

Code	*Meaning*
0	Success.
1405	A null was fetched.
1406	ASCII or string buffer data was truncated. The converted data from the database did not fit into the buffer. Check the value in INDP, if specified, or RLEN, to determine the original length of the data.
1454	Invalid conversion specified: integers not of length 1, 2, or 4; reals not of length 4 or 8; invalid packed decimal conversions; packed decimal with more than 38 digits specified.
1456	Real overflow. Conversion of a database column or expression would overflow a floating–point number on this machine.
3115	Unsupported datatype.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2 (32)	IN / OUT
POS	INTEGER*4	IN
BUF	(Address)	IN (1)
BUFL	INTEGER*4	IN
FTYPE	INTEGER*4	IN
SCALE	INTEGER*4	IN
INDP	INTEGER*2	IN (1)
FMT	CHARACTER*6	IN
FMTL	INTEGER*4	IN
FMTT	INTEGER*4	IN
RLEN	INTEGER*2	IN (1)
RCODE	INTEGER*2	IN (1)

Note 1. Values in the BUF, INDP, RLEN, and RCODE parameters are valid only after the subsequent OFETCH, OFEN, or OEXFET routine returns. These are effectively OUT parameters for those routines.

CURSOR
The CDA specified in the parse call for the SQL statement.

POS
The position index for a select–list item in the query. Position indices start at 1 for the first (or left–most) select–list item. The ODEFIN routine uses the position index to associate output buffers with a given select–list item. If you specify a position index greater than the number of items in the select–list or less than 1, ODEFIN returns a “variable not in select list” error in the *return code* field of the CDA.

> **Note:** If the ODEFIN call is deferred until execution using the deferred mode link option, this error is returned on the OEXEC, OEXN, or OEXFET call.

BUF
The address of the variable in the user program that will receive the data when the fetch is performed. The variable can be of any type into which an Oracle column or expression result can be converted. See Chapter 3 for more information about datatype conversions.

> **Note:** If ODEFIN is being called to define an array fetch operation using OEXFET or OFEN, then the BUF parameter must be the address of an array large enough to hold the set of items that will be fetched.

BUFL
The length in bytes of the buffer being defined. If BUF is an array, this is the size of one element of the array.

> **Note:** The BUFL parameter is an INTEGER. On some systems, however, the underlying parameter type in the OCI library might be only two bytes. When binding LONG VARCHAR and LONG VARRAW buffers, this appears to limit the maximum length of the buffer to 64K bytes. To bind a longer buffer for these datatypes, set BUFL to –1 and pass the actual data area length (total buffer length – 4) in the first four bytes of BUF. Set this value before calling ODEFIN.

FTYPE
The code for the external datatype to which the select–list item is converted before it is moved to the output buffer. See Chapter 3 for a list of the external datatypes and codes.

SCALE
Not normally used in FORTRAN. See the description of the ODEFIN routine in Chapter 5 for information about the packed decimal datatype.

INDP
An INTEGER*2 indicator parameter that must be passed by reference. The value of INDP after OFETCH or OFEN is executed indicates whether the select–list item fetched was null, truncated, or not altered, by returning one of the following values:

Negative	The item fetched was null.
Zero	The item fetched is stored in the output buffer unaltered.
Positive	The length of the item is greater than the specified length of the program buffer; the returned value has been truncated. The positive value returned by the indicator variable is the actual length of the item *before* truncation.

If ODEFIN is being called to define an *array* fetch operation, using the OEXFET or OFEN, then the INDP parameter must be the address of an array large enough to hold indicator variables for all the items that will be fetched.

Note: The INDP parameter offers only a subset of the functionality provided by the RLEN and RCODE parameters.

FMT
Not normally used in FORTRAN. See the description of the ODEFIN routine on page 6 – 31 for more information.

FMTLEN
Not normally used in FORTRAN. See the description of the ODEFIN routine on page 6 – 31 for more information.

FMTTYP
Not normally used in FORTRAN. See the description of the ODEFIN routine on page 6 – 31 for more information.

RLEN
An INTEGER*2 variable or array. Oracle places the actual length of the returned column in this variable after a fetch is executed. If ODEFIN is being used to associate an array with a select–list item, the RLEN parameter must also be an array of INTEGER*2 variables of the same dimension as the BUF parameter. Return lengths are valid after the fetch.

RCODE
An INTEGER*2 variable or array that must be passed by reference. Oracle places the column return code in this variable after the OFETCH, OFEN, or OEXFET operation. The error codes that can be returned in RCODE are those that indicate that data in the column has been truncated or that a null occurred; for example, ORA–01405 or ORA–01406.

If ODEFIN is being used to associate an array with a select–list item, the RCODE parameter must also be an array of INTEGER*2 variables of the same dimension as the BUF array parameter.

See Also ODEFINPS, ODESCR, OEXFET, OFEN, OFETCH, OPARSE.

ODEFINPS

Purpose ODEFINPS defines an output variable for a specified select–list item in a SQL query. This call can also specify if an operation will be performed piecewise or with arrays of structures.

Syntax

```
CALL ODEFINPS(CURSOR, OPCODE, POS, BUFCTX, BUFL, FTYPE,
        [SCALE], [INDP], [FMT], [FMTL], [FMTT], [RLENP],
        [RCODEP], BUFSKIP, INDSKIP, LENSKIP, RCSKIP)
```

Comments ODEFINPS is used to define an output variable for a specified select–list item in a SQL query. Additionally, it can indicate that an application will be fetching data incrementally at runtime. This piecewise fetch is designated in the OPCODE parameter. ODEFINPS is also used when an application will be fetching data into an array of structures.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

With the introduction of ODEFINPS there are now two fully–supported calls for binding input parameters, the other being the older ODEFIN. Application developers should consider the following points when determining which define call to use:

- ODEFINPS is supported only when a program is linked in deferred mode. If it is necessary to link in non–deferred mode, ODEFIN must be used. In this case, the ability to handle piecewise operations and arrays of structures is not supported.
- ODEFINPS is more complex than the older bind call. Users who are not performing piecewise operations and are not using arrays of structures may choose to use ODEFIN.

Unlike older OCI calls, ODEFINPS does not accept –1 for any optional or unused parameters. When it is necessary to pass a value to these parameters NULL or 0 should be used instead.

See the sections “Piecewise Insert, Update and Fetch,” and “Arrays of Structures” in Chapter 2 for more information about piecewise operations, arrays of structures, skip parameters and the ODEFINPS call.

See the description of *odefinps()* on page 4 – 40 for a sample C language program demonstrating the use of ODEFINPS.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT
OPCODE	INTEGER*1	IN
POS	INTEGER*4	IN
BUFCTX	INTEGER*1	IN
BUFL	INTEGER*4	IN
FTYPE	INTEGER*4	IN
SCALE	INTEGER*4	IN
INDP	INTEGER*2	IN
FMT	CHARACTER*6	IN
FMTL	INTEGER*4	IN
FMTT	INTEGER*4	IN
RLENP	INTEGER*2	OUT
RCODEP	INTEGER*2	IN
BUFSKIP	INTEGER*4	IN
INDSKIP	INTEGER*4	IN
LENSKIP	INTEGER*4	IN
RCSKIP	INTEGER*4	IN

Note: Since the ODEFINPS call can be used in a variety of different circumstances, some items in the following list of parameter descriptions include different explanations for how the parameter is used for piecewise operations, arrays of structures and standard scalar or array binds.

Standard scalar and array defines are those defines which were previously possible using ODEFIN.

CURSOR
The CDA associated with the SELECT statement being processed.

OPCODE
Piecewise define: pass as 0.

Arrays of structures or standard define: pass as 1.

POS
An index for the select–list column which needs to be defined. Position indices start from 1 for the first, or left–most, item of the query. The ODEFINPS function uses the position index to associate output

variables with a given select–list item. If you specify a position index greater than the number of items in the select–list, or less than 1, the behavior of ODEFINPS is undefined.

If you do not know the number of items in the select list, use the ODESCR routine to determine it. See the second sample program in Appendix C for an example that does this.

BUFCTX
Piecewise define: A pointer to a context block entirely private to the application. This should be used by the application to store any information about the column being defined. One possible use would be to store a pointer to a file which will be referenced later. Each output variable can then have its own separate file pointer. The pointer can be retrieved by the application during a call to OGETPI.

Array of structures or standard define: This specifies a pointer to the program variable or the beginning of an array of program variables or structures into which the column being defined will be placed when the fetch is performed. This parameter is equivalent to the BUF parameter of the ODEFIN call.

BUFL
Piecewise define: The maximum possible size of the column being defined.

Array of structures or standard define: The length (in bytes) of the variable pointed to by BUFCTX into which the column being defined will be placed when a fetch is performed. For an array define, this should be the length of the first scalar element of the array of variables or structures pointed to by BUFCTX.

FTYPE
The external datatype to which the select–list item is to be converted before it is moved to the output variable. A list of the external datatypes and datatype codes can be found in the "External Datatypes" section in Chapter 3.

For piecewise operations, the valid datatype codes are 1 (VARCHAR2), 5 (STRING), 8 (LONG) and 24 (LONG RAW).

SCALE
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

INDP
A pointer to an indicator variable or an array of indicator variables. If arrays of structures are used, this points to a possibly interleaved array of indicator variables.

FMT
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

FMTL
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

FMTT
Not normally used in FORTRAN. See the description of OBNDRV on page 5 – 21 for more information about this parameter.

RLENP
An element or array of elements which will hold the length of a column or columns after a fetch is done. If arrays of structures are used, this points to a possibly interleaved array of length elements.

RCODEP
An element or array of elements which will hold column–level error codes which are returned by a fetch. If arrays of structures are used, this points to a possibly interleaved array of return code elements.

BUFSKIP
Piecewise define or standard scalar define: pass as 0.

Array of structures or standard array define: this is the skip parameter which specifies the number of bytes to be skipped in order to get to the next program variable element in the array being defined. In general, this will be the size of one program variable for a standard array define, or the size of one structure for an array of structures.

INDSKIP
Piecewise define or standard scalar define: pass as 0.

Array of structures or standard array define: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next indicator variable in the possibly interleaved array of indicator variables pointed to by INDP. In general, this will be the size of one indicator variable for a standard array define, and the size of one indicator variable structure for arrays of structures.

LENSKIP
Piecewise define or standard define: pass as 0.

Array of structures: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next column length in the possibly interleaved array of column lengths pointed to by RLENP. In general, this will be the size of one length variable for a standard array define, and the size of one length variable structure for arrays of structures.

RCSKIP
Piecewise define or standard define: pass as 0.

Array of structures: this is the skip parameter which specifies the number of bytes which must be skipped to get to the next return code structure in the possibly interleaved array of return codes pointed to by RCODEP. In general, this will be the size of one return code variable for a standard array define, and the size of one length variable structure for arrays of structures.

See Also OBINDPS, ODEFIN, OGETPI, OSETPI.

ODESCR

Purpose ODESCR describes select–list items for dynamic SQL queries. The ODESCR routine returns internal datatype and size information for a specified select–list item.

Syntax

```
CALL ODESCR(CURSOR, POS, DBSIZE, [DBTYPE], [CBUF],
            [CBUFL], [DSIZE],
            [PREC], [SCALE], [NULLOK])
```

Comments You can call ODESCR after parsing the SQL statement to determine the number of select–list items in a query and obtain the following information about each select–list item:

- maximum length of the item's name (DBSIZE)
- internal datatype code (DBTYPE)
- item name (CBUF)
- the length in bytes of the name (CBUFL)
- maximum display size (DSIZE)
- precision of a numeric (PREC)
- scale of a numeric (SCALE)
- null status of the column (NULLOK)

This routine is used for interactive or dynamic SQL queries, that is, queries in which the number of select–list items, as well as their datatypes and sizes, might not be known until runtime.

The *return code* field of the CDA indicates success (zero) or failure (non–zero) of the ODESCR call.

The ODESCR routine uses a position index to refer to select–list items in the SQL query statement. For example, the SQL statement

```
SELECT ENAME, SAL FROM EMP WHERE SAL > :MINSAL
```

contains two select–list items: ENAME and SAL. The position index of SAL is 2, and ENAME's index is 1. The following example shows how you can call ODESCR in a loop to describe the first 12 select–list items in an arbitrary SQL statement. See the second OCI sample program in Appendix C for additional information

```
INTEGER        DBSIZE(12), CBUFL(12), DSIZE(12)
INTEGER*2      DBTYPE(12), PREC(12), SCALE(12), NOK(12)
CHARACTER*12   CBUF(12)
CHARACTER*80   SQLSTM
...
```

```
*  After logging on and opening a cursor, get and process
*  SQL statements in a loop

     DO WHILE (0 .EQ. 0)
25      PRINT '(/, ''$'', A)', 'SQL> '
        READ '(A)', SQLSTM
        IF (SQLSTM(1:4) .EQ. 'exit') GOTO 999  ! exit loop
        CALL OPARSE(CDA, SQLSTM, LEN_TRIM(SQLSTM), 1, 2)
        IF (CDA(7) .NE. 0) THEN
          CALL ERRRPT(LDA, CDA)
          GOTO 25
        END IF

     DO 50 COUNT = 1, 12
       CBUFL(COUNT) = 12   ! length of CHARACTER buffer
       CALL ODESCR(CDA, COUNT, DBSIZE(COUNT), DBTYPE(COUNT),
1       CBUF(COUNT), CBUFL(COUNT), DSIZE(COUNT),
2       PREC(COUNT), SCALE(COUNT), NOK(COUNT))
       IF (CDA(7) .EQ. 1007) THEN     ! end of select-list
        GOTO 100
       ELSE IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GOTO 25
       END IF
50   CONTINUE

*  Print out the total count and the names, columns sizes,
*  and types of each select-list item
100     COUNT = COUNT - 1
        PRINT '(1X, A, I3, A, /)', 'There were', COUNT,
     1        ' select-list items.'

        PRINT *,'NAME        ',' COL_LEN',' DB_SIZE','    TYPE'
        PRINT *, '-------------------------------------'

        DO 150 J = 1, COUNT
150       PRINT '(1X, A, 3I8)', CBUF(J), CBUFL(J),
     1           DBSIZE(J), DBTYPE(J)
      END DO   ! end main loop
```

Note: A dependency exists between the results returned by a describe operation (ODESCR) and a bind operation (OBNDRN or OBNDRV). Because a select–list item might contain bind variables, the type returned by ODESCR can vary depending on the results of bind operations.

So, if you have placeholders for bind variables in a SELECT statement and you plan to use ODESCR to obtain the size or datatype of select–list items, you should do the bind operation before the describe.

If you need to rebind any input variables after performing a describe, you must reparse the SQL statement before rebinding. Note that the rebind operation might change the results returned for a select–list item.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT
POS	INTEGER*4	IN
DBSIZE	INTEGER*4	OUT
DBTYPE	INTEGER*2	OUT
CBUF	CHARACTER*n	OUT
CBUFL	INTEGER*4	IN / OUT
DSIZE	INTEGER*4	OUT
PREC	INTEGER*2	OUT
SCALE	INTEGER*2	OUT
NULLOK	INTEGER*2	OUT

CURSOR
A CDA in the program. The ODESCR routine uses the cursor to reference a specific SQL query statement that has been passed to Oracle by a prior OPARSE call.

POS
The position index of the select–list item in the SQL query. Each item is referenced by position as if they were numbered left to right consecutively beginning with 1. If you specify a position index greater than the number of items in the select–list or less than 1, ODESCR returns a "variable not in select–list" error in the *return code* field of the CDA.

DBSIZE
An integer that receives the *maximum* size of the column as stored in the Oracle data dictionary. If the column is defined as VARCHAR2, CHAR, or NUMBER, the length returned is the maximum length specified for the column.

> **Note:** It is generally more efficient to establish a column–level RCODE with ODEFIN rather than using ODESCR after each fetch. For an example of using column–level return codes, see the example code under the OFETCH description in this chapter.

DBTYPE
A two–byte integer that receives the internal datatype code of the select–list item. See Table 3 – 1 for a list of Oracle internal datatype codes, and the possible external conversions for each of them.

CBUF
The address of a character buffer in the program that receives the name of the select–list item (name of the column or wording of the expression).

CBUFL
An integer that is set to the length of CBUF. CBUFL should be set before ODESCR is called. If CBUFL is not specified or if the value contained in CBUFL is zero, then the column name is not returned.

On return from ODESCR, CBUFL contains the length of the column or expression name that was returned in CBUF. The column name in CBUF is truncated if it is longer than the length specified in CBUFL on the call.

DSIZE
An integer that receives the maximum display size of the select–list item if the select–list item is returned as a character string. The DSIZE parameter is especially useful when SQL routines, such as SUBSTR or TO_CHAR, are used to modify the representation of a column. Values returned in DSIZE are

Oracle Column Type	***Value***
CHAR, VARCHAR2, RAW	length of the column in the table
NUMBER	22 (the internal length)
DATE	7 (the internal length)
LONG, LONG RAW	0
ROWID	(system dependent)
Functions returning dataype 1 (such as TO_CHAR())	same as the DSIZE parameter

PREC
An INTEGER*2 variable that receives the precision of select–list items.

SCALE
An INTEGER*2 variable that receives the scale of numeric select–list items.

For Version 6 of the RDBMS, ODESCR returns the correct scale and precision of fixed–point numbers and returns precision and scale of zero for floating–point numbers, as shown below:

SQL Datatype	*Precision*	*Scale*
NUMBER(P)	P	0
NUMBER(P,S)	P	S
NUMBER	0	0
FLOAT(N)	0	0

For Oracle7, the SQL types REAL, DOUBLE PRECISION, FLOAT, and FLOAT(N) return the correct precision and a scale of –127.

NULLOK
An INTEGER*2 variable that is set to 1 if null values are allowed for that column and to 0 if they are not.

See Also OBINDPS, OBNDRA, OBNDRN, OBNDRV, ODEFIN, ODEFINPS, OPARSE.

ODESSP

Purpose ODESSP is used to describe the parameters of a PL/SQL procedure or function stored in an Oracle database.

Syntax

```
CALL ODESSP(LDA, OBJNAM, ONLEN, RSV1, RSV1LN,
       RSV2, RSV2LN, OVRLD, POS, LEVEL, ARGNM,
       ARNLEN, DTYPE, DEFSUP, MODE, DTSIZ,
       PREC, SCALE, RADIX, SPARE, ARRSIZ)
```

Comments You call ODESSP to get the properties of a stored procedure (or function) and the properties of its parameters. When you call ODESSP, pass to it:

- A valid LDA for a connection that has, at the least, execute privileges on the procedure.
- The name of the procedure, optionally including the package name. The package body does not have to exist, as long as the procedure is specified in the package.
- The total length of the procedure name.

If the procedure exists and the connection specified in the LDA parameter has permission to execute the procedure, ODESSP returns information about each parameter of the procedure in a set of array parameters. It also returns information about the return type if it is a function.

Your OCI program must allocate the arrays for all parameters of ODESSP, and you must pass a parameter (ARRSIZ) that indicates the size of the arrays (or the size of the smallest array, if they are not equal). The ARRSIZ parameter returns the number of elements of each array that was returned by ODESSP.

ODESSP returns a non–zero value if an error occurred. The error number is in the *return code* field of the LDA. The following errors can be returned there:

–20000	The object named in the OBJNAM parameter is a package, not a procedure or function.
–20001	The procedure or function named in OBJNAM does not exist in the named package.
–20002	A database link was specified in OBJNAM, either explicitly or by means of a synonym.
ORA–0xxxx	An Oracle code, usually indicating a syntax error in the procedure specification in OBJNAM.

When ODESSP returns successfully, the OUT array parameters contain the descriptive information about the procedure or function parameters, and the return type for a function. As an example, consider a package EMP_RECS in the SCOTT schema. The package contains two stored procedures and a stored function, all named GET_SAL_INFO. Here is the package specification:

```
create or replace package EMP_RECS as

    procedure get_sal_info (
        name    in   emp.ename%type,
        salary  out  emp.sal%type);
    procedure get_sal_info (
        IDnum  in   emp.empno%type,
        salary  out  emp.sal%type);
    function get_sal_info (
        name      emp.ename%type) return emp.sal%type;

end EMP_RECS;
```

A code fragment to describe these procedures and functions follows:

```
*  Declare parameters
     CHARACTER*35     OBJNAM
      INTEGER*2        OVRLD(10), POS(10), LEVEL(10), ARNLEN(10)
     INTEGER*2        DTYPE(10), PREC(10), SCALE(10)
     INTEGER*4        DTSIZE(10), SPARE(10), ARRSIZ, ONLEN, I
     CHARACTER*20     ARGNAM(10)
     LOGICAL*1        DEFSUP(10), MODE(10), RADIX(10)
...
*  Set the OBJECT NAME
     OBJNAM = 'SCOTT.EMP_RECS.GET_SAL_INFO'
     ONLEN  =  LEN_TRIM(OBJNAM)

*  Call the describe routine
     CALL ODESSP(LDA, OBJNAM, ONLEN, 0, 0, 0, 0
    +            OVRLD, POS, LEVEL, ARGNAM, ARNLEN, DTYPE,
    +            DEFSUP, MODE, DTSIZE, PREC, SCALE, RADIX,
    +            SPARE, ARRSIZ)

*  Print out some of the values
     WRITE (*, '(1X, A)')
    +    'Overload  Level  Pos  Procname           Datatype'
     DO 100 I = 1, ARRSIZ
         WRITE (*, 9000) OVRLD(I), LEVEL(I), POS(I), ARGNAM(I),
    +                    DTYPE(I)
9000   FORMAT (1X, I10, I8, I5, A20, I5)
 100   CONTINUE
...
```

When this call to ODESSP completes, the return parameter arrays are filled in as shown in Table 6 – 1. 6 is returned in the ARRSIZ parameter, as there were a total of 5 parameters and one function return type described.

	ARRAY ELEMENT					
PARAMETER	0	1	2	3	4	5
OVRLD	1	1	2	2	3	3
POS	1	2	1	2	0	1
LEVEL	1	1	1	1	1	1
ARGNM	name	salary	ID_num	salary	NULL	name
ARNLEN	4	6	6	6	0	4
DTYPE	1	2	2	2	2	1
DEFSUP	0	0	0	0	0	0
MODE	0	1	0	1	1	0
DTSIZE	10	22	22	22	22	10
PREC		7	4	7	7	
SCALE		2	0	2	2	
RADIX		10	10	10	10	
SPARE[1]	n/a	n/a	n/a	n/a	n/a	n/a
Note 1: Reserved by Oracle for future use.						

Table 6 – 1 ODESSP Return Values

Parameters

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN / OUT
OBJNAM	CHARACTER*N	IN
ONLEN	INTEGER*4	IN
RSV1	(Address)	IN
RSV1LN	INTEGER*4	IN
RSV2	(Address)	IN
RSV2LN	INTEGER*4	IN
OVRLD	INTEGER*2	OUT
POS	INTEGER*2	OUT
LEVEL	INTEGER*2	OUT
ARGNM	CHARACTER*N(30)	OUT

Parameter Name	***Type***	***Mode***
ARNLEN	INTEGER*2	OUT
DTYPE	INTEGER*2	OUT
DEFSUP	LOGICAL*1	OUT
MODE	LOGICAL*1	OUT
DTSIZ	INTEGER*4	OUT
PREC	INTEGER*2	OUT
SCALE	INTEGER*2	OUT
RADIX	LOGICAL*1	OUT
SPARE	INTEGER*4	OUT
ARRSIZ	INTEGER*4	IN / OUT

LDA
The LDA used in the OLOG call.

OBJNAM
The name of the procedure or function, including optional schema and package name. Quoted names are accepted. Synonyms are also accepted and are translated. Multi–byte characters can be used. The string can be null terminated. If it is not, the actual length must be passed in the ONLEN parameter.

ONLEN
The length in bytes of the OBJNAM parameter. If OBJNAM is a null–terminated string, pass ONLEN as –1; otherwise, pass the exact length.

RSV1
Reserved by Oracle for future use.

RSV1LN
Reserved by Oracle for future use.

RSV2
Reserved by Oracle for future use.

RSV2LN
Reserved by Oracle for future use.

OVRLD
An array indicating whether the procedure is overloaded. If the procedure (or function) is not overloaded, zero is returned. Overloaded procedures return *1...n* for *n* overloadings of the name.

POS
An array returning the parameter positions in the parameter list of the procedure. The first, or left–most, parameter in the list is position 1. When *pos* returns a zero, this indicates that a function return type is being described.

LEVEL
For scalar parameters, LEVEL returns zero. For a record parameter, zero is returned for the record itself, then for each parameter in the record the parameter's level in the record is indicated, starting from one, in successive elements of the returned value of LEVEL.

For array parameters, zero is returned for the array itself. The next element in the return array is at level one, and describes the element type of the array.

For example, a procedure that contains three scalar parameters, an array of ten elements, and one record containing three scalar parameters at the same level, you need to pass ODESSP arrays with a minimum dimension of nine: three elements for the scalars, two for the array, and four for the record parameter.

ARGNM
An array of strings that returns the name of each parameter in the procedure or function.

ARNLEN
An array returning the length in bytes of each corresponding parameter name in ARGNM.

DTYPE
The Oracle datatype code for each parameter. See the *PL/SQL User's Guide and Reference* for a list of the PL / SQL datatypes. Numeric types, such as FLOAT, INTEGER, and REAL return a code of 2. VARCHAR2 returns 1. CHAR returns 96. Other datatype codes are shown in Table 3 – 5.

> **Note:** A DTYPE value of zero indicates that the procedure being described has no parameters.

DEFSUP
This parameter indicates whether the corresponding parameter has a default value. Zero returned indicates no default; one indicates that a default value was supplied in the procedure or function specification.

MODE
This parameter indicates the mode of the corresponding parameter. Zero indicates an IN parameter, one an OUT parameter, and two an IN / OUT parameter.

DTSIZ
The size of the datatype in bytes. Character datatypes return the size of the parameter. For example, the EMP table contains a column ENAME. If a parameter in a procedure being described is of the type EMP.ENAME%TYPE, the value 10 is returned for this parameter, since that is the length of the ENAME column.

For number types, 22 is returned. See the description of the DBSIZE parameter under ODESCR in this chapter for more information.

PREC
This parameter indicates the precision of the corresponding parameter if the parameter is numeric.

SCALE
This parameter indicates the scale of the corresponding parameter if the parameter is numeric.

RADIX
This parameter indicates the radix of the corresponding parameter if it is numeric.

SPARE
Reserved by Oracle for future use.

ARRSIZ
When you call ODESSP, pass the length of the arrays of the OUT parameters. If the arrays are not of equal length, you must pass the length of the shortest array.

When ODESSP returns, ARRSIZ returns the number of array elements filled in.

See Also ODESCR.

OERHMS

Purpose OERHMS returns the text of an Oracle error message, given the error code RCODE.

Syntax `CALL OERHMS(LDA, RCODE, BUF, BUFSIZ)`

Comments When you call OERHMS, pass the address of the LDA for the active connection as the first parameter. This is required to retrieve error messages that are correct for the database version being used on the connection.

When using OERHMS to return error messages from PL/SQL blocks (where the error code is between 6550 and 6599), be sure to allocate a large BUF, since several messages can be returned. 1000 bytes should be sufficient to handle most cases.

For more information about the causes of Oracle errors and possible solutions, see the *Oracle7 Server Messages* manual.

The following example shows how to obtain an error message from a specific Oracle instance:

```
      INTEGER*2          LDA(32,2)
      CHARACTER*512   MSGBUF
      ...
      CALL OERHMS(LDA(1,2), CDA(7,2), MSGBUF, 512)
*  Or, on a VAX.
      CALL OERHMS(LDA(1,2), CDA(7,2), %REF(MSGBUF), 512)
```

Parameters

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN/OUT
RCODE	INTEGER*2	IN
BUF	CHARACTER*n	OUT
BUFSIZ	INTEGER*4	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

RCODE
The return code containing an Oracle error number.

BUF
A buffer that receives the error message text. The maximum size of the buffer is essentially unlimited for Oracle7.

BUFSIZ
The size of the buffer in bytes.

See Also OERMSG, OLOG.

OEXEC

Purpose OEXEC executes the SQL statement associated with a cursor.

Syntax

```
CALL OEXEC(CURSOR)
```

Comments Before calling OEXEC, you must call OPARSE to parse the SQL statement; this call must complete successfully. If the SQL statement contains placeholders for bind variables, you must call OBINDPS, OBNDRA, OBNDRN or OBNDRV to bind each placeholder to the address of a program variable before calling OEXEC.

For queries, after OEXEC is called, your program must explicitly request each row of the result using OFEN or OFETCH.

For SQL UPDATE, DELETE, and INSERT statements, OEXEC executes the SQL statement and sets the *return code* field and the *rows processed count* field in the CDA. Note that an UPDATE that does not affect any rows (no rows match the WHERE clause) returns success in the *return code* field and zero in the *rows processed count* field.

Data Definition Language statements are executed on the parse if you have linked in non–deferred mode or if you have liked with the deferred option and the DEFFLG parameter of OPARSE is zero. If you have linked in deferred mode and the DEFFLG parameter is non–zero, you must call OEXN or OEXEC to execute the statement.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when OPARSE is called in non–deferred mode are not detected until the first non–deferred call is made (usually an execute or describe call).

Refer to the description of the OFETCH routine in this chapter for an example showing how OEXEC is used.

Parameter

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT

CURSOR
The CDA specified in the associated OPARSE call.

See Also OBINDPS, OBNDRA, OBNDRN, OEXFET, OEXN, OPARSE.

OEXFET

Purpose OEXFET executes the SQL statement associated with a cursor, then fetches one or more rows. A cancel (effectively, OCAN) of the cursor can also be performed by OEXFET.

Syntax

```
CALL OEXFET(CURSOR, NROWS, CANCEL, EXACT)
```

Comments Before calling OEXFET, the OCI program must first call OPARSE to parse the SQL statement, call OBINDPS, OBNDRA, OBNDRN or OBNDRV (if necessary) to bind input variables, then call ODEFIN or ODEFINPS to define output variables.

If the OCI program was linked using the deferred mode link option, the bind and define steps are deferred until OEXFET is called. If OPARSE was called with the deferred parse flag (DEFFLG) parameter non–zero, the parse step is also delayed until OEXFET is called. This means that your program can complete the processing of a SQL statement using a minimum of message round–trips between the client running the OCI program and the database server.

If you call OEXFET for a DML statement that is not a query, Oracle issues the error

```
ORA–01002:  fetch out of sequence
```

and the execute operation fails.

> **Note:** Using the deferred parse, bind, and define capabilities to process a SQL statement requires more memory on the client system than the non–deferred sequence. So, you gain execution speed at the cost of some additional space.

When running against an Oracle7 database where the SQL statement was parsed using OPARSE with the LNGFLG Parameter set to 1 or 2, a character string that is too large for its associated buffer is truncated, the column return code (RCODE) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the OEXFET call does not return an error indication. If a null is encountered for a select–list item, the associated column return code (RCODE) for that column is set to the error

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

and the indicator parameter is set to –1. The OEXFET call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, OEXFET does return an ORA–01405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

OEXFET both executes the statement and fetches the row or rows that satisfy the query. If you need to fetch additional rows after OEXFET completes, use the OFEN function.

The following example shows how you can use deferred parse, bind, and define operations together with OEXFET to process a SQL statement:

```
      INTEGER*2       CURSOR(32), INDPB, SALI(1000), NAMEI(1000)
      CHARACTER*80    SQLSTM, FMT
      REAL*4          SALS(1000)
      CHARACTER*20    NAMES(1000)
      INTEGER         DEPNUM, SQLSTL
      SQLSTM = ’SELECT ENAME, SAL FROM EMP WHERE DEPTNO = :1’
*   After logging on and opening the cursor, do
*   a deferred parse:
      CALL OPARSE(CURSOR, SQLSTM, LEN_TRIM(SQLSTM), 1, 2)
*   Bind the input variable
      CALL OBNDRV(CURSOR, 1, DEPNUM, 4, 3, –1, INDPB,FMT,–1,–1)
      PRINT ’(’’$’’, a)’, ’Enter department number: ’
      READ ’(I6)’, DEPNUM
*   Define the salary and ename arrays
      CALL ODEFIN(CURSOR, 2, SALS, 4, 4, –1 SALI)
      CALL ODEFIN(CURSOR, 1, NAMES, 20, 1, NAMEI)
*   Call OEXFET to parse, bind, define, execute, and fetch
      CALL OEXFET(CURSOR, 1000, 0, 0)
```

The number of rows that were fetched is returned in the *rows processed count* field of the CDA.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT
NROWS	INTEGER*4	IN
CANCEL	INTEGER*4	IN
EXACT	INTEGER*4	IN

CURSOR
The CDA specified in the associated OPARSE call.

NROWS
The number of rows to fetch. If NROWS is greater than 1, then you must define arrays to receive the select–list values, as well as any indicator variables. See the description of ODEFIN for more information.

If NROWS is greater than the number of rows that satisfy the query, the *rows processed count* field in the CDA is set to the number of rows returned, and Oracle returns the error

```
ORA–01403:  no data found
```

CANCEL
If this parameter is non–zero when OEXFET is called, the cursor is cancelled after the fetch completes. This has exactly the effect of issuing an OCAN call, but does not require the additional call overhead.

EXACT
If this parameter is non–zero when OEXFET is called, OEXFET returns an error if the number of rows that satisfy the query is not exactly the same as the number specified in the NROWS parameter. If the number of rows returned by the query is less than the number specified in the NROWS parameter, Oracle returns the error

```
ORA–01403:  no data found
```

If the number of rows returned by the query is greater than the number specified in the NROWS parameter, Oracle returns the error

```
ORA–01422:  Exact fetch returns more than requested
            number of rows
```

Note: If EXACT is non–zero, a cancel of the cursor is always performed, regardless of the setting of the CANCEL parameter.

See Also OBINDPS, OBNDRA, OBNDRN, OBNDRV, ODEFIN, ODEFINPS, OFEN, OPARSE.

OEXN

Purpose OEXN executes a SQL statement. Array variables can be used as input data.

Syntax `CALL OEXN(CURSOR, ITERS, ROWOFF)`

Comments OEXN is similar to OEXEC, but it allows you to take advantage of the Oracle array interface. OEXN allows operations using an array of bind variables. OEXN is generally much faster than successive calls to OEXEC, especially in a networked client–server environment.

Arrays are bound to placeholders in the SQL statement using OBINDPS, OBNDRA, OBNDRV or OBNDRN. The address of the first element of the array is passed to the bind routine.

The example below declares three arrays, one of ten integers, one of ten indicator parameters, and one of ten 20–character arrays, and defines a SQL statement that inserts multiple rows into the database. After binding the arrays, the program must place data for the first INSERT in ENAMES(1) and EMPNOS(1), for the second INSERT in ENAMES(2) and EMPNOS(2), and so forth. (This step is not shown in the example.) Then OEXN is called to insert the data in the arrays.

```
     INTEGER*2     CURSOR(32), INDPS(10)
     INTEGER       EMPNOS(10), FMTL, FMTT
     CHARACTER*20  ENAMES(10)
     CHARACTER*4   PH1, PH2, FMT
     CHARACTER*100 SQLSTM
     SQLSTM = ’INSERT INTO emp (ename, empno) VALUES (:N, :E)’
     PH1 = ’:N’
     PH2 = ’:E’
     FMTL = –1
*  Parse the statement
     CALL OPARSE(CURSOR, SQLSTM, 46, 1, 1)
*  Bind the arrays to the placeholders. Bind variables will be
*  non–NULL, so pass 0 in the indicator parameter array
     DO 10 I = 1, 10
10       INDPS(I) = 0
     CALL OBNDRV(CURSOR, PH1, 2, ENAMES(1), 20, 1,
    1     SCALE, INDPS(1), FMT, FMTL, FMTT)
     CALL OBNDRV(CURSOR, PH2, 2, EMPNOS(1), 4, 3,
    1     SCALE, INDPS(1), FMT, FMTL, FMTT)
*  After obtaining data in the EMPNOS and ENAMES arrays,
*  execute the statement, inserting the values in the arrays
     CALL OEXN(CURSOR, 10, 0)
```

The completion status of OEXN is indicated in the *return code* field of the CDA. The *rows processed count* in the CDA indicates the number of rows successfully processed.

The *rows processed count* also returns the number of rows successfully processed before an error. If the SQL statement is processing only one row per array element, and if the *rows processed count* is not equal to ITERS, the operation failed on array element *rows processed count* + 1.

You can continue to process the rest of the array even after a failure on one of the array elements as long as a rollback did not occur (obtained from the *flags1* field in the CDA). You do this by using the zero–based ROWOFF parameter to start operations at array elements other than the first. In the above example, if the *rows processed count* was 5 at completion of OEXN, then row 6 was rejected. In this event, to continue the operation at row 7, call OEXN again as follows:

```
CALL OEXN(CURSOR, 10, 6)
```

Note: The maximum number of elements in an array is 32767.

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	INTEGER*2(32)	IN / OUT
ITERS	INTEGER*4	IN
ROWOFF	INTEGER*4	IN

CURSOR
The CDA specified in the OPARSE call.

ITERS
The size of the array of bind variables to be used. The size cannot be greater than 32767 items. If the size is 1, OEXN acts effectively just like OEXEC.

ROWOFF
The zero–based offset within the bind variable array at which to begin operations. OEXN processes (ITERS – ROWOFF) array elements if no error occurs.

See Also OEXEC, OEXFET.

OFEN

Purpose OFEN fetches multiple rows into arrays of variables, taking advantage of the Oracle array interface.

Syntax `CALL OFEN(CURSOR, NROWS)`

Comments OFEN is similar to OFETCH; however, OFEN can fetch multiple rows into an array of variables with a single call. A pointer to the array is bound to a select–list item in the SQL query statement using ODEFIN.

When running against an Oracle7 database where the SQL statement was parsed using OPARSE with the LNGFLG parameter set to 1 or 2, a character string that is too large for its associated buffer is truncated, the column return code (RCODE) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the OFEN call does not return an error indication. If a null is encountered for a select–list item, the associated column return code (RCODE) for that column is set to the error

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

and the indicator parameter is set to –1. The OFEN call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, OFEN does return the ORA–01405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

Even when fetching a single row, Oracle recommends that Oracle7 OCI programs use OEXFET, with the NROWS parameter set to 1, instead of the combination of OEXEC and OFEN. Use OFEN after OEXFET to fetch additional rows when you do not know in advance the exact number of rows that a query returns.

The following example is a complete program that shows how OFEN can be used to extract data, using the array interface

```
      PROGRAM TSTOFN

      INTEGER*2 CDA(32), LDA(32)
      INTEGER*2 HDA(256)
      INTEGER*2 INDARR(10), RLEN(10), RCODE(10)
      INTEGER*4 EMPNO(10), ROWS, UIDL, PWDLEN, DBNLEN, SQLLEN
      CHARACTER*10 ENAMES(10)
      CHARACTER*20 UID, PASSWD
      CHARACTER*28 SQLSTM
      LOGICAL*1 FMT(2), DBN(2), DUMMY(2)

      UID = 'SCOTT'
      UIDLEN = 5
      PASSWD = 'TIGER'
      PWDLEN = 5

      DATA HDA/256*0/
      CALL OLOG(LDA, HDA, UID, UIDLEN, PASSWD, PWDLEN, 0,
                -1, 0)
      IF (LDA(7) .NE. 0) GOTO 911

      CALL OOPEN(CDA, LDA, DBN, DBNLEN, -1, UID, UIDL)
      IF (CDA(7) .NE. 0) GOTO 920

      SQLSTM = 'SELECT ENAME, EMPNO FROM EMP'
      SQLLEN = 28

      CALL OPARSE(CDA, SQLSTM, SQLLEN, 1, 1)
      IF (CDA(7) .NE. 0) GOTO 920

      CALL ODEFIN(CDA, 1, ENAMES, 10, 1, -1, INDARR,
     1   FMT, 0, -1, RLEN, RCODE)
      IF (CDA(7) .NE. 0) GOTO 920
      CALL ODEFIN(CDA, 2, EMPNO, 4, 3, -1, DUMMY,
     1   FMT, 0, -1, DUMMY, DUMMY)
      IF (CDA(7) .NE. 0) GOTO 920

      CALL OEXEC(CDA)
      IF (CDA(7) .NE. 0) GOTO 920

      ROWS = 0
100   CALL OFEN(CDA, 10)
         IF (CDA(7) .NE. 1403 .AND. CDA(7) .NE. 0) GOTO 920
         N = CDA(3) - ROWS
         ROWS = ROWS + N
         DO 110 I = 1, N
            IF (INDARR(I) .NE. 0) THEN
               WRITE (*, 9100)
9100           FORMAT(1X, '    (null)')
            ELSE
               WRITE (*, 9110) ENAMES(I)
9110           FORMAT (1X, A10)
            ENDIF
            WRITE (*, 9120) EMPNO(I)
9120        FORMAT (1X, I8)
110      CONTINUE
         IF (CDA(7) .EQ. 0) GOTO 100
      WRITE (*, 9130) CDA(3)
9130  FORMAT (1X, I4, ’ rows returned.’)

      GOTO 950

911   WRITE(*, 9000) UID
9000  FORMAT(’ Cannot connect to Oracle as ’, A11)
      GOTO 950

920   WRITE(*, 9010) CDA(7)
9010  FORMAT(’ Oracle error’, / , I4)

950   END
```

The completion status of OFEN is indicated in the *return code* field of the CDA. The *rows processed count* field in the CDA indicates the cumulative number of rows successfully fetched. If the rows processed count increases by NROWS, OFEN may be called again to get the next batch of rows. If the rows processed count does not increase by NROWS, then an error, such as “no data found”, occurred.

Parameters

Parameter Name	***Type***	***Mode***
CURSOR	INTEGER*2(32)	IN / OUT
NROWS	INTEGER*4	IN

CURSOR
The CDA associated with the SQL statement by the OPARSE call.

NROWS
The size of the defined variable array on which to operate. That is, the maximum number of rows to fetch at a time. The size cannot be greater than 32767 items. If the size is 1, OFEN acts effectively just like OFETCH.

See Also ODEFIN, ODEFINPS, OEXFET, OFETCH, OPARSE.

OFETCH

Purpose OFETCH returns rows of a query to the user program, one row at a time.

Syntax `CALL OFETCH(CURSOR)`

Comments Each select–list item of the query is placed into a buffer identified by a previous ODEFIN or ODEFINPS call.

When running against an Oracle7 database where the SQL statement was parsed using OPARSE with the LNGFLG parameter set to 1 or 2, a character string that is too large for its associated buffer is truncated, the column return code (RCODE) is set to the error

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

and the indicator parameter is set to the original length of the item. However, the OFETCH call does not return an error indication.

If a null is encountered for a select–list item, the associated column return code (RCODE) for that column is set to the error

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

and the indicator parameter is set to –1. The OFETCH call does not return an error.

However, if no indicator parameter is defined and the program is running against an Oracle7 database, OFETCH does return the 1405 error. It is always an error if a null is selected and no indicator parameter is defined, even if column return codes and return lengths are defined.

Even when fetching a single row, Oracle recommends that Oracle7 OCI programs use OEXFET, with the parameter set to 1, instead of the combination of OEXEC and OFETCH.

The following example shows how you can obtain data from Oracle using OFETCH on a query statement. This example continues the one shown in the description of the ODEFIN routine earlier in this section. In that example, the select–list items in the SQL statement

```
SELECT ename, empno, sal FROM emp WHERE
    sal > :MINSAL
```

were associated with output buffers, and the addresses of column return lengths and return codes were bound. The example continues:

```
     ...
*  Execute the statement
     CALL OEXEC(CURSOR)
```

```
*  Fetch each row of the query
     DO 10 I = 1, 1000000
       CALL OFETCH(CURSOR)
*  Was there a row level error or warning?
     IF (CURSOR(7) .NE. 0) THEN
       GOTO 20
*  Check column level return codes for NULL values
     IF (RCODE(1) .EQ. 1405) THEN
       WRITE (*, 100)
100    FORMAT(1X, ’NULL      ’)
     ELSE
       WRITE (*, 200) ENAME
200  FORMAT(1X, A10)
*  Process remaining two items in select–list in the same way.
10   CONTINUE
*  Check return code (CURSOR(7)), and process error if it’s
*  not 1403 ”no more data found”.
20   IF (CURSOR(7) .NE. 1403) THEN
        CALL ERRRPT(CURSOR)
* continue with program...
```

Each OFETCH call returns the next row from the set of rows that satisfies a query. After each OFETCH call, the *rows processed count* in the CDA is incremented.

There is no way to refetch rows previously fetched except by re–executing the OEXEC call and moving forward through the active set again. After the last row has been returned, the next fetch will return a “no data found” return code. When this happens, *rows processed count* contains the total number of rows recovered by the query.

Parameter

Parameter Name	***Type***	***Mode***
CURSOR	INTEGER*2(32)	IN / OUT

CURSOR
The CDA associated with the SQL statement in the OPARSE or OSQL3 call.

See Also ODEFIN, ODEFINPS, ODESCR, OEXEC, OEXFET, OFEN.

OFLNG

Purpose OFLNG fetches a portion of a LONG or LONG RAW column.

Syntax

```
CALL OFLNG(CURSOR, POS, BUF, BUFL, DTYPE,
             RETL, OFFSET)
```

Comments In Oracle7, LONG and LONG RAW columns can hold up to 2 gigabytes of data. The OFLNG routine allows you to fetch up to 64K bytes, starting at any offset, in the LONG or LONG RAW column of the current row. There can be only one LONG or LONG RAW column in a table; however, a query that includes a join operation can include in its select list several LONG–type items. The POS parameter specifies the LONG–type column that the OFLNG call uses.

Note: Although the datatype of BUFL is INTEGER*4, OFLNG can only retrieve up to 64K at a time. If an attempt is made to retrieve more than 64K, the returned data will not be complete. The use of INTEGER *4 in the interface is for future enhancements.

Before calling OFLNG to retrieve the portion of the LONG column, you must do one or more fetches to position the cursor at the desired row.

Note: With release 7.3, it may be possible to perform piecewise operations more efficiently using the new OBINDPS, ODEFINPS, OGETPI and OSETPI calls. See the section "Piecewise Insert, Update and Fetch" on page 2 – 39 for more information.

The example below shows how you could retrieve 64 Kbytes, starting at offset 70000, from a LONG RAW column. See the third sample program in Appendix C for a complete example that uses OFLNG

```
     LOGICAL*1     DAREA(65536)
      INTEGER*4     OFFSET, RETLEN, DBSIZE, SQLSTL
      INTEGER*4     LNGPOS
      INTEGER*2     CURSOR(32)
      CHARACTER*80  SQLSTM
...
      SQLSTM = 'SELECT idno FROM data_table1 WHERE idno = 100'
      SQLSTL = 44
      DBSIZE = 65536
      CALL OPARSE(CURSOR, SQLSTM, SQLSTL, 1, 1)

*  Do the defines, and then execute
*  and fetch from row desired
      CALL OEXFET(CURSOR, 1, 0, 0)
*  cursor is now at right row
*  set position in row of LONG RAW column
*  (this could also be obtained dynamically using ODESCR)
*  then fetch the portion of the column
      LNGPOS = 2
      CALL OFLNG(CURSOR, LNGPOS, DAREA, DBSIZ
```

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT
POS	INTEGER*2	IN
BUF	(Address)	OUT
BUFL	INTEGER*4	IN
DTYPE	INTEGER*4	IN
RETL	INTEGER*4	OUT
OFFSET	INTEGER*4	IN

CURSOR
The CDA specified in the associated OPARSE or OSQL3 call.

POS
The index position of the LONG–type column in the row. The first position is position one. If the column at the index position is not a LONG–type, a “column does not have LONG datatype” error is returned.

BUF
The buffer that receives the portion of the LONG–type column data. This parameter must be passed by reference.

BUFL
The length of BUF in bytes.

DTYPE
The code corresponding to the external datatype of BUF. See the “External Datatypes” section in Chapter 3 for a list of the datatype codes.

RETL
The number of bytes returned. If more than 65535 bytes were requested and returned, the value 65535 is returned in this parameter.

OFFSET
The zero–based offset of the first byte in the LONG–type column to be fetched.

See Also ODESCR, OEXFET, OFEN, OFETCH.

OGETPI

Purpose OGETPI returns information about the next chunk of data to be processed as part of a piecewise insert, update or fetch.

Syntax
```
CALL OGETPI(CURSOR, PIECEP, CTXPP, ITERP, INDEXP)
```

Comments OGETPI is used (in conjunction with OSETPI) in an OCI application to determine whether more pieces exist to be either inserted, updated, or fetched as part of a piecewise operation.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

See the section “Piecewise Insert, Update and Fetch” in Chatper 2 for more information about piecewise operations and the OGETPI call.

For a sample C language program illustrating the use of OGETPI in an OCI program which performs an piecewise insert, see the description of the *ogetpi()* call in Chapter 4.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT
PIECEP	INTEGER*1	OUT
CTXPP	INTEGER*4	OUT
ITERP	INTEGER*4	OUT
INDEXP	INTEGER*4	OUT

CURSOR
The CDA associated with the SQL or PL / SQL statement being processed.

PIECEP
Specifies whether the next piece to be fetched or inserted is the first piece, an intermediate piece or the last piece. Possible values are one (for the first piece) and two (for a subsequent piece) for a fetch or insert, and three (for the last piece) for fetches only. These values are defined in *ocidfn.h* as OCI_FIRST_PIECE, OCI_NEXT_PIECE, and OCI_LAST_PIECE.

CTXPP
The user–defined context pointer, which is optionally passed as part of an OBINDPS or ODEFINPS call. This pointer is returned to the application during the OGETPI call. If CTXPP is passed as NULL, the parameter is ignored. The application may already know which buffer it needs to pass in OSETPI at run time.

ITERP
The current iteration. During an array insert it will tell you which row you are working with. Starts from 0.

INDEXP
Pointer to the current index of an array mapped to a PL/SQL table, if an array is bound for an insert. The value of INDEXP varies between zero and the value set in the *cursiz* parameter of the OBINDPS call.

See Also OBINDPS, ODEFINPS, OSETPI.

OLOG

Purpose OLOG establishes a connection between an OCI program and an Oracle database.

Syntax

```
CALL OLOG(LDA, HDA, UID, UIDL, [PSWD], [PSWDL],
          [CONN], [CONNL], MODE)
```

Comments An OCI program can connect to one or more Oracle instances multiple times. Communication takes place using the LDA and the HDA defined within the program. It is the OLOG function that connects the LDA to Oracle.

The HDA is a program–allocated data area associated with each OLOG logon call. Its contents are entirely private to Oracle, but the HDA must be allocated by the OCI program. Each concurrent connection requires one LDA–HDA pair.

> **Note:** The HDA must be initialized to all zeros (binary zeros, not the "0" character) before the call to OLOG, or runtime errors will occur. In FORTRAN this can be accomplished through the use of the DATA statement. See the sample code below for an example.

The HDA has a size of 256 bytes on 32–bit systems, and 512 bytes on 64–bit systems. If memory permits, it is possible to allocate a 512–byte HDA on a 32–bit system to increase portability of aplications.

Refer to the section "Host Data Area" in Chapter 2 for more information about HDAs.

After the OLOG call, the HDA and the LDA must remain at the same program address they occupied at the time OLOG was called.

When an OCI program issues an OLOG call, a subsequent OLOGOF call using the same LDA commits all outstanding transactions for that connection. If a program fails to disconnect or terminates abnormally, then all outstanding transactions are rolled back.

The LDA *return code* field indicates the result of the OLOG call. A zero return code indicates a successful connection.

The MODE parameter specifies whether the connection is in blocking or non–blocking mode. For more information on connection modes, see "Non–Blocking Mode" on page 2 – 32. For a short example program in C, see the *onbset()* description on page 4 – 89.

You should also refer to the section on SQL*Net in your Oracle system–specific documentation for any particular notes or restrictions that apply to your operating system.

The following code fragment demonstrates a typical set of declarations and initializations for a call to OLOG.

```
       INTEGER*2          LDA(32)
       INTEGER*2          HDA(256)
       CHARACTER*80       UID, PSWD
       INTEGER*4          UIDL, PSWDL, MODE
...
       UID = 'SCOTT'
       PSWD = 'TIGER'
       UIDL = LEN_TRIM(UID)
       PSWDL = LEN_TRIM(PSWD)
       MODE = 0
...
*  CONNECT TO ORACLE IN NON–BLOCKING MODE.
*  MODE = 0 INDICATES A NON–BLOCKING CONNECTION.
*  HDA MUST BE INITIALIZED TO ZEROS BEFORE CALL TO OLOG.

       DATA HDA/256*0/

       CALL OLOG(LDA, HDA, UID, UIDL, PSWD, PSWDL, 0, –1, MODE)

       IF (LDA(7).NE.0) THEN
           CALL ERRRPT(LDA(1), LDA(1))
           GOTO 999
       END IF

       WRITE (*, '(1X, A, A)') 'Connected to ORACLE as user ', UID
...
```

Parameters

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN / OUT
HDA	INTEGER*2(128)	OUT
UID	CHARACTER*n	IN
UIDL	INTEGER*4	IN
PSWD	CHARACTER*n	IN
PSWDL	INTEGER*4	IN
CONN	CHARACTER*n	IN
CONNL	INTEGER*4	IN
MODE	INTEGER*4	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

HDA
A host data area. See Chapter 2 for more information on host data areas.

UID
A string containing the user ID, an optional password, and an optional host machine identifier. If you include the password as part of the UID parameter, put it immediately after the user ID and separate it from the user ID with a '/'. Put the host system identifier after the password or user ID, separated by the '@' sign.

If you do not include the password in this parameter, it must be in the PSWD parameter. Examples of valid UID strings are

```
NAME
NAME/PASSWORD
NAME@SERVICENAME
NAME/PASSWORD@DSERVICENAME
```

The following string is not a correct example of the UID parameter:

```
NAME@SERVICENAME/PASSWORD
```

UIDL
The length of the UID string.

PSWD
The string containing the password. If the password is specified as part of the string pointed to by UID, this parameter can be omitted.

PSWDL
The length of the password.

CONN
A string containing a SQL*Net V2 connect descriptor to connect to a database. If the connect descriptor is specified as part of the UID string, this parameter can be omitted.

CONNL
The length of the CONN parameter.

MODE
Specifies whether the connection is in blocking or non–blocking mode. Possible values are zero (for blocking mode) or one (for non–blocking mode).

See Also OLOGOF, ONBSET, SQLLDA.

OLOGOF

Purpose

OLOGOF disconnects an LDA from the Oracle program global area and frees all Oracle resources owned by the Oracle user process.

Syntax

```
CALL OLOGOF(LDA)
```

Comments

A COMMIT is automatically issued on a successful OLOGOF call; all currently opened cursors are closed. If a program logs off unsuccessfully or terminates abnormally, all outstanding transactions are rolled back.

If the program has multiple active logons, a separate call to OLOGOF must be performed for each active LDA. If OLOGOF fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also

OLOG.

ONBCLR

Purpose ONBCLR places a database connection in non–blocking mode.

Syntax

```
CALL ONBCLR(LDA)
```

Comments If there is a pending call on a connection and ONBCLR is called, the pending call, when resumed, will block.

Parameters

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OLOG, ONBSET, ONBTST.

ONBSET

Purpose ONBSET places a database connection in non–blocking mode for all subsequent OCI calls on this connection.

Syntax

```
CALL ONBSET(LDA)
```

Comments ONBSET will succeed if the library is linked in deferred mode and if the network driver supports non–blocking operations.

This call also requires SQL*Net Release 2.1 or higher. It is not compatible with a single–task driver.

Parameters

Parameter Name	***Type***	***Mode***
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OLOG, ONBCLR, ONBTST.

ONBTST

Purpose ONBTST tests whether a database connection is in non–blocking mode.

Syntax

```
CALL ONBTST(LDA)
```

Comments If the connection is in blocking mode, the user may call ONBSET to place the channel in non–blocking mode, if allowed by the network driver. Non–blocking connections require Oracle7 Server release 7.2 or higher, and SQL*Net release 2.1 or higher. Non–blocking connections are not possible with a single–task driver.

Parameters

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OLOG, ONBCLR, ONBSET.

OOPEN

Purpose OOPEN opens the specified cursor.

Syntax

```
CALL OOPEN(CURSOR, LDA, <DBN>, <DBNL>,
           <ARSIZE>, <UID>, <UIDL>)
```

Comments OOPEN associates a CDA in the program with data storage areas in the Oracle Server. Oracle uses these data storage areas to maintain state information about the processing of a SQL statement. Status concerning error and warning conditions, as well as other information, such as function codes, is returned to the CDA in your program as Oracle processes the SQL statement.

An OCI program can have many cursors active at the same time.

The OPARSE routine is used to parse a SQL statement and associate it with a cursor. In the OCI routines, SQL statements are always referenced using a cursor as the handle.

The *return code* field of the CDA indicates the result of the OOPEN. A return code value of zero indicates a successful OOPEN call.

It is possible to issue an OOPEN call on a cursor that is already open. This has no effect on the cursor, but it does affect the value in the Oracle OPEN_CURSORS counter. Repeatedly reopening an open cursor may result in an ORA–01000 error ('maximum open cursors exceeded'). Refer to the *Oracle7 Server Messages* manual for information about what to do if this happens.

See the description of the OPARSE routine in this chapter for a code example that uses OOPEN.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	OUT
LDA	INTEGER*2(32)	IN / OUT
DBN	CHARACTER*n	IN
DBNL	INTEGER*4	IN
ARSIZE	INTEGER*4	IN
UID	CHARACTER*n	IN
UIDL	INTEGER*4	IN

CURSOR
A cursor data area associated with the program.

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

DBN
This parameter is included only for Oracle Version 2 compatibility. It is not used in later versions.

DBNL
This parameter is included only for Oracle Version 2 compatibility. It is not used in later versions.

ARSIZE
The ARSIZE (areasize) parameter is no longer used with Oracle7, as the data areas used by cursors in the Oracle Server are resized automatically as required.

UID
A character string containing the user ID and password. The password must be separated from the user ID by a '/'.

If the connection to Oracle was established using the Version 2 OLOGON call, then UID and UIDL are used in the OOPEN call. If the OLON routine was used, UID and UDL are ignored in the OOPEN call.

UDL
The length of the UID parameter.

See Also OLOG, OPARSE.

OOPT

Purpose OOPT is used to set rollback options for non–fatal Oracle errors involving multi–row INSERT and UPDATE SQL statements. It is also used to set wait options in cases where requested resources are not available; for example, whether to wait for locks.

Syntax

```
CALL OOPT(CURSOR, RBOPT, WAITOPT)
```

Comments The RBOPT parameter is not supported in Oracle Server Version 6 or later.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT
RBOPT	INTEGER*4	IN
WAITOPT	INTEGER*4	IN

CURSOR
The CDA used in the OOPEN call for this cursor.

RBOPT
The action to be taken when a non–fatal Oracle error occurs. If this option is set to 0, all errors, even non–fatal errors, cause the current transaction to be rolled back. If this option is set to 2, only the failing row will be rolled back during a non–fatal row–level error. This is the default setting.

WAITOPT
Specifies whether to wait for resources or continue without them if they are currently not available. If this option is set to 0, the program waits indefinitely if resources are not available. This is the default. If this option is set to 4, the program will receive an error return code whenever a resource is requested but is not available. Use of WAITOPT set to 4 can cause many error return codes while waiting for internal resources that are locked for short durations. The only resource errors received are for resources requested by the calling process.

See Also OOPEN.

OPARSE

Purpose OPARSE parses a SQL statement or a PL/SQL block and associates it with a cursor. The parse can optionally be deferred.

Syntax

```
OPARSE(CURSOR, SQLSTM, SQLL, DEFFLG, LNGFLG)
```

Comments OPARSE passes the SQL statement to Oracle for parsing. If the DEFFLG parameter is non–zero, the parse is deferred until the statement is executed or until ODESCR is called to describe the statement. Once the parse is performed, the parsed representation of the SQL statement is stored in the Oracle shared SQL cache. Subsequent OCI calls reference the SQL statement using the cursor name.

An open cursor can be reused by subsequent OPARSE calls within a program, or the program can define multiple concurrent cursors when it is necessary to maintain multiple active SQL statements.

If OPARSE is used to parse a query, the fetch call returns a "fetched column value was truncated" if a column value was truncated and no indicator parameter was defined for that column using ODEFIN.

> **Note:** When OPARSE is called with the DEFFLG parameter set, you cannot receive most error indications until the parse is actually performed. The parse is performed at the first call to ODESCR, OEXEC, OEXN, or OEXFET. However, the SQL statement string is scanned on the client system, and some errors, such as "missing double quote in identifier", can be returned immediately.

The statement can be any valid SQL statement, or PL/SQL anonymous block. Oracle parses the statement and selects an optimal access path to perform the requested function.

Data Definition Language statements are executed on the parse if you have linked in non–deferred mode or if you have liked with the deferred option and the DEFFLG parameter of OPARSE is zero. If you have linked in deferred mode and the DEFFLG parameter is non–zero, you must call OEXN or OEXEC to execute the statement.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment. Note, however, that errors in the SQL statement that would be detected when OPARSE is called in non–deferred mode are not detected until the first non–deferred call is made (usually an execute or describe call).

The following example opens a cursor and parses a SQL statement. The OPARSE call associates the SQL statement with the cursor.

```
     INTEGER*2      LDA(32), CURSOR(32)
     CHARACTER*120  SQLSTM
     ..
     SQLSTM = 'DELETE FROM emp WHERE empno = :EMPLOYEE_NUMBER'

     ...
*  After connecting to Oracle..
     CALL OOPEN(CURSOR, LDA, 0, 0, 0, 0, 0)
     CALL OPARSE(CURSOR, SQLSTM, LEN_TRIM(SQLSTM), 1, 1)
```

SQL syntax error codes are returned in the CDA's *return code* field and *parse error offset* field. Parse error offset indicates the location of the error in the SQL statement text. See "Cursor Data Area (CDA)" on page 2 – 4 for a list of the information fields available in the CDA after an OPARSE call.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN / OUT
SQLSTM	CHARACTER*n	IN
SQLL	INTEGER*4	IN
DEFFLG	INTEGER*4	IN
LNGFLG	INTEGER*4	IN

CURSOR
The CDA specified in the OOPEN call.

SQLSTM
A string containing the SQL statement.

SQLL
The length of the SQL statement string in bytes.

DEFFLG
If non–zero, the parse of the SQL statement is deferred until an ODESCR, OEXEC, OEXN, or OEXFET call is made. Note that bind and define operations are also deferred until the execute step if the program was linked using the deferred mode option. See "Deferred Statement Execution" on page 2 – 14 for more information about the deferred mode link option.

Oracle recommends that you use the deferred parse capability whenever possible. This results in increased performance, especially in a networked environment.

LNGFLG
The LNGFLG parameter determines the way that Oracle handles the SQL statement or PL/SQL anonymous block. To ensure strict ANSI conformance, Oracle7 defines several datatypes and operations in a slightly different way than Version 6. The table below shows the differences between Version 6 and Oracle7:

Behavior	***V6***	***V7***
CHAR columns are fixed length (including created by a CREATE TABLE statement).	NO	YES
An error is issued if an attempt is made to fetch a null value into an output variable that has no associated indicator variable.	NO	YES
An error is issued if a fetched value is truncated and there is no indicator variable.	YES	NO
Describe (ODESCR) returns internal datatype 1 for CHAR columns.	YES	NO
Describe (ODESCR) returns internal datatype 96 for CHAR columns.	n/a	YES

The LNGFLG parameter has three possible settings:

0 Specifies Version 6 behavior (the database you are connected to can be either Version 6 or Oracle7).

1 Specifies the normal behavior for the database version the program is connected to (either Version 6 or Oracle7).

2 Specifies Oracle7 behavior. If you use this value for the parameter and you are not connected to an Oracle7 database, Oracle issues the error
```
ORA–01011:  Cannot use this language type when talking
            to a V6 database
```

See Also ODESCR, OEXEC, OEXFET, OEXN, OOPEN.

OROL

Purpose OROL rolls back the current transaction.

Syntax `CALL OROL(LDA)`

Comments The current transaction is defined as the set of SQL statements executed since the OLOG call or the last OCOM or OROL call. If OROL fails, the reason is indicated in the *return code* field of the LDA.

Parameter

Parameter Name	***Type***	***Mode***
LDA	INTEGER*2(32)	IN / OUT

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

See Also OCOM, OLOG.

OSETPI

Purpose OSETPI sets information about the next chunk of data to be processed as part of a piecewise insert, update or fetch.

Syntax

```
CALL OSETPI(CURSOR, PIECE, BUFP, LENP)
```

Comments An OCI application uses OSETPI to set the information about the next piecewise insert, update, or fetch. The BUFP parameter is either the buffer containing the next piece to be inserted, or to the buffer where the next fetched piece will be stored.

> **Note:** This function is only compatible with Oracle server release 7.3 or later. If a release 7.3 application attempts to use this function against a release 7.2 or earlier server, an error message is likely to be generated. At that point you must restart execution.

See the section "Piecewise Insert, Update and Fetch" in Chatper 2 for more information about piecewise operations and the OSETPI call.

For a sample C language program illustrating the use of OSETPI to perform a piecewise fetch, see the description of the *osetpi()* routine in Chapter 4.

Parameters

Parameter Name	*Type*	*Mode*
CURSOR	INTEGER*2(32)	IN
PIECE	INTEGER*1	IN
BUFP	INTEGER*4	IN
LENP	INTEGER*4	IN / OUT

CURSOR
The cursor data area associated with the SQL or PL / SQL statement.

PIECE
Specifies the piece being provided or fetched. Possible values are one (for the first piece), two (for a subsequent piece) or three (for the last piece). These values are defined in *ocidfn.h* as OCI_FIRST_PIECE, OCI_NEXT_PIECE, and OCI_LAST_PIECE. Relevant when the buffer is being set after error ORA–03129 was returned by a call to OEXEC.

BUFP
A data buffer. If OSETPI is called as part of a piecewise insert, this pointer must point to the next piece of the data to be transmitted. If OSETPI is called as part of a piecewise fetch, this is a buffer to hold the next piece to be retrieved.

LENP
The length in bytes of the current piece. If a piece is provided, the value is unchanged on return. If the buffer is filled up and part of the data is truncated, LENP is modified to reflect the length of the piece in the buffer.

See Also OBINDPS, ODEFINPS, OGETPI.

SQLLD2

Purpose The SQLLD2 routine is provided for OCI programs that operate as application servers in an X/Open distributed transaction processing environment. SQLLD2 fills in fields in an LDA, according to the connection information passed to it.

Syntax `CALL SQLLD2(LDA, CNAME, CNLEN)`

Comments OCI programs that operate in conjunction with a transaction manager do not manage their own connections. However, all OCI programs require a valid LDA. You use SQLLD2 to obtain the LDA.

SQLLD2 fills in the LDA using the connection name passed in the CNAME parameter. The CNAME parameter must match the DB_NAME alias parameter of the XA info string of the XA_OPEN call. If the CNLEN parameter is set to zero, an LDA for the default connection is returned. Your program must declare the LDA, then pass it as a parameter.

If you call SQLLD2 and there is no valid connection, the error

```
ORA–01012:  not logged on
```

is returned in the *return code* field of the LDA parameter.

SQLLD2 must be invoked whenever there is an active XA transaction. This means that it must be invoked after XA_OPEN and XA_START, and before XA_END. Otherwise and ORA–01012 error will result.

SQLLD2 is part of SQLLIB, the Oracle Precompiler library. SQLLIB must be linked into all programs that call SQLLD2. See your Oracle system–specific documentation for information about linking SQLLIB.

This example demonstrates how you can use SQLLD2 to obtain a valid LDA for a specific connection:

```
      INTEGER*2        LDA1(32), LDA2(32)
      CHARACTER*20  DBNAM1, DBNAM2
      INTEGER*4     CNLEN1, CNLEN2

*  Set up handles
      DBNAM1 = 'D:NEWYORK'
      DBNAM2 = 'D:LOSANGLES'

      DB_STRING1 = 'D:NEWYORK'
      DB_STRING2 = 'D:LOSANGELES'
      CNLEN1     = 9
      CNLEN2     = 12
```

```
*  Get the LDAs
     CALL SQLLD2(LDA1, DBNAM1, CNLEN1)
     CALL SQLLD2(LDA2, DBNAM2, CNLEN2)
```

Parameters

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	OUT
CNAME	CHARACTER*n	IN
CNLEN	INTEGER*4	IN

LDA
A local data area. You must declare this data area before calling SQLLD2**.**

CNAME
The name of the database connection. If the name consists of all blanks, SQLLD2 returns the LDA for the default connection.

CNLEN
The length of the CNAME parameter. If CNLEN is passed as zero, SQLLD2 returns the LDA for the default connection, regardless of the contents of CNAME.

SQLLDA

Purpose SQLLDA is part of the SQLLIB library that is used with the Oracle Precompilers. This routine is provided for programs that mix both precompiler code and OCI calls. The address of an LDA is passed to SQLLDA; the precompiler fills in the required fields in the LDA.

Syntax

```
CALL SQLLDA(LDA)
```

Comments If your program contains both precompiler statements and calls to OCI routines, you cannot use OLOG to log on to Oracle. You must use the embedded SQL command

```
EXEC SQL CONNECT ...
```

to log on. However, many OCI routines require a valid LDA. The SQLLDA routine obtains the LDA. SQLLDA is part of SQLLIB, the precompiler library.

SQLLDA fills in the LDA using the connect information from the most recently executed SQL statement. So, you should call SQLLDA immediately after doing the connect with the EXEC SQL CONNECT ... statement.

The example below demonstrates how you can do multiple remote logons in a mixed Precompiler–OCI program. Refer to Chapter 3 in the *Programmer's Guide to the Oracle Precompilers* for additional information about multiple remote logons.

```
EXEC SQL BEGIN DECLARE SECTION;
     CHARACTER*20 USRNAM, PASSWD, DBSTR1, DBSTR2
EXEC SQL END DECLARE SECTION;

...

*  Host program declarations
     INTEGER*2        LDA1(32), LDA2(32)

*  Set up arrays
     USRNAM = 'SCOTT'
     PASSWD = 'TIGER'
     DBSTR1 = 'D:NEWYORK'
     DBSTR2 = 'D:LOSANGELES'
*  Do the connections
     EXEC SQL DECLARE dbn1 DATABASE;
     EXEC SQL CONNECT :USRNAM IDENTIFIED BY :PASSWD
         AT dbn1 USING :DBSTR1;
```

```
*  Get the first LDA for OCI use
     CALL SQLLDA(LDA1)
     EXEC SQL DECLARE dbn2 DATABASE;
     EXEC SQL CONNECT :USRNAM IDENTIFIED BY :PASSWD
       AT dbn2 USING :DBSTR2;
*  Get the second LDA for OCI use
     CALL SQLLDA(LDA2)
```

Parameter

Parameter Name	***Type***	***Mode***
LDA	INTEGER*2(32)	OUT

LDA
A local data area. You must declare this area before calling SQLLDA.

APPENDIX

A Sample Programs in C

This appendix contains sample OCI programs written in C and C++. Also included are the header (.h) files used by these sample programs.

These header files and sample programs are available online. The online location of these programs is system dependent. Refer to your Oracle system–specific documentation for additional information.

This appendix contains listings for the following files:

- header files
  - *oratypes.h*
  - *ocidfn.h*
  - *ocidem.h*
  - *ociapr.h*
  - *ocikpr.h*
  - *cdemo6.h*
- cdemo1.c
- cdemo2.c
- cdemo3.c
- cdemo4.c
- cdemo5.c
- cdemo6.cc

Header Files

All sample programs and sample header files depend on the types that are defined in the *oratypes.h* file. If you attempt to compile and run the sample programs, make sure that the directories containing *oratypes.h* and the OCI header files are on the include path for your compiler.

oratypes.h

The following listing shows an example *oratypes.h* file. This header is system dependent. The example listed is valid only for SunOS C compilers.

```
/*
  oratypes.h – System–dependent external definitions of Oracle
  types (SunOS version)

  This header file defines C types used by Oracle demonstration
  code, and code samples in documentation (such as the
  _Programmer’s Guide to the Oracle Call Interfaces_)

. Porters must modify oratypes.h to reflect the reality of their
  ports
*/

#ifndef ORASTDDEF
# include <stddef.h>
# define ORASTDDEF
#endif

#ifndef ORALIMITS
# include <limits.h>
# define ORALIMITS
#endif

#ifndef  SX_ORACLE
#define  SX_ORACLE
#define  SX
#define  ORATYPES
#ifndef SS_COMPFLAGS
# include <ss_compflags.h>
#endif

/* define TRUE/FALSE;  defined in some stdio’s */
#ifndef TRUE
# define TRUE  1
# define FALSE 0
#endif
```

```
#ifdef lint
# ifndef mips
#  define signed
# endif /* mips */
#endif /* lint */

#ifdef ENCORE_88K
# ifndef signed
#  define signed
# endif /* signed */
#endif /* ENCORE_88K */

#if defined(SYSV_386) || defined(SUN_OS)
# ifdef signed
#  undef signed
# endif /* signed */
# define signed
#endif /* SYSV_386 */

#ifndef lint
typedef          int eword;    /* use where sign not important */
typedef unsigned int uword;    /* use where unsigned important */
typedef   signed int sword;    /* use where   signed important */
#else
#define eword int
#define uword unsigned int
#define sword signed int
#endif /*LINT */

#define  EWORDMAXVAL  ((eword) INT_MAX)
#define  EWORDMINVAL  ((eword)       0)
#define  UWORDMAXVAL  ((uword)UINT_MAX)
#define  UWORDMINVAL  ((uword)       0)
#define  SWORDMAXVAL  ((sword) INT_MAX)
#define  SWORDMINVAL  ((sword) INT_MIN)
#define  MINEWORDMAXVAL  ((eword)  32767)
#define  MAXEWORDMINVAL  ((eword)      0)
#define  MINUWORDMAXVAL  ((uword)  65535)
#define  MAXUWORDMINVAL  ((uword)      0)
#define  MINSWORDMAXVAL  ((sword)  32767)
#define  MAXSWORDMINVAL  ((sword) –32767)
```

```
#ifndef lint
# ifdef mips
typedef   signed char  eb1;
# else
typedef          char  eb1;    /* use where sign not important */
# endif /* mips */

typedef unsigned char  ub1;    /* use where unsigned important */
typedef   signed char  sb1;    /* use where   signed important */
#else
#define eb1 char
#define ub1 unsigned char
#define sb1 signed char
#endif /* LINT */

#define EB1MAXVAL ((eb1)SCHAR_MAX)
#define EB1MINVAL ((eb1)        0)
#if defined(mips)        /* mips compilers barf on x[UB1MAXVAL] */
# ifndef lint
#  define UB1MAXVAL (UCHAR_MAX)
# endif
#endif

#ifndef UB1MAXVAL
# ifdef SCO_UNIX
# define UB1MAXVAL (UCHAR_MAX)
# else
# define UB1MAXVAL ((ub1)UCHAR_MAX)
# endif /* SCO_UNIX */
#endif

#define UB1MINVAL ((ub1)        0)
#define SB1MAXVAL ((sb1)SCHAR_MAX)
#define SB1MINVAL ((sb1)SCHAR_MIN)
#define MINEB1MAXVAL ((eb1)  127)
#define MAXEB1MINVAL ((eb1)    0)
#define MINUB1MAXVAL ((ub1)  255)
#define MAXUB1MINVAL ((ub1)    0)
#define MINSB1MAXVAL ((sb1)  127)
#define MAXSB1MINVAL ((sb1) –127)

/* number of bits in a byte */
#define UB1BITS          CHAR_BIT
#define UB1MASK              0xff
```

```
/* human readable (printable) characters */
typedef  unsigned char text;

#ifndef lint
typedef          short    eb2;  /* use where sign not important */
typedef unsigned short    ub2;  /* use where unsigned important */
typedef   signed short    sb2;  /* use where   signed important */
#else
#define eb2  short
#define ub2  unsigned short
#define sb2  signed short
#endif /* LINT */

#define EB2MAXVAL ((eb2) SHRT_MAX)
#define EB2MINVAL ((eb2)        0)
#define UB2MAXVAL ((ub2)USHRT_MAX)
#define UB2MINVAL ((ub2)        0)
#define SB2MAXVAL ((sb2) SHRT_MAX)
#define SB2MINVAL ((sb2) SHRT_MIN)
#define MINEB2MAXVAL ((eb2) 32767)
#define MAXEB2MINVAL ((eb2)     0)
#define MINUB2MAXVAL ((ub2) 65535)
#define MAXUB2MINVAL ((ub2)     0)
#define MINSB2MAXVAL ((sb2) 32767)
#define MAXSB2MINVAL ((sb2)-32767)

#ifndef lint
typedef          long  eb4;     /* use where sign not important */
typedef unsigned long  ub4;     /* use where unsigned important */
typedef   signed long  sb4;     /* use where   signed important */
#else
#define eb4 long
#define ub4 unsigned long
#define sb4 signed long
#endif /* LINT */

#define EB4MAXVAL ((eb4) LONG_MAX)
#define EB4MINVAL ((eb4)        0)
#define UB4MAXVAL ((ub4)ULONG_MAX)
#define UB4MINVAL ((ub4)        0)
#define SB4MAXVAL ((sb4) LONG_MAX)
#define SB4MINVAL ((sb4) LONG_MIN)
#define MINEB4MAXVAL ((eb4) 2147483647)
#define MAXEB4MINVAL ((eb4)          0)
#define MINUB4MAXVAL ((ub4) 4294967295)
#define MAXUB4MINVAL ((ub4)          0)
#define MINSB4MAXVAL ((sb4) 2147483647)
#define MAXSB4MINVAL ((sb4)-2147483647)
```

```
#ifndef lint
typedef unsigned long  ubig_ora;             /* use where unsigned
important */
typedef   signed long  sbig_ora;             /* use where   signed
important */
#else
#define ubig_ora unsigned long
#define sbig_ora signed long
#endif /* LINT */

#define UBIG_ORAMAXVAL ((ubig_ora)ULONG_MAX)
#define UBIG_ORAMINVAL ((ubig_ora)        0)
#define SBIG_ORAMAXVAL ((sbig_ora) LONG_MAX)
#define SBIG_ORAMINVAL ((sbig_ora) LONG_MIN)
#define MINUBIG_ORAMAXVAL ((ubig_ora) 4294967295)
#define MAXUBIG_ORAMINVAL ((ubig_ora)          0)
#define MINSBIG_ORAMAXVAL ((sbig_ora) 2147483647)
#define MAXSBIG_ORAMINVAL ((sbig_ora)–2147483647)

/* Use CONST as replacement for the ANSI type qualifier ’const’.*/

#undef CONST
#ifdef _olint
# define CONST const
#else
#if defined(PMAX) && defined(__STDC__)
#   define CONST const
#else
# ifdef M88OPEN
#  define CONST const
# else /* M88OPEN */
# define CONST
# endif /* M88OPEN */
#endif /* PMAX and !ULTRIX_MLS */
#endif /* _olint */

/*
** lgenfp_t, a generic function pointer type.  (It’s too bad
** that a level of indirection is hidden, as with the dearly
** departed ptr_t, but typedef’ing a function isn’t portable.)
*/

typedef void (*lgenfp_t)(/*_ void _*/);
```

```
/* dvoid: base type for pointer to arbitrary block of memory    **
** use as ”dvoid *”; in ansi environments, dvoid should be void **
** If your compiler doesn’t support void *, dvoid * should match**
** the return type from memcpy()/memset(), i.e. probably char * **
** VMS defines (as opposed to typedefing) dvoid when linting to **
** allow dvoid* to match void* while being strict about other   **
** typedefs (i.e. short not matching ub2).                      */

#ifdef lint
# define dvoid void
#else
# ifdef UTS2
#  define dvoid char
# else
# define dvoid void
# endif /* UTS2 */
#endif /* lint */

/* type boolean, for TRUE/FALSE function local variables and
** parameters; defined as int for efficient processing and to
** match the natural type for C boolean expressions.  Do not use
** where space efficiency is important.
*/
#ifndef ORASYSTYPES
# include <sys/types.h>
# define ORASYSTYPES
#endif /* !ORAUSRINCLUDESYSTYPES */
#define boolean int

/* SIZE_TMAXVAL is the largest amount of memory that the
** current platform is capable of allocating in one single block.
** The constant MINSIZE_TMAXVAL is the largest allocation across
** all platforms.
*/
#ifdef sparc
# define SIZE_TMAXVAL SB4MAXVAL  /* This case applies for sun4 */
#else
# define SIZE_TMAXVAL UB4MAXVAL  /* This case applies for others
*/
#endif /* sparc */
#define MINSIZE_TMAXVAL (size_t)65535
#endif /* SX_ORACLE */
/*
```

ocidfn.h

```
/*
 *  ocidfn.h
 *  Common header file for OCI C sample programs.
 *  This header declares the cursor and logon data area structure.
 *  The types used are defined in <oratypes.h>.
 */
#ifndef OCIDFN
#define OCIDFN
#include <oratypes.h>

/* The cda_head struct is strictly PRIVATE.  It is used
   internally only. Do not use this struct in OCI programs. */
struct cda_head {
    sb2          v2_rc;
    ub2          ft;
    ub4          rpc;
    ub2          peo;
    ub1          fc;
    ub1          rcs1;
    ub2          rc;
    ub1          wrn;
    ub1          rcs2;
    sword        rcs3;
    struct {
        struct {
           ub4    rcs4;
           ub2    rcs5;
           ub1    rcs6;
        } rd;
        ub4    rcs7;
        ub2    rcs8;
    } rid;
    sword        ose;
    ub1           chk;
    dvoid        *rcsp;
};
# define CDA_SIZE 64
/* the real CDA, padded to CDA_SIZE bytes in size */
struct cda_def {
    sb2          v2_rc;                       /* V2 return code */
    ub2          ft;                        /* SQL function type */
    ub4          rpc;                    /* rows processed count */
    ub2          peo;                      /* parse error offset */
    ub1          fc;                        /* OCI function code */
    ub1          rcs1;                            /* filler area */
    ub2          rc;                          /* V7 return code */
    ub1          wrn;                          /* warning flags */
    ub1          rcs2;                               /* reserved */
    sword        rcs3;                                /* reserved */
    struct {                                   /* rowid structure */
        struct {
           ub4    rcs4;
           ub2    rcs5;
           ub1    rcs6;
        } rd;
        ub4    rcs7;
        ub2    rcs8;
    } rid;
    sword        ose;                    /* OSD dependent error */
    ub1          chk;
    dvoid        *rcsp;             /* pointer to reserved area */
    ub1          rcs9[CDA_SIZE - sizeof (struct cda_head)];
/* filler */
};
typedef struct cda_def Cda_Def;
/* the logon data area (LDA)
   is the same shape as the CDA */
typedef struct cda_def Lda_Def;


/* OCI Environment Modes for opinit call */
#define OCI_EV_DEF 0     /* default single-threaded environment */
#define OCI_EV_TSF 1                /* thread-safe environment */

/* OCI Logon Modes for olog call */
#define OCI_LM_DEF 0                              /* default login */
#define OCI_LM_NBL 1                         /* non-blocking logon */

/*
 * since sqllib uses both ocidef and ocidfn the following defines
 * need to be guarded
 */
#ifndef OCI_FLAGS
#define OCI_FLAGS

/* OCI_*_PIECE defines the piece types that are returned or set */
#define OCI_ONE_PIECE   0    /* there or this is the only piece */
#define OCI_FIRST_PIECE 1           /* the first of many pieces */
#define OCI_NEXT_PIECE  2            /* the next of many pieces */
#define OCI_LAST_PIECE  3      /* the last piece of this column */
#endif

/* input data types */
#define SQLT_CHR  1           /* (ORANET TYPE) character string */
#define SQLT_NUM  2             /* (ORANET TYPE) oracle numeric */
#define SQLT_INT  3                    /* (ORANET TYPE) integer */
#define SQLT_FLT  4      /* (ORANET TYPE) Floating point number */
```

```
#define SQLT_STR  5                    /* zero terminated string */
#define SQLT_VNU  6            /* NUM with preceding length byte */
#define SQLT_PDN  7      /* (ORANET TYPE) Packed Decimal Numeric */
#define SQLT_LNG  8                                      /* long */
#define SQLT_VCS  9                 /* Variable character string */
#define SQLT_NON  10        /* Null/empty PCC Descriptor entry */
#define SQLT_RID  11                                     /* rowid */
#define SQLT_DAT  12                    /* date in oracle format */
#define SQLT_VBI  15                     /* binary in VCS format */
#define SQLT_BIN  23                     /* binary data(DTYBIN) */
#define SQLT_LBI  24                             /* long binary */
#define SQLT_UIN  68                        /* unsigned integer */
#define SQLT_SLS  91          /* Display sign leading separate */
#define SQLT_LVC  94                     /* Longer longs (char) */
#define SQLT_LVB  95                      /* Longer long binary */

#define SQLT_AFC  96                         /* Ansi fixed char */
#define SQLT_AVC  97                           /* Ansi Var char */
#define SQLT_CUR  102                           /* cursor  type */
#define SQLT_LAB  105                             /* label type */
#define SQLT_OSL  106                           /* oslabel type */

#endif  /* OCIDFN */
```

ocidem.h

```
/*  ocidem.h
 *  Declares additional functions and data structures
 *  used in the OCI C sample programs.
 */

#include <oratypes.h>

#ifndef OCIDEM
#define OCIDEM

/*  internal/external datatype codes */
#define VARCHAR2_TYPE            1
#define NUMBER_TYPE              2
#define INT_TYPE<T><T>           3
#define FLOAT_TYPE               4
#define STRING_TYPE              5
#define ROWID_TYPE              11
#define DATE_TYPE               12

/*  ORACLE error codes used in demonstration programs */
#define VAR_NOT_IN_LIST       1007
#define NO_DATA_FOUND         1403
#define NULL_VALUE_RETURNED   1405
```

```
/*  some SQL and OCI function codes */
#define FT_INSERT                3
#define FT_SELECT                4
#define FT_UPDATE                5
#define FT_DELETE                9

#define FC_OOPEN                14

/*
 *  OCI function code labels,
 *  corresponding to the fc numbers
 *  in the cursor data area.
 */

CONST text  *oci_func_tab[] =  {(text *) ”not used”,
/* 1–2 */       (text *) ”not used”, (text *) ”OSQL”,
/* 3–4 */       (text *) ”not used”, (text *) ”OEXEC, OEXN”,
/* 5–6 */       (text *) ”not used”, (text *) ”OBIND”,
/* 7–8 */       (text *) ”not used”, (text *) ”ODEFIN”,
/* 9–10 */      (text *) ”not used”, (text *) ”ODSRBN”,
/* 11–12 */     (text *) ”not used”, (text *) ”OFETCH, OFEN”,
/* 13–14 */     (text *) ”not used”, (text *) ”OOPEN”,
/* 15–16 */     (text *) ”not used”, (text *) ”OCLOSE”,
/* 17–18 */     (text *) ”not used”, (text *) ”not used”,
/* 19–20 */     (text *) ”not used”, (text *) ”not used”,
/* 21–22 */     (text *) ”not used”, (text *) ”ODSC”,
/* 23–24 */     (text *) ”not used”, (text *) ”ONAME”,
/* 25–26 */     (text *) ”not used”, (text *) ”OSQL3”,
/* 27–28 */     (text *) ”not used”, (text *) ”OBNDRV”,
/* 29–30 */     (text *) ”not used”, (text *) ”OBNDRN”,
/* 31–32 */     (text *) ”not used”, (text *) ”not used”,
/* 33–34 */     (text *) ”not used”, (text *) ”OOPT”,
/* 35–36 */     (text *) ”not used”, (text *) ”not used”,
/* 37–38 */     (text *) ”not used”, (text *) ”not used”,
/* 39–40 */     (text *) ”not used”, (text *) ”not used”,
/* 41–42 */     (text *) ”not used”, (text *) ”not used”,
/* 43–44 */     (text *) ”not used”, (text *) ”not used”,
/* 45–46 */     (text *) ”not used”, (text *) ”not used”,
/* 47–48 */     (text *) ”not used”, (text *) ”not used”,
/* 49–50 */     (text *) ”not used”, (text *) ”not used”,
/* 51–52 */     (text *) ”not used”, (text *) ”OCAN”,
/* 53–54 */     (text *) ”not used”, (text *) ”OPARSE”,
/* 55–56 */     (text *) ”not used”, (text *) ”OEXFET”,
/* 57–58 */     (text *) ”not used”, (text *) ”OFLNG”,
/* 59–60 */     (text *) ”not used”, (text *) ”ODESCR”,
/* 61–62 */     (text *) ”not used”, (text *) ”OBNDRA”
};

#endif      /* OCIDEM */
```

ociapr.h

```
/*
 *  Declare the OCI functions.
 *  Prototype information is included.
 *  Use this header for ANSI C compilers.
 */

#ifndef OCIAPR
#define OCIAPR

#include <oratypes.h>
#include <ocidfn.h>

sword  obindps(struct cda_def *cursor, ub1 opcode, text *sqlvar,
              sb4 sqlvl, ub1 *pvctx, sb4 progvl,
              sword ftype, sword scale,
              sb2 *indp, ub2 *alen, ub2 *arcode,
              sb4 pv_skip, sb4 ind_skip, sb4 alen_skip, sb4
              rc_skip, ub4 maxsiz, ub4 *cursiz,
              text *fmt, sb4 fmtl, sword fmtt);
sword  obreak(struct cda_def *lda);
sword  ocan  (struct cda_def *cursor);
sword  oclose(struct cda_def *cursor);
sword  ocof  (struct cda_def *lda);
sword  ocom  (struct cda_def *lda);
sword  ocon  (struct cda_def *lda);
sword  odefinps(struct cda_def *cursor, ub1 opcode, sword pos,ub1
             *bufctx, sb4 bufl, sword ftype, sword scale,
             sb2 *indp, text *fmt, sb4 fmtl, sword fmtt,
             ub2 *rlen, ub2 *rcode,
             sb4 pv_skip, sb4 ind_skip, sb4 alen_skip,
             sb4 rc_skip);
sword  odessp(struct cda_def *cursor, text *objnam, size_t onlen,
             ub1 *rsv1, size_t rsv1ln, ub1 *rsv2, size_t rsv2ln,
             ub2 *ovrld, ub2 *pos, ub2 *level, text **argnam,
             ub2 *arnlen, ub2 *dtype, ub1 *defsup, ub1* mode,
             ub4 *dtsiz, sb2 *prec, sb2 *scale, ub1 *radix,
             ub4 *spare, ub4 *arrsiz);
sword  odescr(struct cda_def *cursor, sword pos, sb4 *dbsize,
             sb2 *dbtype, sb1 *cbuf, sb4 *cbufl, sb4 *dsize,
             sb2 *prec, sb2 *scale, sb2 *nullok);
sword  oerhms(struct cda_def *lda, sb2 rcode, text *buf,
             sword bufsiz);
sword  oermsg(sb2 rcode, text *buf);
sword  oexec (struct cda_def *cursor);
sword  oexfet(struct cda_def *cursor, ub4 nrows,
             sword cancel, sword exact);
sword  oexn  (struct cda_def *cursor, sword iters, sword rowoff);
sword  ofen  (struct cda_def *cursor, sword nrows);
sword  ofetch(struct cda_def *cursor);
sword  oflng (struct cda_def *cursor, sword pos, ub1 *buf,
             sb4 bufl, sword dtype, ub4 *retl, sb4 offset);
sword  ogetpi(struct cda_def *cursor, ub1 *piecep, dvoid **ctxpp,
             ub4 *iterp, ub4 *indexp);
sword  opinit(ub4 mode);
sword  olog  (struct cda_def *lda, ub1* hda,
             text *uid, sword uidl, text *pswd, sword pswdl,
             text *conn, sword connl, ub4 mode);
sword  ologof(struct cda_def *lda);
sword  oopen (struct cda_def *cursor, struct cda_def *lda,
             text *dbn, sword dbnl, sword arsize,
             text *uid, sword uidl);
sword  oparse(struct cda_def *cursor, text *sqlstm, sb4 sqllen,
             sword defflg, ub4 lngflg);
sword  orol  (struct cda_def *lda);
sword  osetpi(struct cda_def *cursor, ub1 piece, dvoid *bufp,
             ub4 *lenp);
void sqlld2  (struct cda_def *lda, text *cname, sb4 *cnlen);
void sqllda  (struct cda_def *lda);

/* non-blocking functions */
sword onbset (struct cda_def *lda );
sword onbtst (struct cda_def *lda );
sword onbclr (struct cda_def *lda );
sword  oopt  (struct cda_def *cursor, sword rbopt, sword waitopt);

sword  obndra(struct cda_def *cursor, text *sqlvar, sword sqlvl,
             ub1 *progv, sword progvl, sword ftype, sword scale,
             sb2 *indp, ub2 *alen, ub2 *arcode, ub4 maxsiz,
             ub4 *cursiz, text *fmt, sword fmtl, sword fmtt);
sword  obndrn(struct cda_def *cursor, sword sqlvn, ub1 *progv,
             sword progvl, sword ftype, sword scale, sb2 *indp,
             text *fmt, sword fmtl, sword fmtt);
sword  obndrv(struct cda_def *cursor, text *sqlvar, sword sqlvl,
             ub1 *progv, sword progvl, sword ftype, sword scale,
             sb2 *indp, text *fmt, sword fmtl, sword fmtt);
sword  odefin(struct cda_def *cursor, sword pos, ub1 *buf,
             sword bufl, sword ftype, sword scale, sb2 *indp,
             text *fmt, sword fmtl, sword fmtt, ub2 *rlen,
             ub2 *rcode);

/* older calls ; preferred equivalent calls above */
sword  oname (struct cda_def *cursor, sword pos, sb1 *tbuf,
             sb2 *tbufl, sb1 *buf, sb2 *bufl);
sword  orlon (struct cda_def *lda, ub1 *hda,
             text *uid, sword uidl, text *pswd, sword pswdl,
             sword audit);
sword  olon  (struct cda_def *lda, text *uid, sword uidl,
              text *pswd, sword pswdl, sword audit);
sword  osql3 (struct cda_def *cda, text *sqlstm, sword sqllen);
sword  odsc  (struct cda_def *cursor, sword pos, sb2 *dbsize,
              sb2 *fsize, sb2 *rcode, sb2 *dtype, sb1 *buf,
              sb2 *bufl, sb2 *dsize);

#endif /* OCIAPR */
```

ocikpr.h

```
/*
 *  Declare the OCI functions.
 *  Prototype information is commented out.
 *  Use this header for non–ANSI C compilers.
 *  Note that you will need to include ocidfn.h in the .c files
 *    to get the definition for cda_def.
 */

#ifndef OCIKPR
#define OCIKPR

#include <oratypes.h>

sword obindps( /*_ struct cda_def *cursor, ub1 opcode, text
            *sqlvar, sb4 sqlvl, ub1 *pvctx, sb4 progvl,
            sword ftype, sword scale,
            sb2 *indp, ub2 *alen, ub2 *arcode,
            sb4 pv_skip, sb4 ind_skip, sb4 alen_skip, sb4 rc_skip,
            ub4 maxsiz, ub4 *cursiz, text *fmt, sb4 fmtl,
            sword fmtt _*/ );
sword obreak( /*_ struct cda_def *lda _*/ );
sword ocan  ( /*_ struct cda_def *cursor _*/ );
sword oclose( /*_ struct cda_def *cursor _*/ );
sword ocof  ( /*_ struct cda_def *lda _*/ );
sword ocom  ( /*_ struct cda_def *lda _*/ );
sword ocon  ( /*_ struct cda_def *lda _*/ );
sword odefinps( /*_ struct cda_def *cursor, ub1 opcode, sword
            pos,ub1 *bufctx, sb4 bufl, sword ftype, sword scale,
            sb2 *indp, text *fmt, sb4 fmtl, sword fmtt,
            ub2 *rlen, ub2 *rcode,
            sb4 pv_skip, sb4 ind_skip, sb4 alen_skip,
            sb4 rc_skip _*/ );
sword odescr( /*_ struct cda_def *cursor, sword pos, sb4 *dbsize,
            sb2 *dbtype, sb1 *cbuf, sb4 *cbufl, sb4 *dsize,
            sb2 *prec, sb2 *scale, sb2 *nullok _*/ );
```

```
sword odessp( /*_ struct cda_def *cursor, text *objnam, size_t
            onlen, ub1 *rsv1, size_t rsv1ln, ub1 *rsv2, size_t
            rsv2ln, ub2 *ovrld, ub2 *pos, ub2 *level, text
            **argnam, ub2 *arnlen, ub2 *dtype, ub1 *defsup, ub1*
            mode, ub4 *dtsiz, sb2 *prec, sb2 *scale, ub1 *radix,
            ub4 *spare, ub4 *arrsiz _*/ );
sword oerhms( /*_ struct cda_def *lda, sb2 rcode, text *buf,
                   sword bufsiz _*/ );
sword oermsg( /*_ sb2 rcode, text *buf _*/ );
sword oexec ( /*_ struct cda_def *cursor _*/ );
sword oexfet( /*_ struct cda_def *cursor, ub4 nrows,
                   sword cancel, sword exact _*/ );
sword oexn  ( /*_ struct cda_def *cursor, sword iters, sword
            rowoff _*/ );
sword ofen  ( /*_ struct cda_def *cursor, sword nrows _*/ );
sword ofetch( /*_ struct cda_def *cursor _*/ );
sword oflng ( /*_ struct cda_def *cursor, sword pos, ub1 *buf,
             sb4 bufl, sword dtype, ub4 *retl, sb4 offset _*/ );
sword ogetpi( /*_ struct cda_def *cursor, ub1 *piecep, dvoid
             **ctxpp, ub4 *iterp, ub4 *indexp _*/ );
sword opinit( /*_ ub4 mode _*/ );
sword olog  ( /*_ struct cda_def *lda, ub1 *hst,
             text *uid, sword uidl,
             text *psw, sword pswl,
             text *conn, sword connl,
             ub4 mode _*/ );
sword ologof( /*_ struct cda_def *lda _*/ );
sword oopen ( /*_ struct cda_def *cursor, struct cda_def *lda,
             text *dbn, sword dbnl, sword arsize,
             text *uid, sword uidl _*/ );
sword oopt  ( /*_ struct cda_def *cursor, sword rbopt, sword
             waitopt _*/ );
sword oparse( /*_ struct cda_def *cursor, text *sqlstm, sb4
             sqllen, sword defflg, ub4 lngflg _*/ );
sword orol  ( /*_ struct cda_def *lda _*/ );
sword osetpi( /*_ struct cda_def *cursor, ub1 piece, dvoid *bufp,
             ub4 *lenp _*/ );
void sqlld2  ( /*_ struct cda_def *lda, text *cname,
             sb4 *cnlen _*/ );
void sqllda  ( /*_ struct cda_def *lda _*/ );

/* non–blocking functions */
sword onbset( /*_ struct cda_def *lda _*/ );
sword onbtst( /*_ struct cda_def *lda _*/ );
sword onbclr( /*_ struct cda_def *lda _*/ );
sword obndra( /*_ struct cda_def *cursor, text *sqlvar, sword
             sqlvl, ub1 *progv, sword progvl, sword ftype, sword
             scale, sb2 *indp, ub2 *alen, ub2 *arcode, ub4 maxsiz,
             ub4 *cursiz, text *fmt, sword fmtl, sword fmtt _*/ );
sword obndrn( /*_ struct cda_def *cursor, sword sqlvn, ub1
             *progv, sword progvl, sword ftype, sword scale, sb2
             *indp, text *fmt, sword fmtl, sword fmtt _*/ );
sword obndrv( /*_ struct cda_def *cursor, text *sqlvar, sword
             sqlvl, ub1 *progv, sword progvl, sword ftype, sword
             scale, sb2 *indp, text *fmt, sword fmtl,
             sword fmtt _*/ );
sword odefin( /*_ struct cda_def *cursor, sword pos, ub1 *buf,
             sword bufl, sword ftype, sword scale, sb2 *indp,
             text *fmt, sword fmtl, sword fmtt, ub2 *rlen, ub2
             *rcode _*/ );

/* older calls ; preferred equivalent calls above */
sword odsc  ( /*_ struct cda_def *cursor, sword pos, sb2 *dbsize,
                   sb2 *fsize, sb2 *rcode, sb2 *dtype, sb1 *buf,
           sb2 *bufl, sb2 *dsize _*/ );
sword oname ( /*_ struct cda_def *cursor, sword pos, sb1 *tbuf,
           sb2 *tbufl, sb1 *buf, sb2 *bufl _*/ );
sword olon  ( /*_ struct cda_def *lda, text *uid, sword uidl,
           text *pswd, sword pswdl, sword audit _*/ );
sword orlon ( /*_ struct cda_def *lda, ub1 *hda, text *uid,
           sword uidl, text *pswd, sword pswdl, sword audit _*/ );
sword osql3 ( /*_ struct cda_def *cda, text *sqlstm,
           sword sqllen _*/ );

#endif  /* OCIKPR */
```

cdemo6.h

```
/* Used with cdemo6.cc */

extern ”C”
{
#include <string.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>
}
/* oparse flags */
#define  DEFER_PARSE        1
#define  NATIVE             1
#define  VERSION_7          2
/* Class forward declarations */
class connection;
class cursor;
/*
```

```
 * This class represents a connection to ORACLE database.
 *
 * NOTE: This connection class is just given as an example and all
 * possible operations on a connection have not been defined.
 */

class connection
{
  friend class cursor;
  public:
    connection()
      { state = not_connected; memset(hda,'\0', HDA_SIZE); }
    ~connection();
    sword connect(const text *username, const text *password);
    sword disconnect();
    void display_error(FILE* file) const;
  private:
    Lda_Def lda;
    ub1 hda[HDA_SIZE];
    enum conn_state
    {
      not_connected,
      connected
    };
    conn_state state;
};
/*
 * This class represents an ORACLE cursor.
 *
 * NOTE: This cursor class is just given as an example and all
 *       possible operations on a cursor have not been defined.
 */
class cursor
{
  public:
    cursor()
      {state = not_opened; conn = (connection *)0; }
    ~cursor();
    sword open(connection *conn_param);
    sword close();
    sword parse(const text *stmt)
      { return (oparse(&cda, (text *)stmt, (sb4)-1,
                  DEFER_PARSE, (ub4) VERSION_7)); }
    /* bind an input variable */
    sword bind_by_position(sword sqlvnum, ub1 *progvar, sword
                           progvarlen, sword datatype, sword
                           scale, sb2 *indicator)
      { return (obndrn(&cda, sqlvnum, progvar, progvarlen,
                             datatype, scale, indicator, (text *)0,
                             –1, –1)); }
    /* define an output variable */
    sword define_by_position(sword position, ub1 *buf, sword bufl,
                             sword datatype, sword scale, sb2
                             *indicator, ub2 *rlen, ub2 *rcode)
      { return (odefin(&cda, position, buf, bufl, datatype, scale,
                             indicator,(text *)0, –1, –1, rlen,
                             rcode)); }
    sword describe(sword position, sb4 *dbsize, sb2 *dbtype, sb1
                             *cbuf, sb4 *cbufl, sb4 *dsize, sb2
                             *prec, sb2 *scale, sb2 *nullok)
      { return (odescr(&cda, position, dbsize, dbtype, cbuf,
                  cbufl, dsize, prec, scale, nullok)); }
    sword execute()
      { return (oexec(&cda)); }
    sword fetch()
      { return (ofetch(&cda)); }
    sword get_error_code() const
      { return (cda.rc); }
    void display_error( FILE* file) const;
  private:
    Cda_Def cda;
    connection *conn;
    enum cursor_state
    {
      not_opened,
      opened
    };
    cursor_state state;
};
/*
 * Error number macros
 */
#define CONERR_ALRCON –1                   /* already connected */
#define CONERR_NOTCON –2                       /* not connected */
#define CURERR_ALROPN –3              /* cursor is already open */
#define CURERR_NOTOPN –4                /* cursor is not opened */
```

cdemo1.c

```
/*
 *      –– cdemo1.c ––
 *  An example program which adds new employee
 *  records to the personnel data base.  Checking
 *  is done to insure the integrity of the data base.
 *  The employee numbers are automatically selected using
 *  the current maximum employee number as the start.
 *
 *  The program queries the user for data as follows:
 *
 *  Enter employee name:
 *  Enter employee job:
 *  Enter employee salary:
 *  Enter employee dept:
 *
 *  The program terminates if return key (CR) is entered
 *  when the employee name is requested.
 *
 *  If the record is successfully inserted, the following
 *  is printed:
 *
 *  ”ename” added to department ”dname” as employee # ”empno”
 *
 *  The size of the HDA is defined by the HDA_SIZE constant,
 *  which is declared in ocidem.h to be 256 bytes for 32–
 *  bit architectures and 512 bytes for 64–bit architectures.
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <oratypes.h>

/* LDA and CDA struct declarations */
#include <ocidfn.h>
#ifdef __STDC__
#include <ociapr.h>
#else
#include <ocikpr.h>
#endif

/* demo constants and structs */
#include <ocidem.h>
```

```
/* oparse flags */
#define  DEFER_PARSE        1
#define  NATIVE             1
#define  VERSION_7          2

text *username = (text *) ”SCOTT”;
text *password = (text *) ”TIGER”;

/* Define SQL statements to be used in program. */
text *insert = (text *) ”INSERT INTO emp(empno, ename, job, sal,
    ndeptno)\
    VALUES (:empno, :ename, :job, :sal, :deptno)”;
text *seldept = (text *) ”SELECT dname FROM dept WHERE deptno =
    :1”;
text *maxemp = (text *) ”SELECT NVL(MAX(empno), 0) FROM emp”;
text *selemp = (text *) ”SELECT ename, job FROM emp”;

/* Define an LDA, a HDA,  and two cursors. */
Lda_Def lda;
ub1     hda[HDA_SIZE];
Cda_Def cda1;
Cda_Def cda2;

void err_report();
void myfflush();

main()
{
    sword empno, sal, deptno;
    sword len, len2, rv, dsize, dsize2;
    sb4   enamelen, joblen, deptlen;
    sb2   sal_ind, job_ind;
    sb2   db_type, db2_type;
    sb1   name_buf[20], name2_buf[20];
    text  *cp, *ename, *job, *dept;

/*
 *  Connect to ORACLE and open two cursors.
 *  Exit on any error.
 */
    if (olog(&lda, hda, username, –1, password, –1,
             (text *) 0, –1, OCI_LM_DEF))
    {
        err_report(&lda);
        exit(EXIT_FAILURE);
    }
    printf(”Connected to ORACLE as %s\n”, username);
```

```
if (oopen(&cda1, &lda, (text *) 0, –1, –1, (text *) 0, –1))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}

if (oopen(&cda2, &lda, (text *) 0, –1, –1, (text *) 0, –1))
{
    err_report(&cda2);
    do_exit(EXIT_FAILURE);
}

/* Turn off auto–commit. Default is off, however. */
if (ocof(&lda))
{
    err_report(&lda);
    do_exit(EXIT_FAILURE);
}

/* Retrieve the current maximum employee number. */
if (oparse(&cda1, maxemp, (sb4) –1, DEFER_PARSE,
           (ub4) VERSION_7))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}

if (odefin(&cda1, 1, (ub1 *) &empno, (sword) sizeof(sword),
           (sword) INT_TYPE,
           (sword) –1, (sb2 *) 0, (text *) 0, –1, –1,
           (ub2 *) 0, (ub2 *) 0))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}

if (oexfet(&cda1, (ub4) 1, FALSE, FALSE))
{
    if (cda1.rc == NO_DATA_FOUND)
        empno = 10;
    else
    {
        err_report(&cda1);
        do_exit(EXIT_FAILURE);
    }
}
```

```
/*  Describe the columns in the select–list
    of ”selemp” to determine the max length of
    the employee name and job title.
 */
if (oparse(&cda1, selemp, (sb4) –1, FALSE, VERSION_7))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}

len = sizeof(name_buf); len2 = sizeof (name2_buf);

if (odescr(&cda1, 1, &enamelen,
           (sb2 *) &db_type, name_buf, (sb4 *) &len,
           (sb4 *) &dsize, (sb2 *) 0, (sb2 *) 0, (sb2 *) 0) ||
    odescr(&cda1, 2, &joblen,
           (sb2 *) &db_type, name2_buf, (sb4 *) &len2,
           (sb4 *) &dsize2, (sb2 *) 0, (sb2 *) 0, (sb2 *) 0))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}

/* Parse the INSERT statement. */
if (oparse(&cda1, insert, (sb4) –1, FALSE, (ub4) VERSION_7))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}

/* Parse the SELDEPT statement. */
if (oparse(&cda2, seldept, (sb4) –1, FALSE, (ub4) VERSION_7))
{
    err_report(&cda2);
    do_exit(EXIT_FAILURE);
}

/*  Allocate output buffers. Allow for \n and ’\0’. */
ename = (text *) malloc((int) enamelen + 2);
job   = (text *) malloc((int) joblen + 2);

/*  Bind the placeholders in the INSERT statement. */
if (obndrv(&cda1, (text *) ”:ENAME”, –1, (ub1 *) ename,
           enamelen+1, STRING_TYPE, –1, (sb2 *) 0,
           (text *) 0, –1, –1) ||
    obndrv(&cda1, (text *) ”:JOB”, –1, (ub1 *) job, joblen+1,
           STRING_TYPE, –1, &job_ind, (text *) 0, –1, –1) ||
        obndrv(&cda1, (text *) ":SAL", –1, (ub1 *) &sal,
               (sword)sizeof (sal),
               INT_TYPE, –1, &sal_ind, (text *) 0, –1, –1) ||
        obndrv(&cda1, (text *) ":DEPTNO",–1, (ub1 *) &deptno,
               (sword) sizeof (deptno), INT_TYPE, –1,
               (sb2 *) 0, (text *) 0, –1, –1) ||
        obndrv(&cda1, (text *) ":EMPNO", –1, (ub1 *) &empno,
               (sword) sizeof (empno), INT_TYPE, –1,
               (sb2 *) 0, (text *) 0, –1, –1))
    {
        err_report(&cda1);
        do_exit(EXIT_FAILURE);
    }

    /*  Bind the placeholder in the "seldept" statement. */
    if (obndrn(&cda2,
               1,
               (ub1 *) &deptno,
               (sword) sizeof(deptno),
               INT_TYPE,
               –1,
               (sb2 *) 0,
               (text *) 0,
               –1,
               –1))
    {
        err_report(&cda2);
        do_exit(EXIT_FAILURE);
    }

    /*  Describe the select–list field "dname". */
    len = sizeof (name_buf);
    if (odescr(&cda2, 1, (sb4 *) &deptlen, &db_type,
               name_buf, (sb4 *) &len, (sb4 *) &dsize, (sb2 *) 0,
               (sb2 *) 0, (sb2 *) 0))
    {
        err_report(&cda2);
        do_exit(EXIT_FAILURE);
    }

/*  Allocate the dept buffer now that you have length. */
    dept = (text *) malloc((int) deptlen + 1);
```

```
/*  Define the output variable for the select–list. */
if (odefin(&cda2,
           1,
           (ub1 *) dept,
           deptlen+1,
           STRING_TYPE,
           –1,
           (sb2 *) 0,
           (text *) 0,
           –1,
           –1,
           (ub2 *) 0,
           (ub2 *) 0))
{
    err_report(&cda2);
    do_exit(EXIT_FAILURE);
}
for (;;)
{
    /* Prompt for employee name.  Break on no name. */
    printf(”\nEnter employee name (or CR to EXIT): ”);
    fgets((char *) ename, (int) enamelen+1, stdin);
    cp = (text *) strchr((char *) ename, ’\n’);
    if (cp == ename)
    {
        printf(”Exiting... ”);
        do_exit(EXIT_SUCCESS);
    }
    if (cp)
        *cp = ’\0’;
    else
  {
        printf(”Employee name may be truncated.\n”);
        myfflush();
  }

    /*  Prompt for the employee’s job and salary. */
    printf(”Enter employee job: ”);
    job_ind = 0;
    fgets((char *) job, (int) joblen + 1, stdin);
    cp = (text *) strchr((char *) job, ’\n’);
    if (cp == job)
    {
       job_ind = –1;            /* make it NULL in table */
       printf(”Job is NULL.\n”);/* using indicator variable */
    }
    else if (cp == 0)
```

```
        {
              printf(”Job description may be truncated.\n”);
              myfflush();
        }
          else
              *cp = ’\0’;

          printf(”Enter employee salary: ”);
          scanf(”%d”, &sal);

         myfflush();
         sal_ind = (sal <= 0) ? –2 : 0; /* set indicator variable */

          /*
           *  Prompt for the employee’s department number; verify
           *  that the entered department number is valid
           *  by executing and fetching.
           */
          do
          {
              printf(”Enter employee dept: ”);
              scanf(”%d”, &deptno);
              myfflush();
              if (oexec(&cda2) ||
                      (ofetch(&cda2) && (cda2.rc != NO_DATA_FOUND)))
              {
                  err_report(&cda2);
                  do_exit(EXIT_FAILURE);
              }
              if (cda2.rc == NO_DATA_FOUND)
                  printf(”The dept you entered doesn’t exist.\n”);
          } while (cda2.rc == NO_DATA_FOUND);

          /*
           *  Increment empno by 10, and execute the INSERT
           *  statement. If the return code is 1 (duplicate
           *  value in index), then generate the next
           *  employee number.
           */
          empno += 10;
          if (oexec(&cda1) && cda1.rc != 1)
          {
              err_report(&cda1);
              do_exit(EXIT_FAILURE);
          }
          while (cda1.rc == 1)
          {
              empno += 10;
              if (oexec(&cda1) && cda1.rc != 1)
```

```
            {
                err_report(&cda1);
                do_exit(EXIT_FAILURE);
            }
        }  /* end for (;;) */

/* Commit the change. */
        if (ocom(&lda))
        {
            err_report(&lda);
            do_exit(EXIT_FAILURE);
        }
        printf(”\n\n%s added to the %s department as employee
                number %d\n”, ename, dept, empno);
    }
    do_exit(EXIT_SUCCESS);
}


void
err_report(cursor)
    Cda_Def *cursor;
{
    sword n;
    text msg[512];

    printf(”\n–– ORACLE error––\n”);
    printf(”\n”);
    n = oerhms(&lda, cursor–>rc, msg, (sword) sizeof msg);
    fprintf(stderr, ”%s\n”, msg);
    if (cursor–>fc > 0)
        fprintf(stderr, ”Processing OCI function %s”,
            oci_func_tab[cursor–>fc]);
}


/*
 *  Exit program with an exit code.
 */
do_exit(exit_code)
    sword exit_code;
{
    sword error = 0;

    if (oclose(&cda1))
    {
        fprintf(stderr, ”Error closing cursor 1.\n”);
        error++;
    }
    if (oclose(&cda2))
    {
        fprintf(stderr, ”Error closing cursor 2.\n”);
        error++;
    }
    if (ologof(&lda))
    {
        fprintf(stderr, ”Error on disconnect.\n”);
        error++;
    }
    if (error == 0 && exit_code == EXIT_SUCCESS)
        printf (”\nG’day\n”);
    exit(exit_code);
}


void
myfflush()
{
  eb1 buf[50];

  fgets((char *) buf, 50, stdin);
}
```

cdemo2.c

```
/* This program accepts arbitrary SQL statements from the user,
   and processes the statement.  Statements may be entered on
   multiple lines, and must be terminated by a semi–colon.
   If a query, the results are printed.
   Statements are entered at the OCISQL prompt.

   To quit the program, type EXIT at the OCISQL prompt.
   The size of the HDA is defined by the HDA_SIZE constant,
   which is declared in ocidem.h to be 256 bytes for 32–
   bit architectures and 512 bytes for 64–bit architectures.
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* Include OCI–specific headers. */
#include <oratypes.h>
#include <ocidfn.h>
#ifdef __STDC__
#include <ociapr.h>
```

```
#else
#include <ocikpr.h>
#endif
#include <ocidem.h>

/* Constants used in this program. */
#define MAX_BINDS               12
#define MAX_ITEM_BUFFER_SIZE    33
#define MAX_SELECT_LIST_SIZE    12
#define MAX_SQL_IDENTIFIER      31
#define PARSE_NO_DEFER           0
#define PARSE_V7_LNG             2

/* Define one logon data area and one cursor data area
   Also define a host data area for olog.
   (See ocidfn.h for declarations). */
Lda_Def lda;
Cda_Def cda;
ub1     hda[HDA_SIZE];

/* Declare an array of bind values. */
text bind_values[MAX_BINDS][MAX_ITEM_BUFFER_SIZE];

/* Declare structures for query information. */
struct describe
{
    sb4             dbsize;
    sb2             dbtype;
    sb1             buf[MAX_ITEM_BUFFER_SIZE];
    sb4             buflen;
    sb4             dsize;
    sb2             precision;
    sb2             scale;
    sb2             nullok;
};

struct define
{
    ub1             buf[MAX_ITEM_BUFFER_SIZE];
    float           flt_buf;
    sword           int_buf;
    sb2             indp;
    ub2             col_retlen, col_retcode;
};

/* Define arrays of describe and define structs. */
struct describe desc[MAX_SELECT_LIST_SIZE];
struct define   def[MAX_SELECT_LIST_SIZE];
```

```
/*  Declare this programs functions. */
sword  connect_user();
sword  describe_define();
sword  do_binds();
void   do_exit();
void   oci_error();
sword  get_sql_statement();
void   print_header();
void   print_rows();

/* Globals */
static text sql_statement[2048];
static sword sql_function;
static sword numwidth = 8;


main()
{
    sword col, errno, n, ncols;
    text *cp;

    /* Connect to ORACLE. */
    if (connect_user())
        exit(–1);

    /* Open a cursor, exit on error (unrecoverable). */
    if (oopen(&cda, &lda, (text *) 0, –1, –1, (text *) 0, –1))
    {
        printf(”Error opening cursor.  Exiting...\n”);
        ologof(&lda);
        exit(–1);
    }

    /* Process user’s SQL statements. */
    for (;;)
    {
        /* Get the statement, exit on ”exit”. */
        if (get_sql_statement())
            do_exit(0);

        /* Parse the statement; do not defer the parse,
           so that errors come back right away. */
        if (oparse(&cda, (text *) sql_statement, (sb4) –1,
                 (sword) PARSE_NO_DEFER, (ub4) PARSE_V7_LNG))
        {
            oci_error(&cda);
            continue;
        }
        /* Save the SQL function code right after parse. */
        sql_function = cda.ft;

        /* Bind any input variables. */
        if ((ncols = do_binds(&cda, sql_statement)) == –1)
            continue;

        /* If the statement is a query, describe and define
           all select–list items before doing the oexec. */
        if (sql_function == FT_SELECT)
            if ((ncols = describe_define(&cda)) == –1)
                continue;

        /* Execute the statement. */
        if (oexec(&cda))
        {
            oci_error(&cda);
            continue;
        }

        /* Fetch and display the rows for the query. */
        if (sql_function == FT_SELECT)
        {
            print_header(ncols);
            print_rows(&cda, ncols);
        }

        /* Print the rows–processed count. */
        if (sql_function == FT_SELECT ||
            sql_function == FT_UPDATE ||
            sql_function == FT_DELETE ||
            sql_function == FT_INSERT)
            printf(”\n%d row%c processed.\n”, cda.rpc,
                   cda.rpc == 1 ? ’\0’ : ’s’);
        else
            printf(”\nStatement processed.\n”);
    } /* end for (;;) */

}     /* end main() */


sword
connect_user()
{
    text username[132];
    text password[132];
    sword n;
    /* Three tries to connect. */
    for (n = 3; --n >= 0; )
    {
        printf("Username: ");
        gets((char *) username);
        printf("Password: ");
        gets((char *) password);
        if (olog(&lda, hda, username, -1, password, -1,
                 (text *) 0, -1, OCI_LM_DEF))
        {
            printf("Cannot connect as %s.\n", username);
            printf("Try again.\n\n");
        }
        else
        {
            return 0;
        }
    }
    printf("Connection failed.  Exiting...\n");
    return -1;
}

/*  Describe select-list items. */
sword
describe_define(cda)
Cda_Def *cda;
{
    sword col, deflen, deftyp;
    static ub1 *defptr;

    /* Describe the select-list items. */
    for (col = 0; col < MAX_SELECT_LIST_SIZE; col++)
    {
        desc[col].buflen = MAX_ITEM_BUFFER_SIZE;
        if (odescr(cda, col + 1, &desc[col].dbsize,
                   &desc[col].dbtype, &desc[col].buf[0],
                   &desc[col].buflen, &desc[col].dsize,
                   &desc[col].precision, &desc[col].scale,
                   &desc[col].nullok))
        {
            /* Break on end of select list. */
            if (cda->rc == VAR_NOT_IN_LIST)
                break;
            else
            {
                oci_error(cda);
                return -1;
            }
        }
        /* adjust sizes and types for display */
        switch (desc[col].dbtype)
        {
        case NUMBER_TYPE:
            desc[col].dbsize = numwidth;
            /* Handle NUMBER with scale as float. */
            if (desc[col].scale != 0)
            {
                defptr = (ub1 *) &def[col].flt_buf;
                deflen = (sword) sizeof(float);
                deftyp = FLOAT_TYPE;
                desc[col].dbtype = FLOAT_TYPE;
            }
            else
            {
                defptr = (ub1 *) &def[col].int_buf;
                deflen = (sword) sizeof(sword);
                deftyp = INT_TYPE;
                desc[col].dbtype = INT_TYPE;
            }
            break;
        default:
            if (desc[col].dbtype == DATE_TYPE)
                desc[col].dbsize = 9;
            if (desc[col].dbtype == ROWID_TYPE)
                desc[col].dbsize = 18;
            defptr = def[col].buf;
            deflen = desc[col].dbsize > MAX_ITEM_BUFFER_SIZE ?
              MAX_ITEM_BUFFER_SIZE : desc[col].dbsize + 1;
            deftyp = STRING_TYPE;
            break;
        }

        if (odefin(cda, col + 1,
                   defptr, deflen, deftyp,
                   –1, &def[col].indp, (text *) 0, –1, –1,
                   &def[col].col_retlen,
                   &def[col].col_retcode))
        {
            oci_error(cda);
            return –1;
        }
    }
    return col;
}
```

```
/*  Bind input variables. */
sword
do_binds(cda, stmt_buf)
Cda_Def *cda;
text *stmt_buf;
{
    sword i, in_literal, n;
    text *cp, *ph;

    /* Find and bind input variables for placeholders. */
    for (i = 0, in_literal = FALSE, cp = stmt_buf;
              *cp && i < MAX_BINDS; cp++)
    {
        if (*cp == '\'')
            in_literal = ~in_literal;
        if (*cp == ':' && !in_literal)
        {
            for (ph = ++cp, n = 0;
                 *cp && (isalnum(*cp) || *cp == '_')
                     && n < MAX_SQL_IDENTIFIER;
                 cp++, n++
                )
                ;
            *cp = '\0';
            printf("Enter value for %s: ", ph);
            gets((char *) &bind_values[i][0]);
            /* Do the bind, using obndrv().
               NOTE:  the bind variable address must be static.
               This would not work if bind_values were an
               auto on the do_binds stack. */
            if (obndrv(cda, ph, -1, &bind_values[i][0], -1,
                        VARCHAR2_TYPE, -1, (sb2 *) 0, (text *) 0,
                        -1, -1))
            {
                oci_error(cda);
                return -1;
            }
            i++;
        }   /* end if (*cp == ...) */
    }       /* end for () */
    return i;
}


/*  Clean up and exit.  LDA and CDA are
    global. */
void
do_exit(rv)
sword rv;
```

```
{
    if (oclose(&cda))
        fputs(”Error closing cursor!\n”, stdout);
    if (ologof(&lda))
        fputs(”Error logging off!\n”, stdout);
    exit(rv);
}


void
oci_error(cda)
Cda_Def *cda;
{
    text msg[512];
    sword n;

    fputs(”\n–– ORACLE ERROR ––\n”, stderr);
    n = oerhms(&lda, cda–>rc, msg, (sword) sizeof (msg));
    fprintf(stderr, ”%.*s”, n, msg);
    fprintf(stderr, ”Processing OCI function %s\n”,
            oci_func_tab[cda–>fc]);
    fprintf(stderr, ”Do you want to continue? [yn]: ”);
    fgets((char *) msg, (int) sizeof (msg), stdin);
    if (*msg != ’\n’ && *msg != ’y’ && *msg != ’Y’)
        do_exit(1);
    fputc(’\n’, stdout);
}


sword
get_sql_statement()
{
    text cbuf[1024];
    text *cp;
    sword stmt_level;

    for (stmt_level = 1; ;)
    {
        if (stmt_level == 1)
        {
            /* Init statement buffer and print prompt. */
            *sql_statement = ’\0’;
            fputs(”\nOCISQL> ”, stdout);
        }
        else
        {
            printf(”%3d     ”, stmt_level);
        }
        /* Get (part of) a SQL statement. */
        gets((char *) cbuf);
        if (*cbuf == ’\0’)
            continue;
        if (strncmp((char *) cbuf, ”exit”, 4) == 0)
            return –1;

        /* Concatenate to statement buffer. */
        if (stmt_level > 1)
            strcat((char *) sql_statement, ” ”);
        strcat((char *) sql_statement, (char *) cbuf);

        /* Check for possible terminator. */
        cp = &sql_statement[strlen((char *) sql_statement) – 1];

        while (isspace(*cp))
            cp––;
        if (*cp == ’;’)
        {
            *cp = ’\0’;
            break;
        }
        stmt_level++;
    }
    return 0;
}


void
print_header(ncols)
sword ncols;
{
    sword col, n;
    fputc(’\n’, stdout);

    for (col = 0; col < ncols; col++)
    {
        n = desc[col].dbsize – desc[col].buflen;
        if (desc[col].dbtype == FLOAT_TYPE ||
            desc[col].dbtype == INT_TYPE)
        {
            printf(”%*c”, n, ’ ’);
            printf(”%*.*s”, desc[col].buflen,
                   desc[col].buflen, desc[col].buf);
        }
        else
        {
            printf(”%*.*s”, desc[col].buflen,
                   desc[col].buflen, desc[col].buf);
```

```
                printf("%*c", n, ' ');
            }
            fputc(' ', stdout);
        }
        fputc('\n', stdout);
        for (col = 0; col < ncols; col++)
        {
            for (n = desc[col].dbsize; --n >= 0; )
                fputc('-', stdout);
            fputc(' ', stdout);
        }
        fputc('\n', stdout);
    }

    void
    print_rows(cda, ncols)
    Cda_Def *cda;
    sword ncols;
    {
        sword col, n;

        for (;;)
        {
            fputc('\n', stdout);
            /* Fetch a row.  Break on end of fetch,
               disregard null fetch "error". */
            if (ofetch(cda))
            {
                if (cda->rc == NO_DATA_FOUND)
                    break;
                if (cda->rc != NULL_VALUE_RETURNED)
                    oci_error(cda);
            }
            for (col = 0; col < ncols ; col++)
            {
                /* Check col. return code for null.  If
                   null, print n spaces, else print value. */
                if (def[col].indp < 0)
                    printf("%*c", desc[col].dbsize, ' ');
                else
                {
                    switch (desc[col].dbtype)
                    {
                    case FLOAT_TYPE:
                       printf("%*.*f", numwidth, 2, def[col].flt_buf);
                       break;
                    case INT_TYPE:
                       printf("%*d", numwidth, def[col].int_buf);
                       break;
                    default:
                       printf(”%s”, def[col].buf);
                       n = desc[col].dbsize – strlen((char *)
                           def[col].buf);
                       if (n > 0)
                           printf(”%*c”, n, ’ ’);
                       break;
                    }
                }
                fputc(’ ’, stdout);
            }
        }  /* end for (;;) */
    }
```

cdemo3.c

```
/*
 *  cdemo3.c
 *
 *  Demonstrates using the oflng function to retrieve
 *  a portion of a LONG column.
 *
 *  This example ”plays” a digitized voice message
 *  by repeatedly extracting 64 Kbyte chunks of the message
 *  from the table and sending them to a converter buffer
 *  (for example, a digital–to–analog converter’s FIFO buffer).
 *
 *  To better understand this example, the table is created by
 *  the program, and some dummy data is inserted into it.
 *
 *  The converter subroutine is only simulated in this example
 *  program.
 *
 *  The size of the HDA is defined by the HDA_SIZE constant,
 *  which is declared in ocidem.h to be 256 bytes for 32–
 *  bit architectures and 512 bytes for 64–bit architectures.
 *
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MSG_SIZE        200000

#include <oratypes.h>
#include <ocidfn.h>
```

```
#ifdef __STDC__
#include <ociapr.h>
#else
#include <ocikpr.h>
#endif
#include <ocidem.h>

Cda_Def cda;
Lda_Def lda;
ub1     hda[HDA_SIZE];

dvoid do_exit();
dvoid oci_error();
dvoid play_msg();

main()
{
    text sql_statement[256];
    register sword i;
    sb2 indp;
    ub2 retl, rcode;
    sword msg_id;
    sb4 msg_len, len, offset;
    ub1 *ucp;
    register ub1 *ucp1;
    ub4 ret_len;

    /* Connect to ORACLE. */
    if (olog(&lda, hda, (text *) ”scott/tiger”, –1,
              (text *) 0, –1, (text *) 0, –1, OCI_LM_DEF))
    {
        fputs(”Cannot connect as SCOTT. Exiting...\n”, stderr);
        exit(EXIT_FAILURE);
    }
    fputs(”Connected to ORACLE as user SCOTT.\n”, stdout);

    /* Open a cursor. */
    if (oopen(&cda, &lda, (text *) 0, –1,
              –1, (text *) 0, –1))
    {
        fputs(”Cannot open cursor. Exiting...\n”, stderr);
        exit(EXIT_FAILURE);
    }
    fputs(”Program is about to drop the VOICE_MAIL table.\n”,
           stdout);
    fputs(”Is this OK (Y or N)? : ”, stdout);
    fflush(stdout);
```

```
gets((char *) sql_statement);
if (*sql_statement != ’y’ && *sql_statement != ’Y’)
    do_exit(EXIT_SUCCESS);

/* Parse, set defflg parameter for non–deferred parse
   (to execute the DDL statement immediately). */
if (oparse(&cda, (text *) ”DROP TABLE voice_mail”, –1,
           FALSE, 2))
{
    if (cda.rc == 942)
        fputs(”Table did not exist.\n”, stdout);
    else
        oci_error(&cda);
}
else
    fputs(”Dropped table \”voice_mail\”.\n”, stdout);

strcpy((char *) sql_statement, ”CREATE TABLE voice_mail\
    (msg_id NUMBER(6), msg_len NUMBER(12), msg LONG RAW)”);
if (oparse(&cda, sql_statement, –1, FALSE, 2))
    oci_error(&cda);
fputs(”Created table \”voice_mail\”.\n”, stdout);

/* Create a dummy message. */
strcpy((char *) sql_statement,
       ”INSERT INTO voice_mail (msg_id, msg_len, msg) \
                       VALUES (:1, :2, :3)”);
if (oparse(&cda, sql_statement, –1, FALSE, 2))
    oci_error(&cda);

if (obndrn(&cda, 1, (ub1 *) &msg_id, 4, 3, –1,
           (sb2 *) 0, (text *) 0, 0, –1))
    oci_error(&cda);

/* Set buffer address before binding. */
ucp = (ub1 *) malloc(MSG_SIZE);
if (ucp == 0)
{
    fputs(”malloc error\n”, stderr);
    do_exit(EXIT_FAILURE);
}
if (obndrn(&cda, 2, (ub1 *) &msg_len, 4, 3, –1,
           (sb2 *) 0, (text *) 0, 0, –1))
    oci_error(&cda);

if (obndrn(&cda, 3, ucp, MSG_SIZE, 24, –1,
           (sb2 *) 0, (text *) 0, 0, –1))
    oci_error(&cda);
    /* Set bind vars before oexn. */
    msg_id  = 100;
    msg_len = MSG_SIZE;
    for (i = 0, ucp1 = ucp; i < MSG_SIZE; i++)
        *ucp1++ = (ub1) i % 128;

    if (oexn(&cda, 1, 0))
        oci_error(&cda);
    fputs(”Data inserted in table \”voice_mail\”.\n”, stdout);


/*
 *  After setting up the test data, do the
 *  select and fetch the message chunks        */
    strcpy((char *) sql_statement, ”select msg_id, msg_len, msg\
                  from voice_mail where msg_id = 100”);
    if (oparse(&cda, sql_statement, –1, 0, 2))
        oci_error(&cda);

    if (odefin(&cda,
               1,                /* index */
               (ub1 *) &msg_id, /* output variable */
               4,                /* length */
               3,                /* datatype */
               –1,               /* scale */
               (sb2 *) 0,        /* indp */
               (text *) 0,       /* fmt */
               0,                /* fmtl */
               –1,               /* fmtt */
               (ub2 *) 0,        /* retl */
               (ub2 *) 0))       /* rcode */
        oci_error(&cda);

    if (odefin(&cda,
               2,                /* index */
               (ub1 *) &msg_len,/* output variable */
               4,                /* length */
               3,                /* datatype */
               –1,               /* scale */
               (sb2 *) 0,        /* indp */
               (text *) 0,       /* fmt */
               0,                /* fmtl */
               –1,               /* fmtt */
               (ub2 *) 0,        /* retl */
               (ub2 *) 0))       /* rcode */
        oci_error(&cda);
```

```
if (odefin(&cda,
           3,                /* index */
           ucp,              /* output variable */
           100,              /* length */
           24,               /* LONG RAW datatype code */
           –1,               /* scale */
           &indp,            /* indp */
           (text *) 0,       /* fmt */
           0,                /* fmtl */
           –1,               /* fmtt */
           &retl,            /* retl */
           &rcode))          /* rcode */
    oci_error(&cda);

/* Do the query, getting the msg_id and the first
   100 bytes of the message. */
if (oexfet(&cda,
           (ub4) 1,          /* nrows */
           0,                /* cancel (FALSE) */
           0))               /* exact (FALSE) */
{
        oci_error(&cda);
}

fprintf(stdout,
  ”Message %d is available, length is %d.\n”, msg_id,
         msg_len);
fprintf(stdout,
  ”indp = %d, rcode = %d, retl = %d\n”, indp, rcode, retl);

/* Play the message, looping until there are
   no more data points to output. */
for (offset = (ub4) 0; ; offset += (ub4) 0x10000)
{
    len = msg_len < 0x10000 ? msg_len : 0x10000;
    if (oflng(&cda,
              3,                 /* position */
              ucp,               /* buf */
              len,               /* bufl */
              24,                /* datatype */
              &ret_len,          /* retl */
              offset))           /* offset */
        oci_error(&cda);
    /* Output the message chunk. */
    play_msg(ucp, len);
    msg_len –= len;
    if (msg_len <= 0)
        break;
}
```

```
    do_exit(EXIT_SUCCESS);
}


dvoid
play_msg(buf, len)
ub1 *buf;
sword len;
{
    fprintf(stdout, ”\”playing\” %d bytes.\n”, len);
}



dvoid
oci_error(cda)
Cda_Def *cda;
{
    text msg[200];
    sword n;
    fputs(”\n–– ORACLE ERROR ––\n”, stderr);
    n = oerhms(&lda, (sb2) cda–>rc, msg, 200);
    fprintf(stderr, ”%.*s”, n, msg);
    fprintf(stderr, ”Processing OCI function %s\n”,
            oci_func_tab[(int) cda–>fc]);
    do_exit(EXIT_FAILURE);
}


dvoid
do_exit(rv)
sword rv;
{
    fputs(”Exiting...\n”, stdout);
    if (oclose(&cda))
    {
        fputs(”Error closing cursor.\n”, stderr);
        rv = EXIT_FAILURE;
    }
    if (ologof(&lda))
    {
        fputs(”Error logging off.\n”, stderr);
        rv = EXIT_FAILURE;
    }
    exit(rv);
}
```

cdemo4.c

```
/* cdemo4.c
 *
 *  Demonstrates doing a FETCH from a cursor
 *  into PL/SQL tables.  The tables are bound to C
 *  arrays using the obndra routine.
 *  The fully–commented script to create the stored procedure
 *  is in the demo program file calldemo.sql.
 *
 *  Execute this script using SQL*DBA or SQL*Plus
 *  to store the package before executing this program.
 *
 * The script is:
 * create or replace package calldemo as
 *
 *   type char_array is table of varchar2(20) index by
 *         binary_integer;
 *   type num_array is table of float index by binary_integer;
 *
 *   procedure get_employees(
 *     dept_number in     integer,   –– which department to query
 *     batch_size  in     integer,   –– how many rows at a time
 *     found       in out integer,   –– n of rows actually
 *                                      returned
 *     done_fetch  out    integer,   –– all done flag
 *     emp_name    out    char_array,–– arrays of employee names,
 *     job         out    char_array,––                   jobs,
 *     sal         out    num_array);––                   salaries
 *
 * end;
 * /
 *
 * create or replace package body calldemo as
 *
 *   cursor get_emp(
 *     dept_number in     integer) 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 this procedure).  Procedure may 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     integer,
 *     batch_size  in     integer,
 *     found       in out integer,
 *     done_fetch  out    integer,
 *     emp_name    out    char_array,
 *     job         out    char_array,
 *     sal         out    num_array) is
 *
 *    begin
 *      if NOT get_emp%ISOPEN then     –– open the cursor if it is
 *        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 end
 * –– of fetch is encountered, close the cursor and exit
 * –– the loop, returning only the last set of rows found.
 *
 *     done_fetch := FALSE;
 *     found := 0;
 *     for i in 1..batch_size loop
 *        fetch get_emp               –– get one emp table row
 *          into emp_name(i), job(i), sal(i);
 *
 *        if get_emp%notfound then    –– if no row was found, then
 *          close get_emp;            –– close the cursor
 *          done_fetch := TRUE;       –– indicate all done
 *          exit;                     –– exit the loop
 *        else
 *          found := found + 1;       –– else count the row and
 *                                       continue
 *          end if;
 *     end loop;
 *    end;
 * end;
 * /
 */

#include <stdio.h>
#include <string.h>
#include <oratypes.h>
#include <ocidfn.h>

#ifdef __STDC__
#include <ociapr.h>
#else
#include <ocikpr.h>
#endif
```

```
#include <ocidem.h>

#define MAX_ARRAY_SIZE     5
#define NO_PARSE_DEFER     0
#define V7_LNGFLG          2
#define VC_LENGTH         20

/* Declare the data areas. */
Cda_Def cda;
Lda_Def lda;
ub1     hda[512];


/*  Declare routines in this program */
dvoid do_fetch(/*_ void _*/);
dvoid oci_error(/*_ void _*/);


main(argc, argv)
sword argc;
text **argv;
{
    text username[128];

    if (argc > 1)
        strncpy((char *) username, (char *) argv[1],
                sizeof (username) – 1);
    else
        strcpy((char *) username, ”SCOTT/TIGER”);
    if (olog(&lda, hda, username, –1, (text *) 0, –1,
             (text *) 0, –1, OCI_LM_DEF))
    {
        printf(”Cannot connect as %s. Exiting...\n”, username);
        exit(–1);
    }
    else
        printf(”Connected.\n”);

    /* Open the OCI cursor. */
    if (oopen(&cda, &lda, (text *) 0, –1, –1, (text *) 0, –1))
    {
        printf(”Cannot open cursor data area, exiting...\n”);
        exit(–1);
    }
    /* Fetch and print the data. */
    do_fetch();
    /* Close the OCI cursor. */
    if (oclose(&cda))
    {
        printf(”Error closing cursor!\n”);
        exit(–1);
    }

    /* Disconnect from ORACLE. */
    if (ologof(&lda))
    {
        printf(”Error logging off!\n”);
        exit(–1);
    }
    exit(0);
}


/*  Set up an anonymous PL/SQL call to the stored
    procedure that fetches the data. */
dvoid
do_fetch(/*_ void _*/)
{
    text *call_fetch = (text *) ”\
      begin\
        calldemo.get_employees(:deptno, :t_size, :num_ret,\
                           :all_done, :e_name, :job, :sal);\
      end;”;
    sword table_size = MAX_ARRAY_SIZE;
    sword i, n_ret, done_flag;
    sword dept_num;
    sb2 n_ret_indp;
    ub2 n_ret_len, n_ret_rcode;
    ub4 n_ret_cursiz = 0;

    text emp_name[MAX_ARRAY_SIZE][VC_LENGTH];
    sb2 emp_name_indp[MAX_ARRAY_SIZE];
    ub2 emp_name_len[MAX_ARRAY_SIZE];
    ub2 emp_name_rcode[MAX_ARRAY_SIZE];
    ub4 emp_name_cursiz = (ub4) MAX_ARRAY_SIZE;

    text job[MAX_ARRAY_SIZE][VC_LENGTH];
    sb2 job_indp[MAX_ARRAY_SIZE];
    ub2 job_len[MAX_ARRAY_SIZE];
    ub2 job_rcode[MAX_ARRAY_SIZE];
    ub4 job_cursiz = (ub4) MAX_ARRAY_SIZE;

    float salary[MAX_ARRAY_SIZE];
    sb2 salary_indp[MAX_ARRAY_SIZE];
    ub2 salary_len[MAX_ARRAY_SIZE];
```

```
ub2 salary_rcode[MAX_ARRAY_SIZE];
ub4 salary_cursiz = (ub4) MAX_ARRAY_SIZE;

/* parse the anonymous SQL block */
if (oparse(&cda, call_fetch, –1,
           NO_PARSE_DEFER, V7_LNGFLG))
{
    oci_error();
    return;
}

/* initialize the bind arrays */
for (i = 0; i < MAX_ARRAY_SIZE; i++)

{
    emp_name_len[i] = VC_LENGTH;
    job_len[i] = VC_LENGTH;
    salary_len[i] = sizeof (float);
}

n_ret_len = sizeof (sword);

/* bind the department number IN parameter */
if (obndrv(&cda, (text *) ”:deptno”, –1, (ub1 *) &dept_num,
           (sword) sizeof (sword), INT_TYPE, –1,
           (sb2 *) 0, (text *) 0, –1, –1))
{
    oci_error();
    return;
}

/* bind the table size IN parameter */
if (obndrv(&cda, (text *) ”:t_size”, –1, (ub1 *) &table_size,
           (sword) sizeof (sword),
           INT_TYPE, –1, (sb2 *) 0, (text *) 0, –1, –1))
{
    oci_error();
    return;
}

/* bind the fetch done OUT parameter */
if (obndrv(&cda, (text *) ”:all_done”, –1, (ub1 *) &done_flag,
           (sword) sizeof (sword),
           INT_TYPE, –1, (sb2 *) 0, (text *) 0, –1, –1))
{
    oci_error();
    return;
}
```

```
/* Bind the OUT n_ret using obndra. obndrv could
   have been used just as well, since no arrays
   are involved, but it is possible to use obndra
   for scalars as well. */
if (obndra(&cda,
       (text *) ”:num_ret”,
       –1,
       (ub1 *) &n_ret,
       (sword) sizeof (sword),
       INT_TYPE,
       –1,
       &n_ret_indp,
       &n_ret_len,
       &n_ret_rcode,
       (ub4) 0,  /* pass as 0, not 1, when binding a scalar */
       (ub4 *) 0,   /* pass as the null pointer when scalar */
       (text *) 0,
       –1,
       –1))
{
    oci_error();
    return;
}

/* bind the employee name array */
if (obndra(&cda,
       (text *) ”:e_name”,
       –1,
       (ub1 *) emp_name,
       VC_LENGTH,
       VARCHAR2_TYPE,
       –1,
       emp_name_indp,
       emp_name_len,
       emp_name_rcode,
       (ub4) MAX_ARRAY_SIZE,
       &emp_name_cursiz,
       (text *) 0,
       –1,
       –1))
{
    oci_error();
    return;
}
```

```
/* bind the job array */
if (obndra(&cda,
       (text *) ”:job”,
       –1,
       (ub1 *) job,
       VC_LENGTH,
       VARCHAR2_TYPE,
       –1,
       job_indp,
       job_len,
       job_rcode,
       (ub4) MAX_ARRAY_SIZE,
       &job_cursiz,
       (text *) 0,
       –1,
       –1))
{
    oci_error();
    return;
}

/* bind the salary array */
if (obndra(&cda,
       (text *) ”:sal”,
       –1,
       (ub1 *) salary,
       (sword) sizeof (float),
       FLOAT_TYPE,
       –1,
       salary_indp,
       salary_len,
       salary_rcode,
       (ub4) MAX_ARRAY_SIZE,
       &salary_cursiz,
       (text *) 0,
       –1,
       –1))
{
    oci_error();
    return;
}

printf(”\nenter deptno: ”);
scanf(”%d”, &dept_num);

for (;;)
{
    /* execute the fetch */
    if (oexec(&cda))
```

```
        {
            oci_error();
            return;
        }

        printf(”\n%d row%c returned\n”,
               n_ret, n_ret == 1 ? ’\0’ : ’s’);

        if (n_ret > 0)
        {
            printf(”\n%–*.*s%–*.*s%s\n”,
                   VC_LENGTH, VC_LENGTH, ”Employee Name”,
                   VC_LENGTH, VC_LENGTH, ”Job”, ”   Salary”);
            for (i = 0; i < n_ret; i++)
            {
                printf(”%.*s”, emp_name_len[i], emp_name[i]);
                printf(”%*c”, VC_LENGTH – emp_name_len[i], ’ ’);
                printf(”%.*s”, job_len[i], job[i]);
                printf(”%*c”, VC_LENGTH – job_len[i], ’ ’);
                printf(”%8.2f\n”, salary[i]);
            }
        }
        if (done_flag != 0)
        {
            printf(”\n”);
            break;
        }
    }
    return;
}


dvoid
oci_error(/*_ void _*/)
{
    text msg[900];
    sword rv;

    rv = oerhms(&lda, cda.rc, msg, (sword) sizeof (msg));
    printf(”\n\n%.*s”, rv, msg);
    printf(”Processing OCI function %s\n”,
           oci_func_tab[(int) cda.fc]);
    return;
}
```

cdemo5.c

```
/*
 *     -- cdemo5.c --
 *
 *  An example program which demonstrates the use of
 *  Variable Cursors in an OCI program.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocikpr.h>

 /* oparse flags */
#define DEFER_PARSE 1
#define VERSION_7 2
#define NPOS 16
#define DSCLEN 240

text *username = (text *) ”SCOTT”;
text *password = (text *) ”TIGER”;

static  sword retval;
static  ub1   hstb[512];
static  text  errorb[4095];
static  text  cbuf[NPOS][DSCLEN];
static  sb4   cbufl[NPOS];
static  sb4   dbsize[NPOS];
static  sb4   dsize[NPOS];
static  sb2   dbtype[NPOS];
static  sb2   prec[NPOS];
static  sb2   scale[NPOS];
static  sb2   nullok[NPOS];

static Lda_Def lda1;
static text plsql_block[] =
      ”begin \
      OPEN :cursor1 FOR select empno, ename, job, mgr,
hiredate,sal,deptno\
               from emp where job=:job order by empno;\
      end;”;

/* CLIENT CURSORS */
static Cda_Def cursor, cursor_emp;
```

```
/* Prototype */
void oracle_error();


int main ()
{
  Lda_Def *ldap = &lda1;
  ub4    empno;
  text   ename[11];
  text   job[10];
  ub4    mgr;
  text   hidate[10];
  ub4    sal;
  ub4    deptno;
  int    i;
  text   job_kind[50];
  ub4    pos;

  strcpy((char *) job_kind, ”ANALYST”);
  fprintf(stdout,”\n\nFETCHING for job=%s\n\n”,job_kind);

  /* Connect to Oracle as SCOTT/TIGER.  *
   * Exit on any error.                 */
    if (olog(ldap, hstb, username, –1, password, –1,
           (text *) 0, –1, OCI_LM_DEF))
  {
      printf(”Unable to connect as %s\n”, username);
      exit(EXIT_FAILURE);
  }

  printf(”Connected to Oracle as %s\n\n”, username);

  /*
   * Open a cursor for executing the PL/SQL block.
   */
  if (oopen(&cursor, ldap, (text *) 0, –1, 0, (text *) 0, –1))
  {
     oracle_error(&cursor);
     exit(EXIT_FAILURE);
  }

  /* Parse the PL/SQL block. */

  if (oparse(&cursor, plsql_block, (sb4) –1, (sword) TRUE,
             (ub4) 2))
  {
     oracle_error(&cursor);
     exit(EXIT_FAILURE);
  }
```

```
/*
 * Bind a variable of cursor datatype, the cursor will be opened
 * inside the PL/SQL block.
 */

if (obndra(&cursor,(text *) ":cursor1", –1, (ub1 *) &cursor_emp,
           –1, SQLT_CUR, –1, (sb2 *) 0, (ub2 *) 0, (ub2 *) 0,
           (ub4) 0, (ub4 *) 0, (text *) 0, 0, 0))
{
   oracle_error(&cursor);
   exit(EXIT_FAILURE);
}

/*
 * Bind a variable of string datatype.
 */

if (obndra(&cursor, (text *) ":job", –1, (ub1 *) job_kind,
           –1, SQLT_STR, –1, (sb2 *) 0, (ub2 *) 0, (ub2 *) 0,
           (ub4) 0, (ub4 *) 0, (text *) 0, 0, 0))
{
   oracle_error(&cursor);
   exit(EXIT_FAILURE);
}

/*
 * Execute the PL/SQL block.
 */

 if (oexec(&cursor))
 {
    oracle_error(&cursor);
    exit(EXIT_FAILURE);
 }

/*
 * Close the cursor on which the PL/SQL block executed.
 */

if (oclose(&cursor))
{
   oracle_error(&cursor);
   exit(EXIT_FAILURE);
}
```

```
/*
 * Do describe on cursor initialized and returned from the
   PL/SQL block.
 */
for (pos = 0; pos < NPOS; pos++)
{
   cbufl[pos] = DSCLEN;
   if (odescr(&cursor_emp, (sword) (pos+1), &dbsize[pos],
            &dbtype[pos],(sb1 *) cbuf[pos], &cbufl[pos],
            &dsize[pos], &prec[pos], &scale[pos], &nullok[pos]))
   {
      if (cursor_emp.rc == 1007)
         break;
      oracle_error(&cursor_emp);
      exit(EXIT_FAILURE);
   }
}

printf(”Describe select–list returns:\n\n”);
printf(”–––––––––––––––––––––––––––––––––––––––\n”);
printf(”Item\t\tMaxSize\t\tType\n”);
printf(”–––––––––––––––––––––––––––––––––––––––\n”);

for (i = 0; i < pos; i++)
{
   cbuf[i][cbufl[i]] = ’\0’;
   printf(”%s\t\t%d\t\t%d\n”, cbuf[i], dbsize[i], dbtype[i]);
}

/*
 * Do client defines.
 */

if (odefin(&cursor_emp, 1, (ub1 *) &empno, (sword) sizeof(ub4),
           SQLT_INT, –1, (sb2 *) –1, (text *) 0, (sword) 0,
          (sword) 0, (ub2 *) 0,(ub2 *) 0))
{
   oracle_error(&cursor_emp);
   exit(EXIT_FAILURE);
}

 if (odefin(&cursor_emp, 2, (ub1 *) ename, (sword)
            sizeof(ename), SQLT_STR, –1, (sb2 *) –1, (text *)0,
            (sword) 0, (sword) 0,(ub2 *) 0, (ub2 *) 0))
 {
    oracle_error(&cursor_emp);
    exit(EXIT_FAILURE);
 }
```

```
  if (odefin(&cursor_emp, 3, (ub1 *) job, (sword) sizeof(job),
             SQLT_STR, –1, (sb2 *) –1, (text *) 0, (sword) 0,
             (sword) 0, (ub2 *) 0, (ub2 *) 0))
   {
      oracle_error(&cursor_emp);
      exit(EXIT_FAILURE);
   }

  if (odefin(&cursor_emp, 4, (ub1 *)&mgr, (sword) sizeof(ub4),
             SQLT_INT, –1, (sb2 *) –1, (text *)0, (sword) 0,
             (sword) 0, (ub2 *) 0, (ub2 *) 0))
   {
      oracle_error(&cursor_emp);
      exit(EXIT_FAILURE);
   }

  if (odefin(&cursor_emp, 5, (ub1 *)hidate, (sword)sizeof(hidate),
             SQLT_STR, –1, (sb2 *) –1, (text *) 0, (sword) 0,
            (sword) 0, (ub2 *) 0, (ub2 *) 0))
   {
      oracle_error(&cursor_emp);
      exit(EXIT_FAILURE);
   }

   if (odefin(&cursor_emp, 6, (ub1 *) &sal, (sword) sizeof(ub4),
              SQLT_INT, –1, (sb2 *) –1, (text *) 0, (sword) 0,
             (sword) 0, (ub2 *) 0, (ub2 *) 0))
   {
      oracle_error(&cursor_emp);
      exit(EXIT_FAILURE);
   }

   if (odefin(&cursor_emp, 7, (ub1 *) &deptno,
             (sword)sizeof(deptno), SQLT_INT, –1, (sb2 *) –1,
             (text *) 0, (sword) 0, (sword) 0, (ub2 *) 0,
             (ub2 *) 0))
   {
      oracle_error(&cursor_emp);
      exit(EXIT_FAILURE);
   }
  printf(”\nFETCH from variable cursor:\n\n”);

printf(”––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
––\n”);

  printf(”empno\tename\tjob\tmgr\thiredate\tsalary\tdept\n”);

printf(”––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
––\n”);
```

```
  /*
   * Now fetch the result set and display.
   */

  while (1)
  {
    sb4 err = 0;
    if (err = ofetch(&cursor_emp))
    {
      if (cursor_emp.rc == 1403)
         break;
      else
      {
         oracle_error(&cursor_emp);
         exit(EXIT_FAILURE);
      }
    }
    else
    {
      /* A row was returned; have to do the fetch. */

      fprintf(stdout, ”%d\t%s\t%s\t%d\t%s\t%d\t%d\n\n”,
              empno, ename, job, mgr, hidate, sal, deptno);
    }
  }

  /*
   * Log off.
   */

  if (ologof(ldap))
  {
     oracle_error(&cursor_emp);
     exit(EXIT_FAILURE);
  }
} /* end of main */


void oracle_error(lda)
Lda_Def * lda;
{
  char msgbuf[512];
  int n=oerhms(lda, lda–>rc, msgbuf, (int) sizeof(msgbuf) );

  printf(”\n\n%.*s\n”,n,msgbuf);
}
```

cdemo6.cc

```
/*      -- cdemo6.cc --
 *  An example program which illustrates how a C++ program
 *  can use the OCI interface to access ORACLE database.
 *
 *  This program retrieves department name, given the
 *  department number.
 *
 *  The program queries the user for data as follows:
 *
 *  Enter department number:
 *
 *  The program terminates if –1 is entered
 *  when the department number is requested.
 */

extern ”C”
{
#include <stdio.h>
#include <oratypes.h>
#include <ociapr.h>
/* demo constants and structs */
#include <ocidem.h>
}
extern ”C”
{
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>
}
/* oparse flags */
#define  DEFER_PARSE        1
#define  NATIVE             1
#define  VERSION_7          2
/* Class forward declarations */
class connection;
class cursor;
/*
 * This class represents a connection to ORACLE database.
 *
 * NOTE: This connection class is just given as an example and
 * all possible operations on a connection have not been defined.
 */
class connection
{
  friend class cursor;
  public:
    connection()
```

```
      { state = not_connected; }
    ~connection();
    sword connect(const text *username, const text *password);
    sword disconnect();
    void display_error(FILE* file) const;
  private:
    Lda_Def lda;
    ub1 hda[HDA_SIZE];
    enum conn_state
    {
      not_connected,
      connected
    };
    conn_state state;
};
/*
 * This class represents an ORACLE cursor.
 *
 * NOTE: This cursor class is just given as an example and all
 * possible operations on a cursor have not been defined.
 */
class cursor
{
  public:
    cursor()
      {state = not_opened; conn = (connection *)0; }
    ~cursor();
    sword open(connection *conn_param);
    sword close();
    sword parse(const text *stmt)
      { return (oparse(&cda, (text *)stmt, (sb4)–1,
                       DEFER_PARSE, (ub4) VERSION_7)); }
    /* bind an input variable */
    sword bind_by_position(sword sqlvnum, ub1 *progvar,
                           sword progvarlen, sword datatype,
                           sword scale, sb2 *indicator)
      { return (obndrn(&cda, sqlvnum, progvar, progvarlen,
                datatype, scale, indicator, (text *)0, –1, –1)); }
    /* define an output variable */
    sword define_by_position(sword position, ub1 *buf,
                sword bufl, sword datatype, sword scale,
                sb2 *indicator, ub2 *rlen, ub2 *rcode)
      { return (odefin(&cda, position, buf, bufl, datatype,
                       scale, indicator,
                       (text *)0, –1, –1, rlen, rcode)); }
    sword describe(sword position, sb4 *dbsize, sb2 *dbtype,
                   sb1 *cbuf, sb4 *cbufl, sb4 *dsize, sb2 *prec,
                   sb2 *scale, sb2 *nullok)
      { return (odescr(&cda, position, dbsize, dbtype,
                        cbuf, cbufl, dsize,  prec, scale, nullok));
      }
    sword execute()
      { return (oexec(&cda)); }
    sword fetch()
      { return (ofetch(&cda)); }
    sword get_error_code() const
      { return (cda.rc); }
    void display_error( FILE* file) const;
  private:
    Cda_Def cda;
    connection *conn;
    enum cursor_state
    {
      not_opened,
      opened
    };
    cursor_state state;
};
/* Error number macros */

#define CONERR_ALRCON –1            /* already connected */
#define CONERR_NOTCON –2            /* not connected */
#define CURERR_ALROPN –3            /* cursor is already open */
#define CURERR_NOTOPN –4            /* cursor is not opened */

/* exit status upon failure */
#define  EXIT_FAILURE       1

const text *username = (text *) ”SCOTT”;
const text *password = (text *) ”TIGER”;
/* define SQL statements to be used in the program */
const text *seldept = (text *) ”SELECT dname FROM dept WHERE
deptno = :1”;
void err_report(FILE *file, text *errmsg, sword func_code);
void myfflush();
/* connection destructor */
connection::~connection()
{
  // disconnect if connection exists
  if (state == connected)
  {
    if (disconnect())
    {
      display_error(stderr);
    }
  }
}
```

```
/* connect to ORACLE */
sword connection::connect(const text *username, const text
*password)
{
  sword status;
  if (state == connected)
  {
    // this object is already connected
    return (CONERR_ALRCON);
  }
  if ((status = olog(&lda, hda, (text *)username, –1
                     (text *)password, –1, (text *) 0, –1,
                     OCI_LM_DEF)) == 0)
  {
    // successful login
    state = connected;
    printf(”Connected to ORACLE as %s\n”, username);
  }
  return (status);
}
/* disconnect from ORACLE */
sword connection::disconnect()
{
  sword status;
  if (state == not_connected)
  {
    // this object has not been connected
    return (CONERR_NOTCON);
  }
  if ((status = ologof(&lda)) == 0)
  {
    // successful logout
    state = not_connected;
  }
  return (status);
}
/* write error message to the given file */
void connection::display_error(FILE *file) const
{
  if (lda.rc != 0)
  {
    sword n;
    text msg[512];
    n = oerhms((cda_def *)&lda, lda.rc, msg, (sword) sizeof(msg));
    err_report(file, msg, lda.fc);
  }
}
/* cursor destructor */
cursor::~cursor()
```

```
{
  if (state == opened)
  {
    if (close())
      display_error(stderr);
  }
}
/* open the cursor */
sword cursor::open(connection *conn_param)
{
  sword status;
  if (state == opened)
  {
    // this cursor has already been opened
    return (CURERR_ALROPN);
  }
  if ((status = oopen(&cda, &conn_param–>lda, (text *)0, –1, –1,
                      (text *)0, –1)) == 0)
  {
    // successfull open
    state = opened;
    conn = conn_param;
  }
  return (status);
}
/* close the cursor */
sword cursor::close()
{
  sword status;
  if (state == not_opened)
  {
    // this cursor has not been opened
    return (CURERR_NOTOPN);
  }
  if ((status = oclose(&cda)) == 0)
  {
    // successful cursor close
    state = not_opened;
    conn = (connection *)0;
  }
  return (status);
}
/* write error message to the given file */
void cursor::display_error(FILE *file) const
{
  if (cda.rc != 0)
  {
    sword n;
    text msg[512];
```

```
        n = oerhms(&conn–>lda, cda.rc, msg, (sword) sizeof(msg));
        err_report(file, msg, cda.fc);
      }
    }
    int main()
    {
        sword deptno;
        sword len, dsize;
        sb4   deptlen;
        sb2   db_type;
        sb1   name_buf[20];
        text  *dept;

    /*
     *  Connect to ORACLE and open a cursor.
     *  Exit on any error.
     */

        connection conn;
        if (conn.connect(username, password))
        {
          conn.display_error(stderr);
          return(EXIT_FAILURE);
        }
        cursor crsr;
        if (crsr.open(&conn))
        {
          crsr.display_error(stderr);
          return(EXIT_FAILURE);
        }
        /* parse the SELDEPT statement */
        if (crsr.parse(seldept))
        {
          crsr.display_error(stderr);
          return(EXIT_FAILURE);
        }

        /* bind the placeholder in the SELDEPT statement */
        if (crsr.bind_by_position(1, (ub1 *) &deptno,
                                  (sword) sizeof(deptno),
                                  INT_TYPE, –1, (sb2 *) 0))
        {
          crsr.display_error(stderr);
          return(EXIT_FAILURE);
        }

        /* describe the select–list field ”dname” */
        len = sizeof (name_buf);
```

```
if (crsr.describe(1, (sb4 *) &deptlen, &db_type,
                  name_buf, (sb4 *) &len, (sb4 *) &dsize,
                 (sb2 *) 0, (sb2 *) 0, (sb2 *) 0))
{
  crsr.display_error(stderr);
  return(EXIT_FAILURE);
}

/* allocate space for dept name now that you have length */
dept = new text((int) deptlen + 1);

/* define the output variable for the select-list */
if (crsr.define_by_position(1, (ub1 *) dept, (sword)deptlen+1,
                            STRING_TYPE, -1, (sb2 *) 0,
                           (ub2 *) 0, (ub2 *) 0))
{
  crsr.display_error(stderr);
  delete dept;
  return(EXIT_FAILURE);
}
for (;;)
{
    /* prompt for department number, */
    /* break if given number == -1 */
    printf("\nEnter department number (or -1 to EXIT): ");
    while (scanf("%d", &deptno) != 1)
    {
      myfflush();
      printf("Invalid input, please enter a number \
             (-1 to EXIT): ");
    }
    if (deptno == -1)
    {
      printf("Exiting... ");
      break;
    }

    /* display the name of the corresponding department */
    if (crsr.execute() || crsr.fetch())
    {
      if (crsr.get_error_code() != NO_DATA_FOUND)
      {
        crsr.display_error(stderr);
        delete dept;
        return(EXIT_FAILURE);
      }
      else
```

```
          printf(”\n  The department number that you entered \
                 doesn’t exist.\n”);
        }
        else
       {
       printf(”\n  Department name = %s    \
              Department number = %d\n”,
              dept, deptno);
       }
    }
    delete dept;
    printf (”\nG’day\n”);
  return 0;
}

void err_report(FILE *file, text *errmsg, sword func_code)
{
    fprintf(file, ”\n–– ORACLE error––\n\n%s\n”, errmsg);
    if (func_code > 0)
        fprintf(file, ”Processing OCI function %s\n”,
            oci_func_tab[func_code]);
}

void myfflush()
{
  eb1 buf[50];

  fgets((char *) buf, 50, stdin);
}
```

APPENDIX

B

Sample Programs in COBOL

This appendix contains three sample OCI programs written in COBOL. The first adds a new employee to a database, the second processes dynamic SQL statements, and the third fetches a portion of a LONG or LONG RAW column using OFLNG.

Each of these sample programs is available online. The exact name and storage location of these progams is system dependent. See your Oracle system–specific documentation for details.

This appendix contains listings for the following files:

- CBDEM1.COB
- CBDEM2.COB
- CBDEM3.COB

CBDEM1.COB

```
* ------------------------------------------------------------
* CBDEM1 IS A SIMPLE EXAMPLE PROGRAM WHICH ADDS NEW EMPLOYEE
* ROWS TO THE PERSONNEL DATA BASE. CHECKING IS DONE TO
* INSURE THE INTEGRITY OF THE DATA BASE. EMPLOYEE NUMBERS
* ARE AUTOMATICALLY SELECTED USING THE CURRENT MAXIMUM
* EMPLOYEE NUMBER AS THE START. DUPLICATE NUMBERS ARE SKIPPED.
* THE PROGRAM QUERIES THE USER FOR DATA AS FOLLOWS:
*
*      Enter employee name  :
*      Enter employee job   :
*      Enter employee salary:
*      Enter employee dept  :
*
* TO EXIT THE PROGRAM, ENTER A CARRIAGE RETURN AT THE
* PROMPT FOR EMPLOYEE NAME. IF THE ROW IS SUCCESSFULLY
* INSERTED, THE FOLLOWING IS PRINTED:
*
* ENAME added to DNAME department as employee # NNNNN
*
* THE MAXIMUM LENGTHS OF THE 'ENAME', 'JOB', AND 'DNAME'
* COLUMNS WILL BE DETERMINED BY THE ODESCR CALL.
*------------------------------------------------------------

 IDENTIFICATION DIVISION.
 PROGRAM-ID.  CBDEM1.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  LDA.
     02   LDA-V2RC       PIC S9(4) COMP.
     02   FILLER         PIC X(10).
     02   LDA-RC         PIC S9(4) COMP.
     02   FILLER         PIC X(50).
 01  HDA                 PIC X(512).

 01  CURSOR-1.
     02   C-V2RC         PIC S9(4) COMP.
     02   C-TYPE         PIC S9(4) COMP.
     02   C-ROWS         PIC S9(9) COMP.
     02   C-OFFS         PIC S9(4) COMP.
     02   C-FNC          PIC S9(4) COMP.
     02   C-RC           PIC S9(4) COMP.
     02   FILLER         PIC X(50).
 01  CURSOR-2.
     02   C-V2RC         PIC S9(4) COMP.
     02   C-TYPE         PIC S9(4) COMP.
```

```
    02   C–ROWS          PIC S9(9) COMP.
    02   C–OFFS          PIC S9(4) COMP.
    02   C–FNC           PIC S9(4) COMP.
    02   C–RC            PIC S9(4) COMP.
    02   FILLER          PIC X(50).

77   USER–ID             PIC X(5)  VALUE ”SCOTT”.
77   USER–ID–L           PIC S9(9) VALUE 5 COMP.
77   PSW                 PIC X(5)  VALUE ”tiger”.
77   PSW–L               PIC S9(9) VALUE 5 COMP.
77   CONN                PIC S9(9) VALUE 0 COMP.
77   CONN–L              PIC S9(9) VALUE 0 COMP.
77   CONN–MODE           PIC S9(9) VALUE 0 COMP.

77   SQL–SEL             PIC X(38) VALUE
        ”SELECT DNAME FROM DEPT WHERE DEPTNO=:1”.
77   SQL–SEL–L           PIC S9(9) VALUE 38 COMP.

77   SQL–INS             PIC X(150) VALUE
        ”INSERT INTO EMP (EMPNO,ENAME,JOB,SAL,DEPTNO)
–       ” VALUES (:EMPNO,:ENAME,:JOB,:SAL,:DEPTNO)”.
77   SQL–INS–L           PIC S9(9) VALUE 150 COMP.

77   SQL–SELMAX          PIC X(33) VALUE
        ”SELECT NVL(MAX(EMPNO),0) FROM EMP”.
77   SQL–SELMAX–L        PIC S9(9) VALUE 33 COMP.

77   SQL–SELEMP          PIC X(26) VALUE
        ”SELECT ENAME,JOB FROM EMP”.
77   SQL–SELEMP–L        PIC S9(9) VALUE 26 COMP.

77   EMPNO               PIC S9(9) COMP.
77   EMPNO–D             PIC ZZZZ9.
77   ENAME               PIC X(12).
77   JOB                 PIC X(12).
77   SAL                 PIC X(10).
77   DEPTNO              PIC X(10).
77   FMT                 PIC X(6).
77   CBUF                PIC X(10).
77   DNAME               PIC X(15).

77   ENAME–L             PIC S9(9) VALUE 12 COMP.
77   ENAME–SIZE          PIC S9(4) COMP.
77   JOB–L               PIC S9(9) VALUE 12 COMP.
77   JOB–SIZE            PIC S9(4) COMP.
77   SAL–L               PIC S9(9) VALUE 10 COMP.
77   DEPTNO–L            PIC S9(9) VALUE 10 COMP.
77   DNAME–L             PIC S9(9) VALUE 15 COMP.
77   DNAME–SIZE          PIC S9(4) COMP.
```

```
 77   EMPNO–N            PIC X(6) VALUE ”:EMPNO”.
 77   ENAME–N            PIC X(6) VALUE ”:ENAME”.
 77   JOB–N              PIC X(4) VALUE ”:JOB”.
 77   SAL–N              PIC X(4) VALUE ”:SAL”.
 77   DEPTNO–N           PIC X(7) VALUE ”:DEPTNO”.
 77   EMPNO–N–L          PIC S9(9) VALUE 6 COMP.
 77   ENAME–N–L          PIC S9(9) VALUE 6 COMP.
 77   JOB–N–L            PIC S9(9) VALUE 4 COMP.
 77   SAL–N–L            PIC S9(9) VALUE 4 COMP.
 77   DEPTNO–N–L         PIC S9(9) VALUE 7 COMP.

 77   INTEGER            PIC S9(9) COMP VALUE 3.
 77   ASC                PIC S9(9) COMP VALUE 1.
 77   ZERO–A             PIC S9(9) COMP VALUE 0.
 77   ZERO–B             PIC S9(4) COMP VALUE 0.
 77   ONE                PIC S9(9) COMP VALUE 1.
 77   TWO                PIC S9(9) COMP VALUE 2.
 77   FOUR               PIC S9(9) COMP VALUE 4.
 77   SIX                PIC S9(9) COMP VALUE 6.
 77   EIGHT              PIC S9(9) COMP VALUE 8.
 77   ERR–RC             PIC S9(4) COMP.
 77   ERR–FNC            PIC S9(4) COMP.
 77   ERR–RC–D           PIC ZZZ9.
 77   ERR–FNC–D          PIC ZZ9.
 77   MSGBUF             PIC X(160).
 77   MSGBUF–L           PIC S9(9) COMP VALUE 160.

 77   ASK–EMP            PIC X(25) VALUE
                           ”Enter employee name: ”.
 77   ASK–JOB            PIC X(25) VALUE
                           ”Enter employee job: ”.
 77   ASK–SAL            PIC X(25) VALUE
                           ”Enter employee salary: ”.
 77   ASK–DEPTNO         PIC X(25) VALUE
                           ”Enter employee dept: ”.

 PROCEDURE DIVISION.
 BEGIN.

*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
* CONNECT TO ORACLE IN NON–BLOCKING MODE.
* HDA MUST BE INITIALIZED TO ALL ZEROS BEFORE CALL TO OLOG.
*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

     MOVE LOW–VALUES TO HDA.

     CALL ”OLOG” USING LDA, HDA, USER–ID, USER–ID–L,
           PSW, PSW–L, CONN, CONN–L, CONN–MODE.
```

```
          IF LDA–RC NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–STOP.
          DISPLAY ”Connected to ORACLE as user ”, USER–ID.

     *––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
     * OPEN THE CURSORS.
     *––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

          CALL ”OOPEN” USING CURSOR–1, LDA.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–LOGOF.

          CALL ”OOPEN” USING CURSOR–2, LDA.
          IF C–RC IN CURSOR–2 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–LOGOF.

     *––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
     * DISABLE AUTO–COMMIT.
     * NOTE: THE DEFAULT IS OFF, SO THIS COULD BE OMITTED.
     *––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

          CALL ”OCOF” USING LDA.
          IF LDA–RC NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

     *––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
     * RETRIEVE THE CURRENT MAXIMUM EMPLOYEE NUMBER.
     *––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

          CALL ”OPARSE” USING CURSOR–1, SQL–SELMAX, SQL–SELMAX–L,
                ZERO–A, TWO.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          CALL ”ODEFIN” USING CURSOR–1, ONE, EMPNO, FOUR,
                INTEGER, ZERO–A, ZERO–B, FMT, ZERO–A, ZERO–A,
                ZERO–B, ZERO–B.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.
          CALL ”OEXEC” USING CURSOR–1.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.
          CALL ”OFETCH” USING CURSOR–1.
          IF C–RC IN CURSOR–1 NOT = 0
             IF C–RC IN CURSOR–1 NOT = 1403
                PERFORM ORA–ERROR
                GO TO EXIT–CLOSE
             ELSE
                MOVE 10 TO EMPNO.

     *-----------------------------------------------------------
     * DETERMINE THE MAX LENGTH OF THE EMPLOYEE NAME AND
     * JOB TITLE.  PARSE THE SQL STATEMENT –
     * IT WILL NOT BE EXECUTED.
     * DESCRIBE THE TWO FIELDS SPECIFIED IN THE SQL STATEMENT.
     *-----------------------------------------------------------

          CALL ”OPARSE” USING CURSOR–1, SQL–SELEMP, SQL–SELEMP–L,
                ZERO–A, TWO.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          CALL ”ODESCR” USING CURSOR–1, ONE, ENAME–SIZE, ZERO–B,
                CBUF, ZERO–A, ZERO–A, ZERO–B, ZERO–B, ZERO–B.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          CALL ”ODESCR” USING CURSOR–1, TWO, JOB–SIZE, ZERO–B,
                CBUF, ZERO–A, ZERO–A, ZERO–B, ZERO–B, ZERO–B.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          IF ENAME–SIZE > ENAME–L
             DISPLAY ”ENAME too large for buffer.”
             GO TO EXIT–CLOSE.
          IF JOB–SIZE > JOB–L
             DISPLAY ”JOB too large for buffer.”
             GO TO EXIT–CLOSE.

      *-----------------------------------------------------------
     * PARSE THE INSERT AND SELECT STATEMENTS.
     *-----------------------------------------------------------

          CALL ”OPARSE” USING CURSOR–1, SQL–INS, SQL–INS–L,
                ZERO–A, TWO.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.
```

```
          CALL ”OPARSE” USING CURSOR–2, SQL–SEL, SQL–SEL–L,
               ZERO–A, TWO.
          IF C–RC IN CURSOR–2 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

     *–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
     * BIND ALL SQL SUBSTITUTION VARIABLES.
     *–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

          CALL ”OBNDRV” USING CURSOR–1, EMPNO–N, EMPNO–N–L,
                EMPNO, FOUR, INTEGER, ZERO–A.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          CALL ”OBNDRV” USING CURSOR–1, ENAME–N, ENAME–N–L,
               ENAME, ENAME–L, ASC.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          CALL ”OBNDRV” USING CURSOR–1, JOB–N, JOB–N–L,
                JOB, JOB–L, ASC.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          CALL ”OBNDRV” USING CURSOR–1, SAL–N, SAL–N–L, SAL,
                SAL–L, ASC.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

          CALL ”OBNDRV” USING CURSOR–1, DEPTNO–N, DEPTNO–N–L,
                DEPTNO, DEPTNO–L, ASC.
          IF C–RC IN CURSOR–1 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.

     *–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
     * BIND THE DEPTNO SUBSTITUTION VAR IN THE SELECT STATEMENT.
     *–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

          CALL ”OBNDRN” USING CURSOR–2, ONE, DEPTNO,
                DEPTNO–L, ASC.
          IF C–RC IN CURSOR–2 NOT = 0
             PERFORM ORA–ERROR
             GO TO EXIT–CLOSE.
```

```
*-----------------------------------------------------------
* DESCRIBE THE ’DNAME’ COLUMN – ONLY THE LENGTH.
*-----------------------------------------------------------

     CALL ”ODSC” USING CURSOR–2, ONE, DNAME–SIZE.
     IF C–RC IN CURSOR–2 NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–CLOSE.

     IF DNAME–SIZE > DNAME–L
        DISPLAY ”DNAME is to large for buffer.”
        GO TO EXIT–CLOSE.

*-----------------------------------------------------------
* DEFINE THE BUFFER TO RECEIVE ’DNAME’.
*-----------------------------------------------------------

     CALL ”ODEFIN” USING CURSOR–2, ONE, DNAME,
           DNAME–L, ASC.
     IF C–RC IN CURSOR–2 NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–CLOSE.

*-----------------------------------------------------------
* ASK THE USER FOR EMPLOYEE NAME, JOB, SAL, AND DEPTNO.
*-----------------------------------------------------------

 NEXT–EMP.

     DISPLAY ASK–EMP WITH NO ADVANCING.
     ACCEPT ENAME.
     IF ENAME = ” ”
        GO TO EXIT–CLOSE.

     DISPLAY ASK–JOB WITH NO ADVANCING.
     ACCEPT JOB.

     DISPLAY ASK–SAL WITH NO ADVANCING.
     ACCEPT SAL.

 ASK–DPT.
     DISPLAY ASK–DEPTNO WITH NO ADVANCING.
     ACCEPT DEPTNO.
```

```
*-----------------------------------------------------------
* CHECK FOR A VALID DEPARTMENT NUMBER BY EXECUTING.
* THE SELECT STATEMENT.
*-----------------------------------------------------------


     CALL ”OEXEC” USING CURSOR–2.
     IF C–RC IN CURSOR–2 NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–CLOSE.

*-----------------------------------------------------------
* FETCH THE ROWS – DEPTNO IS A PRIMARY KEY SO A MAX.
* OF 1 ROW WILL BE FETCHED. IF CURSOR RETURN CODE IS 1403
* THEN NO SUCH DEPARTMENT EXISTS.
*-----------------------------------------------------------

     MOVE SPACES TO DNAME.
     CALL ”OFETCH” USING CURSOR–2.
     IF C–RC IN CURSOR–2 = 0 THEN GO TO ADD–ROW.
     IF C–RC IN CURSOR–2 = 1403
        DISPLAY ”No such department.”
        GO TO ASK–DPT.

*-----------------------------------------------------------
* INCREMENT EMPNO BY 10.
* EXECUTE THE INSERT STATEMENT.
*-----------------------------------------------------------

 ADD–ROW.
     ADD 10 TO EMPNO.
     IF EMPNO > 9999
        MOVE EMPNO TO EMPNO–D
        DISPLAY ”Employee number ” EMPNO–D ” too large.”
        GO TO EXIT–CLOSE.
     CALL ”OEXEC” USING CURSOR–1.
     IF C–RC IN CURSOR–1 = 0 THEN GO TO PRINT–RESULT.

*-----------------------------------------------------------
* IF THE RETURN CODE IS 1 (DUPLICATE VALUE IN INDEX),
* THEN GENERATE THE NEXT POSSIBLE EMPLOYEE NUMBER.
*-----------------------------------------------------------

     IF C–RC IN CURSOR–1 = 1
        ADD 10 TO EMPNO
        GO TO ADD–ROW
     ELSE
        PERFORM ORA–ERROR
        GO TO EXIT–CLOSE.
```

```
  PRINT–RESULT.
    MOVE EMPNO TO EMPNO–D.
    DISPLAY ENAME ” added to the ” DNAME
      ” department as employee number ” EMPNO–D.

*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
* THE ROW HAS BEEN ADDED – COMMIT THIS TRANSACTION.
*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
    CALL ”OCOM” USING LDA.
    IF LDA–RC NOT = 0
       PERFORM ORA–ERROR
       GO TO EXIT–CLOSE.
    GO TO NEXT–EMP.

*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
* CLOSE CURSORS AND LOG OFF.
*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

 EXIT–CLOSE.

    CALL ”OCLOSE” USING CURSOR–1.
    IF C–RC IN CURSOR–1 NOT = 0
       PERFORM ORA–ERROR.

    CALL ”OCLOSE” USING CURSOR–2.
    IF C–RC IN CURSOR–2 NOT = 0
       PERFORM ORA–ERROR.

 EXIT–LOGOF.

    CALL ”OLOGOF” USING LDA.
    IF LDA–RC NOT = 0
       PERFORM ORA–ERROR.

 EXIT–STOP.

    DISPLAY ”End of the OCIDEMO1 program.”
    STOP RUN.

*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
* DISPLAY ORACLE ERROR NOTICE.
*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

 ORA–ERROR.

    IF LDA–RC NOT = 0
       DISPLAY ”OLOGON error”
       MOVE LDA–RC TO ERR–RC
       MOVE ”0” TO ERR–FNC
```

```
     ELSE IF C–RC IN CURSOR–1 NOT = 0
        MOVE C–RC IN CURSOR–1 TO ERR–RC
        MOVE C–FNC IN CURSOR–1 TO ERR–FNC
     ELSE
        MOVE C–RC IN CURSOR–2 TO ERR–RC
        MOVE C–FNC IN CURSOR–2 TO ERR–FNC.
     DISPLAY ”ORACLE error” WITH NO ADVANCING.
     IF ERR–FNC NOT = 0
        MOVE ERR–FNC TO ERR–FNC–D
        DISPLAY ” processing OCI function”
             ERR–FNC–D ”.”
     ELSE
        DISPLAY ”.”.
     MOVE ” ” TO MSGBUF.
     CALL ”OERHMS” USING LDA, ERR–RC, MSGBUF, MSGBUF–L.
     DISPLAY MSGBUF.
```

CBDEM2.COB

```
* CBDEM2.COB
*
* The program CBDEM2 accepts SQL statements from the
* user at run time and processes them.
* If the statement was a Data Definition Language (DDL),
* Data Control Language (DCL), or Data Manipulation
* Language (DML) statement, it is parsed and executed,
* and the next statement is retrieved.  (Note that
* performing the execute step for a DDL or DCL statement
* is not necessary, but it does no harm, and simplifies
* the program logic.)
* If the statement was a query, the program describes
* the select list, and defines output variables of the
* appropriate type and size, depending on the internal
* datatype of the select–list item.
* Then, each row of the query is fetched, and the results
* are displayed.
* To keep the size of this example program to a
* reasonable limit for this book, the following
* restrictions are present:
* (1) The SQL statement can contain only 25 elements (words
*   and punctuation), and must be entered on a single line.
*   There is no terminating ’;’.
* (2) A maximum of 8 bind (input) variables is permitted.
*   Additional input variables are not bound, which will
*   cause an error at execute time.  Input values must be
*   enterable as character strings
*   (numeric or alphanumeric).
*   Placeholders for bind variables are :bv,
*   as for OBNDRV.
* (3) A maximum of 8 select–list items per table are
*   described and defined.  Additional columns are
*   not defined, which will cause unpredictable behavior
*   at fetch time.
* (4) Not all internal datatypes are handled for queries.
*   Selecting a RAW or LONG column could cause problems.

 IDENTIFICATION DIVISION.
 PROGRAM–ID.  CBDEM2.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING–STORAGE SECTION.

* Logon, cursor, and host data areas.
 01  LDA.
     02      LDA–V2RC    PIC S9(4) COMP.
     02      FILLER      PIC X(10).
     02      LDA–RC      PIC S9(4) COMP.
     02      FILLER      PIC X(50).
 01  CDA.
     02      C–V2RC      PIC S9(4) COMP.
     02      C–TYPE      PIC S9(4) COMP.
     02      C–ROWS      PIC S9(9) COMP.
     02      C–OFFS      PIC S9(4) COMP.
     02      C–FNC       PIC S9(4) COMP.
     02      C–RC        PIC S9(4) COMP.
     02      FILLER      PIC X(50).
 01  HDA                 PIC X(512).

* Error message variables for the OERHMS routine.
 01  MSGBUF              PIC X(256).
 01  MSGBUF–L            PIC S9(9) VALUE 256 COMP.
 01  ERR–FNC–D           PIC ZZZ.

* Connect info.  Link single–task, or modify to use
* SQL*Net connect string appropriate to your site.
 01  USER–ID             PIC X(5)  VALUE ”SCOTT”.
 01  USER–ID–L           PIC S9(9) VALUE 5 COMP.
 01  PSW                 PIC X(5)  VALUE ”TIGER”.
 01  PSW–L               PIC S9(9) VALUE 5 COMP.
 01  CONN                PIC S9(9) VALUE 0 COMP.
 01  CONN–L              PIC S9(9) VALUE 0 COMP.
 01  CONN–MODE           PIC S9(9) VALUE 0 COMP.
```

```
* Parameters for OPARSE.
 01  SQL-STMT            PIC X(132).
 01  SQLL                PIC S9(9) COMP.
 01  DEF-MODE            PIC S9(9) VALUE 1 COMP.
 01  NO-DEF-MODE         PIC S9(9) VALUE 0 COMP.
 01  V7-FLG              PIC S9(9) VALUE 2 COMP.

* Parameters for OBNDRV.
 01  BVNX.
     03  BV-NAME         OCCURS 25 TIMES.
         05 BV-NAMEX     OCCURS 10 TIMES PIC X.
 01  BVVX.
     03  BV-VAL          OCCURS 10 TIMES PIC X(10).
 01  BV-VAL-L            PIC S9(9) VALUE 10 COMP.
 01  N-BV                PIC S9(9) COMP.

* Parameters for ODESCR.  Note: some are two bytes (S9(4))
* some are four bytes (S9(9)).
 01  DBSIZEX.
     03  DBSIZE          OCCURS 8 TIMES PIC S9(9) COMP.
 01  DBTYPEX.
     03  DBTYPE          OCCURS 8 TIMES PIC S9(4) COMP.
 01  NAMEX.
     03  DBNAME            OCCURS 8 TIMES PIC X(10).
 01  NAME-LX.
     03  NAME-L          OCCURS 8 TIMES PIC S9(9) COMP.
 01  DSIZEX.
     03  DSIZE           OCCURS 8 TIMES PIC S9(9) COMP.
 01  PRECX.
     03  PREC            OCCURS 8 TIMES PIC S9(4) COMP.
 01  SCALEX.
     03  SCALE           OCCURS 8 TIMES PIC S9(4) COMP.
 01  NULL-OKX.
     03  NULL-OK         OCCURS 8 TIMES PIC S9(4) COMP.

* Parameters for ODEFIN.
 01  OV-CHARX.
     03  OV-CHAR         OCCURS 8 TIMES PIC X(10).
 01  OV-NUMX.
     03  OV-NUM          OCCURS 8 TIMES
                            PIC S99999V99 COMP-3.
 01  INDPX.
     03  INDP            OCCURS 8 TIMES PIC S9(4) COMP.
 01  N-OV                PIC S9(9) COMP.
 01  N-ROWS              PIC S9(9) COMP.
 01  N-ROWS-D            PIC ZZZ9 DISPLAY.
 01  OV-CHAR-L           PIC S9(9) VALUE 10 COMP.
 01  SEVEN               PIC S9(9) VALUE 7 COMP.
 01  PACKED-DEC-L        PIC S9(9) VALUE 4 COMP.
 01  PACKED–DEC–T        PIC S9(9) VALUE 7 COMP.
 01  NUM–DISP            PIC ZZZZZ.ZZ.
 01  FMT                 PIC X(6) VALUE ”08.+02”.
 01  FMT–L               PIC S9(9) VALUE 6 COMP.
 01  FMT–NONE            PIC X(6).

* Miscellaneous parameters.
 01  ZERO–A              PIC S9(9) VALUE 0 COMP.
 01  ZERO–B              PIC S9(9) VALUE 0 COMP.
 01  ZERO–C              PIC S9(4) VALUE 0 COMP.
 01  ONE                 PIC S9(9) VALUE 1 COMP.
 01  TWO                 PIC S9(9) VALUE 2 COMP.
 01  FOUR                PIC S9(9) VALUE 4 COMP.
 01  INDX                PIC S9(9) COMP.
 01  NAME–D8             PIC X(8).
 01  NAME–D10            PIC X(10).
 01  VARCHAR2–T          PIC S9(9) VALUE 1 COMP.
 01  NUMBER–T            PIC S9(9) VALUE 2 COMP.
 01  INTEGER–T           PIC S9(9) VALUE 3 COMP.
 01  DATE–T              PIC S9(9) VALUE 12 COMP.
 01  CHAR–T              PIC S9(9) VALUE 96 COMP.

 PROCEDURE DIVISION.
 BEGIN.

* Connect to ORACLE in non–blocking mode.
* HDA must be initialized to all zeros before call to OLOG.
     MOVE LOW–VALUES TO HDA.

     CALL ”OLOG” USING LDA, HDA, USER–ID, USER–ID–L,
           PSW, PSW–L, CONN, CONN–L, CONN–MODE.

* Check for error, perform error routine if required.
     IF LDA–RC NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–STOP.
     DISPLAY ”Logged on to ORACLE as user ” USER–ID ”.”.
     DISPLAY ”Type EXIT at SQL prompt to quit.”

* Open a cursor.  Only the first two parameters are
* used, the remainder (for V2 compatibility) are ignored.
     CALL ”OOPEN” USING CDA, LDA, USER–ID, ZERO–A,
           ZERO–A, USER–ID, ZERO–A.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
```

```
* Process each SQL statement.
 STMT–LOOP.
     PERFORM DO–SQL–STMT.
     GO TO STMT–LOOP.
 EXIT–CLOSE.
     CALL ”OCLOSE” USING CDA.
 EXIT–LOGOFF.
     CALL ”OLOGOF” USING LDA.
 EXIT–STOP.
     STOP RUN.

* Perform paragraphs.
 DO–SQL–STMT.
     MOVE ” ” TO SQL–STMT.
     DISPLAY ” ”.
     DISPLAY ”SQL > ” NO ADVANCING.
     ACCEPT SQL–STMT.

* Get first word of statement.
     UNSTRING SQL–STMT DELIMITED BY ALL ” ”
              INTO BV–NAME(1).
     IF (BV–NAME(1) = ”exit” OR BV–NAME(1) = ”EXIT”)
        GO TO EXIT–CLOSE.
     MOVE 132 TO SQLL.

* Use non–deferred parse, to catch syntax errors
* right after the parse.
     CALL ”OPARSE” USING CDA, SQL–STMT, SQLL,
           NO–DEF–MODE, V7–FLG.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO DO–SQL–STMT.
     PERFORM BIND–VARS.
     DISPLAY ” ”.
     MOVE N–BV TO ERR–FNC–D.
     DISPLAY ”There were” ERR–FNC–D
             ” bind variables.”.

* Execute the statement.
     CALL ”OEXN” USING CDA, ONE, ZERO–B.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO DO–SQL–STMT.

* Describe the SQL statement, and define output
* variables if it is a query.  Limit output variables
* to eight.
     PERFORM DESCRIBE–DEFINE THRU DESCRIBE–DEFINE–EXIT.
     SUBTRACT 1 FROM N–OV.
```

```
          IF (N–OV > 0)
              MOVE N–OV TO ERR–FNC–D
              DISPLAY ”There were” ERR–FNC–D
                      ” define variables.”
              DISPLAY ” ”
              PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > N–OV
                 IF (DBTYPE(INDX) NOT = 2)
                    MOVE DBNAME(INDX) TO NAME–D10
                    DISPLAY NAME–D10 NO ADVANCING
                 ELSE
                    MOVE DBNAME(INDX) TO NAME–D8
                    DISPLAY NAME–D8 NO ADVANCING
                 END–IF
                 DISPLAY ” ” NO ADVANCING
              END–PERFORM
              DISPLAY ” ”
              PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > N–OV
                 DISPLAY ”––––––––” NO ADVANCING
                 IF DBTYPE(INDX) NOT = 2
                    DISPLAY ”––” NO ADVANCING
                 END–IF
                 DISPLAY ” ” NO ADVANCING
              END–PERFORM
              DISPLAY ” ”
          END–IF.

     * If the statement was a query, fetch the rows and
     * display them.
          IF (C–TYPE IN CDA = 4)
             PERFORM FETCHN THRU FETCHN–EXIT
             MOVE N–ROWS TO N–ROWS–D
             DISPLAY ” ”
             DISPLAY N–ROWS–D ” rows returned.”.
     * End of DO–SQL–STMT.

      BIND–VARS.
          MOVE 0 TO N–BV.
          PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > 25
            MOVE ” ” TO BV–NAME(INDX)
          END–PERFORM.
          UNSTRING SQL–STMT
            DELIMITED BY ”(” OR ”,” OR ”;” OR ”=”
                      OR ”)” OR ALL ” ”
              INTO BV–NAME(1)
                   BV–NAME(2)
                   BV–NAME(3)
                   BV–NAME(4)
                   BV–NAME(5)
                   BV–NAME(6)
                   BV–NAME(7)
                   BV–NAME(8)
                   BV–NAME(9)
                   BV–NAME(10)
                   BV–NAME(11)
                   BV–NAME(12)
                   BV–NAME(13)
                   BV–NAME(14)
                   BV–NAME(15)
                   BV–NAME(16)
                   BV–NAME(17)
                   BV–NAME(18)
                   BV–NAME(19)
                   BV–NAME(20)
                   BV–NAME(21)
                   BV–NAME(22)
                   BV–NAME(23)
                   BV–NAME(24)
                   BV–NAME(25).

     * Scan the words in the SQL statement.  If the
     * word begins with ’:’, it is a placeholder for
     * a bind variable.  Get a value for it (as a string)
     * and bind using the OBNDRV routine, datatype 1.

          MOVE 0 TO INDP(1).
          PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > 25
             IF BV–NAMEX(INDX,1) = ’:’
                ADD 1 TO N–BV
                MOVE 0 TO SQLL
                INSPECT BV–NAME(INDX) TALLYING SQLL
                   FOR CHARACTERS BEFORE INITIAL ’ ’
                DISPLAY ”Enter value for ” BV–NAME(INDX) ” ––> ”
                   NO ADVANCING
                ACCEPT BV–VAL(N–BV)
                CALL ”OBNDRV” USING CDA, BV–NAME(INDX), SQLL,
                     BV–VAL(N–BV), BV–VAL–L, VARCHAR2–T,
                     ZERO–A, INDP(1), FMT–NONE, ZERO–A, ZERO–A
                IF C–RC IN CDA NOT = 0
                   PERFORM ORA–ERROR
                   GO TO EXIT–CLOSE
                ELSE
                   DISPLAY ”Bound ” BV–VAL(N–BV)
                END–IF
             END–IF
          END–PERFORM.
      DESCRIBE–DEFINE.
          MOVE 0 TO N–OV.
          PERFORM 9 TIMES
```

```
            ADD 1 TO N–OV
            IF (N–OV > 8)
               GO TO DESCRIBE–DEFINE–EXIT
            END–IF
            MOVE 10 TO NAME–L(N–OV)
            MOVE ” ” TO DBNAME(N–OV)

            CALL ”ODESCR” USING CDA, N–OV, DBSIZE(N–OV),
                  DBTYPE(N–OV),
                  DBNAME(N–OV), NAME–L(N–OV), DSIZE(N–OV),
                  PREC(N–OV), SCALE(N–OV), NULL–OK(N–OV)

* Check for end of select list.
            IF (C–RC IN CDA = 1007)
               GO TO DESCRIBE–DEFINE–EXIT
            END–IF

* Check for error.
            IF (C–RC IN CDA NOT = 0)
               PERFORM ORA–ERROR
               GO TO DESCRIBE–DEFINE–EXIT
            END–IF

* Define an output variable for the select–list item.
* If it is a number, define a packed decimal variable,
* and create a format string for it.
            IF (DBTYPE(N–OV) = 2)
               CALL ”ODEFIN” USING CDA, N–OV, OV–NUM(N–OV),
                    PACKED–DEC–L, PACKED–DEC–T, TWO,
                    INDP(N–OV), FMT, FMT–L, PACKED–DEC–T,
                    ZERO–C, ZERO–C
            ELSE

* For all other types, convert to a VARCHAR2 of length 10.
               CALL ”ODEFIN” USING CDA, N–OV, OV–CHAR(N–OV),
                    OV–CHAR–L, VARCHAR2–T, ZERO–A, INDP(N–OV),
                    FMT, ZERO–A, ZERO–A, ZERO–C, ZERO–C
            END–IF
            IF (C–RC IN CDA NOT = 0)
               PERFORM ORA–ERROR
               GO TO DESCRIBE–DEFINE–EXIT
            END–IF
        END–PERFORM.
 DESCRIBE–DEFINE–EXIT.
 FETCHN.

        MOVE 0 TO N–ROWS.
        PERFORM 10000 TIMES
```

```
* Clear any existing values from storage buffers
        MOVE SPACES TO OV–CHARX
        MOVE LOW–VALUES TO OV–NUMX

        CALL ”OFETCH” USING CDA

* Check for end of fetch (”no data found”)
        IF C–RC IN CDA = 1403
           GO TO FETCHN–EXIT
         END–IF
        IF C–RC IN CDA NOT = 0
           PERFORM ORA–ERROR
           GO TO FETCHN–EXIT
        END–IF
        ADD 1 TO N–ROWS
        PERFORM VARYING INDX FROM 1
                BY 1 UNTIL INDX > N–OV
           IF (DBTYPE(INDX) = 2)
              MOVE OV–NUM(INDX) TO NUM–DISP
              INSPECT NUM–DISP REPLACING ALL ”.00” BY ”   ”
              DISPLAY NUM–DISP NO ADVANCING
           ELSE
              DISPLAY OV–CHAR(INDX) NO ADVANCING
           END–IF
           DISPLAY ” ” NO ADVANCING
        END–PERFORM
        DISPLAY ” ”
     END–PERFORM.
     DISPLAY ”LEAVING FETCHN...”.
 FETCHN–EXIT.

* Report an error.  Obtain the error message
* text using the OERHMS routine.
 ORA–ERROR.
     IF LDA–RC IN LDA NOT = 0
        DISPLAY ”OLOGON error”
        MOVE 0 TO C–FNC IN CDA
        MOVE LDA–RC IN LDA TO C–RC IN CDA.
     DISPLAY ”ORACLE error ” NO ADVANCING.
     IF C–FNC NOT = 0
        DISPLAY ”processing OCI function” NO ADVANCING
        MOVE C–FNC IN CDA TO ERR–FNC–D
        DISPLAY ERR–FNC–D
     ELSE
        DISPLAY ”:”.

     MOVE ” ” TO MSGBUF.
     CALL ”OERHMS” USING LDA, C–RC IN CDA, MSGBUF,MSGBUF–L.
     DISPLAY MSGBUF.
```

CBDEM3.COB

```
*  The program CBDEM3 creates a table called
*  ”VOICE_MAIL” that contains three fields:
*  a message ID, and message length, and a LONG RAW
*  column that contains a digitized voice
*  message.  The program fills one row of the table with a
*  (simulated) message, then plays the message by
*  extracting 64 kB chunks of it using the OFLNG routine,
*  and sending them to a (simulated) digital–to–analog
*  (DAC) converter routine.

 IDENTIFICATION DIVISION.
 PROGRAM–ID.  CBDEM3.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING–STORAGE SECTION.
 01  LDA.
     02      LDA–V2RC    PIC S9(4) COMP.
     02      FILLER      PIC X(10).
     02      LDA–RC      PIC S9(4) COMP.
     02      FILLER      PIC X(50).
 01  CDA.
     02      C–V2RC      PIC S9(4) COMP.
     02      C–TYPE      PIC S9(4) COMP.
     02      C–ROWS      PIC S9(9) COMP.
     02      C–OFFS      PIC S9(4) COMP.
     02      C–FNC       PIC S9(4) COMP.
     02      C–RC        PIC S9(4) COMP.
     02      FILLER      PIC X(50).
 01  HDA                 PIC X(512).

 01  ERRMSG              PIC X(256).
 01  ERRMSG–L            PIC S9(9) VALUE 256 COMP.
 01  ERR–RC              PIC S9(9) COMP.
 01  ERR–FNC–D           PIC ZZ9.
 01  USER–ID             PIC X(5)  VALUE ”SCOTT”.
 01  USER–ID–L           PIC S9(9) VALUE 5 COMP.
 01  PSW                 PIC X(5)  VALUE ”tiger”.
 01  PSW–L               PIC S9(9) VALUE 5 COMP.
 01  CONN                PIC S9(9) VALUE 0 COMP.
 01  CONN–L              PIC S9(9) VALUE 0 COMP.
 01  CONN–MODE           PIC S9(9) VALUE 0 COMP.
 01  SQL–STMT            PIC X(132).
 01  SQLL                PIC S9(9) COMP.
 01  ZERO–A              PIC S9(9) VALUE 0 COMP.
 01  ZERO–B              PIC S9(9) VALUE 0 COMP.
 01  FMT                 PIC X(6).
```

```
*  Establish a 200000 byte buffer.  (On most systems,
*  including the VAX, a PIC 99 reserves one byte.)
 01  MSGX.
     02 MSG               OCCURS 200000 TIMES PIC 99.
 01  MSGX–L               PIC S9(9) VALUE 200000 COMP.
 01  MSG–L                PIC S9(9) COMP.
 01  MSG–L–D              PIC ZZZZZZ.
 01  MSG–ID               PIC S9(9) COMP.
 01  MSG–ID–L             PIC S9(9) VALUE 4 COMP.
 01  MSG–ID–D             PIC ZZZZ.
 01  LEN                  PIC 9(9) COMP.
 01  LEN–D                PIC ZZZZ9.
 01  INDX                 PIC S9(9) COMP.
 01  INTEGER–T            PIC S9(9) VALUE 3 COMP.
 01  DEF–MODE             PIC S9(9) VALUE 1 COMP.
 01  LONG–RAW             PIC S9(9) VALUE 24 COMP.
 01  ONE                  PIC S9(9) VALUE 1 COMP.
 01  TWO                  PIC S9(9) VALUE 2 COMP.
 01  THREE                PIC S9(9) VALUE 3 COMP.
 01  ANSX.
     02      ANSWER       OCCURS 6 TIMES PIC X.
 01  VERSION–7            PIC S9(9) VALUE 2 COMP.
 01  INDP                 PIC S9(4) COMP.
 01  RCODE                PIC S9(4) COMP.
 01  RLEN                 PIC S9(4) COMP.
 01  RETL                 PIC S9(9) COMP.
 01  OFF1                 PIC S9(9) COMP.

 PROCEDURE DIVISION.
 BEGIN.

*  Connect to ORACLE in non–blocking mode.
*  HDA must be initialized to all zeros before call to OLOG.
     MOVE LOW–VALUES TO HDA.

     CALL ”OLOG” USING LDA, HDA, USER–ID, USER–ID–L,
           PSW, PSW–L, CONN, CONN–L, CONN–MODE.
     IF LDA–RC NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–STOP.
     DISPLAY ”Logged on to ORACLE as user ”, USER–ID.

*  Open a cursor.
     CALL ”OOPEN” USING CDA, LDA, USER–ID, ZERO–A,
           ZERO–A, USER–ID, ZERO–A.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
```

```
*  Drop the VOICE_MAIL table.
     DISPLAY ”OK to drop VOICE_MAIL table (Y or N)? : ”
–  WITH NO ADVANCING.
     ACCEPT ANSX.
     IF (ANSWER(1) NOT = ’y’ AND ANSWER(1) NOT = ’Y’)
        DISPLAY ”Exiting program now.”
        GO TO EXIT–CLOSE.
     MOVE ”DROP TABLE VOICE_MAIL” TO SQL–STMT.
     MOVE 132 TO SQLL.

*  Call OPARSE with no deferred parse to execute the DDL
*  statement immediately.
     CALL ”OPARSE” USING CDA, SQL–STMT, SQLL,
          ZERO–A, VERSION–7.
     IF C–RC IN CDA NOT = 0
        IF (C–RC IN CDA = 942)
           DISPLAY ”Table did not exist.”
        ELSE
           PERFORM ORA–ERROR
           GO TO EXIT–LOGOFF
        END–IF
     ELSE
        DISPLAY ”Table dropped.”
     END–IF

*  Create the VOICE_MAIL table anew.
     MOVE ”CREATE TABLE VOICE_MAIL (MSG_ID NUMBER(6),
–    ”MSG_LEN NUMBER(12), MSG LONG RAW)” TO SQL–STMT.
     MOVE 132 TO SQLL.

*  Non–deferred parse to execute the DDL SQL statement.
     DISPLAY ”Table VOICE_MAIL ” NO ADVANCING.
     CALL ”OPARSE” USING CDA, SQL–STMT, SQLL,
          ZERO–A, VERSION–7.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
     DISPLAY ”created.”.

*  Insert some data into the table.
     MOVE ”INSERT INTO VOICE_MAIL VALUES (:1, :2, :3)”
          TO SQL–STMT.
     MOVE 132 TO SQLL.
     CALL ”OPARSE” USING CDA, SQL–STMT, SQLL,
          ZERO–A, VERSION–7.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
```

```
*  Bind the inputs.
     MOVE 0 TO INDP.
     CALL ”OBNDRN” USING CDA, ONE, MSG–ID, MSG–ID–L,
     INTEGER–T, ZERO–A, INDP, FMT, ZERO–A, ZERO–A.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
     CALL ”OBNDRN” USING CDA, TWO, MSG–L, MSG–ID–L,
          INTEGER–T, ZERO–A, INDP, FMT, ZERO–A, ZERO–A.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
     CALL ”OBNDRN” USING CDA, THREE, MSGX, MSGX–L,
          LONG–RAW, ZERO–A, INDP, FMT, ZERO–A, ZERO–A.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.

*  Set input variables, then execute the INSERT statement.
     MOVE 100 TO MSG–ID.
     MOVE 200000 TO MSG–L.
     PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > MSG–L
        MOVE 42 TO MSG(INDX)
     END–PERFORM.
     CALL ”OEXN” USING CDA, ONE, ZERO–B.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
     MOVE ”SELECT MSG_ID, MSG_LEN, MSG FROM VOICE_MAIL
–    ” WHERE MSG_ID = 100” TO SQL–STMT.

*  Call OPARSE in deferred mode to select a message.
     CALL ”OPARSE” USING CDA, SQL–STMT, SQLL,
          DEF–MODE, VERSION–7.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.

*  Define the output variables.
     CALL ”ODEFIN” USING CDA, ONE, MSG–ID,
          MSG–ID–L, INTEGER–T, ZERO–A, ZERO–A, ZERO–A,
          ZERO–A, ZERO–A, ZERO–A, ZERO–A.
     IF C–RC IN CDA NOT = 0
        PERFORM ORA–ERROR
        GO TO EXIT–LOGOFF.
     CALL ”ODEFIN” USING CDA, TWO, MSG–L,
          MSG–ID–L, INTEGER–T, ZERO–A, ZERO–A, ZERO–A,
          ZERO–A, ZERO–A, ZERO–A, ZERO–A.
     IF C–RC IN CDA NOT = 0
```

```
              PERFORM ORA–ERROR
              GO TO EXIT–LOGOFF.
           MOVE 100 TO MSG–ID–L.


           CALL ”ODEFIN” USING CDA, THREE, MSGX,
                MSG–ID–L, LONG–RAW, ZERO–A, INDP, ANSX, ZERO–A, ZERO–A,
                RLEN, RCODE.
           IF C–RC IN CDA NOT = 0
              PERFORM ORA–ERROR
              GO TO EXIT–LOGOFF.

      *  Do the query, getting the message ID and just the first
      *  100 bytes of the message.  This query basically just sets
      *  the cursor to the right row.  The message contents are
      *  fetched by the OFLNG routine.
           CALL ”OEXFET” USING CDA, ONE, ZERO–A, ZERO–A.
           IF C–RC IN CDA NOT = 0
              PERFORM ORA–ERROR
              GO TO EXIT–LOGOFF.
           MOVE MSG–ID TO MSG–ID–D.
           DISPLAY ” ”.
           DISPLAY ”Message ” MSG–ID–D ” is available.”.
           MOVE MSG–L TO MSG–L–D.
           DISPLAY ”The length is ” MSG–L–D ” bytes.”.
           PERFORM VARYING OFF1 FROM 0 BY 65536
                 UNTIL MSG–L <= 0
              IF (MSG–L < 65536)
                 MOVE MSG–L TO LEN
              ELSE
                 MOVE 65536 TO LEN
              END–IF
              PERFORM PLAY–MSG THRU PLAY–MSG–EXIT
              SUBTRACT LEN FROM MSG–L
      *        IF (MSG–L < 0 OR MSG–L = 0)
      *           GO TO END–LOOP
      *        END–IF
           END–PERFORM.
       END–LOOP.
           DISPLAY ” ”.
           DISPLAY ”End of message.”.
       EXIT–CLOSE.
           CALL ”OCLOSE” USING CDA.
       EXIT–LOGOFF.
           CALL ”OLOGOF” USING LDA.
       EXIT–STOP.
           STOP RUN.
       PLAY–MSG.
           MOVE LEN TO LEN–D.
```

```
     DISPLAY ”Playing ” LEN–D ” bytes.”.
 PLAY–MSG–EXIT.



* Report an error.  Obtain the error message
* text using the OERHMS routine.
 ORA–ERROR.
     IF LDA–RC IN LDA NOT = 0
        DISPLAY ”OLOGON error”
        MOVE 0 TO C–FNC IN CDA
        MOVE LDA–RC IN LDA TO C–RC IN CDA.
     DISPLAY ”ORACLE error” NO ADVANCING.
     IF C–FNC NOT = 0
        DISPLAY ” processing OCI function ” NO ADVANCING
        MOVE C–FNC IN CDA TO ERR–FNC–D
        DISPLAY ERR–FNC–D
     ELSE
        DISPLAY ”.”.

     MOVE ” ” TO ERRMSG.
     CALL ”OERHMS” USING LDA, C–RC IN CDA, ERRMSG, ERRMSG–L.
     DISPLAY ERRMSG.
```

APPENDIX

C

Sample Programs in FORTRAN

This appendix contains three sample OCI programs written in FORTRAN. The first adds a new employee to a database, the second processes dynamic SQL statements, and the third fetches a portion of a LONG or LONG RAW column using OFLNG.

Each of these sample programs is available online. The exact name and storage location of these progams is system dependent. Refer to your Oracle installation or user’s guide for details.

This appendix contains listings for the following files:

- FDEMO1.FOR
- FDEMO2.FOR
- FDEMO3.FOR

FDEMO1.FOR

```
      PROGRAM FDEMO1

*--------------------------------------------------------------
*  FDEMO1 is a demonstration program that adds new employee
*  rows to the personnel data base.  Checking
*  is done to insure the integrity of the data base.
*  The employee numbers are automatically selected using
*  the current maximum employee number as the start.
*  If any employee number is a duplicate, it is skipped.
*  The program queries the user for data as follows:
*
*    Enter employee name  :
*    Enter employee job   :
*    Enter employee salary:
*    Enter employee dept  :
*
*  If just <cr> is entered for the employee name,
*  the program terminates.
*
*  If the row is successfully inserted, the following
*  is printed:
*
*  ENAME added to DNAME department as employee N.
*
*  The maximum lengths of the ’ename’, ’job’, and ’dname’
*  columns are determined by an ODESCR call.
*
*  Note: VAX FORTRAN, by default, passes all CHARACTER variables
*  (variables declared as CHARACTER*N) by descriptor.
*  To compile this program on systems that pass character
*  variables by descriptor, insert %REF() where necessary.
*--------------------------------------------------------------

      IMPLICIT INTEGER (A–Z)

      INTEGER*2         LDA(32)
      INTEGER*2         CURS(32,2)
      INTEGER*2         HDA(256)
      CHARACTER*20      UID, PSW
      INTEGER*4         NOBLOK

*  CHARACTER string vars to hold the SQL statements

      CHARACTER*60      SMAX
      CHARACTER*60      SEMP
      CHARACTER*150     INS
      CHARACTER*60      SEL
      INTEGER*4         SMAXL, SEMPL, INSL, SELL

*  Program vars to be bound to SQL placeholders and
*  select–list fields.

      INTEGER*4         EMPNO, DEPTNO, SAL
      CHARACTER*10      ENAME
      CHARACTER*10      JOB
      CHARACTER*14      DNAME

*  Actual lengths of columns.

      INTEGER*4         ENAMES, JOBS, DNAMES

*  Character strings for SQL placeholders.

      CHARACTER*6       ENON
      CHARACTER*6       ENAN
      CHARACTER*4       JOBN
      CHARACTER*4       SALN
      CHARACTER*7       DEPTN

*  Lengths of character strings for SQL placeholders.

      INTEGER*4         ENONL, ENANL, JOBNL, SALNL, DEPTNL

*  Parameters for OPARSE.

      INTEGER*4         NODEFP, V7FLAG

*  Parameters for ODESCR.

      INTEGER*2     DTYPE, PREC, SCALE, NULLOK
      INTEGER*4     DSIZE, CNAMEL
      CHARACTER*80  CNAME

*---------------------------------------------------------------
*  Initialize variables.
*---------------------------------------------------------------

      SMAX = ’SELECT NVL(MAX(EMPNO),0) FROM EMP’
      SMAXL = LEN_TRIM(SMAX)

      SEMP = ’SELECT ENAME,JOB FROM EMP’
      SEMPL=  LEN_TRIM(SEMP)

      INS = ’INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,
     + DEPTNO) VALUES (:EMPNO,:ENAME,:JOB,:SAL,:DEPTNO)’
      INSL = LEN_TRIM(INS)
      SEL = ’SELECT DNAME FROM DEPT WHERE DEPTNO = :1’
      SELL = LEN_TRIM(SEL)

*  All in Deferred Mode

      NODEFP = 1
      V7FLAG = 2
      ENAMEL = 10
      JOBL   = 10
      EMPNOL = 4
      DEPTL  = 4
      SALL   = 4

      ENON = ’:EMPNO’
      ENAN = ’:ENAME’
      JOBN = ’:JOB’
      SALN = ’:SAL’
      DEPTN = ’:DEPTNO’

      ENONL = 6
      ENANL = 6
      JOBNL = 4
      SALNL = 4
      DEPTNL = 7

*---------------------------------------------------------------
*  Connect to ORACLE in non–blocking mode.
*  HDA must be initialized to all zeros before call to OLOG.
*---------------------------------------------------------------

      UID = ’SCOTT’
      PSW = ’TIGER’
      NOBLOK = 0
      DATA HDA/256*0/

      CALL OLOG(LDA, HDA, UID, LEN_TRIM(UID),
     +           PSW, LEN_TRIM(PSW), 0, –1, NOBLOK)
      IF (LDA(7).NE.0) THEN
          CALL ERRRPT(LDA(1), LDA(1))
          GO TO 900
      END IF

     WRITE (*, ’(1X, A, A20)’) ’Logged on to ORACLE as user ’,
     +      UID
```

```
*---------------------------------------------------------------
*  Open two cursors for the personnel data base.
*---------------------------------------------------------------

      CALL OOPEN(CURS(1,1), LDA, 0, –1, –1, 0, –1)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      CALL OOPEN(CURS(1,2),LDA(1),0, –1, –1, 0, –1)
      IF (CURS(1,2).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,2))
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Turn off auto–commit.  Note: the default is off,
*  so this could be omitted.
*---------------------------------------------------------------

      CALL OCOF(LDA(1))
      IF (LDA(1).NE.0) THEN
        CALL ERRRPT(LDA(1), LDA(1))
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Retrieve the current maximum employee number.
*----------------------------------------------------------------

*  Parse the SQL statement.

      CALL OPARSE(CURS(1,1), SMAX, SMAXL, NODEFP, V7FLAG)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

*  Define a buffer to receive the MAX(EMPNO) from ORACLE.

      CALL ODEFIN(CURS(1,1), 1, EMPNO, 4, 3, –1, 0, 0,
+                  –1, –1, 0, 0)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF
```

```
*  Execute the SQL statement.

      CALL OEXEC(CURS(1,1))
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

*  Fetch the data from ORACLE into the defined buffer.

      CALL OFETCH(CURS(1,1))
      IF (CURS(1,1).EQ.0) GO TO 50
      IF (CURS(7,1).NE.1403) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

*  A cursor return code of 1403 means that no row
*  satisfied the query, so generate the first empno.

      EMPNO=10
50    CONTINUE

*---------------------------------------------------------------
*  Determine the max length of the employee name and job title.
*  Parse the SQL statement – it will not be executed.
*  Describe the two fields specified in the SQL statement.
*----------------------------------------------------------------

      CALL OPARSE(CURS(1,1), SEMP, SEMPL, NODEFP, V7FLAG)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      CNAMEL = 80
      CALL ODESCR(CURS(1,1), 1, ENAMES, DTYPE, CNAME,
     +            CNAMEL, DSIZE, PREC, SCALE, NULLOK)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      IF (ENAMES .GT. ENAMEL) THEN
        WRITE (*, ’(1X, A, I2, A, I2)’) ’ENAME too large (’,
     +   ENAMES, ’ for buffer (’, ENAMEL, ’).’
        GO TO 700
      END IF
      CNAMEL = 80
      CALL ODESCR(CURS(1,1), 2, JOBS, DTYPE, CNAME,
     +            CNAMEL, DSIZE, PREC, SCALE, NULLOK)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      IF (JOBS .GT. JOBL) THEN
        WRITE (*, '(1X, A, I2, A, I2)') 'JOB too large (',
     +   JOBS, ' for buffer (', JOBL, ').'
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Parse the insert and select statements.
*---------------------------------------------------------------

      CALL OPARSE(CURS(1,1), INS, INSL, NODEFP, V7FLAG)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      CALL OPARSE(CURS(1,2), SEL, SELL, NODEFP, V7FLAG)
      IF (CURS(1,2).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,2))
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Bind all placeholders.
*---------------------------------------------------------------

      CALL OBNDRV(CURS(1,1),ENON,LEN(ENON),EMPNO,EMPNOL,3,-1,
+                 0,0,-1,-1)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      CALL OBNDRV(CURS(1,1),ENAN,ENANL,ENAME,ENAMEL,1,-1,
+                 0,0,-1,-1)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF
      CALL OBNDRV(CURS(1,1),JOBN,JOBNL,JOB,JOBL,1,–1,0,0,–1,–1)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700

      END IF
      CALL OBNDRV(CURS(1,1),SALN,SALNL,SAL,SALL,3,–1,0,0,–1,–1)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      CALL OBNDRV(CURS(1,1),DEPTN,DEPTNL,DEPTNO,DEPTL,3,–1,
+                 0,0,–1,–1)
      IF (CURS(1,1).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Bind the DEPTNO variable.
*---------------------------------------------------------------

      CALL OBNDRN(CURS(1,2), 1, DEPTNO, DEPTL, 3,–1,0,0,–1,–1)
      IF (CURS(1,2).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,2))
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Describe the DNAME column – get the name and length.
*---------------------------------------------------------------

      DNAMEL = 14
      CALL ODESCR(CURS(1,1), 1, DNAMES, DTYPE, DNAME,
     +            DNAMEL, DSIZE, PREC, SCALE, NULLOK)
      IF (CURS(1,2).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,2))
        GO TO 700
      END IF

      IF (DNAMES .GT. DNAMEL) THEN
        WRITE (*, ’(1X, A)’) ’DNAME too large for buffer.’
        GO TO 700
      END IF
```

```
*---------------------------------------------------------------
*  Define the buffer to receive DNAME.
*---------------------------------------------------------------

      CALL ODEFIN(CURS(1,2),1,DNAME,DNAMEL,1,-1,0,0,-1,-1,0,0)
      IF (CURS(1,2).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,2))
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Read the user's input.  Statement 100
*  starts the main program loop.
*---------------------------------------------------------------

100   WRITE (*, '( A)')
     +  'Enter employee name (CR to QUIT) : '
      READ (*, '(A)'), ENAME
      IF (LEN_TRIM(ENAME) .EQ. 0) GO TO 700
      WRITE (*, '( A)'), 'Enter employee job   : '
      READ (*, '(A)'), JOB
      WRITE (*, '( A)') 'Enter employee salary: '
      READ (*, '(I6)'), SAL
300   WRITE (*, '( A)') 'Enter employee dept  : '
      READ (*, '(I6)'), DEPTNO

*---------------------------------------------------------------
*  Check for a valid department number by
*  executing the SELECT statement.
*---------------------------------------------------------------

      CALL OEXEC(CURS(1,2))
      IF (CURS(1,2).NE.0) THEN
        CALL ERRRPT(LDA(1), CURS(1,2))
        GO TO 700
      END IF

*---------------------------------------------------------------
*  Fetch the rows - DEPTNO is a primary key, so a max of
*  one row will be fetched.
*  If the return code is 1403 no such department exists.
*---------------------------------------------------------------

      CALL OFETCH(CURS(1,2))
      IF (CURS(1,2).EQ.0) GO TO 500
      IF (CURS(7,2).NE.1403) THEN
        CALL ERRRPT(LDA(1), CURS(1,2))
        GO TO 700
      END IF
      WRITE (*, '(1X, A)') 'No such department number'
      GO TO 300

*--------------------------------------------------------------
*  Increment EMPNO by 10.
*  Execute the insert statement.
*--------------------------------------------------------------

500   EMPNO = EMPNO + 10
      CALL OEXEC(CURS(1,1))
      IF (CURS(1,1).EQ.0) GO TO 600

*--------------------------------------------------------------
*  If the call returns code 1 (duplicate value in index),
*  generate the next possible employee number.
*--------------------------------------------------------------

      IF (CURS(7,1).NE.1) THEN
        CALL ERRRPT(LDA(1), CURS(1,1))
        GO TO 700
      END IF

      EMPNO=EMPNO+10
      GO TO 500

600   WRITE (*, 610) ENAME, DNAME, EMPNO
610   FORMAT(/, 1X, A10, ' added to the ', A14,
     +     ' department as employee# ', I4, /)

*--------------------------------------------------------------
*  The row has been added – commit this transaction.
*--------------------------------------------------------------

      CALL OCOM(LDA(1))
      IF (LDA(1).NE.0) THEN
        CALL ERRRPT(LDA(1), LDA(1))
        GO TO 700
      END IF

      GO TO 100

*--------------------------------------------------------------
*  Either a fatal error has occurred or the user typed
*  <CR> for the employee name.
*  Close the cursors, disconnect, and end the program.
*--------------------------------------------------------------

700   CONTINUE
      CALL OCLOSE(CURS(1,1))
      IF (CURS(1,1).NE.0) CALL ERRRPT(LDA(1), CURS(1,1))
      CALL OCLOSE(CURS(1,2))
      IF (CURS(1,2).NE.0) CALL ERRRPT(LDA(1), CURS(1,2))

      CALL OLOGOF(LDA(1))
      IF (LDA(1).NE.0) CALL ERRRPT(LDA(1), LDA(1))
900   STOP
      END

*---------------------------------------------------------------
*  ERRRPT prints the cursor number, the error code, and the
*  OCI function code.
*
*  CURS is a cursor. N is the cursor number.
*---------------------------------------------------------------

      SUBROUTINE ERRRPT(LDA, CURS)
      INTEGER*2 CURS(32), LDA(32)

      CHARACTER*160 ERRMSG

      IF (CURS(6) .GT. 0) THEN
        WRITE (*, '(1X, A, I3)') 'ORACLE error processing OCI
     + function ', CURS(6)
      END IF

      CALL OERHMS(LDA(1), CURS(7), ERRMSG, 160)
      WRITE (*, '(1X, A)') ERRMSG

      RETURN
      END

      INTEGER FUNCTION LEN_TRIM(STRING)
      CHARACTER*(*) STRING

      INTEGER NEXT

      DO 10 NEXT = LEN(STRING), 1, -1
        IF (STRING(NEXT : NEXT) .NE. ' ') THEN
          LEN_TRIM = NEXT
          RETURN
        ENDIF
10    CONTINUE

      LEN_TRIM = 0


      RETURN
      END
```

FDEMO2.FOR

```
*  FDEMO2.FOR
*
*  A dynamic SQL OCI example program.  Processes
*  SQL statements entered interactively by the user.
*
*  There is a 132–character limit on the length of
*  the SQL statements.  It is not necessary to
*  terminate the SQL statement with a semicolon.
*
*  To end the demo, type 'exit' or 'EXIT' at the
*  prompt.

      PROGRAM FDEMO2
      IMPLICIT INTEGER*4 (A–Z)

*  Data structures
*  Logon and cursor areas
      INTEGER*2      CDA(32), LDA(32), HDA(256)

*  Bind values
      CHARACTER*20   BVARV(8)
      INTEGER        NBV

*  Output values
      CHARACTER*10   DVARC(8)
      INTEGER        DVARI(8)
      REAL*4         DVARF(8)
      INTEGER*2      DBTYPE(8), RLEN(8), RCODE(8)
      INTEGER*2      INDP(8)
      INTEGER        NOV

*  Column names for SELECT
      CHARACTER*10   COLNAM(8)

*  SQL statement buffer and logon info
      CHARACTER*80   SQLSTM, UID, PWD, PROMPT
      INTEGER        UIDL, PWDL, SQLL, NOBLOK
      UID = 'SCOTT'
      PWD = 'TIGER'
      UIDL = LEN_TRIM(UID)
      PWDL = LEN_TRIM(PWD)
      NOBLOK = 0

*  Connect to ORACLE in non–blocking mode.
*  HDA must be initialized to all zeros before call to OLOG.
      DATA HDA/256*0/
      CALL OLOG(LDA, HDA, UID, UIDL, PWD, PWDL, 0, –1, NOBLOK)
      IF (LDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GO TO 999
      ENDIF
      WRITE (*, '(1X, A, A)') 'Connected to ORACLE as user ', UID
*  Open a cursor.
      CALL OOPEN(CDA, LDA, UID, 0, -1, PWD, 0)
      IF (LDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GO TO 900
      ENDIF

*  Beginning of the main program loop.
*  Get and process SQL statements.
      PROMPT = 'Enter SQL statement (132 char max)
+               or EXIT to quit >'
100   WRITE (*, '(/, A)') PROMPT
      READ '(A)', SQLSTM
      SQLL = LEN_TRIM(SQLSTM)
      IF (SQLL .EQ. 0) GO TO 100
      I = INDEX(SQLSTM, ';')
      IF (I .GT. 0) THEN
        SQLL = I - 1
      ENDIF

      IF ((SQLSTM(1:4) .EQ. 'exit') .OR.
     +    (SQLSTM(1:4) .EQ. 'EXIT')) GO TO 900

*  Parse the statement.
      CALL OPARSE(CDA, SQLSTM, SQLL, 0, 2)
      IF (CDA(7) .NE. 0) THEN
        CALL ERRRPT(LDA, CDA)
        GO TO 100
      ENDIF

*  If there are bind values, obtain them from user.
      CALL GETBNV(LDA, CDA, SQLSTM, BVARV, NBV)
      IF (NBV .LT. 0) GO TO 100

*  Define the output variables.  If the statement is not a
*  query, NOV returns as 0.  If there were errors defining
*  the output variables, NOV returns as -1.
      CALL DEFINE(LDA, CDA, COLNAM, DBTYPE, DVARC, DVARI,
     +            DVARF, INDP, RLEN, RCODE, NOV)
      IF (NOV .LT. 0) GO TO 100

*  Execute the statement.
      CALL OEXN(CDA, 1, 0)
      IF (CDA(7) .NE. 0) THEN
```

```
            CALL ERRRPT(LDA, CDA)
            GO TO 100
          ENDIF

*  Fetch rows and display output if the statement was a query.
          CALL FETCHN(LDA, CDA, COLNAM, NOV, DBTYPE, DVARC,
     +            DVARI, DVARF, INDP, RV)
          IF (RV .LT. 0) GO TO 100

*  Loop back to statement 100 to process
*  another SQL statement.
          GO TO  100
*  End of main program loop.  Here on exit or fatal error.
900       CALL OCLOSE(CDA)
          CALL OLOGOF(LDA)

*  End of program.  Come here if connect fails.
999       END

*  Begin subprograms.

          SUBROUTINE GETBNV(LDA, CDA, STMT, BVARV, N)
          IMPLICIT INTEGER*4 (A–Z)
          INTEGER*2      LDA(32), CDA(32)
          CHARACTER*(*) STMT
          CHARACTER*(*) BVARV(8)

*         Arrays for bind variable info.
          INTEGER        BVARI(8), BVARL(8)

*  Scan the SQL statement for placeholders (:ph).
*  Note that a placeholder must be terminated with
*  a space, a comma, or a close parentheses.
*  Two arrays are maintained: an array of starting
*  indices in the string (BVARI), and an array of
*  corresponding lengths (BVARL).
          POS = 1
          DO 300 K = 1, 8            ! maximum of 8 per statement
            I = INDEX(STMT(POS:), ':')
            IF (I .EQ. 0) GO TO 400
            POS = I + POS – 1
            BVARI(K) = POS
            DO 100 J = POS, LEN(STMT)
              IF (STMT(J:J) .EQ. ' '
     +          .OR. STMT(J:J) .EQ. ','
     +          .OR. STMT(J:J) .EQ. ')') THEN
                BVARL(K) = J – POS
                GO TO 200
              ENDIF
```

```
100     CONTINUE

200     POS = POS + 1               ! index past the ':'
300   CONTINUE

400   N = K - 1                     ! N is the number of BVs
      DO 500 K = 1, N
        CALL OBNDRV(CDA, STMT(BVARI(K) :), BVARL(K),
     +             BVARV(K), 20, 1,-1,0,0,-1,-1)
        IF (CDA(7) .NE. 0) THEN
          CALL ERRRPT(LDA, CDA)
          N = -1
          RETURN
        ENDIF
        WRITE (*, '( A, A, A)') 'Enter value for ',
     +        STMT(BVARI(K)+1:BVARI(K)+BVARL(K)-1), '  --> '
        READ '(A)', BVARV(K)
500   CONTINUE

      RETURN
      END

*  Define output variables for queries.
*  Returns the number of select-list items (N)
*  and the names of the select-list items (COLNAM).
*  A maximum of 8 select-list items is permitted.
*  (Note that this program does not check if there
*   are more, but a production-quality program
*   must do this.)

      SUBROUTINE DEFINE(LDA, CDA, COLNAM, DBTYPE, DVARC,
     +                 DVARI, DVARF, INDP, RLEN, RCODE, RV)
      IMPLICIT INTEGER*4 (A-Z)
      INTEGER*2       LDA(32), CDA(32), DBTYPE(8)
      INTEGER*2       RLEN(8), RCODE(8), INDP(8)
      CHARACTER*(*)   DVARC(8), COLNAM(8)
      INTEGER         DVARI(8), RV
      REAL*4          DVARF(8)

      INTEGER         DBSIZE(8), COLNML(8), DSIZE(8)
      INTEGER*2       PREC(8), SCALE(8), NOK(8)

*  If not a query (SQL function code .ne. 4), return.
      IF (CDA(2) .NE. 4) THEN
        RV = 0
        RETURN
      ENDIF
```

```
*  Describe the select–list (up to 8 items max),
*  and define an output variable for each item, with the
*  external (hence, FORTRAN) type depending on the
*  internal ORACLE type, and its attributes.

      DO 100 N = 1, 8
        COLNML(N) = 10  ! COL length must be set on the call
        CALL ODESCR(CDA, N, DBSIZE(N), DBTYPE(N),
     +       COLNAM(N), COLNML(N), DSIZE(N),
     +       PREC(N), SCALE(N), NOK(N))

*  If the return code from ODESCR is 1007, then you have
*  reached the end of the select list.
        IF (CDA(7) .EQ. 1007) THEN
          GO TO 200
*  Otherwise, if the return code is non–zero, an
*  error occurred. Exit the subroutine, signalling
*  an error.
        ELSE IF (CDA(7) .NE. 0) THEN
          CALL ERRRPT(LDA, CDA)
          RV = –1                ! Error on return
          RETURN
        ENDIF

*  Check the datatype of the described item.  If it’s a
*  NUMBER, check if the SCALE is 0.  If so, define the
*  output variable as INTEGER (3). If it’s NUMBER with SCALE != 0,
*  define the output variable as REAL (4).  Otherwise,
*  it’s assumed to be a DATE, LONG, CHAR, or VARCHAR2,
*  so define the output as 1 (VARCHAR2).
        IF (DBTYPE(N) .EQ. 2) THEN
          IF (SCALE(N) .EQ. 0) THEN
            DBTYPE(N) = 3
          ELSE
            DBTYPE(N) = 4
          ENDIF
        ELSE
          DBTYPE(N) = 1
        ENDIF

*  Define the output variable.  Do not define RLEN if
*  the external datatype is 1.
        IF (DBTYPE(N) .EQ. 3) THEN
          CALL ODEFIN(CDA, N, DVARI(N), 4, 3, 0, INDP(N),
     +                FMT, 0, 0, RLEN(N), RCODE(N))
        ELSE IF (DBTYPE(N) .EQ. 4) THEN
          CALL ODEFIN(CDA, N, DVARF(N), 4, 4, 0, INDP(N),
     +                FMT, 0, 0, RLEN(N), RCODE(N))
```

```
        ELSE
          CALL ODEFIN(CDA, N, DVARC(N), 10, 1, 0, INDP(N),
     +                FMT, 0, 0, %VAL(-1), RCODE(N))
        ENDIF
        IF (CDA(7) .NE. 0) THEN
          CALL ERRRPT(LDA, CDA)
          RV = -1
          RETURN
        ENDIF
100   CONTINUE

200   RV = N - 1             ! Decrement to get correct count

      RETURN
      END

*  FETCHN uses OFETCH to fetch the rows that satisfy
*  the query, and displays the output.  The data is
*  fetched 1 row at a time.

      SUBROUTINE FETCHN(LDA, CDA, NAMES, NOV, DBTYPE, DVARC,
     +                  DVARI, DVARF, INDP, RV)
      IMPLICIT INTEGER*4 (A-Z)
      INTEGER*2     LDA(32), CDA(32), DBTYPE(8), INDP(8)
      CHARACTER*(*) NAMES(8), DVARC(8)
      INTEGER       DVARI(8), NOV, RV
      REAL*4        DVARF(8)

      IF (CDA(2) .NE. 4) THEN         ! not a query
        RV = 0
        RETURN
      ENDIF

      DO 50 COL = 1, NOV
        IF (DBTYPE(COL) .EQ. 1) THEN
          WRITE (*, 900) NAMES(COL), ' '
900       FORMAT ('+', A10, A1, $)
        ELSE
          WRITE (*, 902) NAMES(COL), ' '
902       FORMAT ('+', A8, A1, $)
        ENDIF
50    CONTINUE

      WRITE (*, '(1X, A, /)') '------------------------------
     +-------------------------------------------------'
      DO 200 NROWS = 1, 10000
        CALL OFETCH(CDA)
        IF (CDA(7) .EQ. 1403) GO TO 300
        IF (CDA(7) .NE. 0 .AND. CDA(7) .NE. 1406) THEN
```

```
              CALL ERRRPT(LDA, CDA)
              RV = –1
              RETURN
            ENDIF
            DO 100 COL = 1, NOV
              IF (INDP(COL) .LT. 0 .AND. DBTYPE(COL) .NE. 1) THEN
                WRITE (*, 903), '          '
903             FORMAT ('+', A9, $)
              ELSE IF (INDP(COL) .LT. 0 .AND. DBTYPE(COL) .EQ. 1) THEN
                WRITE (*, 905), '            '
905             FORMAT ('+', A11, $)
              ELSE
                IF (DBTYPE(COL) .EQ. 3) THEN
                  WRITE (*, 904) DVARI(COL), '   '
904               FORMAT ('+', I6, A3, $)
                ELSE IF (DBTYPE(COL) .EQ. 4) THEN
                  WRITE (*, 906) DVARF(COL), ' '
906               FORMAT ('+', F8.2, A1, $)
                ELSE
                  WRITE (*, 908) DVARC(COL), ' '
908               FORMAT ('+', A10, A1, $)
                ENDIF
              ENDIF
100         CONTINUE
            WRITE (*, '(1X)')
200       CONTINUE

300       NROWS = NROWS – 1
          WRITE (*, '(/, 1X, I3, A)') NROWS, ' rows returned'
          RETURN
          END

          SUBROUTINE ERRRPT(LDA, CDA)
          INTEGER*2 LDA(32), CDA(32)

          CHARACTER*132  MSG

          MSG = ' '
          IF (LDA(7) .NE. 0) THEN
             CDA(7) = LDA(7)
             CDA(6) = 0
          ENDIF

          IF (CDA(6) .NE. 0) THEN
            WRITE (*, '(1X, A, I3)') 'Error processing OCI function',
         +   CDA(6)
          ENDIF

          CALL OERHMS (LDA, CDA(7), MSG, 132)
```

```
      WRITE (*, '(1X, A)') MSG

      RETURN
      END

      INTEGER FUNCTION LEN_TRIM(STRING)
      CHARACTER*(*) STRING

      INTEGER NEXT

      DO 10 NEXT = LEN(STRING), 1, -1
        IF (STRING(NEXT : NEXT) .NE. ' ') THEN
          LEN_TRIM = NEXT
          RETURN
        ENDIF
10    CONTINUE

      LEN_TRIM = 0

      RETURN
      END
```

FDEMO3.FOR

```
*   FDEMO3.FOR
*   OCI FORTRAN Sample Program 3
*
*   Demonstrates using the OFLNG routine to retrieve
*   part of a LONG RAW column
*
*   This example "plays" a digitized voice message
*   by repeatedly extracting 64 kB chunks of the message
*   from the row in the table, and sending them to a
*   converter buffer (for example, a Digital-to-Analog
*   Converter (DAC) FIFO buffer).
*
*   The hardware-specific DAC routine is merely simulated
*   in this example.

      PROGRAM FDEMO3
      IMPLICIT INTEGER(A-Z)

*   Connect and Cursor Data Structures

      INTEGER*2        CDA(32)
      INTEGER*2        LDA(32)
      INTEGER*2        HDA(256)
```

```
*  Program Variables

      CHARACTER*132    SQLSTM
      CHARACTER*20     UID, PWD
      INTEGER          MSGID, MSGLEN
      INTEGER*2        INDP, RLEN, RCODE
      INTEGER          RETL
      BYTE             DBIN(200000)
      CHARACTER*6      FMT
      INTEGER*4        NOBLOK

*  Connect to ORACLE in non–blocking mode.
*  The HDA must be initialized to all zeros before calling OLOG.
      UID = ’SCOTT’
      PWD = ’TIGER’
      DATA HDA/256*0/
      CALL OLOG(LDA, HDA, UID, LEN_TRIM(UID),
     +          PWD, LEN_TRIM(PWD), 0, –1, NOBLOK)
      IF (LDA(1) .NE. 0) THEN
          WRITE (*, ’(1X, A)’) ’Cannot connect as scott/tiger...’
          WRITE (*, ’(1X, A)’) ’Application terminating...’
          GOTO 999
      END IF
*  Open the cursor.  (Use UID as a dummy parameter––it
*  won’t be looked at.)
      CALL OOPEN(CDA, LDA, UID, 0, 0, UID, 0)
      IF (CDA(7) .NE. 0) THEN
          CALL ERRPT(CDA)
          GOTO 900
      END IF

*  Drop the old table.
      WRITE (*, ’( A)’) ’OK to drop table VOICE_MAIL (Y or N)? : ’
      READ ’(A)’, FMT
      IF (FMT(1:1) .EQ. ’y’ .OR. FMT(1:1) .EQ. ’Y’) THEN
         GO TO 10
      ELSE
         GO TO 900
      ENDIF

*  Parse the DROP TABLE statement.
10    SQLSTM = ’DROP TABLE VOICE_MAIL’
      CALL OPARSE(CDA, SQLSTM, LEN_TRIM(SQLSTM), 0, 2)
      IF (CDA(7) .EQ. 0) THEN
          WRITE (*, ’(1X, A)’) ’Table VOICE_MAIL dropped.’
      ELSEIF (CDA(7) .EQ. 942) THEN
         WRITE (*, ’(1X, A)’) ’Table did not exist.’
      ELSE
```

```
            CALL ERRPT(LDA, CDA)
            GO TO 900
         ENDIF
*     Create new table.  Parse with DEFFLG set to zero,
*     to immediately execute the DDL statement.  The LNGFLG
*     is set to 2 (Version 7).

         SQLSTM = 'CREATE TABLE VOICE_MAIL
        +   (MSG_ID NUMBER(6), MSG_LEN NUMBER(12), MSG LONG RAW)'

*     Parse the statement.  Do not defer the parse, so that the
*     DDL statement is executed immediately.
         CALL OPARSE(CDA, SQLSTM, LEN_TRIM(SQLSTM), 0, 2)
         IF (CDA(7) .EQ. 0) THEN
             WRITE (*, '(1X, A)') 'Created table VOICE_MAIL.'
         ELSE
             CALL ERRPT(LDA, CDA)
             GOTO 900
         END IF

*     Insert some dummy data into the table.
         SQLSTM = 'INSERT INTO VOICE_MAIL VALUES (:1, :2, :3)'
         CALL OPARSE(CDA, SQLSTM, LEN_TRIM(SQLSTM), 1, 2)
         IF (CDA(7) .NE. 0) THEN
             CALL ERRPT(LDA, CDA)
             GOTO 900
         END IF

*     Do the binds for the input data to set values
*     in the new table.
         INDP = 0
         CALL OBNDRN(CDA, 1, MSGID, 4, 3, 0, INDP, FMT, 0, 0)
         IF (CDA(7) .NE. 0) THEN
             CALL ERRPT(LDA, CDA)
             GOTO 900
         END IF

         CALL OBNDRN(CDA, 2, MSGLEN, 4, 3, 0, INDP, FMT, 0, 0)
         IF (CDA(7) .NE. 0) THEN
             CALL ERRPT(LDA, CDA)
             GOTO 900
         END IF

         CALL OBNDRN(CDA, 3, DBIN, 200000, 24, 0, INDP, FMT, 0, 0)
         IF (CDA(7) .NE. 0) THEN
             CALL ERRPT(LDA, CDA)
             GOTO 900
         END IF
```

```
*  Fill the input buffer with some dummy data.
      MSGID = 100
      MSGLEN = 200000
      DO 100 I = 1, 200000
100       DBIN(I) = 42

*  Execute the statement to INSERT the data
      WRITE (*, '(1X, A)') 'Inserting data into the table.'
      CALL OEXN(CDA, 1, 0)
      IF (CDA(7) .NE. 0) THEN
          CALL ERRPT(LDA, CDA)
          GOTO 900
      END IF

*  Do the selects.  First position the cursor at the
*  proper row, using the MSG_ID.  Then fetch the data
*  in 64K chunks, using OFLNG.
      SQLSTM = 'SELECT MSG_ID, MSG_LEN, MSG
     + FROM VOICE_MAIL WHERE MSG_ID = 100'
      CALL OPARSE(CDA, SQLSTM, LEN_TRIM(SQLSTM), 1, 2)
      IF (CDA(7) .NE. 0) THEN
          CALL ERRPT(LDA, CDA)
          GOTO 900
      END IF

*  Define the output variables for the SELECT.
      CALL ODEFIN(CDA, 1, MSGID, 4, 3, 0, %VAL(-1), %VAL(-1),
     +     0, 0, %VAL(-1), %VAL(-1))
      IF (CDA(7) .NE. 0) THEN
          CALL ERRPT(LDA, CDA)
          GOTO 900
      END IF

      CALL ODEFIN(CDA, 2, MSGLEN, 4, 3, 0, %VAL(-1), %VAL(-1),
     +     0, 0, %VAL(-1), %VAL(-1))
      IF (CDA(7) .NE. 0) THEN
          CALL ERRPT(LDA, CDA)
          GOTO 900
      END IF

      CALL ODEFIN(CDA, 3, DBIN, 200000, 24, 0, INDP, %VAL(-1),
     +     0, 0, RLEN, RCODE)
      IF (CDA(7) .NE. 0) THEN
          CALL ERRPT(LDA, CDA)
          GOTO 900
      END IF
```

```
*  Do the query, getting the MSG_ID to position the cursor, and
*  the first 100 bytes of the message.
*  CANCEL and EXACT are FALSE.
      CALL OEXFET(CDA, 1, 0, 0)
      IF (CDA(7) .NE. 0) THEN
          CALL ERRPT(LDA, CDA)
          GOTO 900
      END IF
      WRITE (*, '(1X, A, I4, A)') 'Message', MSGID,
     +' is available.'
      WRITE (*, '(1X, A, I7, A)') 'The length is', MSGLEN,
     +' bytes.'

*  Play out the message, calling the DAC routine for each
*  64K chunk fetched by OFLNG.

      OFFSET = 0
      N = MSGLEN/65536 + 1
      DO 200 J = 1, N
        IF (MSGLEN .LT. 65536) THEN
           LEN = MSGLEN
        ELSE
           LEN = 65536
        ENDIF
        CALL OFLNG(CDA, 3, DBIN, LEN, 24, RETL, OFFSET)
        IF (CDA(7) .NE. 0) THEN
           CALL ERRPT(LDA, CDA)
           GOTO 900
        ENDIF
        CALL PLAYMSG(DBIN, LEN)
        MSGLEN = MSGLEN – LEN
        IF (MSGLEN .LT. 0) GO TO 900

200   CONTINUE

900   CALL OCLOSE(CDA)
      IF (CDA(7) .NE. 0) THEN
         CALL ERRPT(LDA, CDA)
         GOTO 900
      END IF
      CALL OLOGOF(LDA)
      IF (LDA(7) .NE. 0) THEN
         CALL ERRPT(LDA, CDA)
         GOTO 900
      END IF

999   STOP 'End of OCIDEMO3.'
      END
      SUBROUTINE PLAYMSG(OUT, LEN)
      BYTE     OUT(65536)
      INTEGER  LEN
      WRITE (*, '(1X, A, I7, A)') 'Playing', LEN, ' bytes.'
      RETURN
      END
      SUBROUTINE ERRPT(LDA, CDA)
      INTEGER*2 LDA(32), CDA(32)

      CHARACTER*132  MSG

      MSG = ' '
      IF (LDA(7) .NE. 0) THEN
         CDA(7) = LDA(7)
         CDA(6) = 0
      ENDIF

      IF (CDA(6) .NE. 0) THEN
        WRITE (*, '(1X, A, I3)') 'Error processing OCI function',
     +   CDA(6)
      ENDIF

      CALL OERHMS (LDA, CDA(7), MSG, 132)
      WRITE (*, '(1X, A)') MSG

      RETURN
      END

      INTEGER FUNCTION LEN_TRIM(STRING)
      CHARACTER*(*) STRING

      INTEGER NEXT

      DO 10 NEXT = LEN(STRING), 1, –1
        IF (STRING(NEXT : NEXT) .NE. ' ') THEN
          LEN_TRIM = NEXT
          RETURN
        ENDIF
10    CONTINUE

      LEN_TRIM = 0

      RETURN
      END
```

APPENDIX

D

Additional OCI Functions (C)

This appendix describes OCI functions that have been superseded by newer functions in more recent versions of the Oracle7 OCI. The new functions offer increased functionality or performance and should be used in new OCI programs that access Oracle7.

These older functions are still available in the current Oracle7 OCI library. However, they might not be available in future versions of the Oracle Call Interface.

odsc

Purpose *odsc()* describes select–list items for dynamic SQL queries. The *odsc()* function returns internal datatype and size information for a specified select–list item. Use *odescr()* in place of *odsc()* in Oracle7 or later OCI programs.

Syntax

```
odsc(struct cda_def *cursor, sword pos,
     sb2 *dbsize, [sb2 *fsize], [sb2 *rcode],
     [sb2 *dbtype], [text *cbuf],
     [sb2 *cbufl], [sb2 *dsize]);
```

Comments You can call *odsc()* after an *osql3()* or *oexec()* call to obtain the maximum size (*dbsize*), internal datatype code (*dbtype*), column name (*cbuf*), and maximum display size (*dsize*) of select–list items in a query. This function is used for dynamic SQL queries, that is, queries in which the number of select–list items and their datatypes and sizes may not be known until runtime.

The *odsc()* function may also be called after an *ofetch()*, to return the actual size (*fsize*) of the item just fetched.

The *return code* field of the cursor data area indicate success (zero) or failure (non–zero) of the *odsc()* call.

The *odsc()* function uses a position index to refer to select–list items in the SQL query statement. For example, the SQL statement

```
SELECT ename, sal FROM emp WHERE sal > :Min_sal
```

contains two select–list items: “ename” and “sal”. The position index of sal is 2; ename’s index is 1. The following example shows how you call *odsc()* to describe the two select–list items in this SQL statement:

```
struct cda_def *cursor;
sb2 ename_len, sal_len, ename_type, sal_type;
sb2 col_name_len, max_disp_len;
text col_name[240];
...
odsc(&cursor, 1, &ename_len, (sb2 *) –1, (sb2 *) –1,
     &ename_type, col_name, &col_name_len, &max_disp_len);
/* Make use of values returned. */
odsc(&cursor, 2, &sal_len, (sb2 *) –1, (sb2 *) –1,
     &sal_type, col_name, &col_name_len, &max_disp_len);
```

Note: A dependency exists between the results returned by a describe operation (*odsc()*) and a bind operation (*obndrn()* or *obndrv()*). Because a select–list item might contain bind variables, the type returned by *odsc()* can vary depending on the results of bind operations.

So, if you have placeholders for bind variables in a SELECT statement and you will use *odsc()* to obtain the size or datatype of select–list items, you should do the bind operation before the describe. If you need to rebind any input variables after performing a describe, you must reparse the SQL statement before rebinding. Note that the rebind operation might change the results returned for a select–list item.

Parameters

Parameter Table	***Type***	***Status***
cursor	struct cda_def *	IN / OUT
pos	sword	IN
dbsize	sb2 *	OUT
fsize	sb2 *	IN 1
rcode	sb2 *	IN 1
dbtype	sb2 *	OUT
cbuf	text *	OUT
cbuflen	sb2 *	IN / OUT
dsize	sb2 *	OUT

Note 1. These are OUT parameters for *oexfet()*, *ofetch()*, or *ofen()*.

cursor
A pointer to the cursor data area for the statement being described.

pos
The position index of the select–list item in the SQL query. Each item is referenced by position index, starting at 1 for the first (or left–most) item. If you specify a position index greater than the number of items in the select–list or less than 1, *odsc()* returns a “variable not in select–list” error in the *return code* field of the cursor data area.

dbsize
Receives the maximum size of the column as stored in the Oracle data dictionary. If the column is defined as VARCHAR2, CHAR or NUMBER, the length returned is the maximum length specified for the column. For LONG and LONG RAW columns, zero is returned.

fsize
Receives the actual size of the data value returned by the last *ofetch()* operation. This parameter is only meaningful when *odsc()* is issued after an *ofetch()* call. The value returned is the actual length of the value as stored in the database before it is moved to the user buffer where padding or truncation may take place. Oracle suppresses leading zeros

on numeric data and trailing blanks on VARCHAR character data before storing the values in the database.

Oracle recommends using the *rlen* parameter of *odefin()* to obtain column return lengths. This is more efficient than calling *odsc()* after a fetch.

rcode
Receives the column return code returned by the last fetch operation. This parameter is meaningful only when *odsc()* is issued after an *ofetch()* call.

> **Note:** It is more efficient to establish a column–level *rcode* with *odefin()* rather than using *odsc()* after each fetch. For an example of using column–level return codes, see the example code under the *ofetch()* description on page 4 – 74.

dbtype
Receives the internal datatype code of the select–list item. A list of Oracle internal datatype codes and the possible external conversions for each of them is provided in Table 3 – 2.

cbuf
The *cbuf* buffer receives the name of the select–list item (name of the column or wording of the expression). If *cbuf* is passed as NULL, the column or expression name is not returned.

cbufl
Set *cbufl* to the length of *cbuf* on the call. If *cbufl* is not specified (that is, passed as NULL) or if the value contained in *cbufl* is zero, then the column name is not returned. The column name is truncated if it is longer than *cbufl*.

On return from *odsc()*, *cbufl* contains the length of the returned string.

dsize
This parameter receives the maximum display size of the select–list item if the select–list item is returned as a character string. The *dsize* parameter is especially useful when SQL functions, like SUBSTR or TO_CHAR, are used to modify the representation of a column.

See Also *odescr()*.

oermsg

Purpose *oermsg()* returns the Oracle error message text corresponding to the return code (rcode). Use *oerhms()* instead of *oermsg()* in new Oracle7 OCI programs, or wherever error messages longer than 70 bytes must be returned.

Syntax

```
oermsg(ub2 rcode, text *msg);
```

Comments The *oermsg()* function is similar to *oerhms()* in that it returns an Oracle error message text. If there is no message that corresponds to the return code, the message “unknown Oracle error code” is returned.

> **Note:** Programs that have multiple active connections should not use the *oermsg()* function to obtain error message text. Use the *oerhms()* function instead. Also, precompiler programs that contain OCI calls must use *oerhms()* instead of *oermsg()*.

The following example shows a function that is used to report Oracle errors:

```
void
errrpt(struct cda_def *cur)
{
  char msg_buf[71];

  printf(”Oracle error: code is %d, op is %d\n”,
         (int) cur–>rc, (int) cur–>fc);
  oermsg(cur–>rc, msg_buf);
  printf(”%s\n”, (char *) msg_buf);
}
```

Parameters

Parameter Name	*Type*	*Mode*
rcode	ub2	IN
msg	text *	OUT

rcode
The return code value for which the message text is to be returned. This can be either the V2 return code or the normal return code.

msg
A pointer to a program buffer to receive the error message text. The returned message text is null terminated. The maximum message size that can be returned is 70 bytes. Longer messages are truncated.

See Also *oerhms()*.

olon

Purpose

olon() logs a program on to an Oracle database. Communication between the program and Oracle is established using the logon data area defined in the program. Use *olog()* instead of *olon()* in new Oracle7 OCI programs.

Syntax

```
olon(struct lda_def *lda, text *uid, [sword uidl],
     [text *pswd], [sword pswdl], <sword audit>);
```

Comments

A program can log on to Oracle many times, but if the program logs on using *olon()*, only one connection can be active at a time. To switch to another user ID using the same LDA, you must first log off using the *ologof()* function, then log on again using *olon()*, with the different user ID. For multiple concurrent connections using different LDAs, use the *orlon()* function.

olon() cannot be used in a program that mixes OCI calls and precompiler statements. See the description of *sqllda()* in Chapter 4 for more information.

OCI programs can use *orlon()*, with the *hda* parameter set to the null pointer (0), as a direct replacement for *olon()*. For example, to log on to Oracle:

```
Lda_Def lda;
text uid[32], pwd[32];

strcpy(uid, “system”);
strcpy(pwd, “manager”);
if (olon(&lda, uid, –1, pwd, –1, 0))
{
    printf(“Cannot logon as %s\/%s\n”, uid, pwd);
    exit(1);
}
```

Parameters

Parameter Name	*Type*	*Mode*
lda	struct lda_def *	IN / OUT
uid	text *	IN
uidl	sword	IN
pswd	text *	IN
pswdl	sword	IN
audit	sword	IN

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle. The *return code* field indicates the result of the *olon()* call. A zero indicates a successful logon.

uid
A pointer to a character string containing the user ID and possibly the password. If the password is included as part of the user ID, it must be separated from the user ID by a '/'.

If the user ID is simply '/' and a valid automatic logon account exists on the database, then the logon succeeds under the user ID of the person running the program.

uidl
The length of the string pointed to by *uid*. This parameter can be omitted if the *uid* string is null terminated.

pswd
A pointer to a string containing the password. If the password is specified as part of *uid*, this parameter can be omitted.

If the *uid* parameter does not include a password and the *pswd* parameter is invalid, null, or omitted, the logon will fail with an error code in the *return code* field of the LDA.

pswdl
The length of the string pointed to by *pswd*. This parameter can be omitted if the *pswd* string is null terminated.

audit
This parameter is not currently used. It should be passed as –1.

See Also *olog()*.

oname

Purpose *oname()* retrieves the names of select–list items in a SQL query statement. Use *odescr()* instead of *oname()* in new Oracle7 OCI programs.

Syntax

```
oname(struct cda_def *cursor, sword pos,
    <sb1 *tbuf>, <sb2 *tbufl>,
    sb1 *cbuf, sb2 *cbufl);
```

Comments *oname()* can be called only after the SQL statement has been parsed using *oparse()* or *osql3()*. The maximum length of an item name that can be returned is 240 bytes.

> **Note:** *oname()* obtains the names of select–list items, not their values. In the SQL statement SELECT AVG(SAL) FROM EMP there is one select–list item. Its name is AVG(SAL).

oname() operates by referencing the select–list items by position index. Indices are numbered sequentially from left to right, beginning with 1. If you specify a position index greater than the number of select–list items, *oname()* returns a "no data found" status in the *return code* field of the cursor data area.

For queries such as SELECT * FROM EMP, in which the number of columns is unknown, you can obtain the column names dynamically by repeatedly calling *oname()* until a "no data found" error occurs. For example, the following code fragment shows how to use *oname()* to obtain column names:

```
struct cda_def *cursor;    /* cursor data area */
text name_buf[241];        /* output buffer */
sb2 i, nb_len;
...

osql3(&cursor, "SELECT * FROM EMP", –1);
for (nb_len = 240, i = 1;
  !oname(&cursor, i, NULL, (sb2 *) 0, name_buf,
    &nb_len);
  nb_len = 240, i++)
    printf("%*.*s\n", (int) nb_len, (int) nb_len,
           (char *) name_buf);
```

Observe the *printf()* statement in this example carefully. The *oname()* function does not null terminate the returned string.

> **Note:** *oname()* provides only a limited subset of the functionality of the *odescr()* function.

Parameters

Parameter Name	*Type*	*Mode*
cursor	struct cda_def *	IN / OUT
pos	sword	IN
tbuf	text *	IN
tbufl	sb2 *	IN
cbuf	text *	OUT
cbufl	sb2 *	IN / OUT

cursor
A pointer to the 64–byte cursor data area for the query. The *oname()* function uses the cursor address to obtain the names from the parsed SQL statement in the Oracle shared SQL cache.

pos
The positional index of the select–list item in the SQL query. The first (left–most) select–list item is position 1.

tbuf
This parameter is included only for Oracle Version 2 compatibility. For later versions, it should be passed as NULL.

tbufl
This parameter is included only for Oracle Version 2 compatibility. For later versions, it should be passed as (**sb2** *) –1.

cbuf
A pointer to a data buffer within the program that receives the name of the select–list item. If *cbufl* is passed as zero, the name is not stored.

The name returned in *cbuf* is blank padded to the length specified in the *cbufl* input parameter. It is not null terminated.

cbufl
A pointer to a short integer. Before calling *oname()*, this parameter must be set to the length of the *cbuf* buffer. After *oname()* completes, *cbuflen* contains the length of the name that was returned in *cbuf*. If the value contained in *cbufl* is zero, no name is returned in *cbuf*. If the select–list item name is longer than the value pointed to by *cbufl*, the returned name is truncated.

See Also *odescr()*.

orlon

Purpose *orlon()* establishes concurrent communications between an OCI program and an Oracle database. Use *olog()* instead of *orlon()* in new Oracle7 OCI programs.

Syntax

```
orlon(struct lda_def *lda, ub1 *hda, text *uid,
      [sword uidl], [text *pswd], [sword pswdl],
      <sword audit>);
```

Comments An OCI program can connect to one or more Oracle instances multiple times. Communication takes place using the logon data area and the host data area defined within the program. The *orlon()* function connects the LDA to Oracle.

A host data area is a program–allocated data area associated with each *orlon()* logon call. Its contents are entirely private to Oracle, but the HDA must be allocated by the OCI program. Each concurrent connection requires one LDA–HDA pair.

> **Note:** Refer to the section "Host Data Area" in Chapter 2 for important information about allocating an HDA.

After the *orlon()* call, the HDA and the LDA must remain at the same program address they occupied at the time *orlon()* was called.

For example:

```
#include <stdlib.h>

Lda_Def lda[2];        /* establish two logon data areas */
ub1 hda[2][256];                        /* and two HDA's */
text *uid1 = "SCOTT/TIGER@T:KERVMS:V7_R2";
text *uid2 = "SYSTEM@T:KR2VMS:V7_R3";
text *pwd = "MANAGER";
...
/* first connect as scott */
if (orlon(&lda[0], &hda[0], uid1, –1, (text *) 0, –1, 0))
{
  error_handler(&lda[0]);
  exit(EXIT_FAILURE);
}
...
/* and later as the system manager */
if (orlon(&lda[1], &hda[1], uid2, –1, pwd, –1, 0))
{
  error_handler(&lda[1]);
  exit(EXIT_FAILURE);
}
```

When the program has issued an *orlon()* call, a subsequent *ologof()* call using the same LDA commits all outstanding transactions for that connection. If a program fails to disconnect or terminates abnormally, then all outstanding transactions are rolled back.

The LDA *return code* field indicates the result of the *orlon()* call. A zero return code indicates a successful connection.

You should also refer to the section on SQL*Net in your Oracle installation or user's guide for any particular notes or restrictions that apply to your operating system.

Parameters

Parameter Name	***Type***	***Mode***
lda	struct lda_def *	IN / OUT
hda	ub1 *	OUT
uid	text *	IN
uidl	sword	IN
pswd	text *	IN
pswdl	sword	IN
audit	sword	IN

lda
A pointer to the LDA specified in the *olog()* call that was used to make this connection to Oracle.

hda
A pointer to a host data area struct. See Chapter 2 for more information on host data areas.

uid
Specifies a string containing the username, an optional password, and an optional host machine identifier. If you include the password as part of the *uid* parameter, put it immediately after the username and separate it from the username with a '/'. Put the host machine identifier after the username or the password, preceded by the '@' sign.

If the password is not included in this parameter, it must be in the *pswd* parameter. Examples of valid *uid* parameters are

```
name
name/password
name@d:nodename:dbname
name/password@d:nodename:dbname
```

The following example is not a correct example of a *uid*:

```
name@d:nodename:dbname/password
```

uidl
The length of the string pointed to by *uid*. If the string pointed to by *uid* is null terminated, this parameter should be passed as –1.

pswd
A pointer to a string containing the password. If the password is specified as part of the string pointed to by *uid*, this parameter should be passed as 0.

pswdl
The length of the password. If the string pointed to by *pswd* is null or null terminated, this parameter should be passed as –1.

audit
This parameter is no longer supported; the only permissible values are 0 or –1.

See Also *olog()*.

osql3

Purpose

osql3() parses a SQL statement or a PL / SQL block and associates it with a cursor. The *osql3()* function also executes Data Definition Language statements. Use *oparse()* in place of *osql3()* in new Oracle7 OCI programs.

Syntax

```
osql3(struct cda_def *cursor, text *sqlstm,
      [sword sqll]);
```

Comments

osql3() passes the SQL statement to Oracle for parsing. The parsed representation of the SQL statement is stored in the shared SQL statement cache in the Oracle server, and state information about the cursor is stored in the private SQL area. Subsequent OCI calls reference the SQL statement using the cursor name. An open cursor may be reused by subsequent *osql3()* calls within a program, or the program may define multiple concurrent cursors when it is necessary to maintain multiple active SQL statements.

The SQL statement may be any valid SQL query, data manipulation, data definition, or data control statement. Oracle parses the statement and selects an optimal access path to perform the requested function. Data Definition Language statements are executed when *osql3()* is called. Data Manipulation Language statements and queries are not executed until you call the *oexec()* function.

The following example opens a cursor and parses a SQL statement. The *osql3()* call associates the SQL statement with the cursor.

```
lda_def lda;
struct cda_def cursor;
text *sql_stmt =
    ”DELETE FROM emp WHERE empno = :EMPLOYEE_NUMBER”;

...

oopen(&lda, &cursor, (text *) 0, 0, 0, (text *) 0, 0);
osql3(&cursor, sql_stmt, –1);
```

SQL syntax error codes are returned in the cursor data area’s *return code* field and *parse error offset* field. Parse error offset indicates the location of the error in the SQL statement text. See the section “Cursor Data Area” on page 2 – 4 for a list of the information fields available in the cursor data area after an *osql3()* call.

Parameters

Parameter Name	*Type*	*Mode*
cursor	struct cda_def *	IN / OUT
sqlstm	text *	IN
sqll	sword	IN

cursor
A pointer to a cursor data area specified in the *oopen()* call.

sqlstm
A pointer to a string containing a SQL statement.

sqll
Specifies the length of the SQL statement. If the SQL statement string pointed to by *sqlstm* is null terminated, this parameter can be omitted.

See Also *oparse()*.

APPENDIX

E

Additional OCI Routines (COBOL)

This appendix describes OCI routines that have been superseded by newer ones in more recent versions of the Oracle7 OCI. The new routines offer increased functionality or performance and should be used in new OCI programs that access Oracle7.

These older routines are still available in Oracle7. However, they might not be available in future versions of the OCI.

ODSC

Purpose ODSC describes select–list items for dynamic SQL queries. The ODSC routine returns internal datatype and size information for a specified select–list item. Use ODESCR in place of ODSC in new Oracle7 OCI programs.

Syntax

```
CALL ”ODSC” USING CURSOR, POS, DBSIZE, [FSIZE],
     [RCODE], [DBTYPE], [CBUF], [CBUFL], [DSIZE].
```

Comments You can call ODSC after an OSQL3 or OEXEC call to obtain the maximum size (DBSIZE), internal datatype code (DBTYPE), column name (CBUF), and maximum display size (DSIZE) of select–list items in a query. This routine is used for *dynamic* SQL queries, that is, queries in which the number of select–list items, as well as their datatypes and sizes, may not be known until runtime.

The ODSC routine may also be called after an OFETCH to return the actual size (FSIZE) of the item just fetched.

The *return code* field of the cursor data area indicates success (zero) or failure (non–zero) of the ODSC call.

The ODSC routine uses a position index to refer to select–list items in the SQL query statement. For example, the SQL statement

```
SELECT ENAME, SAL FROM EMP WHERE SAL > :MIN_SAL
```

contains two select–list items: ENAME and SAL. The position index of SAL is 2, and ENAME’s index is 1. The following example shows how you would call ODSC to describe the two select–list items in this SQL statement:

```
DATA DIVISION.
77   ENAME–L     PIC S9(4) COMP.
77   SAL–L       PIC S9(4) COMP.
77   ENAME–TYPE  PIC S9(4) COMP.
77   SAL–TYPE    PIC S9(4) COMP.
77   MAX–DISP–L  PIC S9(4) COMP.
77   FSIZE       PIC S9(4) COMP.
77   RCODE       PIC S9(4) COMP.
77   COL–NAME    PIC X(240).
77   C–NAME–L    PIC S9(4) COMP.
...
PROCEDURE DIVISION.
CALL ”ODSC” USING CURSOR, 1, ENAME–L, FSIZE, RCODE,
    ENAME–TYPE, COL–NAME, C–NAME–L, MAX–DISP–L.
...
CALL ”ODSC” USING CURSOR, 2, SAL–L, FSIZE, RCODE,
    SAL–TYPE, COL–NAME, C–NAME–L, MAX–DISP–L.
```

Note: A dependency exists between the results returned by a describe operation (ODSC) and a bind operation (OBNDRN or OBNDRV). Because a select–list item might contain bind variables, the type returned by ODSC can vary depending on the results of bind operations.

So, if you have placeholders for bind variables in a SELECT statement and you plan to use ODSC to obtain the size or datatype of select–list items, you should do the bind operation before the describe. If you need to rebind any input variables after performing a describe, you must reparse the SQL statement before rebinding. Note that the rebind operation might change the results returned for a select–list item.

Parameters

Parameter Name	*Type*	*Status*
CURSOR	Address	IN / OUT
POS	PIC S9(9) COMP	IN
DBSIZE	PIC S9(4) COMP	IN
FSIZE	PIC S9(4) COMP	IN 1
RCODE	PIC S9(4) COMP	IN 1
DBTYPE	PIC S9(4) COMP	OUT
CBUF	PIC X(n)	OUT
CBUFL	PIC S9(4) COMP	IN / OUT
DSIZE	PIC S9(4) COMP	OUT

Note 1. Parameter can also be an OUT after OFETCH or OFEN.

CURSOR
The cursor data area associated with the SQL statement being described. The ODSC routine uses the cursor to reference a specific SQL query statement that has been passed to Oracle by a prior OSQL3 call.

POSITION
The position index of the select–list item in the SQL query. Each item is referenced by position as if they were numbered left to right consecutively beginning with 1. If you specify a position index greater than the number of items in the select–list or less than 1, ODSC returns a “variable not in select–list” error in the *return code* field of the cursor data area.

DBSIZE
DBSIZE receives the *maximum* size of the column as stored in the Oracle data dictionary. If the column is defined as VARCHAR2, CHAR, or NUMBER, the length returned is the maximum length specified for the column.

FSIZE
FSIZE receives the actual size of the data value returned by the last OFETCH operation. This parameter is only meaningful when ODSC is issued after an OFETCH call. The value returned is the actual length of the value as stored in the database before it is moved to the user buffer where padding or truncation may take place. Oracle suppresses leading zeros on numeric data and trailing blanks on VARCHAR data before storing the values in the database.

RCODE
RCODE receives the column return code returned by the last fetch operation. This parameter is meaningful only when ODSC is issued after an OFETCH call.

> **Note:** It is generally more efficient to establish a column–level RCODE with ODEFIN rather than using ODSC after each fetch. For an example of using column–level return codes, see the example code under the OFETCH description on page 5 – 70.

DBTYPE
DBTYPE receives the internal datatype code of the select–list item. A list of Oracle internal datatype codes and the possible external conversions for each of them is provided in Table 3 – 2.

CBUF
The address of a character buffer in the program that receives the name of the select–list item (name of the column or wording of the expression).

CBUFL
CBUFL is set to the length of CBUF. The column name is truncated if it is longer than CBUFL. If CBUFL is not specified or if the value contained in CBUFL is 0, then the column name is not returned. On return from ODSC, CBUFL contains the length of the column or expression name.

DSIZE
DSIZE receives the maximum display size of the select–list item if the select–list item is returned as a character string. The DSIZE parameter is especially useful when routines, like SUBSTR or TO_CHAR, are used to modify the representation of a column.

See Also ODESDCR.

OERMSG

Purpose OERMSG returns the Oracle error message text corresponding to the return code parameter RCODE. Use OERHMS instead of OERMSG in new Oracle7 OCI programs or wherever longer error messages must be returned.

Syntax

```
CALL ”OERMSG” USING RCODE, MSG.
```

Comments The OERMSG routine is similar to OERHMS in that it returns an Oracle error message text. If there is no message that corresponds to the return code, the message “unknown Oracle error code” is returned.

> **Note:** Programs that have multiple active logons should not use OERMSG to obtain error message text. Use OERHMS instead. Also, Precompiler programs that contain OCI calls cannot use OERMSG; use OERHMS instead.

The following example shows a subroutine that can be used to report oracle errors:

```
ORA–ERR
.   IF LDA–RC NOT = 0
        DISPLAY ”OLOGON ERROR”
        MOVE LDA–RC TO ERR–RC
        MOVE ”0” TO ERR–FUNC
    ELSE IF C–RC IN CURSOR–1 NOT = 0
        MOVE C–RC IN CURSOR–1 TO ERR–RC
        MOVE C–FNC IN CURSOR–1 TO ERR–FUNC
    ELSE MOVE C–RC IN CURSOR–2 TO ERR–RC
        MOVE C–FNC IN CURSOR–2 TO ERR–FUNC
        MOVE ERR–RC TO ERR–RCX
.   DISPLAY ”Oracle ERROR. CODE IS ”, ERR–RCX ,”,FUNCTION IS ”,
             ERR–FUNC
.   CALL ”OERMSG” USING ERR–RC, MSGBUF
.   DISPLAY MSGBUF.
```

Parameters

Parameter Name	*Type*	*Status*
RCODE	PIC S9(4) COMP	IN
MSG	PIC X(n)	OUT

RCODE
The return code value for which the message text is to be returned. This can be either the V2 return code or the normal return code.

MSG
The address of a program buffer to receive the error message text. The maximum message size that can be returned is 70 characters. If the message is longer than the buffer size or 70 characters, the message is truncated.

See Also OERHMS.

OLON

Purpose OLON logs a program on to an Oracle database. Communication between the program and Oracle is established using the logon data area defined in the program. Use OLOG in place of ORLON in new Oracle7 OCI programs.

Syntax

```
CALL ”OLON” USING LDA, UID, UIDL, [PSWD],
     [PSWDL], <AUDIT>.
```

Comments A program can log on to Oracle many times, but if the program logs on using OLON, only one connection can be active at a time. To switch to another connection you must first log off using the OLOGOF procedure, then log on again using OLON. For multiple *concurrent* logons using different LDAs, use the ORLON routine.

OLON cannot be used in a program that mixes OCI calls and Precompiler statements. See the description of SQLLDA in Chapter 5 for more information. To connect to Oracle:

```
DATA DIVISION.
WORKING STORAGE DIVISION
.   77    USED–ID       PIC X(6)  VALUE ”system”
.   77    USER–ID–L     PIC S9(9) VALUE 6 COMP
.   77    PASSWORD      PIC X(7)  VALUE ”manager”
.   77    PASSWORD–L    PIC S9(9) VALUE 7 COMP
.   77    NO–AUDIT      PIC S9(9) VALUE 0 COMP
.   ...
PROCEDURE DIVISION.
..
.   CALL ”OLON” USING LDA, USER–ID, USER–ID–L, PASSWORD,
                      PASSWORD–L, NO–AUDIT
.   ...
```

Parameters

Parameter Name	*Type*	*Status*
LDA	(Address)	IN / OUT
UID	PIC X(n)	IN
UIDL	PIC S9(9) COMP	IN
PSWD	PIC X(n)	IN
PSWDL	PIC S9(9) COMP	IN
AUDIT	PIC S9(9) COMP	IN

LDA
The LDA specified in the OLOG call that was used to make this

connection to Oracle. The *return code* field indicates the result of the OLON call. Zero indicates a successful logon.

UID
A character string containing the user ID and possibly the password. If the password is included as part of the user ID, it must be separated from the user ID by a '/'.

If the user ID is '/' and a valid automatic logon account exists on the database, then the logon succeeds under the user ID of the person running the program.

UIDL
The length of the UID string.

PSWD
A character string containing the password. If the password is specified as part of USER–ID, this parameter can be omitted.

If the UID parameter does not include a password and the PSWD parameter is invalid or omitted, the logon will fail with an error code in the *return code* field of the LDA.

PSWDL
The length of the PSWD string.

AUDIT
This parameter is not currently used. It should be passed as zero.

See Also OLOG.

ONAME

Purpose ONAME retrieves the names of select–list items in a SQL query statement. Use ODESCR in place of ONAME in new Oracle7 OCI programs.

Syntax

```
CALL ”ONAME” USING CURSOR, POS, <TBUF>,
     <TBUFL>, CBUF, CBUFL.
```

Comments ONAME can be called only after the SQL statement has been parsed using OSQL3. The maximum length of an item name that can be returned is 240 bytes.

> **Note:** ONAME obtains the *names* of select–list items, not their values. In the SQL statement “SELECT AVG(SAL) FROM EMP”, there is one select–list item. Its name is “AVG(SAL)”.

ONAME operates by referencing the select–list items by position index. Indices are numbered sequentially from left to right, beginning with 1. If you specify a position number greater than the number of select–list items, ONAME returns a “no data found” status in the *return code* field of the cursor data area.

For queries such as “SELECT * FROM EMP”, in which the number of columns is unknown, you can obtain the column names dynamically by repeatedly calling ONAME until a “no data found” error occurs. For example, the following code fragment shows how to use ONAME to obtain column names:

```
DATA DIVISION.
WORKING STORAGE SECTION.
77   NAME–BUFFER    PIC X(240).
77   SQL–STMT       PIC X(17) VALUE ”SELECT * FROM EMP”.
77   SQL–STMT–LEN   PIC S9(9) VALUE 17 COMP.
77   NAME–BUFFER–L  PIC S9999 COMP.
77   J              PIC S9(9) COMP.
...
PROCEDURE DIVISION.
...
*  Parse the statement.
CALL ”OSQL3” USING CURSOR, SQL–STMT, SQL–STMT–LEN.
*  Get and display the names of each select–list item.
PERFORM DISP–ITEM VARYING J FROM 1 BY 1
    UNTIL C–RC IN CURSOR NOT = 0.
...
DISP–ITEM
.   MOVE 240 TO NAME–BUFFER–L
.   CALL ”ONAME” USING CURSOR, J, OMITTED, OMITTED,
         NAME–BUFFER, NAME–BUFFER–L
```

```
.   DISPLAY NAME–BUFFER.
...
```

Note: ONAME provides only a subset of the functionality of the ODSC and ODESCR routines.

Parameters

Parameter Name	*Type*	*Status*
CURSOR	Address	IN / OUT
POS	PIC S9(9) COMP	IN
TBUF	PIC X(n)	IN
TBUFL	PIC S9(4) COMP	IN
CBUF	PIC X(n)	OUT
CBUFL	PIC S9(4) COMP	IN / OUT

CURSOR
The cursor data area for the query. The ONAME routine uses the cursor address to obtain the names from the parsed SQL statement in the Oracle shared SQL cache.

POS
The position index of the select–list item in the SQL query. The first (or left–most) select–list item is position 1.

TBUF
A character string. This parameter is included only for Oracle Version 2 compatibility. For later versions, it is unused.

TBUFL
This parameter is included only for Oracle Version 2 compatibility. For later versions, it is unused.

CBUF
The name of the select–list item. If CBUFL is passed as 0, the name is not stored. The name returned in CBUF is blank padded to the length specified in the CBUFL input parameter.

CBUFL
Before calling ONAME, CBUFL must be set to the length of CBUF. After ONAME completes, CBUFL contains the length of the name that was returned in CBUF. If CBUFL is not specified or if the value contained in CBUFL is 0, no name is returned in CBUF. If the select–list item name is longer than the value in CBUFL, the returned name is truncated.

See Also ODESCR.

ORLON

Purpose ORLON establishes concurrent communications between a user's program and Oracle through SQL*Net. Use OLOG in place of ORLON in new Oracle7 OCI programs.

Syntax

```
CALL ”ORLON” USING LDA, HDA, UID, UIDL,
     [PSWD], [PSWDL], <AUDIT>.
```

Comments A program can log on to Oracle multiple times. Communication takes place using the logon data area and the host data area defined within the program. The ORLON routine connects the LDA to Oracle.

The *host data area* is a program–allocated data area associated with each ORLON logon call. Its contents are entirely private to Oracle but the HDA must be allocated by the user program, just like the logon data area and cursor data area.

> **Note:** Refer to the section ”Host Data Area” in Chapter 2 for important information about allocating an HDA.

Each concurrent logon requires one LDA–HDA pair. For example:

```
DATA DIVISION.
WORKING STORAGE SECTION.
01  LDA–1.
    02  LDA–V2RC    PIC S9(4) COMP.
    02  FILLER      PIC X(10).
    02  LDA–RC      PIC S9(4) COMP.
    02  FILLER      PIC X(50).
01  LDA–2.
    02  LDA–V2RC    PIC S9(4) COMP.
    02  FILLER      PIC X(10).
    02  LDA–RC      PIC S9(4) COMP.
    02  FILLER      PIC X(50).
01  HDA–1.
    02  FILLER      PIC X(256).
01  HDA–2.
    02  FILLER      PIC X(256).
77  USER–ID–1       PIC X(14)     VALUE
    ”SCOTT/TIGER@T:KERVMS:ORA60”.
77  USER–ID–2       PIC X(14)     VALUE
    ”SYSTEM/MANAGER@T:WRVMS:V60TSTD”.
77  USER–ID–1–L     PIC S9(9) VALUE 26 COMP.
77  USER–ID–2–L     PIC S9(9) VALUE 30 COMP.
77  PASSWORD        PIC X(1).
...
PROCEDURE DIVISION.
...
CALL ”ORLON” USING LDA–1, HDA–1, USER–ID–1, USER–ID–1–L,
     OMITTED, ZERO, ZERO.
IF LDA–RC IN LDA–1 NOT = 0
   PERFORM ERRRPT.
*  And later...

CALL ”ORLON” USING LDA–2, HDA–2, USER–ID–2, USER–ID–2–L,
     OMITTED, ZERO, ZERO.
IF LDA–RC IN LDA–2 NOT = 0
   PERFORM ERRRPT.
```

When the program has issued an ORLON call, a subsequent OLOGOF call commits all outstanding transactions. If a program fails to log off or terminates abnormally, all outstanding transactions are rolled back.

The LDA *return code* field indicates the result of the ORLON call. A zero return code indicates a successful logon.

You should also refer to the section on SQL*Net in the Oracle installation or user's guide for your operating system for any particular notes or restrictions that apply to your operating system.

Parameters

Parameter Name	*Type*	*Mode*
LDA	(Address)	IN / OUT
HDA	PIC X(256)	OUT
UID	PIC X(n)	IN
UIDL	PIC S9(9) COMP	IN
PSWD	PIC X(n)	IN
PSWDL	PIC S9(9) COMP	IN
AUDIT	PIC S9(9) COMP	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

HDA
A host data area. See Chapter 2 for more information on host data areas.

UID
A string containing the user ID, an optional password, and an optional host machine identifier. If you include the password as part of the UID parameter, put it immediately after the user ID and separate it from the user ID with a '/'. Put the host system identifier after the password or user ID, separated by the '@' sign.

If you do not include the password in this parameter, it must be in the PSWD parameter. Examples of valid UID strings are

```
NAME
NAME/PASSWORD
NAME@D:NODENAME:DBNAME
NAME/PASSWORD@D:NODENAME:DBNAME
```

where D is the network identifier, NODENAME is the name of the remote server, and DBNAME is the name of the database instance. The following string is not a correct example of the USERID parameter:

```
NAME@D:NODENAME:DBNAME/PASSWORD
```

UIDL
The length of the UID string.

PSWD
A string containing the password. If the password is specified as part of the UID string, this parameter can be omitted.

PSWDL
The length of the PSWD parameter.

AUDIT
This parameter is not used. Pass it as 0 or –1.

See Also OLOG.

OSQL3

Purpose OSQL3 parses a SQL statement and associates it with a cursor. The OSQL3 routine also executes Data Definition Language statements. Use OPARSE in place of OSQL3 in new Oracle7 OCI programs.

Syntax

```
CALL ”OSQL3” USING CURSOR, SQLSTM, SQLL.
```

Comments OSQL3 passes the SQL statement to Oracle for parsing. The parsed representation of the SQL statement is stored in the shared SQL statement cache in the Oracle server, and state information about the cursor is stored in the private SQL area. Subsequent OCI calls reference the SQL statement using the cursor name. An open cursor may be reused by subsequent OSQL3 calls within a program, or the program may define multiple concurrent cursors when it is necessary to maintain multiple active SQL statements.

The SQL statement may be any valid SQL query, data manipulation, data definition, or data control statement. Oracle parses the statement and selects an optimal access path to perform the requested routine. Data Definition Language statements are executed when OSQL3 is called. Data Manipulation Language statements and queries are not executed until the OEXEC, OEXN, or OEXFET routine is called.

See the description of the OFETCH routine on page 5 – 70 for an example of the OSQL3 routine.

SQL syntax error codes are returned in the cursor data area’s *return code* field and *parse error offset* field. *Parse error offset* indicates the location of the error in the SQL statement text. See the description of the cursor data area on page 2 – 4 for a list of the information fields available in the cursor data area after an OSQL3 call.

Parameters

Parameter Name	*Type*	*Status*
CURSOR	Address	IN / OUT
SQLSTM	PIC X(n)	IN
SQLL	PIC S9(9) COMP	IN

CURSOR
The cursor data area specified in the associated OOPEN call.

SQLSTM
A character string containing a SQL statement. The SQL statement may contain placeholders in any place where a constant is permitted. Placeholders are indicated by placing a colon (:) immediately before the placeholder name. Placeholders are bound to program variables using the OBNDRV or OBNDRN routines.

SQLL
Specifies the length of the SQL statement.

See Also OPARSE.

APPENDIX

F

Additional OCI Routines (FORTRAN)

This appendix describes OCI routines that have been superseded by newer ones in more recent versions of the Oracle7 OCI. The new routines offer increased functionality or performance and should be used in new OCI programs that access Oracle7.

These older routines are still available in Oracle7. However, they might not be available in future versions of the Oracle Call Interface.

ODSC

Purpose ODSC describes select–list items for dynamic SQL queries. The ODSC routine returns internal datatype and size information for a specified select–list item. Use ODESCR in place of ODSC in new Oracle7 OCI programs.

Syntax

```
CALL ODSC(CURSOR, POS, DBSIZE, [FSIZE], [RCODE],
          [DBTYPE], [CBUF], [CBUFL], [DSIZE])
```

Comments You can call ODSC after an OSQL3 or OEXEC call to obtain the maximum size (DBSIZE), internal datatype code (DBTYPE), column name (CBUF), and maximum display size (DSIZE) of select–list items in a query. This routine is used for *dynamic* SQL queries, that is, queries in which the number of select–list items, as well as their datatypes and sizes, may not be known until runtime.

The ODSC routine may also be called after an OFETCH to return the actual size (FSIZE) of the item just fetched.

The *return code* field of the cursor data area indicates success (zero) or failure (non–zero) of the ODSC call.

The ODSC routine uses a position index to refer to select–list items in the SQL query statement. For example, the SQL statement

```
SELECT ename, sal FROM emp WHERE sal > :MIN_SAL
```

contains two select–list items: “ename” and “sal”. The position index of sal is 2, and ename’s index is 1. The following example shows how you would call ODSC to describe the two select–list items in this SQL statement:

```
     INTEGER*2     CURSOR(32)
     INTEGER*2     ENAME_LEN, SAL_LEN, ENAME_TYPE, SAL_TYPE
     INTEGER*2     COL_NAME_LEN, MAX_DISP_LEN
C  The following two variables are for unused optional parameters
     INTEGER*2     FSIZE, RCODE
     CHARACTER*240 COL_NAME
     ..
.    CALL ODSC(CURSOR, 1, ENAME_LEN, FSIZE, RCODE,
    1    ENAME_TYPE, COL_NAME, COL_NAME_LEN, MAX_DISP_LEN)
C  Make use of the values returned..
.    CALL ODSC(CURSOR, 2, SAL_LEN, FSIZE, RCODE,
    1      SAL_TYPE, COL_NAME, COL_NAME_LEN, MAX_DISP_LEN)
```

Note: A dependency exists between the results returned by a describe operation (ODSC) and a bind operation (OBNDRN or OBNDRV). Because a select–list item might contain bind variables, the type returned by ODSC can vary depending on the results of bind operations.

So, if you have placeholders for bind variables in a SELECT statement and you plan to use ODSC to obtain the size or datatype of select–list items, you should do the bind operation before the describe. If you need to rebind any input variables after performing a describe, you must reparse the SQL statement before rebinding. Note that the rebind operation might change the results returned for a select–list item.

Parameters

Parameter Name	*Type*	*Status*
CURSOR	INTEGER*2(32)	IN / OUT
POS	INTEGER*4	IN
DBSIZE	INTEGER*2	OUT
FSIZE	INTEGER*2	IN
RCODE	INTEGER*2	IN 1
DBTYPE	INTEGER*2	OUT
CBUF	CHARACTER*n	OUT
CBUFL	INTEGER*2	IN / OUT
DSIZE	INTEGER*2	OUT

Note 1. Parameter is an OUT after OFETCH or OFEN.

CURSOR
The cursor data area for the SQL statement being described.

POS
The position index of the select–list item in the SQL query. Each item is referenced by position as if numbered left to right consecutively beginning with 1. If you specify a position index greater than the number of items in the select–list or less than 1, ODSC returns a "variable not in select–list" error in the *return code* field of the cursor data area.

DBSIZE
DBSIZE receives the *maximum* size of the column as stored in the Oracle data dictionary. If the column is defined as VARCHAR2, CHAR, or NUMBER, the length returned is the maximum length specified for the column.

FSIZE
FSIZE receives the actual size of the data value returned by the last OFETCH operation. This parameter is only meaningful when ODSC is issued after an OFETCH call. The value returned is the actual length of the value as stored in the database before it is moved to the user buffer where padding or truncation may take place. Oracle suppresses leading zeros on numeric data and trailing blanks on VARCHAR data before storing the values in the database.

RCODE
RCODE receives the column return code returned by the last fetch operation. This parameter is meaningful only when ODSC is issued after an OFETCH call.

> **Note:** It is generally more efficient to establish a column–level RCODE with ODEFIN rather than using ODSC after each fetch. For an example of using column–level return codes, see the example code under the OFETCH description in Chapter 6.

DBTYPE
DBTYPE receives the internal datatype code of the select–list item. See page 3 – 3 for a list of Oracle internal datatype codes and the possible external conversions for each of them.

CBUF
A character buffer in the program that receives the name of the select–list item (name of the column or wording of the expression).

CBUFL
A short integer that is set to the length of CBUF. The column name is truncated if it is longer than CBUFL. If CBUFL is not specified or if the value contained in CBUFL is 0, then the column name is not returned. On return from ODSC, CBUFL contains the length of the returned column or expression name.

DSIZE
DSIZE receives the maximum display size of the select–list item if the select–list item is returned as a character string. The DSIZE parameter is especially useful when SQL functions, like SUBSTR or TO_CHAR, are used to modify the representation of a column.

See Also ODESCR.

OERMSG

Purpose OERMSG returns the Oracle error message text corresponding to the return code parameter RCODE. Use OERHMS in place of OERMSG in V7 OCI programs or when longer error messages must be returned.

Syntax `CALL OERMSG(RCODE, MSG)`

Comments The OERMSG routine is similar to OERHMS, because it returns an Oracle error message text. If there is no message that corresponds to the return code, the message "unknown Oracle error code" is returned.

> **Note:** Programs that have multiple active connections cannot use OERMSG to obtain error message text. Use OERHMS instead. Pro*FORTRAN Precompiler programs that contain OCI calls also must use OERHMS.

The following example is a subroutine that reports Oracle errors:

```
     SUBROUTINE ERRRPT(CURSOR, N)
     INTEGER*2 CURSOR(32), N
     CHARACTER*70 MSG_BUF
     WRITE (*, 100) N, CURSOR(7), CURSOR(6)
100  FORMAT(1X, 'Oracle error on cursor', I3,
    1    ': Code is ', I5, ', OP is ', I5)
     CALL OERMSG(CURSOR(1), MSG_BUF)
     WRITE (*, 200) MSG_BUF
200  FORMAT(1X, A70)
     RETURN
     END
```

Parameters

Parameter Name	*Type*	*Status*
RCODE	INTEGER*2	IN
MSG	CHARACTER*70	OUT

RCODE
The return code value for which the message text is to be returned. This can be either the V2 or the normal return code.

MSG
The address of a program buffer to receive the error message text. The maximum message size that can be returned is 70 characters. If the message is longer than the buffer size or 70 characters, it is truncated.

See Also OERHMS.

OLON

Purpose OLON logs a program on to an Oracle database. Communication between the program and Oracle is established using the logon data area defined in the program. Use OLOG in place of OLON in new Oracle7 OCI programs.

Syntax

```
CALL OLON(LDA, UID, UIDL, [PSWD], [PSWDL], <AUDIT>)
```

Comments A program can log on to Oracle many times, but if the program logs on using OLON, only one connection can be active at a time. To make another connection, you must first log off using the OLOGOF procedure, then log on again using OLON. For multiple *concurrent* logons using different LDAs, use the ORLON routine.

OLON cannot be used in a program that mixes OCI calls and Precompiler statements. See the description of SQLLDA in Chapter 6 for more information.

For example, to log on to Oracle:

```
    INTEGER*2  LDA(32)
    CHARACTER*20 USERID, PASSWD
    ..
.   USERID = ’SCOTT’
    PASSWD = ’TIGER’
    CALL OLON(LDA, USERID, 5, PASSWD, 5, 0)
```

Parameters

Parameter Name	*Type*	*Status*
LDA	INTEGER*2(32)	IN / OUT
USID	CHARACTER*n	IN
UIDL	INTEGER*4	IN
PSWD	CHARACTER*n	IN
PSWDL	INTEGER*4	IN
AUDIT	INTEGER*4	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle. The *return code* field indicates the result of the OLON call. Zero indicates a successful logon.

UID
A character string containing the user ID and possibly the password. If the password is included as part of the user ID, it must be separated from the user ID by a '/'.

If the user ID is '/' and a valid automatic logon account exists on the database, then the logon succeeds under the user ID of the person running the program.

UIDL
The length of the UID string.

PSWD
A character string containing the password. If the password is specified as part of the UID, this parameter can be omitted.

If the UID parameter does not include a password and the PSWD parameter is invalid, null, or omitted, the logon will fail with an error code in the *return code* field of the LDA.

PSWDL
The length of the PSWD array.

AUDIT
This parameter is not currently used. It should be passed as 0 or –1.

See Also OLOG.

ONAME

Purpose ONAME retrieves the names of select–list items in a SQL query statement. Use ODESCR in place of ONAME in new Oracle7 OCI programs.

Syntax `CALL ONAME(CURSOR, POS, TBUF, TBUFL, CBUF, CBUFL)`

Comments ONAME can be called only after the SQL statement has been parsed using OPARSE or OSQL3. The maximum length of an item name that can be returned is 240 bytes.

> **Note:** ONAME obtains the *names* of select–list items, not their values. In the SQL statement "SELECT AVG(sal) FROM emp", there is one select–list item. Its name is "AVG(sal)".

ONAME operates by referencing the select–list items by position index. They are numbered sequentially from left to right, beginning with 1. If you specify a position number greater than the number of select–list items, ONAME returns a "no data found" status in the *return code* field of the cursor data area.

For queries such as "SELECT * FROM emp", in which the number of columns is unknown, you can obtain the column names dynamically by repeatedly calling ONAME until a "no data found" error occurs. For example, the following code fragment uses ONAME to obtain column names:

```
     INTEGER*2       CURSOR(32), NBUFL, TBUFL
     INTEGER         SQLL
     CHARACTER*240   NBUF
     CHARACTER*(*)   SQLSTM, TBUF
     ..
.    PARAMETER(SQLSTM = 'SELECT * FROM EMP')
     SQLL = LEN(SQLSTM)
*  Parse the statement
.    CALL OSQL3(CURSOR, SQLSTM, SQLL)
*  Get the names of each items in the select list
     DO 10 I = 1, 1000000
       NBUFL = 240
       CALL ONAME(CURSOR, I, TBUF, TBUFL, NBUF, NBUFL)
*  Check for error or "no more data"
       IF (CURSOR(7) .NE. 0) GOTO 20
       WRITE (*, 9000) NBUF
9000   FORMAT(1X, A<NBUFL>)
10   CONTINUE
20    ...
```

Observe the WRITE statement in this example carefully. The ONAME routine blank–pads the returned string, but the actual length of the returned name is in the CBUFL parameter on return.

> **Note:** ONAME provides a subset of the functionality of the ODESCR routine.

Parameters

Parameter Name	*Type*	*Status*
CURSOR	INTEGER*2(32)	IN / OUT
POS	INTEGER*4	IN
TBUF	CHARACTER*n	IN
TBUFL	INTEGER*2	IN
CBUF	CHARACTER*n	OUT
CBUFL	INTEGER*2	IN / OUT

CURSOR
The cursor data area for the query. The ONAME routine uses the cursor address to obtain the names from the parsed SQL statement in the Oracle shared SQL cache.

POS
The position index of the select–list item in the SQL query. The left–most select–list item is position 1.

TBUF
This parameter is included only for Oracle Version 2 compatibility. For later versions, it is unused.

TBUFL
This parameter is included only for Oracle Version 2 compatibility.

CBUF
A character buffer within the program that receives the name of the select–list item. If CBUFL is passed as 0, the name is not stored.

The name returned in CBUF is blank padded to the length specified in the CBUFL input parameter.

CBUFL
Before calling ONAME, set CBUFL to the length of the CBUF buffer. After ONAME completes, CBUFL contains the length of the name that was returned in CBUF. If CBUFL is not specified or if the value contained in CBUFL is zero, no name is returned in CBUF. If the select–list item name is longer than the value pointed to by CBUFL, the returned name is truncated.

See Also ODESCR.

ORLON

Purpose ORLON establishes communication between a user's program and Oracle. Use OLOG in place of ORLON in new Oracle7 OCI programs.

Syntax

```
CALL ORLON(LDA, HDA, UID, UIDL, [PSWD],
            [PSWDL], <AUDIT>)
```

Comments An OCI program can connect to Oracle multiple times. Communication takes place using the logon data area and the host data area defined within the program. The ORLON routine connects the LDA to Oracle.

The host data area is a program–allocated data area associated with each ORLON logon call. Its contents are entirely private to Oracle, but the HDA must be allocated by the OCI program.

> **Note:** Refer to the section "Host Data Area" in Chapter 2 for important information about allocating an HDA.

Each concurrent logon requires one LDA–HDA pair. For example:

```
* Establish two logon data areas and host data areas
     INTEGER*2  LDA(32,2), HDA(128,2)
* Declare user ID arrays.
*  (DPWD is a "dummy" for an unused parameter)
     CHARACTER*14 UID1, UID2, DPWD, PWD2
     INTEGER*4 UID1L, UID2L, PWD2L
     ..
.    UID1 = 'SCOTT/TIGER@T:KERVMS:ORA60'
     UID1L = 26
     UID2 = 'SYSTEM@T:WRVMS:V60TSTD'
     UID2L = 22
     PWD2 = 'MANAGER'
     PWD2L = 7
..
.    CALL ORLON(LDA(1,1), HDA(1,1), UID1, UID1L, DPWD, –1, 0))
     IF (LDA(7,1) .NE. 0) THEN
     CALL ERRRPT(LDA(1,1))
...
* and later ..
.    CALL ORLON(LDA(1,2), HDA(1,2), UID2, UID2L, PWD2, PWD2L, 0)
     IF (LDA(7,2) .NE. 0) THEN
     CALL ERRRPT(LDA(1,2))
```

When the program has issued an ORLON call, a subsequent OLOGOF call commits all outstanding transactions. If a program fails to log off or terminates abnormally, all outstanding transactions are rolled back.

The LDA *return code* field indicates the result of the ORLON call. A zero return code indicates a successful logon.

You should also refer to the section on SQL*Net in the Oracle installation or user's guide for your system for any particular notes or restrictions that apply to your operating system.

Parameters

Parameter Name	*Type*	*Mode*
LDA	INTEGER*2(32)	IN / OUT
HDA	INTEGER*2(128)	OUT
UID	CHARACTER*n	IN
UIDL	INTEGER*4	IN
PSWD	CHARACTER*n	IN
PSWDL	INTEGER*4	IN
AUDIT	INTEGER*4	IN

LDA
The LDA specified in the OLOG call that was used to make this connection to Oracle.

HDA
A host data area. See Chapter 2 for more information on host data areas.

UID
A string containing the user ID, an optional password, and an optional host machine identifier. If you include the password as part of the UID parameter, put it immediately after the user ID and separate it from the user ID with a '/'. Put the host system identifier after the password or user ID, separated by the '@' sign.

If you do not include the password in this parameter, it must be in the PSWD parameter. Examples of valid UID strings are

```
NAME
NAME/PASSWORD
NAME@D:NODENAME:DBNAME
NAME/PASSWORD@D:NODENAME:DBNAME
```

where D is the network identifier (for DECNet, in this case), NODENAME is the name of the remote server, and DBNAME is the name of the database instance.

The following string is not a correct example of the USERID parameter:

```
NAME@D:NODENAME:DBNAME/PASSWORD
```

UIDL
The length of the UID string.

PSWD
The string containing the password. If the password is specified as part of the string pointed to by UID, this parameter can be omitted.

PSWDL
The length of the password.

AUDIT
This parameter is not currently used. Pass it as zero or –1.

See Also OLOG.

OSQL3

Purpose OSQL3 parses a SQL statement and associates it with a cursor. The OSQL3 routine also executes Data Definition Language statements. Use OPARSE in place of OSQL3 in new Oracle7 OCI programs.

Syntax

```
CALL OSQL3(CURSOR, SQLSTM, SQLL)
```

Comments OSQL3 passes the SQL statement to Oracle for parsing. The parsed representation of the SQL statement is stored in the Oracle shared SQL statement cache, and state information about the cursor is stored in the private SQL area of the server. Subsequent OCI calls reference the SQL statement using the cursor name. A cursor may be reused by subsequent OSQL3 calls within a program, or the program may define multiple concurrent cursors when it is necessary to maintain multiple active SQL statements.

The SQL statement may be any valid SQL query, data manipulation, data definition, or data control statement. Data Definition Language statements are executed immediately by OSQL3. For Data Manipulation Language statements, the operation is not executed until the OEXEC, OEXN, or OEXFET routine is called.

The following example shows how to call OSQL3:

```
      INTEGER*2         CURSOR(32)
      CHARACTER*(*)     SQL_STATEMENT
      INTEGER*4         SQL_LEN
      ...
C  After connecting and opening the cursor..
.     PARAMETER(SQL_STATEMENT = ’SELECT sname, grade FROM
     1          student_table WHERE sid = :1’)
      CALL OSQL3(CURSOR, SQL_STATEMENT, SQL_LEN)
```

SQL syntax error codes are returned in the cursor data area’s *return code* field and *parse error offset* field. *Parse error offset* indicates the location of the error in the SQL statement text. See the section “Cursor Data Area” on page 2 – 4 for a list of the information fields available in the cursor data area after an OSQL3 call.

Parameters

Parameter Name	*Type*	*Status*
CURSOR	INTEGER*2(32)	IN / OUT
SQLSTM	CHARACTER*n	IN
SQLL	INTEGER*4	IN

CURSOR
The cursor data area specified in the associated OOPEN call.

SQLSTM
A character string containing a SQL statement. The SQL statement may contain placeholders in any place where a constant is permitted. Placeholders, such as “:Employee_number”, are indicated by placing a colon (:) immediately before the placeholder name. Placeholders are bound to program variables using the OBNDRV or OBNDRN routine.

SQLL
Specifies the length of the SQL statement.

See Also OPARSE.

APPENDIX

G

Operating System Dependencies

Some details of OCI programming vary from system to system. For convenience, this appendix collects all references in this guide to system–dependent information.

Chapter 1

Compiler and Linking

Oracle Corporation supplies runtime libraries for most popular third–party compilers. The details of linking an OCI program vary from system to system. See the Oracle system–specific documentation for your system for information on supported compilers on your platform and how to link your OCI application.

Chapter 2

Host Data Area (HDA)

The HDA is 256 bytes long on 32–bit systems only. On 64–bit systems the HDA is typically 512 bytes long. If your system is of a different size, check your Oracle system–specific documentation for the correct size of the HDA.

Many OCI programs, including the demos and sample code in this manual, have defined the HDA as a block of 256 one–byte integers (e.g., **ub1**[256] in C). On some platforms this may cause errors or unpredictable behavior as a result of the integers in the data block not being properly aligned. If your system automatically aligns four–byte integers, you can eliminate the problem by defining the HDA as a block of 64 four–byte integers (e.g., **ub4**[64] in C).

Logon Data Area (LDA)

The layout, field lengths, and byte offsets of the logon data area are hardware and operating–system dependent. C programmers should use the definition of the LDA listed on page A – 8 (and available online in the header file *ocidfn.h*). COBOL and FORTRAN programmers must check the Oracle installation and user's guide for your system for a description of the size of the LDA and the offsets of fields in the LDA.

Cursor Data Area (CDA)

The size of the cursor data area and the layout, field lengths, and byte offsets are hardware and operating–system dependent. The cursor data area is at least 64 bytes in size, but check the Oracle installation or user's guide for your system to see the exact size and configuration of the cursor data area. C programmers should use the definition of the CDA listed on page A – 8 (and available online in the header file *ocidfn.h*). COBOL and FORTRAN programmers must check the Oracle installation and user's guide for your system for a description of the size of the CDA and the offsets of fields in the CDA.

The ROWID field in the cursor data area can differ in size from system to system. This depends on the way that internal fields in a C structure are aligned. On systems that align all fields on byte boundaries, the ROWID field is 13 bytes long. On systems that align fields on different boundaries, the ROWID field can be 16 or more bytes long. Check your Oracle installation or user's guide for the exact length on your system.

Internal ROWID

The length of the ROWID field is system dependent. C programmers should see the listing of the CDA structure on page A – 8. Using the **sizeof** operator on the ROWID substructure returns the correct length of ROWID, which can differ due to different alignment practices from one C compiler to another. COBOL and FORTRAN programmers should see your Oracle installation or user's guide for the length of ROWID on your system.

Deferred Mode Linking

Deferred mode linking is the default. The link time option that you use to select non–deferred mode linking is system dependent. Refer to the Oracle installation or user's guide for your operating system for further information on how to select non–deferred mode linking.

Thread Safety

Thread safety is not available on all platforms. Check your Oracle installation or user's guide for your platform to determine if you can use thread safety.

Chapter 4

Data Structures

In the example code in this section, the *lda_def* and *cda_def* types, as defined in the header file *ocidfn.h*, are used to declare LDAs and CDAs. See page A – 8 for a listing of the *ocidfn.h* file. This file is available on line; see the Oracle installation or user's guide for your system for further information on the location of this file.

olog() function

See the section on SQL*Net in the installation or user's guide for your operating system for any particular notes or restrictions that apply to your operating system.

Chapter 5

Data Structures

The size and offsets of members of the logon data area and cursor data area are system dependent. The Oracle installation and user's guide for your system describes them.

The size of the ROWID part of the CDA is system dependent.

OLOG Routine

You should refer to the section on SQL*Net in the Oracle installation or user's guide for your operating system for any particular notes or restrictions that apply to your operating system.

Chapter 6

Data Structures

The size and offsets of members of the logon data area and cursor data area are system dependent. The Oracle installation and user's guide for your system describes them.

The size of the ROWID part of the CDA is system dependent.

OLOG Routine

You should refer to the section on SQL*Net in the Oracle installation or user's guide for your operating system for any particular notes or restrictions that apply to your operating system.

Appendix A, B, C

File Locations

Each of the sample files in Appendix A, B, and C are available online. The exact name and storage location of these programs is system dependent. See your Oracle installation or user's guide for details.

APPENDIX

H

Oracle Reserved Words, Keywords, and Namespaces

This appendix lists words that have a special meaning to Oracle. Each word plays a specific role in the context in which it appears. For example, in an INSERT statement, the reserved word INTO introduces the tables to which rows will be added. But, in a FETCH or SELECT statement, the reserved word INTO introduces the output host variables to which column values will be assigned.

Oracle Reserved Words

The following words are reserved by Oracle. That is, they have a special meaning to Oracle and so cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes.

ACCESS	ELSE	MODIFY	START
ADD	EXCLUSIVE	NOAUDIT	SELECT
ALL	EXISTS	NOCOMPRESS	SESSION
ALTER	FILE	NOT	SET
AND	FLOAT	NOTFOUND	SHARE
ANY	FOR	NOWAIT	SIZE
ARRAYLEN	FROM	NULL	SMALLINT
AS	GRANT	NUMBER	SQLBUF
ASC	GROUP	OF	SUCCESSFUL
AUDIT	HAVING	OFFLINE	SYNONYM
BETWEEN	IDENTIFIED	ON	SYSDATE
BY	IMMEDIATE	ONLINE	TABLE
CHAR	IN	OPTION	THEN
CHECK	INCREMENT	OR	TO
CLUSTER	INDEX	ORDER	TRIGGER
COLUMN	INITIAL	PCTFREE	UID
COMMENT	INSERT	PRIOR	UNION
COMPRESS	INTEGER	PRIVILEGES	UNIQUE
CONNECT	INTERSECT	PUBLIC	UPDATE
CREATE	INTO	RAW	USER
CURRENT	IS	RENAME	VALIDATE
DATE	LEVEL	RESOURCE	VALUES
DECIMAL	LIKE	REVOKE	VARCHAR
DEFAULT	LOCK	ROW	VARCHAR2
DELETE	LONG	ROWID	VIEW
DESC	MAXEXTENTS	ROWLABEL	WHENEVER
DISTINCT	MINUS	ROWNUM	WHERE
DROP	MODE	ROWS	WITH

Oracle Keywords

The following words also have a special meaning to Oracle but are not reserved words and so can be redefined. However, some might eventually become reserved words.

ADMIN	CURSOR	FOUND	MOUNT
AFTER	CYCLE	FUNCTION	NEXT
ALLOCATE	DATABASE	GO	NEW
ANALYZE	DATAFILE	GOTO	NOARCHIVELOG
ARCHIVE	DBA	GROUPS	NOCACHE
ARCHIVELOG	DEC	INCLUDING	NOCYCLE
AUTHORIZATION	DECLARE	INDICATOR	NOMAXVALUE
AVG	DISABLE	INITRANS	NOMINVALUE
BACKUP	DISMOUNT	INSTANCE	NONE
BEGIN	DOUBLE	INT	NOORDER
BECOME	DUMP	KEY	NORESETLOGS
BEFORE	EACH	LANGUAGE	NORMAL
BLOCK	ENABLE	LAYER	NOSORT
BODY	END	LINK	NUMERIC
CACHE	ESCAPE	LISTS	OFF
CANCEL	EVENTS	LOGFILE	OLD
CASCADE	EXCEPT	MANAGE	ONLY
CHANGE	EXCEPTIONS	MANUAL	OPEN
CHARACTER	EXEC	MAX	OPTIMAL
CHECKPOINT	EXPLAIN	MAXDATAFILES	OWN
CLOSE	EXECUTE	MAXINSTANCES	PACKAGE
COBOL	EXTENT	MAXLOGFILES	PARALLEL
COMMIT	EXTERNALLY	MAXLOGHISTORY	PCTINCREASE
COMPILE	FETCH	MAXLOGMEMBERS	PCTUSED
CONSTRAINT	FLUSH	MAXTRANS	PLAN
CONSTRAINTS	FREELIST	MAXVALUE	PLI
CONTENTS	FREELISTS	MIN	PRECISION
CONTINUE	FORCE	MINEXTENTS	PRIMARY
CONTROLFILE	FOREIGN	MINVALUE	PRIVATE
COUNT	FORTRAN	MODULE	PROCEDURE

Oracle Keywords (continued):

PROFILE	SAVEPOINT	SQLSTATE	TRACING
QUOTA	SCHEMA	STATEMENT_ID	TRANSACTION
READ	SCN	STATISTICS	TRIGGERS
REAL	SECTION	STOP	TRUNCATE
RECOVER	SEGMENT	STORAGE	UNDER
REFERENCES	SEQUENCE	SUM	UNLIMITED
REFERENCING	SHARED	SWITCH	UNTIL
RESETLOGS	SNAPSHOT	SYSTEM	USE
RESTRICTED	SOME	TABLES	USING
REUSE	SORT	TABLESPACE	WHEN
ROLE	SQL	TEMPORARY	WRITE
ROLES	SQLCODE	THREAD	WORK
ROLLBACK	SQLERROR	TIME	

PL/SQL Reserved Words

The following PL/SQL keywords may require special treatment when used in embedded SQL statements.

ABORT	BETWEEN	CRASH	DIGITS
ACCEPT	BINARY_INTEGER	CREATE	DISPOSE
ACCESS	BODY	CURRENT	DISTINCT
ADD	BOOLEAN	CURRVAL	DO
ALL	BY	CURSOR	DROP
ALTER	CASE	DATABASE	ELSE
AND	CHAR	DATA_BASE	ELSIF
ANY	CHAR_BASE	DATE	END
ARRAY	CHECK	DBA	ENTRY
ARRAYLEN	CLOSE	DEBUGOFF	EXCEPTION
AS	CLUSTER	DEBUGON	EXCEPTION_INIT
ASC	CLUSTERS	DECLARE	EXISTS
ASSERT	COLAUTH	DECIMAL	EXIT
ASSIGN	COLUMNS	DEFAULT	FALSE
AT	COMMIT	DEFINITION	FETCH
AUTHORIZATION	COMPRESS	DELAY	FLOAT
AVG	CONNECT	DELETE	FOR
BASE_TABLE	CONSTANT	DELTA	FORM
BEGIN	COUNT	DESC	FROM

PL / SQL Reserved Words (continued):

FUNCTION	NEW	RELEASE	SUM
GENERIC	NEXTVAL	REMR	TABAUTH
GOTO	NOCOMPRESS	RENAME	TABLE
GRANT	NOT	RESOURCE	TABLES
GROUP	NULL	RETURN	TASK
HAVING	NUMBER	REVERSE	TERMINATE
IDENTIFIED	NUMBER_BASE	REVOKE	THEN
IF	OF	ROLLBACK	TO
IN	ON	ROWID	TRUE
INDEX	OPEN	ROWLABEL	TYPE
INDEXES	OPTION	ROWNUM	UNION
INDICATOR	OR	ROWTYPE	UNIQUE
INSERT	ORDER	RUN	UPDATE
INTEGER	OTHERS	SAVEPOINT	USE
INTERSECT	OUT	SCHEMA	VALUES
INTO	PACKAGE	SELECT	VARCHAR
IS	PARTITION	SEPARATE	VARCHAR2
LEVEL	PCTFREE	SET	VARIANCE
LIKE	POSITIVE	SIZE	VIEW
LIMITED	PRAGMA	SMALLINT	VIEWS
LOOP	PRIOR	SPACE	WHEN
MAX	PRIVATE	SQL	WHERE
MIN	PROCEDURE	SQLCODE	WHILE
MINUS	PUBLIC	SQLERRM	WITH
MLSLABEL	RAISE	START	WORK
MOD	RANGE	STATEMENT	XOR
MODE	REAL	STDDEV	
NATURAL	RECORD	SUBTYPE	

Oracle Reserved Namespaces

Table H – 1 contains a list of namespaces that are reserved by Oracle. The initial characters of function names in Oracle libraries are restricted to the character strings in this list. Because of potential name conflicts, use function names that do not begin with these characters.

For example, the SQL*Net Transparent Network Service functions all begin with the characters "NS," so you need to avoid naming functions that begin with "NS."

Namespace	Library
O	OCI functions
S	function names from SQLLIB and system–dependent libraries
XA	external functions for XA applications only
GEN KP L NA NC ND NL NM NR NS NT NZ TTC UPI	Internal functions

Table H – 1 Oracle Reserved Namespaces

Index

Reader's Comment Form

Programmer's Guide to the Oracle Call Interface™
Part No. A32546–1

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

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

If you find any errors or have any other suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Oracle Languages Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065 U.S.A.
Fax: (415) 506–7200

If you would like a reply, please give your name, address, and telephone number below:

Thank you for helping us improve our documentation.