Oracle® Call Interface

Programmer's Guide

Release 9.0.1

June 2001

Part No. A89857-01

ORACLE®

Oracle Call Interface Programmer's Guide, Release 9.0.1

Part No. A89857-01

Copyright © 1996, 2001, Oracle Corporation. All rights reserved.

Primary Authors: Jack Melnick

Contributing Author: Eric Belden, Phil Locke

Contributors: G. Arora, P. Balaji, M. Bastawala, S. Chandrasekaran, D. Chatterjee, E. Chen, L. Chidambaran, S. Gollapudi, G. Govindarajan, W. He, T. Hoang, R.-S. Hwu, M. Joglekar, S. Kaluskar, R. Kambo, R. Kasamsetty, A. Katti, S. Kotsovolos, V. Krishnamurthy, S. Krishnaswamy, R. Krishnan, S. Kumar, S. Lari, C. Lei, C. Lim, Meghna Mehta, Sambit Mishra, Shailendra Mishra, P. Mitra, K. Mohan, R. Murthy, R. Ratnam, P. Reilly, A. Roy, D. Saha, E. Soylemez, A. Surpur, A. Swaminathan, P. Tucker, S. Vedala, Wei Wang, R. Wessman, D. Wong

Graphic Designer: Valarie Moore

Contents

Part I   OCI Concepts

1   Introduction and Upgrading

2   OCI Programming Basics

3   Datatypes

4   Using SQL Statements in OCI

5   Binding and Defining

6   Describing Schema Metadata

7 LOB and FILE Operations

8 Managing Scalable Platforms

9   OCI Programming Advanced Topics

Part II   OCI Object Concepts

10   OCI Object-Relational Programming

11 Object-Relational Datatypes

12   Direct Path Loading

13   Object Cache and Object Navigation

14 The Object Type Translator (OTT)

Part III   OCI Reference

15   OCI Relational Functions

16 More OCI Relational Functions

17   OCI Navigational and Type Functions

18   OCI Datatype Mapping and Manipulation Functions

19   OCI Cartridge Functions

20 OCI Any Type and Data Functions

Part IV   Appendixes

A   Handle and Descriptor Attributes

B OCI Demonstration Programs

C OCI Function Server Round-trips

Index

Send Us Your Comments

Oracle Call Interface Programmer's Guide, Release 9.0.1

Part No. A89857-01

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

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

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

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227   Attn: Information Development Department
- Postal service:
  Oracle Corporation
  Information Development Department
  500 Oracle Parkway MS 4op11
  Redwood Shores, CA 94065
  USA

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

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

Preface

The Oracle Call Interface (OCI) is an application programming interface (API) that allows applications written in C or C++ to interact with one or more Oracle database servers. OCI gives your programs the capability to perform the full range of database operations that are possible with an Oracle database server, including SQL statement processing and object manipulation.

The Preface includes the following sections:

- Audience
- Organization
- Related Documentation
- Conventions
- Documentation Accessibility

Audience

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

This guide assumes that you have a working knowledge of application programming using C. Readers should also be familiar with the use of Structured Query Language (SQL) to access information in relational database systems. In addition, some sections of this guide also assume a knowledge of the basic concepts of object-oriented programming.

> **See Also:**
>
> - For information about SQL, refer to the *Oracle9i SQL Reference* and the *Oracle9i Database Administrator's Guide.*
>
> - For information about basic Oracle concepts, see *Oracle9i Database Concepts.*
>
> - For information about the differences between the Standard Edition and the Enterprise Edition and all the features and options that are available to you, see *Oracle9i Database New Features.*

Organization

The *Oracle Call Interface Programmer's Guide* contains four parts, split between two volumes. A brief summary of what you will find in each chapter and appendix follows:

PART I: OCI CONCEPTS

Part I (Chapter 1 through Chapter 9) provides conceptual information about how to program with OCI to build scalable application solutions that provide access to relational data in an Oracle database.

Chapter 1, "Introduction and Upgrading"

This chapter introduces you to the Oracle Call Interface and describes special terms and typographical conventions that are used in describing the interface. This chapter also discusses features new to the current release.

Chapter 2, "OCI Programming Basics"

This chapter gives you the basic concepts needed to develop an OCI program. It discusses the essential steps each OCI program must include, and how to retrieve and understand error messages

Chapter 3, "Datatypes"

Understanding how data is converted between Oracle tables and variables in your host program is essential for using OCI. This chapter discusses Oracle internal and external datatypes, and data conversions.

Chapter 4, "Using SQL Statements in OCI"

This chapter discusses the steps involved in SQL statements using OCI.

Chapter 5, "Binding and Defining"

This chapter discusses OCI bind and define operations in detail, including a discussion of advanced bind and define operations.

Chapter 6, "Describing Schema Metadata"

This chapter discusses how to use the `OCIDescribeAny()` call to obtain information about schema objects and their associated elements.

Chapter 7, "LOB and FILE Operations"

This chapter describes OCI support for LOB, FILE, and temporary LOB datatypes.

Chapter 8, "Managing Scalable Platforms"

This chapter describes password management, session management, and thread safety.

Chapter 9, "OCI Programming Advanced Topics"

This chapter covers more advanced OCI programming topics, including the OCI thread support, descriptions of user callbacks, application failover callbacks, advanced queuing, and publish-subscribe notification.

PART II: OCI OBJECT CONCEPTS

Part II (Chapter 10 through Chapter 14) describes OCI functionality for accessing object-relational data with OCI.

Chapter 10, "OCI Object-Relational Programming"

This chapter provides an introduction to the concepts involved when using OCI to access objects in an Oracle database server. The chapter includes a discussion of basic object concepts and object navigational access, and the basic structure of object-relational applications.

Chapter 11, "Object-Relational Datatypes"

This chapter outlines the object datatypes used in OCI programming. This chapter discusses the C mappings of user-defined datatypes in an Oracle database, and the functions that manipulate such data. Binding and defining using these C mappings is also covered.

Chapter 12, "Direct Path Loading"

Discusses loading of data from files into scalar and object columns.

Chapter 13, "Object Cache and Object Navigation"

This chapter provides an introduction to the concepts involved when using OCI to access objects in an Oracle database server. This chapter also discusses the Object Cache, and the use of OCI navigational calls to manipulate objects retrieved from the server.

Chapter 14, "The Object Type Translator (OTT)"

This chapter discusses the use of the Object Type Translator to convert database object definitions to C structure representations for use in OCI applications.

PART III: OCI REFERENCE

Part III (Chapter 15 through Chapter 20) lists OCI function calls in the OCI library.

Chapter 15, "OCI Relational Functions"

This chapter contains a list of OCI relational functions, including syntax, comments, parameter descriptions, and other useful information.

Chapter 16, "More OCI Relational Functions"

Continues the OCI relational functions started in the last chapter. It covers LOB, threads, transaction management and miscellaneous functions.

Chapter 17, "OCI Navigational and Type Functions"

This chapter contains a list of OCI navigational functions, including syntax, comments, parameter descriptions, and other useful information.

Chapter 18, "OCI Datatype Mapping and Manipulation Functions"

This chapter contains a list of OCI datatype mapping and manipulation functions, including syntax, comments, parameter descriptions, and other useful information.

Chapter 19, "OCI Cartridge Functions"

This chapter discusses special OCI functions used by external procedures and cartridge functions.

Chapter 20, "OCI Any Type and Data Functions"

This chapter describes the OCI Any Type and Data functions.

PART IV: APPENDIXES

Part IV (Appendix A through Appendix C) provides additional reference information about OCI programming.

Appendix A, "Handle and Descriptor Attributes"

This appendix describes the attributes of OCI application handles that can be set or read using OCI calls.

Appendix B, "OCI Demonstration Programs"

This appendix gives the names of important OCI demonstration programs that are included with the Oracle installation.

Appendix C, "OCI Function Server Round-trips"

This appendix includes tables which show the estimated number of server roundtrips required by various OCI applications.

Where to Begin

Because of the many enhancements to OCI, both new and experienced users should read the conceptual material in Part I.

Readers familiar with the current version of OCI and interested in its object capabilities may want to skim Part 1 and then begin reading the chapters in Part II.

Readers looking for reference information, such as OCI function syntax and handle attribute descriptions, should refer to Part III and Part IV.

Related Documentation

In North America, printed documentation is available for sale in the Oracle Store at

```
http://oraclestore.oracle.com/
```

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

```
http://www.oraclebookshop.com/
```

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

```
http://technet.oracle.com/membership/index.htm
```

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

```
http://technet.oracle.com/docs/index.htm
```

The *Oracle Call Interface Programmer's Guide* does not contain all information that describes the features and functionality of OCI in the Standard Edition and the Enterprise Edition products.

Oracle C++ Call Interface

For C++ programmers, the Oracle C++ Call Interface provides OCI functionality for C++ programs and lets you manipulate database objects (of user-defined types) as C++ objects.

Other Sources of Information about OCI

For other sources of information about OCI:

See Also:

- For information about the C++ Call interface, refer to *Oracle C++ Call Interface Programmer's Guide.*

- For information about cartridge services, and the OCI calls pertaining to development of data cartridges, refer to *Oracle9i Data Cartridge Developer's Guide.*

- For information about OCI calls pertaining to National Language and Globalization Support, see the chapter "OCI Programming" in *Oracle9i Globalization and National Language Support Guide.*

- For information about OCI calls pertaining to Advanced Queuing, see *Oracle9i Application Developer's Guide - Advanced Queuing.*

- For information about using OCI with the XA library, see *Oracle9i Application Developer's Guide - Fundamentals.*

- For information about using OCI calls to manipulate LOBs, including code examples, see *Oracle9i Application Developer's Guide - Large Objects (LOBs).*

- For a more detailed explanation of object types, see *Oracle9i Application Developer's Guide - Object-Relational Features.*

- For a list of Frequently Asked Questions (FAQs) and a tutorial on OCI, see the web site at

  http://www.oracle.com/

  and select Technologies, OCI, Tech Info.

Conventions

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

. . .
In code fragments, an ellipsis means that code not relevant to the discussion has been omitted. In syntax, an ellipsis means that the previous item can be repeated.

```
monospace font
```
SQL and C code examples, OCI function names, database objects, packages, usernames, file names, and directory names are shown in monospace font. Syntax examples are in monospace font also.

italics
Italics are also used for emphasis and for the titles of documents.

```
monospace italic
```
Monospace italics are used for OCI parameters, and user-supplied data fields, when used in body text. Plain font is used for these items when used in tables and in lists.

```
MONOSPACE UPPERCASE
```
Monospace uppercase is used for SQL or PL/SQL keywords, such as `SELECT` or `UPDATE`.

> **See Also:** *Oracle9i SQL Reference* and the *PL/SQL User's Guide and Reference* to see the lists of the keywords and reserved words for SQL and PL/SQL

bold
Boldface type is used to identify the names of C datatypes, such as **ub4**, **sword**, or **OCINumber**. Bold is sometimes used in code examples for emphasis.

This guide uses special text formatting to draw the reader's attention to some information. A paragraph that is indented and begins with a bold text label may have special meaning. The following paragraphs describe the different types of information that are flagged this way.

> **Note:** The **Note** flag indicates that the reader should pay particular attention to the information to avoid a common problem or increase understanding of a concept.

> **7.x Upgrade Note:** An item marked with "7.x Upgrade Note" typically alerts the programmer to something that is done much differently in the releases 8 and later OCI than in the 7.x OCIs.

> **Caution:**   An item marked **Caution** indicates something that an OCI programmer must be careful to do or not do in order for an application to work correctly.

> **See Also:**   Text marked **See Also** points you to another section of this guide, or to other documentation, for additional information about the topic being discussed.

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

```
http://www.oracle.com/accessibility/
```

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

What's New in Oracle Call Interface?

The following sections describe the new features in Oracle Call Interface:

- Oracle9i Release 1 (9.0.1) New Features in Oracle Call Interface
- Oracle9i Release 9.0.0 New Features in Oracle Call Interface
- Oracle8i Release 2 (8.1.6) New Features in Oracle Call Interface

Oracle9*i* Release 1 (9.0.1) New Features in Oracle Call Interface

- **Defining LOB Output Variables**

This section has been re-written.

> **See Also:** "Defining LOB Output Variables" on page 5-22

- **Supporting UTF-16 Unicode**

This discussion has been re-written.

> **See Also:**
>
> - "Globalization Support" on page 2-46
> - "Character Conversion Issues in Binding and Defining" on page 5-35
> - "Bind Handle Attributes" on page A-32
> - "Define Handle Attributes" on page A-35

- **Advanced Queuing**

Changes were made in the interfaces of publish-subscribe notification and in the OCI function OCISubscriptionRegister(). Several subscription handle attributes were modified and several were added. Open registration for publish-subscribe has been added.

> **See Also:**
>
> - "Publish-Subscribe Notification" on page 9-46
> - "Publish-Subscribe Registration Functions" on page 9-47
> - "OCISubscriptionRegister()" on page 16-91
> - "Subscription Handle Attributes" on page A-53
> - "Publish-Subscribe Registration Functions" on page 9-47
> - "OCIServerDNs Descriptor Attributes" on page A-51
> - "Environment Handle Attributes" on page A-4
> - "Publish-Subscribe LDAP Registration Example" on page 9-59

- **Direct Path Loading**

Direct path loading of data into object columns as well as scalar columns, is now supported. Direct path loading is moved to chapter 12, so that it now appears after the discussion of objects and their use. Sections on binding and defining object datatypes are now at the end of chapter 11.

> **See Also:**
> - Chapter 12, "Direct Path Loading"
> - "Direct Path Loading Handle Attributes" on page A-58

Oracle9*i* Release 9.0.0 New Features in Oracle Call Interface

This document has these new features. Each of these features is discussed in greater detail in the cross-referenced sections:

- **Connection Pooling**

This feature allows you to use many logical connections in a single physical connection.

> **See Also:**
> - "Connection Pooling" on page 9-14
> - "Connect, Authorize, and Initialize Functions" on page 15-4
> - "Connection Pool Handle Attributes" on page A-21

- **Scrollable cursors.**

Members of a result set can be accessed in non-sequential order.

> **See Also:**
> - "Scrollable Cursors" on page 4-18
> - "Statement Functions" on page 15-83

- **Globalization support.**

Various OCI calls support UTF-16 for SQL statements, data, metadata, objects, and error messages.

> **See Also:** "Globalization Support" on page 2-46

- **Middle-tier applications.**

New attributes have been added for client authentication.

> **See Also:** "Middle-tier Applications" on page 8-15

- **New datatypes.**

Datetime and Interval and Daylight Savings datatypes are described in the following sections:

> **See Also:**
> - "Datetime and Interval Datatype Descriptors" on page 3-23
> - "Datetime and Interval (OCIDateTime, OCIInterval)" on page 11-9
> - "Data Conversions for Datetime and Interval Datatypes" on page 3-29
> - and "OCI Date, Datetime, and Interval Functions" on page 18-28

- **Any Type, AnyData, AnyDataSet.**

An **OCIAnyData** encapsulates type information as well as a data instance of that type (i.e. self descriptive data). An **OCIAnyDataSet** encapsulates type information as well as a set of instances of that type.

> **See Also:** "Any Type, AnyData and AnyDataSet Interfaces" on page 11-29 and the corresponding new functions in Chapter 20, "OCI Any Type and Data Functions"

- **Using LOB columns instead of LONG columns.**

> **See Also:** "Runtime Data Allocation and Piecewise Operations" on page 5-44 has been rewritten with new features for support of LOBs.

- **Subtypes of objects can be defined.**

    See Also:

    - "Type Inheritance" on page 10-37

    - "OTT Support for Type Inheritance" on page 14-17

- **Type evolution.**

How the attributes of types can be changed.

    See Also: "Type Evolution" on page 10-42

- **Multilevel collection types.**

Collections whose elements are collections.

    See Also: "Multilevel Collection Types" on page 11-26

- **Externally initialized context.**

An externally initialized context is an application context whose attributes can be initialized from OCI.

    See Also:

    - "Externally Initialized Context" on page 8-22

    - "User Session Handle Attributes" on page A-17

- **Caching.**

    See Also: "Oracle9i Application Server Cache" on page 8-26 for this performance-enhancing feature

- **Structural changes to this document:**

    - Chapter 15 of release 8.1.6 has been split into chapters 15 and 16.

    - The sections in chapters 15 and 16 have been rearranged in a more logical order.

    - Chapters 17, 18, 19 were chapters 16, 17, 18 in release 8.1.6.

    - Chapter 20 has been added.

Oracle8*i* Release 2 (8.1.6) New Features in Oracle Call Interface

The 8.1 releases of OCI have the following new features and performance advantages:

- A revised callback mechanism has been implemented.

- A discussion of middle-tier authentication attributes is now in this guide.

- Cartridge service functions are now documented in this guide.

- Ability to create new object with non-NULL attribute values.

- Support for universal ROWIDs.

- Support for fixed-width Unicode.

- OCIThread package for thread manipulation.

- Ability to register user-created callback functions.

- Enhanced application failover processing ability.

- Support for publish/subscribe notification.

- No-wait locking option for objects.

- Ability to detect object changes when flushing.

- Support for temporary LOBs.

- Enhancements to LOB support.

- Enhanced array DML statement execution allowing all errors to be returned in a batch.

- Enhanced DML...RETURNING support.

- Ability to create objects based on object views or user-created object IDs.

- Support for non-blocking mode.

- Additional functional and performance enhancements.

- Publish-subscribe functionality for client notification of events.

- Direct path loading calls that provide access to the direct block formatter of the Oracle server.

- Reduced memory usage at runtime.

- Increased runtime performance with code reduction.

- Increased query performance with streamlined and more efficient fetch protocol.

Part I

OCI Concepts

This part of the guide contains chapters that describe OCI programming concepts:

- Chapter 1, "Introduction and Upgrading", provides an introduction to the OCI and discusses features that are new to this release.

- Chapter 2, "OCI Programming Basics", discusses the basic concepts of OCI programming.

- Chapter 3, "Datatypes", describes datatypes used in OCI applications and within the server.

- Chapter 4, "Using SQL Statements in OCI", discusses how to process SQL statements using OCI.

- Chapter 5, "Binding and Defining", discusses bind and define operations in detail.

- Chapter 6, "Describing Schema Metadata", discusses the OCIDescribeAny() function.

- Chapter 7, "LOB and FILE Operations", discusses the OCI functions that perform operations on large objects (LOBs) in a database and external LOBs.

- Chapter 8, "Managing Scalable Platforms", discusses password and session management, middle-tier applications, and externally initialized context.

- Chapter 9, "OCI Programming Advanced Topics", covers advanced topics in OCI programming, such as threads, connection pooling, user-defined callbacks, advanced queuing, and publish-subscribe notification.

1

Introduction and Upgrading

This chapter introduces you to the Oracle Call Interface (OCI). It provides background information that you need to develop applications using OCI. This chapter also introduces special terms that are used in discussing OCI.

The following topics are covered:

- Overview of OCI
- Compatibility, Upgrading, and Migration

Overview of OCI

The Oracle Call Interface (OCI) is an application programming interface (API) that lets you create applications that use the native procedures or function calls of a third-generation language to access an Oracle database server and control all phases of SQL statement execution. OCI supports the datatypes, calling conventions, syntax, and semantics of C and C++.

See Also:

- For information about the C++ Call interface, refer to *Oracle C++ Call Interface Programmer's Guide*

- "Other Sources of Information about OCI" on page xl

OCI provides:

- Improved performance and scalability through the efficient use of system memory and network connectivity.

- Consistent interfaces for dynamic session and transaction management in a two-tier client-server or multitier environment.

- N-tiered authentication.

- Comprehensive support for application development using Oracle objects.

- Access to external databases.

- Applications that can service an increasing number of users and requests without additional hardware investments.

OCI lets you manipulate data and schemas in an Oracle database using a host programming language, such as C. It provides a library of standard database access and retrieval functions in the form of a dynamic runtime library (OCI library) that can be linked in an application at runtime. This eliminates the need to embed SQL or PL/SQL within 3GL programs.

OCI has many new features that can be categorized into several primary areas:

- Encapsulated/opaque (meaning that the details of which are unknown) interfaces.

- Simplified user authentication and password management.

- Extensions to improve application performance and scalability.

- Consistent interface for transaction management.

- OCI extensions to support client-side access to Oracle objects.

Advantages of OCI

OCI provides significant advantages over other methods of accessing an Oracle database:

- More fine-grained control over all aspects of the application design.

- High degree of control over program execution.

- Use of familiar 3GL programming techniques and application development tools such as browsers and debuggers.

- Support of dynamic SQL.

- Availability on the broadest range of platforms of all the Oracle programmatic interfaces.

- Dynamic bind and define using callbacks.

- Describe functionality to expose layers of server metadata.

- Asynchronous event notification for registered client applications.

- Enhanced array data manipulation language (DML) capability for array inserts, updates, and deletes.

- Ability to associate a commit request with an execute to reduce roundtrips.

- Optimization for queries using transparent prefetch buffers to reduce roundtrips.

- Thread safety; you do not have to use mutual exclusive locks (mutexes) on OCI handles.

Building an OCI Application

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.

Figure 1–1   The OCI Development Process
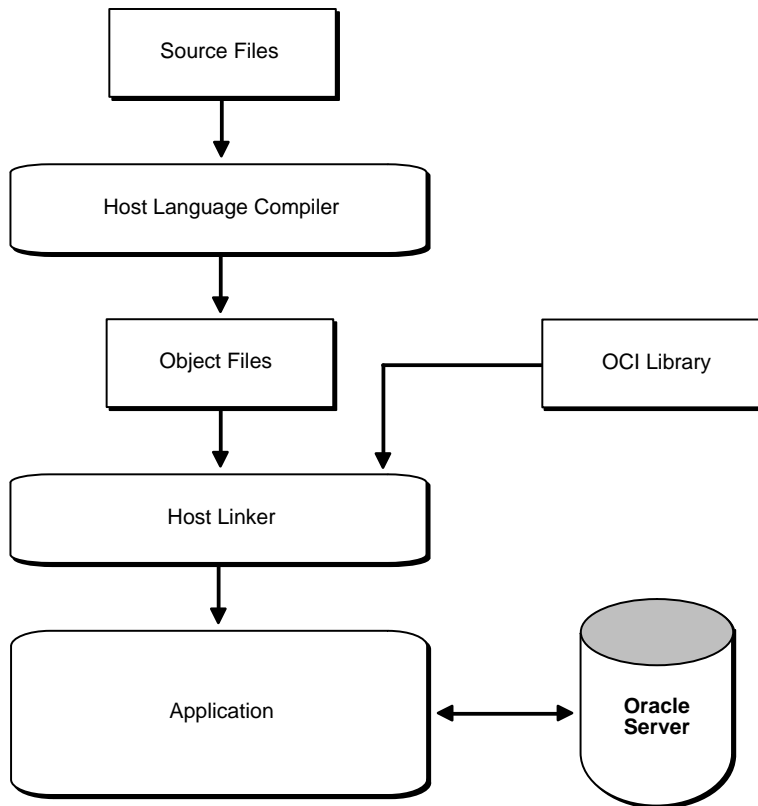
Oracle supports most popular third-party compilers. The details of linking an OCI program vary from system to system. On some platforms, it may be necessary to include other libraries, in addition to the OCI library, to properly link your OCI programs. See your Oracle system-specific documentation and the installation guide for more information about compiling and linking an OCI application for your specific platform.

Parts of OCI

OCI has this functionality:

- APIs to design a scalable, multithreaded application that can support large numbers of users securely.

- SQL access functions, for managing database access, processing SQL statements, and manipulating objects retrieved from an Oracle database server.

- Datatype mapping and manipulation functions, for manipulating data attributes of Oracle types.

- Data loading functions, for loading data directly into the database without using SQL statements.

- External procedure functions, for writing C callbacks from PL/SQL.

Procedural and Non-Procedural Elements

The Oracle Call Interface (OCI) lets you develop scalable, multithreaded applications on multitier architecture that combine the non-procedural data access power of Structured Query Language (SQL) with the procedural capabilities of C and C++.

- In a non-procedural language program, the set of data to be operated on is specified, but what operations will be performed, or how the operations are to be carried out is not specified. The non-procedural nature of SQL makes it an easy language to learn and to use to perform database transactions. It is also the standard language used to access and manipulate data in modern relational and object-relational database systems.

- In a procedural language program, the execution of most statements depends on previous or subsequent statements and on control structures, such as loops or conditional branches, which are not available in SQL. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.

The combination of both non-procedural and procedural language elements in an OCI program provides easy access to an Oracle database in a structured programming environment.

OCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through an Oracle database server. For example, an OCI program can run a query against an Oracle database. The queries can require the program to supply data to the database using input (bind) variables, as follows:

```
SELECT name FROM employees WHERE empno = :empnumber
```

In the above SQL statement, :empnumber is a placeholder for a value that will be supplied by the application.

You can also take advantage of PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. OCI also provides facilities for accessing and manipulating objects in an Oracle database server.

Object Support

OCI has facilities for working with *object types* and *objects.* An object type is a user-defined data structure representing an abstraction of a real-world entity. For example, the database might contain a definition of a `person` object. That object might have *attributes*—`first_name`, `last_name`, and `age`—which represent a person's identifying characteristics.

- The object type definition serves as the basis for creating objects, which represent instances of the object type. Using the object type as a structural definition, a `person` object could be created with the attributes 'John', 'Bonivento', and '30'. Object types may also contain *methods*—programmatic functions that represent the behavior of that object type.

> **See Also:** For a more detailed explanation of object types and objects, see *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Object-Relational Features.*

OCI includes functions that extend the capabilities of OCI to handle objects in an Oracle database server. Specifically, the following capabilities have been added to OCI:

- Support for execution of SQL statements that manipulate object data and schema information.

- Support for passing object references and instances as input variables in SQL statements.

- Support for declaring object references and instances as variables to receive the output of SQL statements.

- Support for fetching object references and instances from a database.

- Support for describing the properties of SQL statements that return object instances and references.

- Support for describing PL/SQL procedures or functions with object parameters or results.

- Commit and rollback calls have been extended to synchronize object and relational functionality.

Additional OCI calls are provided to support manipulation of objects after they have been accessed by way of SQL statements. For a more detailed description of enhancements and new features, refer to "Encapsulated Interfaces" on page 1-12.

SQL Statements

One of the main tasks of an OCI application is to process SQL statements. Different types of SQL statements require different processing steps in your program. It is important to take this into account when coding your OCI application. Oracle recognizes several types of SQL statements:

- Data Definition Language
- Control Statements
    - Transaction Control
    - Session Control
    - System Control
- Data Manipulation Language (DML)
- Queries

> **Note:** Queries are often classified as DML statements, but OCI applications process queries differently, so they are considered separately here.

- PL/SQL.
- Embedded SQL.

> **See Also:** Chapter 4, "Using SQL Statements in OCI"

Data Definition Language

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

The following is an example of creating and specifying access to a table:

```
CREATE TABLE employees
    (name        VARCHAR2(20),
     ssn         VARCHAR2(12),
     empno       NUMBER(6),
     mgr         NUMBER(6),
     salary      NUMBER(6))

GRANT UPDATE, INSERT, DELETE ON employees TO donna
REVOKE UPDATE ON employees FROM jamie
```

DDL statements also allow you to work with objects in the Oracle database server, as in the following series of statements which creates an object table:

```
CREATE TYPE person_t AS OBJECT (
    name     VARCHAR2(30),
    ssn      VARCHAR2(12),
    address  VARCHAR2(50))

CREATE TABLE person_tab OF person_t
```

Control Statements

- OCI applications treat transaction control, session control, and system control statements like DML statements.

    See Also: *Oracle9i SQL Reference* for information about these types of statements

Data Manipulation Language

Data manipulation language (DML) statements can change data in the database tables. For example, DML 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 an application to supply data to the database using input (bind) variables.

> **See Also:** "What is Binding?" on page 4-6 for more information
> about input bind variables

DML statements also allow you to work with objects in the Oracle database server, as in the following example, which inserts an instance of type `person_t` into the object table `person_tab`:

```
INSERT INTO person_tab
    VALUES (person_t('Steve May','123-45-6789','146 Winfield Street'))
```

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, as in the following example:

```
SELECT dname FROM dept
    WHERE deptno = 42
```

Queries access data in tables, and they are often classified with DML statements. However, OCI applications process queries differently, so they are considered separately in this guide.

Queries can require the program to supply data to the database using input (bind) variables, as in the following example:

```
SELECT name
    FROM employees
    WHERE empno = :empnumber
```

In the above SQL statement, `:empnumber` is a placeholder for a value that will be supplied by the application.

- When processing a query, an OCI application also needs to define output variables to receive the returned results. In the above statement, you would need to define an output variable to receive any `name` values returned from the query.

See Also:

- "Binding" on page 5-2 for more information about input bind variables. See the section "Defining" on page 5-19 for information about defining output variables.

- See Chapter 4, "Using SQL Statements in OCI", for detailed information about how SQL statements are processed in an OCI program.

PL/SQL

PL/SQL is Oracle's procedural 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 are:

- One or more SQL statements.

- Variable declarations.

- Assignment statements.

- Procedural control statements (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.

- Access and manipulate objects in an Oracle database server.

The following PL/SQL example issues a SQL statement to retrieve values from a table of employees, given a particular employee number. This example also demonstrates the use of placeholders in PL/SQL statements.

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

```
END;
```

- Note that the placeholders in this statement are not PL/SQL variables. They represent input values passed to Oracle when the statement is processed. These placeholders need to be bound to C language variables in your program.

    See Also:

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

    - See the section "Binding Placeholders in PL/SQL" on page 5-5 for information about working with placeholders in PL/SQL.

Embedded SQL

OCI processes SQL statements as text strings, which an application passes to Oracle on execution. The Oracle precompilers (Pro*C/C++, Pro*COBOL, Pro*FORTRAN) allow you to embed SQL statements directly into your application code. A separate precompilation step is then necessary to generate an executable application.

- It is possible to mix OCI calls and embedded SQL in a precompiler program.

    See Also: Refer to the *Pro*C/C++ Precompiler Programmer's Guide* for more information.

Special OCI/SQL Terms

This guide uses special terms to refer to the different parts of a SQL statement. For example, a SQL statement such as

```
SELECT customer, address
FROM customers
WHERE bus_type = 'SOFTWARE'
AND sales_volume = :sales
```

contains the following parts:

- A SQL *command* — SELECT.

- Two *select-list items* — customer and address.

- A *table name* in the FROM clause — customers.

- Two *column names* in the WHERE clause — bus_type and sales_volume.

- A *literal input value* in the WHERE clause — 'SOFTWARE'.

- A *placeholder* for an input variable in the `WHERE` clause — `:sales`.

When you develop your OCI application, you call routines that specify to the Oracle database server the address (location) of input and output variables in your program. In this guide, specifying the address of a placeholder variable for data input is called a *bind operation*. Specifying the address of a variable to receive select-list items is called a *define operation*.

For PL/SQL, both input and output specifications are called bind operations. These terms and operations are described in Chapter 4, "Using SQL Statements in OCI".

Encapsulated Interfaces

All the data structures that are used by OCI are encapsulated in the form of opaque interfaces that are called handles. A handle is an opaque pointer to a storage area allocated by the OCI library that stores context information, connection information, error information, or bind information about a SQL or PL/SQL statement. A client allocates a certain type of handle, populates one or more of those handles through well-defined interfaces, and sends requests to the server using those handles. In turn, applications can access the specific information contained in the handle by using accessor functions. The OCI library manages a hierarchy of handles. Encapsulating the OCI interfaces using these handles has several benefits to the application developer including:

- Reduction in the amount of server side state information that needs to be retained thereby reducing server side memory usage.

- Improved application developer productivity by eliminating the need for global variables, making error reporting easier and providing consistency in the way OCI variables are accessed and used.

- Further, the encapsulation of OCI structures in the form of handles makes them opaque, allowing changes to be made to the underlying structure without affecting applications.

Simplified User Authentication and Password Management

OCI provides application developers simplified user authentication and password management in several ways:

- Allows a single OCI application to authenticate and maintain multiple users.

- Allows the application to update a user's password which is particularly helpful if an expired password message is returned by an authentication attempt.

OCI supports two types of login sessions:

- A simplified login function for sessions where a single user connects to the database using a login name and password.

- A setup in which a single OCI application authenticates and maintains multiple sessions by separating the login session, which is the session created when a user logs into an Oracle database, from the user sessions, which are all other sessions created by a user. This is an important difference from Oracle 7.3, in which sessions could be created implicitly by starting new transactions once the user has logged in to the database, a process called *session cloning*. These *user* sessions in Oracle 7.3 inherited the privileges and security context from the login session. OCI requires a client to provide all the necessary authentication information for each user session. This allows an OCI application to support multiple users.

Extensions to Improve Application Performance and Scalability

OCI has several enhancements to improve application performance and scalability. Application performance has been improved by reducing the number of client to server round trips required and scalability improvements have been facilitated by reducing the amount of state information that needs to be retained on the server side. Some of these features include:

- Increased client-side processing, and reduced server-side requirements on queries.

- Implicit prefetching of SELECT statement result sets to eliminate the describe round trip, reduce roundtrips, and reduce memory usage.

- Elimination of open and close cursor round trips.

- Improved support for multithreaded environments.

- Session multiplexing over connections.

- Consistent support for a variety of configurations including standard 2-tier client-server configurations, server-to-server transaction coordination, and 3-tier TP-monitor configurations.

- Consistent support for local and global transactions including support for the XA interface's TM_JOIN operation.

- Improved scalability by providing the ability to concentrate connections, processes, and sessions across users on connections and eliminating the need for separate sessions to be created for each branch of a global transaction.

- Allowing applications to authenticate multiple users and allow transactions to be started on their behalf.

OCI Object Support

OCI provides the most comprehensive application programming interface for programmers seeking to use the Oracle server's object capabilities. These features can be divided into five major categories:

- Client-Side Object Cache.

- Associative and navigational interfaces to access and manipulate objects.

- Runtime environment for objects.

- Type management functions to access information about object types in an Oracle database.

- Type mapping and manipulation functions for manipulating data attributes of Oracle types.

- Object Type Translator utility, which maps internal Oracle schema information to client-side language bind variables.

Client-Side Object Cache

The object cache is a client-side memory buffer that provides lookup and memory management support for objects. It stores and tracks objects instances which have been fetched by an OCI application from the server to the client side. The object cache is created when the OCI environment is initialized. Multiple applications running against the same server will each have their own object cache. The cache tracks the objects which are currently in memory, maintains references to objects, manages automatic object swapping and tracks the meta-attributes or type information about objects. The cache provides the following to OCI applications:

- Improved application performance by reducing the number of client-to-server round trips required to fetch and operate on objects.

- Enhanced scalability by supporting object swapping from the client-side cache.

- Improved concurrency by supporting object-level locking.

Associative and Navigational Interfaces

Applications using OCI can access objects in the Oracle server through several types of interfaces:

- Using SQL `SELECT`, `INSERT`, and `UPDATE` statements.

- Using a C-style *pointer chasing* scheme to access objects in the client-side cache by traversing the corresponding smart pointers or REFs.

OCI provides a set of functions with extensions to support object manipulation using SQL `SELECT`, `INSERT`, and `UPDATE` statements. To access Oracle objects these SQL statements use a consistent set of steps as if they were accessing relational tables. OCI provides the following sets of functions required to access objects using SQL statements for:

- Binding and defining object type instances and references as input and output variables of SQL statements.

- Executing SQL statements that contain object type instances and references.

- Fetching object type instances and references.

- Describing a select-list item of an Oracle object type.

OCI also provides a set of functions using a C-style pointer chasing scheme to access objects once they have been fetched into the client-side cache by traversing the corresponding smart pointers or REFs. This *navigational interface* provides functions for:

- Instantiating a copy of a referenceable persistent object, that is, of a persistent object with object ID in the client-side cache by *pinning* its smart pointer or `REF`.

- Traversing a sequence of objects that are *connected* to each other by traversing the `REF`s that point from one to the other.

- Dynamically getting and setting values of an object's attributes.

Runtime Environment for Objects

OCI provides a runtime environment for objects that offers a set of functions for managing how Oracle objects are used on the client-side. These functions provide the necessary functionality for:

- Connecting to an Oracle server in order to access its object functionality including initializing a session, logging on to a database server, and registering a connection.

- Setting up the client-side object cache and tuning its parameters.

- Getting errors and warning messages.

- Controlling transactions that access objects in the server.

- Associatively accessing objects through SQL.

- Describing a PL/SQL procedure or function whose parameters or result are of Oracle type system types.

Type Management, Mapping and Manipulation Functions

OCI provides two sets of functions to work with Oracle objects:

- Type Mapping functions allow applications to map attributes of an Oracle schema which are represented in the server as internal Oracle datatypes such as Oracle's number, date and string types to their corresponding host language types such as integer, months and days.

- Type Manipulation functions allow host language applications to manipulate individual attributes of an Oracle schema such as setting/getting their values and flushing their values to the server.

Additionally, the `OCIDescribeAny()` function can provide information about objects stored in the database.

Object Type Translator

The Object Type Translator (OTT) utility translates schema information about Oracle object types into client-side language bindings. That is, the Oracle OTT translates type information into declarations of host language variables, such as structures and classes. The OTT takes an *intype* file which contains metadata information about Oracle schema objects as input. The OTT generates an *outtype* file and the necessary header and implementation files that must be included in a C application that runs against the object schema. Both OCI applications and Pro*C/C++ precompiler applications may include code generated by the OTT. The OTT has many benefits including:

- *Improves application developer productivity*: OTT eliminates the need for application developers to write by hand the host language variables that correspond to schema objects.

- *Maintains SQL as the data-definition language of choice*: By providing the ability to automatically map Oracle schema objects that are created using SQL to host language variables, OTT facilitates using SQL as the data-definition language of choice. This in turn allows Oracle to support a consistent model of the user's data, enterprise-wide.

- *Facilitates schema evolution of object types*: OTT provides the ability to regenerate included header files when the schema is changed allowing Oracle applications to support schema evolution.

OTT is typically invoked from the command line by specifying the intype file, the outtype file and the specific database connection. With Oracle, OTT can only generate C structs which can either be used with OCI programs or with the Pro*C/C++ precompiler programs.

OCI Support for Oracle Advanced Queuing

OCI provides an interface to Oracle's Advanced Queuing (AQ) feature. Oracle AQ provides message queuing as an integrated part of the Oracle server. Oracle AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database.* By providing an integrated solution Oracle AQ frees you to devote your efforts to your specific business logic rather than having to construct a messaging infrastructure.

> **See Also:** For more information about the OCI AQ features, refer to "OCI and Advanced Queuing" on page 9-42.

XA Library Support

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for information about support for the Oracle XA library.

Simplified Migration of Existing Applications

OCI has been significantly improved with many features. Applications written to work with OCI release 7 have a smooth migration path to this OCI release because of the interoperability of OCI release 7 clients with this release of the server, and of clients of this release with an Oracle7 database server.

Specifically:

- Applications that use OCI release 7.3 work unchanged against this release of the server.

- Applications that use this release of OCI work against an Oracle7 server provided they do not use any of the new capabilities of OCI or the server.

- OCI release 7 and the OCI calls of this release can be mixed in the same application and in the same transaction provided they are not mixed within the statement.

As a result, when migrating an existing OCI release 7 application you have the following three alternatives:

- *Retain Oracle7 OCI client*: You can retain your Oracle7 OCI applications without making any modifications - they will continue to work against a current server.

- *Upgrade to the current OCI client but do not modify application*: You who choose to upgrade from an Oracle7 OCI client to the current release OCI client need only relink the new version of the OCI library and need *not* recompile your application. The relinked Oracle7 OCI applications work unchanged against a current server.

- *Upgrade to Oracle9i OCI client and modify application*: To use the performance and scalability benefits provided by the new OCI, however, you will need to modify your existing applications to use the new OCI programming paradigm, relink them with the new OCI library, and run them against the current release of the server.

Further, if you need to use any of the object capabilities of the current server release, you will need to upgrade your client to use the this release of OCI.

Compatibility, Upgrading, and Migration

This OCI release provides support for applications written with either the 7.x OCI or the 8.x or later OCI. This section discusses issues concerning compatibility between different versions of OCI and server, changes in the OCI library routines, and migrating an application from the release 7.x OCI to this release of OCI.

> **See Also:** For the most recently updated information about compatibility, upgrading, and migration, refer to *Oracle9i Database Migration*

Obsolescent OCI Routines

Release 8.0 of the Oracle Call Interface introduced an entirely new set of functions which were not available in release 7.3. Release 8.1 added more new functions. *Oracle9i OCI continues to support these new functions, and adds more new calls.* The earlier 7.x calls are still available, but Oracle strongly recommends that existing applications use the new calls to improve performance and provide increased functionality.

Table 1–1, "Obsolescent OCI Routines" lists the 7.x OCI calls with their release 8.x or later equivalents. For more information about the OCI calls, see the function

descriptions in Part III of this guide. For more information about the 7.x calls, see the *Programmer's Guide to the Oracle Call Interface, Release 7.3*. These 7.x calls are obsoleted, meaning that OCI has replaced them with newer calls. While the obsoleted calls are supported at this time, they may not be supported in all future versions of OCI.

> **Note:** In many cases the new OCI routines do not map directly onto the 7.x routines, so it may not be possible to simply replace one function call and parameter list with another. Additional program logic may be required before or after the new call is made. See the remaining chapters of this guide for more information.

Table 1–1   Obsolescent OCI Routines

7.x OCI Routine	Equivalent or Similar 8.x or Later OCI Routine
obindps(), obndra(), obndrn(), obndrv()	OCIBindByName(), OCIBindByPos() (Note: additional bind calls may be necessary for some data types)
obreak()	OCIBreak()
ocan()	none
oclose()	Note: cursors are not used in release 8.x or later
ocof(), ocon()	OCIStmtExecute() with OCI_COMMIT_ON_SUCCESS mode
ocom()	OCITransCommit()
odefin(), odefinps()	OCIDefineByPos() (Note: additional define calls may be necessary for some data types)
odescr()	Note: schema objects are described with OCIDescribeAny(). A describe, as used in release 7.x, will most often be done by calling OCIAttrGet() on the statement handle after SQL statement execution.
odessp()	OCIDescribeAny()
oerhms()	OCIErrorGet()
oexec(), oexn()	OCIStmtExecute()
oexfet()	OCIStmtExecute(), OCIStmtFetch() (Note: result set rows can be implicitly prefetched)
ofen(), ofetch()	OCIStmtFetch()
oflng()	none

Table 1–1   Obsolescent OCI Routines (Cont.)

7.x OCI Routine	Equivalent or Similar 8.x or Later OCI Routine
ogetpi()	OCIStmtGetPieceInfo()
olog()	OCILogon()
ologof()	OCILogoff()
onbclr(), onbset(), onbtst()	Note: non-blocking mode can be set or checked by calling OCIAttrSet() or OCIAttrGet() on the server context handle or service context handle
oopen()	Note: cursors are not used in release 8.x or later
oopt()	none
oparse()	OCIStmtPrepare(); however, it is all local
opinit()	OCIEnvCreate()
orol()	OCITransRollback()
osetpi()	OCIStmtSetPieceInfo()
sqlld2()	SQLSvcCtxGet or SQLEnvGet
sqllda()	SQLSvcCtxGet or SQLEnvGet
odsc()	Note: see odescr() above
oermsg()	OCIErrorGet()
olon()	OCILogon()
orlon()	OCILogon()
oname()	Note: see odescr() above
osql3()	Note: see oparse() above

> **See Also:**   For information about the additional functionality
> provided by new functions not listed here, see the remaining
> chapters of this guide.

OCI Routines Not Supported

Some OCI routines that were available in previous versions of OCI are not
supported in Oracle8*i or Oracle9i*. They are listed in Table 1–2, "OCI Routines Not
Supported":

Table 1–2   OCI Routines Not Supported

OCI Routine	Equivalent or Similar 8.x or Later OCI Routine
obind()	OCIBindByName(), OCIBindByPos() (Note: additional bind calls may be necessary for some data types)
obindn()	OCIBindByName(), OCIBindByPos() (Note: additional bind calls may be necessary for some data types)
odfinn()	OCIDefineByPos() (Note: additional define calls may be necessary for some data types)
odsrbn()	Note: see odescr() in Table 1–1
ologon()	OCILogon()
osql()	Note: see oparse() Table 1–1

Compatibility

This section addresses compatibility between different versions of OCI and Oracle server.

Existing 7.x applications with no new release 8.x or later OCI calls have two choices:

- Do not relink the application.

- Relink with the new 8.x or later OCI library.

In either case, the application will work against both Oracle7 and Oracle8*i* or later, with the exception that the function ocom() should be substituted for ocon(). ocon() enables AUTOCOMMIT (automatic commit of every DML statement), and thus leads to an error in a subsequent fetch statement.

The application will not be able to use the object features of Oracle8*i* or later, and will not get any of the performance or scalability benefits provided by those OCI releases.

New applications written completely in OCI will work seamlessly against both Oracle7 and Oracle8*i* or later, with the following exceptions:

- Against Oracle7 servers, none of Oracle's object features are supported, and the following datatypes are not supported:

    - SQLT_NTY - named data type

    - SQLT_REF - reference to named data type in host language representation.

    - SQLT_CLOB - a character LOB data type.

- SQLT_BLOB - a binary LOB data type.

- SQLT_BFILE - a binary FILE LOB data type.

- SQLT_RSET - result set data type.

- SQLT_DATE - ANSI DATE

- SQLT_TIMESTAMP - TIMESTAMP

- SQLT_TIMESTAMP_TZ - TIMESTAMP WITH TIME ZONE

- SQLT_TIMESTAMP_LTZ - TIMESTAMP WITH LOCAL TIME ZONE

- SQLT_INTERVAL_DS - INTERVAL DAY TO SECOND

- SQLT_INTERVAL_YM - INTERVAL YEAR TO MONTH

- Against Oracle7 Servers, the following calls or features are not supported, or are supported with restrictions:

Table 1–3   Oracle8i or Later OCI Restrictions When Running Against Oracle7 Servers

Function	Restrictions
OCIBindObject()	not supported
OCIPasswordChange()	not supported
OCIDefineObject()	not supported
OCIDescribeAny()	only supports description of select lists or stored procedures
OCIErrorGet()	only a subset of Oracle error codes can be returned
OCIStmtFetch()	prefetching options not supported
OCILob*()	LOB/FILE calls are not supported
OCIAttrSet()	setting NCHAR attributes not supported
OCIAttrGet()	getting NCHAR attributes not supported

Upgrading

Programmers who wish to incorporate release 8.x or later functionality into existing OCI applications have two options:

- Completely rewrite the application to use only new OCI calls (recommended).

- Incorporate new OCI release 8.x or later calls into the application, while still using 7.x calls for some operations.

This manual should provide the information necessary to rewrite an existing application to use only new OCI calls.

Adding 8.x or Later OCI Calls to 7.x Applications

The following guidelines apply to programmers who want to incorporate new Oracle datatypes and features by using new OCI calls, while keeping 7.x calls for some operations:

- Change the existing logon to use `OCILogon()` instead of `olog()` (or other logon call). The service context handle can be used with new OCI calls or can be converted into a **Lda_Def** to be used with 7.x OCI calls.

  > **See Also:** See the description of `OCIServerAttach()` on page 16-71 and the description of `OCISessionBegin()` on page 16-71 for information about the logon calls necessary for applications which are maintaining multiple sessions.

- After the server context handle has been initialized, it can be used with OCI release 8.x or later calls.

- To use Oracle7 OCI calls, convert the server context handle to an **Lda_Def** using `OCISvcCtxToLda()`, and pass the resulting **Lda_Def** to the 7.x calls.

  > **Note:** If there are multiple service contexts which share the same server handle, only one can be in Oracle7 mode at any time.

- To begin using 8.x or later OCI calls again, the application must convert the **Lda_Def** back to a server context handle using `OCILdaToSvcCtx()`.

- The application may toggle between the **Lda_Def** and server context as often as necessary in the application.

This approach allows an application to use a single connection, but two different APIs, to accomplish different tasks.

You can mix and match OCI 7.x and OCI 8.x or later calls within a transaction, but *not* within a statement. This lets you execute one SQL or PL/SQL statement with OCI 7.x calls and the next SQL or PL/SQL statement within that transaction with Oracle8.x or later OCI calls.

> **Caution:** You *cannot* open a cursor, parse with OCI 7.x calls and then execute the statement with OCI 8.x or later calls.

2

OCI Programming Basics

This chapter introduces you to the basic concepts involved in programming with the OCI. This chapter covers the following topics:

- Overview of OCI Programming
- OCI Program Structure
- OCI Data Structures
- Handles
- Descriptors
- OCI Programming Steps
- OCI Environment Initialization
- Processing SQL Statements
- Commit or Rollback
- Terminating the Application
- Error Handling
- Additional Coding Guidelines
- Using PL/SQL in an OCI Program
- Globalization Support

Overview of OCI Programming

This chapter provides an introduction to the concepts and procedures involved in developing an OCI application. After reading this chapter, you should have most of the tools necessary to understand and create a basic OCI application.

This chapter is broken down into the following major sections:

- OCI Program Structure - covers the basic overall structure of an OCI application, including the major steps involved in creating one.

- OCI Data Structures - discusses handles, and descriptors.

- OCI Programming Steps - discusses in detail each of the steps involved in coding an OCI application.

- Error Handling - covers error handling in OCI applications.

- Additional Coding Guidelines - provides useful information to keep in mind when coding an OCI application.

- Non-Blocking Mode - this section covers the use of non-blocking mode to connect to an Oracle database server.

- Using PL/SQL in an OCI Program - discusses some important points to keep in mind when working with PL/SQL in an OCI application.

New users should pay particular attention to the information presented in this chapter, because it forms the basis for the rest of the material presented in this guide. The information in this chapter is supplemented by information in later chapters.

See Also:

- For a discussion of the OCI functions that apply to a multilingual environment, see the *Oracle9i Globalization and National Language Support Guide*

- For a discussion of the OCI functions that apply to cartridge services, see the *Oracle9i Data Cartridge Developer's Guide.*

OCI Program Structure

The general goal of an OCI application is to operate on behalf of multiple users. In an n-tiered configuration, multiple users are sending HTTP requests to the client application. The client application may need to perform some data operations that include exchanging data and performing data processing.

The OCI uses the following basic program structure:

1. Initialize the OCI programming environment and threads.

2. Allocate necessary handles, and establish server connections and user sessions.

3. Exchange data with the database server by executing SQL statements on the server, and perform necessary application data processing.

4. Reexecute prepared statements, or prepare a new statement for execution.

5. Terminate user sessions and server connections.

6. Free handles.

Figure 2–1, "Basic OCI Program Flow" illustrates the flow of steps in an OCI application. Each step is described in more detail in the section "OCI Programming Steps" on page 2-20.

Figure 2–1   Basic OCI Program Flow
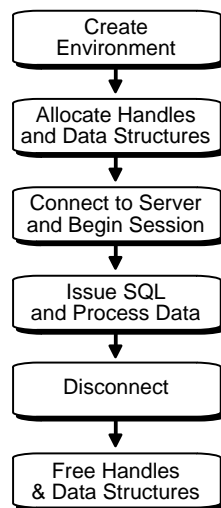
Keep in mind that the diagram and the list of steps present a simple generalization of OCI programming steps. Variations are possible, depending on the functionality of the program. OCI applications that include more sophisticated functionality, such as managing multiple sessions and transactions and using objects, require additional steps.

All OCI function calls are executed in the context of an environment. There can be multiple environments within an OCI process, as illustrated in Figure 2–2, "Multiple Environments Within an OCI Process". If an environment requires any process-level initialization then it is performed automatically.

> **Note:** In previous releases, a separate explicit process-level initialization was required. This requirement has been simplified and no explicit process-level initialization is required.

Figure 2–2 Multiple Environments Within an OCI Process
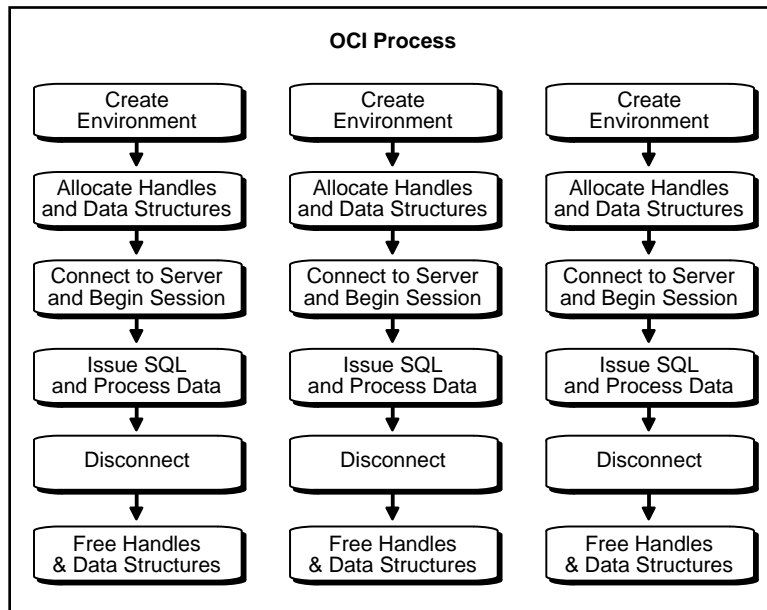
> **Note:** It is possible to have more than one active connection and statement in an OCI application.

> **See Also:** For information about accessing and manipulating objects, see Chapter 10, "OCI Object-Relational Programming" and the chapters that follow it

OCI Data Structures

Handles and *descriptors* are opaque data structures which are defined in OCI applications and may be allocated directly, through specific allocate calls, or may be implicitly allocated by OCI functions.

> **7.x Upgrade Note:**   Programmers who have previously written 7.x OCI applications need to become familiar with these new data structures which are used by most OCI calls

Handles and descriptors store information pertaining to data, connections, or application behavior. Handles are defined in more detail in the following section. Descriptors are discussed in the section "Descriptors" on page 2-15.

Handles

Almost all OCI calls include in their parameter list one or more handles. A handle is an opaque pointer to a storage area allocated by the OCI library. You use a handle to store context or connection information, (e.g., an environment or service context handle), or it may store information about OCI functions or data (e.g., an error or describe handle). Handles can make programming easier, because the library, rather than the application, maintains this data.

Most OCI applications need to access the information stored in handles. The get and set attribute OCI calls, `OCIAttrGet()` and `OCIAttrSet()`, access this information.

> **See Also:**   For more information about using handle attributes, see the section "Handle Attributes" on page 2-13

The following table lists the handles defined for the OCI. For each handle type, the C datatype and handle type constant used to identify the handle type in OCI calls are listed.

Table 2–1   OCI Handle Types

Description	C Type	Handle Type
OCI environment handle	OCIEnv	OCI_HTYPE_ENV
OCI error handle	OCIError	OCI_HTYPE_ERROR
OCI service context handle	OCISvcCtx	OCI_HTYPE_SVCCTX
OCI statement handle	OCIStmt	OCI_HTYPE_STMT
OCI bind handle	OCIBind	OCI_HTYPE_BIND
OCI define handle	OCIDefine	OCI_HTYPE_DEFINE
OCI describe handle	OCIDescribe	OCI_HTYPE_DESCRIBE
OCI server handle	OCIServer	OCI_HTYPE_SERVER
OCI user session handle	OCISession	OCI_HTYPE_SESSION
OCI connection pool	OCIPool	OCI_HTYPE_CPOOL
OCI transaction handle	OCITrans	OCI_HTYPE_TRANS
OCI complex object retrieval (COR) handle	OCIComplexObject	OCI_HTYPE_COMPLEXOBJECT
OCI thread handle	OCIThreadHandle	N/A
OCI subscription handle	OCISubscription	OCI_HTYPE_SUBSCRIPTION
OCI direct path context handle	OCIDirPathCtx	OCI_HTYPE_DIRPATH_CTX
OCI direct path function context handle	OCIDirPathFuncCtx	OCI_HTYPE_DIRPATH_FN_CTX
OCI direct path column array handle	OCIDirPathColArray	OCI_HTYPE_DIRPATH_COLUMN_ARRAY
OCI direct path stream handle	OCIDirPathStream	OCI_HTYPE_DIRPATH_STREAM
OCI process handle		OCI_HTYPE_PROC

Allocating and Freeing Handles

Your application allocates all handles (except the bind, define, and thread handles) with respect to particular environment handle. You pass the environment handle as one of the parameters to the handle allocation call. The allocated handles is then specific to that particular environment.

The bind and define handles are allocated with respect to a statement handle, and contain information about the statement represented by that handle.

> **Note:** The bind and define handles are implicitly allocated by the OCI library, and do not require user allocation.

Figure 2–3, "Hierarchy of Handles" illustrates the relationship between the various types of handles.

All user-allocated handles are allocated using the OCI handle allocation call, `OCIHandleAlloc()`.

> **Note:** The environment handle is allocated and initialized with a call to `OCIEnvCreate()`, which is required by all OCI applications.

The thread handle is allocated with the `OCIThreadHndInit()` call.

An application must free all handles when they are no longer needed. The `OCIHandleFree()` function frees handles.

> **Note:** When a parent handle is freed, all child handles associated with it are also freed, and can no longer be used. For example, when a statement handle is freed, any bind and define handles associated with it are also freed.

Figure 2–3   Hierarchy of Handles
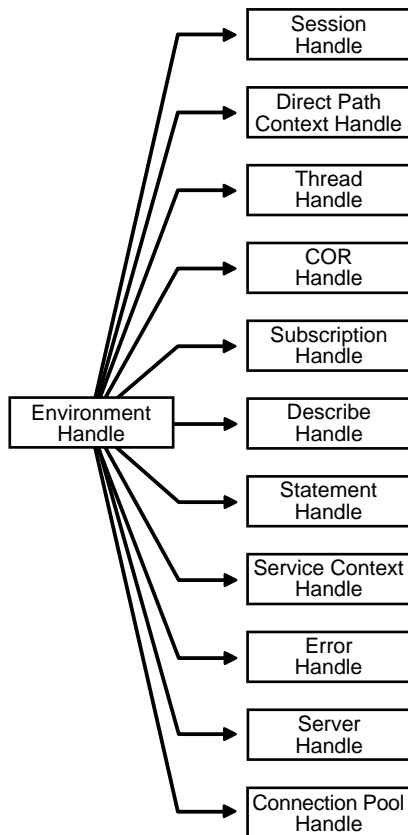
Handles lessen the need for global variables. Handles also make error reporting easier. An error handle is used to return errors and diagnostic information.

> **See Also:**   For sample code demonstrating the allocation and use of OCI handles, see the example programs listed in Appendix B, "OCI Demonstration Programs"

The various handle types are described in more detail in the following sections.

Environment Handle

The *environment handle* defines a context in which all OCI functions are invoked. Each environment handle contains a memory cache, which allows for fast memory access. All memory allocation under the environment handle is done from this cache. Access to the cache is serialized if multiple threads try to allocate memory under the same environment handle. When multiple threads share a single environment handle, they may block on access to the cache.

The environment handle is passed as the *parent* parameter to the `OCIHandleAlloc()` call to allocate all other handle types. Bind and define handles are allocated implicitly.

Error Handle

The *error handle* is passed as a parameter to most OCI calls. The error handle maintains information about errors that occur during an OCI operation. If an error occurs in a call, the error handle can be passed to `OCIErrorGet()` to obtain additional information about the error that occurred.

Allocating the error handle is one of the first steps in an OCI application because most OCI calls require an error handle as one of its parameters.

Service Context and Associated Handles

A *service context handle* defines attributes that determine the operational context for OCI calls to a server. The service context contains three handles as its attributes, that represent a server connection, a user session, and a transaction. These attributes are illustrated in :

Figure 2–4   Components of a Service Context
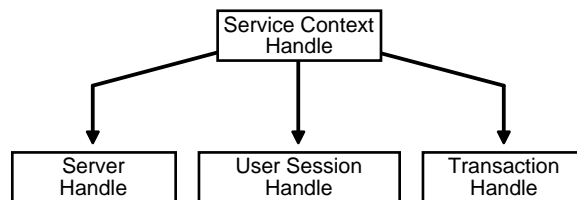
- A *server handle* identifies a connection to a database. It translates into a physical connection in a connection-oriented transport mechanism.

- A *user session handle* defines a user's roles and privileges (also known as the user's security domain), and the operational context on which the calls execute.

- A *transaction handle* defines the transaction in which the SQL operations are performed. The transaction context includes user session state information, including the fetch state and package instantiation, if any.

Breaking the service context down in this way provides scalability and enables programmers to create sophisticated three-tiered applications and transaction processing (TP) monitors to execute requests on behalf of multiple users on multiple application servers and different transaction contexts.

You must allocate and initialize the service context handle with `OCIHandleAlloc()` or `OCILogon()` before you can use it. The service context handle is allocated explicitly by `OCIHandleAlloc()`. It can be initialized using `OCIAttrSet()` with the server, session, and transaction handle. If the service context handle is allocated implicitly using `OCILogon()`, it is already initialized.

Applications maintaining only a single user session per database connection at any time can call `OCILogon()` to get an initialized service context handle.

In applications requiring more complex session management, the service context must be explicitly allocated, and the server handle and user session handle must be explicitly set into the service context. `OCIServerAttach()` and `OCISessionBegin()`, calls initialize the server and user session handle respectively.

An application will only define a transaction explicitly if it is a global transaction or there are multiple transactions active for sessions. It will be able to work correctly with the implicit transaction created automatically by OCI when the application makes changes to the database.

See Also:

- For more information about transactions, see the section "OCI Support for Transactions" on page 8-2

- For more information about establishing a server connection and user session, see the sections "OCI Environment Initialization" on page 2-21, and "Password and Session Management" on page 8-11

Statement Handle, Bind Handle, and Define Handle

A *statement handle* is the context that identifies a SQL or PL/SQL statement and its associated attributes.

Figure 2–5 Statement Handles
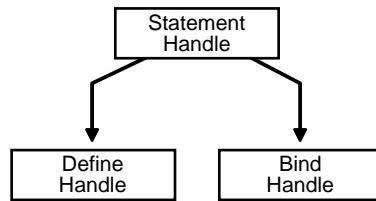
Information about input and output bind variables is stored in *bind handles*. The OCI library allocates a bind handle for each placeholder bound with the `OCIBindByName()` or `OCIBindByPos()` function. The user does not need to allocate bind handles. They are implicitly allocated by the bind call.

Fetched data returned by a query (select statement) is converted and retrieved according to the specifications of the *define handles*. The OCI library allocates a define handle for each output variable defined with `OCIDefineByPos()`. The user does not need to allocate define handles. They are implicitly allocated by the define call.

Bind and define handles are freed when the statement handle is freed or when a new statement is prepared on the statement handle.

Statement context data, the data associated with a statement handle, can be shared.

> **See Also:** For information about OCI shared mode, see "Shared Data Mode" on page 2-22

Describe Handle

The *describe handle* is used by the OCI describe call, `OCIDescribeAny()`. This call obtains information about schema objects in a database (e.g., functions, procedures). The call takes a describe handle as one of its parameters, along with information about the object being described. When the call completes, the describe handle is populated with information about the object. The OCI application can then obtain describe information through the attributes of parameter descriptors.

> **See Also:** Chapter 6, "Describing Schema Metadata", for more information about using the `OCIDescribeAny()` function

Complex Object Retrieval Handle

The *complex object retrieval (COR) handle* is used by some OCI applications that work with objects in an Oracle database server. This handle contains *COR descriptors,* which provide instructions about retrieving objects referenced by another object.

> **See Also :**   For information about complex object retrieval and the complex object retrieval handle, refer to "Complex Object Retrieval" on page 10-22

Thread Handle

For information about the thread handle, which is used in multithreaded applications, refer to "The OCIThread Package" on page 9-5.

Subscription Handle

The subscription handle is used by an OCI client application that is interested in registering for subscriptions to receive notifications of database events or events in the AQ namespace. The subscription handle encapsulates all information related to a registration from a client.

> **See Also:**   For information about publish-subscribe and allocating the subscription handle, refer to "Publish-Subscribe Notification" on page 9-46

Direct Path Handles

The direct path handles are necessary for an OCI application that utilizes the direct path load engine in the Oracle database server. The direct path load interface allows the application to access the direct block formatter of the Oracle server.

Figure 2–6 Direct Path Handles
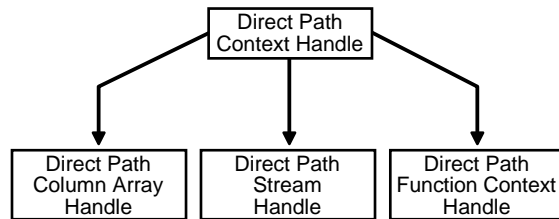
See Also:

- For information about direct path loading and allocating the direct path handles, refer to "Direct Path Loading Overview" on page 12-2

- For information about the handle attributes, refer to "Direct Path Loading Handle Attributes" on page A-58

Process Handle

The *process handle* is a specialized handle for OCI applications that utilize shared data structures mode to set global parameters.

> **See Also:** "Shared Data Mode" on page 2-22

Connection Pool Handle

The *connection pool handle* is used for applications that pool physical connections into virtual connections, by calling specific OCI functions.

> **See Also:** "Connection Pooling" on page 9-14

Handle Attributes

All OCI handles have *attributes* associated with them. These attributes represent data stored in that handle. You can read handle attributes using the attribute get call, OCIAttrGet(), and you can change them with the attribute set call, OCIAttrSet().

For example, the following statements set the username in the session handle by writing to the OCI_ATTR_USERNAME attribute:

```
text username[] = "scott";
```

```
err = OCIAttrSet ((dvoid*) mysessp, OCI_HTYPE_SESSION, (dvoid*) username,
     (ub4) strlen(username), OCI_ATTR_USERNAME,
     (OCIError *) myerrhp);
```

Some OCI functions require that particular handle attributes be set before the function is called. For example, when OCISessionBegin() is called to establish a user's login session, the username and password must be set in the user session handle before the call is made.

Other OCI functions provide useful return data in handle attributes after the function completes. For example, when OCIStmtExecute() is called to execute a SQL query, describe information relating to the select-list items is returned in the statement handle.

```
ub4 parmcnt;
/* get the number of columns in the select list */
err = OCIAttrGet ((dvoid *)stmhp, (ub4)OCI_HTYPE_STMT, (dvoid *)
        &parmcnt, (ub4 *) 0, (ub4)OCI_ATTR_PARAM_COUNT, errhp);
```

See Also:

- See the description of OCIAttrGet() on page 15-33 for an example showing the username and password handle attributes being set

- For a list of all handle attributes, refer to Appendix A, "Handle and Descriptor Attributes"

User Memory Allocation

The OCIEnvCreate() call, which initializes the environment handle, and the generic handle allocation (OCIHandleAlloc()) and descriptor allocation (OCIDescriptorAlloc()) calls have an *xtramem_sz* parameter in their parameter list. This parameter is used to specify memory chunk size which is allocated along with that handle for the user. This memory is not used by OCI and is for use by the application only.

Typically, an application uses this parameter to allocate an application-defined structure, such as for an application *bookkeeping* or storing context information, that has the same lifetime as the handle.

Using the *xtramem_sz* parameter means that the application does not need to explicitly allocate and deallocate memory as each handle is allocated and

deallocated. The memory is allocated along with the handle, and freeing the handle frees up the user's data structures as well.

Descriptors

OCI *descriptors* and *locators* are opaque data structures that maintain data-specific information. The following table lists them, along with their C datatype, and the OCI type constant that allocates a descriptor of that type in a call to `OCIDescriptorAlloc()`. The `OCIDescriptorFree()` function frees descriptors and locators.

Table 2–2   Descriptor Types

Description	C Type	OCI Type Constant
snapshot descriptor	OCISnapshot	OCI_DTYPE_SNAP
LOB datatype locator	OCILobLocator	OCI_DTYPE_LOB
FILE datatype locator	OCILobLocator	OCI_DTYPE_FILE
read-only parameter descriptor	OCIParam	OCI_DTYPE_PARAM
ROWID descriptor	OCIRowid	OCI_DTYPE_ROWID
ANSI DATE descriptor	OCIDateTime	OCI_DTYPE_DATE
TIMESTAMP descriptor	OCIDateTime	OCI_DTYPE_TIMESTAMP
TIMESTAMP WITH TIME ZONE descriptor	OCIDateTime	OCI_DTYPE_TIMESTAMP_TZ
TIMESTAMP WITH LOCAL TIME ZONE descriptor	OCIDateTime	OCI_DTYPE_TIMESTAMP_LTZ
INTERVAL YEAR TO MONTH descriptor	OCIInterval	OCI_DTYPE_INTERVAL_YM
INTERVAL DAY TO SECOND descriptor	OCIInterval	OCI_DTYPE_INTERVAL_DS
complex object descriptor	OCIComplexObjectComp	OCI_DTYPE_COMPLEXOBJECTCOMP
advanced queuing enqueue options	OCIAQEnqOptions	OCI_DTYPE_AQENQ_OPTIONS
advanced queuing dequeue options	OCIAQDeqOptions	OCI_DTYPE_AQDEQ_OPTIONS
advanced queuing message properties	OCIAQMsgProperties	OCI_DTYPE_AQMSG_PROPERTIES

Table 2–2   Descriptor Types (Cont.)

Description	C Type	OCI Type Constant
advanced queuing agent	OCIAQAgent	OCI_DTYPE_AQAGENT
advanced queuing notification	OCIAQNotify	OCI_DTYPE_AQNFY
the distinguished names of the database servers in a registration request	OCIServerDNs	OCI_DTYPE_SRVDN

> **Note:**   Although there is a single C type for **OCILobLocator**, this locator is allocated with a different OCI type constant for internal and external LOBs. The section below on LOB locators discusses this difference.

The main purpose of each descriptor type is listed here, and each descriptor type is described in the following sections:

- **OCISnapshot** - used in statement execution

- **OCILOBLocator** - used for LOB (OCI_DTYPE_LOB) or FILE (OCI_DTYPE_FILE) calls

- **OCIParam** - used in describe calls

- **OCIRowid** - used for binding or defining ROWID values

- **OCIDateTime** and **OCIInterval** - used for datetime and interval datatypes

- **OCIComplexObjectComp** - used for complex object retrieval

- **OCIAQEnqOptions**, **OCIAQDeqOptions**, **OCIAQMsgProperties**, **OCIAQAgent** - used for Advanced Queuing

- **OCIAQNotify** - used for publish-subscribe notification

- **OCIServerDNs** - used for LDAP-based publish-subscribe notification

Snapshot Descriptor

The *snapshot descriptor* is an optional parameter to the execute call, OCIStmtExecute(). It indicates that a query is being executed against a particular database snapshot. A database snapshot represents the state of a database at a particular point in time.

You allocate a snapshot descriptor with a call to OCIDescriptorAlloc(), by passing OCI_DTYPE_SNAP as the *type* parameter.

> **See Also:** For more information about OCIStmtExecute() and database snapshots, see the section "Execution Snapshots" on page 4-7

LOB/FILE Datatype Locator

A LOB (large object) is an Oracle datatype that can hold up to 4 gigabytes of binary (BLOB) or character (CLOB) data. In the database, an opaque data structure called a *LOB locator* is stored in a LOB column of a database row, or in the place of a LOB attribute of an object. The locator serves as a pointer to the actual LOB value, which is stored in a separate location.

The OCI *LOB locator* is used to perform OCI operations against a LOB (BLOB or CLOB) or FILE (BFILE). OCILob* functions take the LOB locator as a parameter instead of the LOB value. OCI LOB functions do not take actual LOB data as parameters. These functions take the LOB locators as parameters and operate on the LOB data referenced by these locators.

Hence, the old long interface can operate on the actual LOB value. This descriptor—**OCILobLocator**—is also used for operations on FILEs.

The LOB locator is allocated with a call to OCIDescriptorAlloc(), by passing OCI_DTYPE_LOB as the *type* parameter for BLOBs or CLOBs, and OCI_DTYPE_FILE for BFILEs.

> **Caution:** The two LOB locator types are *not* interchangeable. When binding or defining a BLOB or CLOB, the application must take care that the locator is properly allocated using OCI_DTYPE_LOB. Similarly, when binding or defining a BFILE, the application must be sure to allocate the locator using OCI_DTYPE_FILE.

An OCI application can retrieve a LOB locator from the server by issuing a SQL statement containing a LOB column or attribute as an element in the select list. In this case, the application would first allocate the LOB locator and then use it to define an output variable. Similarly, a LOB locator can be used as part of a bind operation to create an association between a LOB and a placeholder in a SQL statement.

The LOB locator datatype (**OCILobLocator**) is not a valid datatype when connected to an Oracle7 Server.

> **See Also:** For more information about OCI LOB operations, see Chapter 7, "LOB and FILE Operations"

Parameter Descriptor

OCI applications use *parameter descriptors* to obtain information about select-list columns or schema objects. This information is obtained through a describe operation.

The parameter descriptor is the one descriptor type that is *not* allocated using `OCIDescriptorAlloc()`. You can obtain it only as an attribute of a describe, statement, or complex object retrieval handle by specifying the position of the parameter using an `OCIParamGet()` call.

> **See Also:** See Chapter 6, "Describing Schema Metadata", and "Describing Select-List Items" on page 4-12 for more information about obtaining and using parameter descriptors

ROWID Descriptor

The ROWID descriptor, **OCIRowid**, is used by applications that need to retrieve and use Oracle ROWIDs. The size and structure of the ROWID has changed from Oracle release 7 to Oracle release 8, and is opaque to the user. To work with a ROWID using OCI release 8 or later, an application can define a ROWID descriptor for a rowid position in a SQL select-list, and retrieve a ROWID into the descriptor. This same descriptor can later be bound to an input variable in an INSERT statement or WHERE clause.

ROWIDs are also redirected into descriptors using `OCIAttrGet()` on the statement handle following an execute.

Datetime and Interval Descriptors

These descriptors are used by applications which use the datetime or interval datatypes (**OCIDateTime** and **OCIInterval**). These descriptors can be used for binding and defining, and are passed as parameters to the functions `OCIDescAlloc()` and `OCIDescFree()` to allocate and free memory.

> **See Also:** For more information about these datatypes refer to Chapter 3, "Datatypes". The functions which operate on these datatypes are listed in Chapter 18, "OCI Datatype Mapping and Manipulation Functions"

> **Note:** The functions which operate on **OCIDateTime** and **OCIInterval** datatypes also work on the **OCIDate** datatype

---

Complex Object Descriptor

For information about the complex object descriptor and its use, refer to "Complex Object Retrieval" on page 10-22.

Advanced Queuing Descriptors

For information about Advanced Queuing and its related descriptors, refer to "OCI and Advanced Queuing" on page 9-42.

LDAP-based Publish-Subscribe Notification

For information about LDAP-based publish-subscribe notification, see "Publish-Subscribe Registration Functions" on page 9-47.

User Memory Allocation

The OCIDescriptorAlloc() call has an `xtramem_sz` parameter in its parameter list. This parameter is used to specify an amount of user memory which should be allocated along with a descriptor or locator.

Typically, an application uses this parameter to allocate an application-defined structure that has the same lifetime as the descriptor or locator. This structure maybe used for application *bookkeeping* or storing context information.

Using the `xtramem_sz` parameter means that the application does not need to explicitly allocate and deallocate memory as each descriptor or locator is allocated and deallocated. The memory is allocated along with the descriptor or locator, and freeing the descriptor or locator (with OCIDescriptorFree()) frees up the user's data structures as well.

The OCIHandleAlloc() call has a similar parameter for allocating user memory which has the same lifetime as the handle.

The OCIEnvCreate() and OCIEnvInit() calls have a similar parameter for allocating user memory which has the same lifetime as the environment handle.

OCI Programming Steps

Each of the steps that you perform in an OCI application is described in greater detail in the following sections. 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.

Note: For an example showing the use of OCI calls for processing SQL statements, see the first sample program in Appendix B, "OCI Demonstration Programs"

See Also:

- The special case of dynamically providing data at run time is described in detail in the section "Runtime Data Allocation and Piecewise Operations" on page 5-44

- Special considerations for operations involving arrays of structures are described in the section "Binding and Defining Arrays of Structures" on page 5-26

- Refer to the section "Error Handling" on page 2-31 for an outline of the steps involved in processing a SQL statement within an OCI program

- For information on using the OCI to write multi-threaded applications, refer to "Thread Safety" on page 9-2

- For more information about types of SQL statements, refer to the section "SQL Statements" on page 1-7

The following sections describe the steps that are required of an OCI application:

- OCI Environment Initialization.

- Processing SQL Statements.

- Commit or Rollback.

- Terminating the Application.

- Error Handling.

Application-specific processing will also occur in between any and all of the OCI function steps.

> **7.x Upgrade Note:**   OCI programmers should take note that OCI programs no longer require an explicit parse step. This means that 8.0 or later applications must issue an execute command for both DML and DDL statements.

OCI Environment Initialization

This section describes how to initialize the OCI environment, establish a connection to a server, and authorize a user to perform actions against a database.

First, the three main steps in initializing the OCI environment are described in the following sections:

1.   Creating the OCI environment.

2.   Allocating Handles and Descriptors.

3.   Initializing the Application, Connection, and Session.

Creating the OCI Environment

Each OCI function call is executed in the context of an environment that is created with the OCIEnvCreate() call. This call must be invoked before any other OCI call. The only exception is when setting a process-level attribute for the OCI shared mode.

> **See Also:**   "Shared Data Mode" on page 2-22

The *mode* parameter of OCIEnvCreate() specifies whether the application calling the OCI library functions will:

- Run in a threaded environment (*mode* = OCI_THREADED).

- Use objects (*mode* = OCI_OBJECT).

- Use shared data structures (*mode* = OCI_SHARED).

- Use subscriptions (*mode* = OCI_EVENTS).

The mode can be set independently in each environment.

Initializing in object mode is necessary if the application will be binding and defining objects, or if the application will be using the OCI's object navigation calls. The program may also choose to use none of these features (*mode* = OCI_DEFAULT) or some combination of them, separating the options with a vertical bar. For example if *mode* = (OCI_THREADED | OCI_OBJECT), then the application runs in a threaded environment and use objects.

You can also specify user-defined memory management functions for each OCI environment.

> **Note:** In previous releases, a separate explicit process-level initialization was required. This requirement has been simplified and no explicit process-level initialization is required.

See Also:

- See the description of `OCIEnvCreate()` on page 15-9 and `OCIInitialize()` on page 15-14 for more information about the initialization calls.

- For information about using the OCI to write multi-threaded applications, refer to "Thread Safety" on page 9-2.

- For information about OCI programming with objects, refer to Chapter 10, "OCI Object-Relational Programming" and the chapters that follow it.

- For information about using the publish-subscribe feature, see "Publish-Subscribe Notification" on page 9-46.

Shared Data Mode

When a SQL statement is processed, certain underlying data is associated with the statement. This data includes information about statement text and bind data, as well as define and describe information for queries. For applications where the same set of SQL statements is executed on multiple instances of the application on the same host, the data can be shared.

When an OCI application is initialized in shared mode, common statement data is shared between multiple statement handles, thus providing memory savings for the application. This savings may be particularly valuable for applications which create multiple statement handles which execute the same SQL statement on different users' sessions but in the same schema, either on the same or multiple connections.

Without the shared mode feature, each execution of the query using an OCI statement handle requires its own memory for storing the metadata. The total amount of memory required is roughly equal to the number of statements being executed in all the processes combined multiplied by the memory required for each statement handle.

A large part of the common memory in a statement handle is shared among all the processes executing the same statement with the shared mode feature. The total amount of memory in all the processes combined is much less than in the previous case for the same number of processes. The memory requirement per statement handle is much smaller than in the case where there is no sharing, as the number of such statements increases to a large number.

Shared data structure mode can be useful in the following scenarios:

- When several instances of the same application are running on the same machine to service multiple clients. Each of these instances may be executing identical SQL statements, differentiated by different bind values.

- When an application process forks service threads to execute the same statement for different users either on the same connection or on multiple connections. The same saving as above can be realized in this scenario too.

- Where the types of applications are SQL drivers and other middle-tiered applications.

> **Note:** Small applications, which execute single queries non-concurrently do not benefit from this feature.

There are several ways to use the shared OCI functionality. Existing applications can quickly examine the benefits of this feature without changing any code. These applications can be initialized in OCI shared mode by setting environment variables. New applications should use OCI API calls to initialize shared mode functionality.

Using OCI Functions

To initialize OCI shared mode functionality, process handle parameters must be set and `OCIEnvCreate()` must be called with the mode flag set to OCI_SHARED. For example:

```
ub4 mode = OCI_SHARED | OCI_THREADED;
OCIEnvCreate (&envhp, mode, (CONST dvoid *)0, 0, 0, 0, (size_t)0 (dvoid **)0);
```

The first application that initializes OCI in shared mode starts up the shared subsystem using the parameters set by that OCI application. When subsequent applications initialize using the shared mode, they use the previously started shared subsystem.

> **See Also:** For information on the parameters that can be set and read for the OCI shared mode system, see "Process Handle Attributes" on page A-72.

If an OCI application has been initialized in shared mode, all statements that are prepared and executed use the shared subsystem by default. If you do not want to use the shared subsystem to execute a specific SQL statement, then you can use the OCI_NO_SHARING flag in OCIStmtPrepare(). For example:

```
OCIStmtPrepare(stmthp, (CONST text *)createstmt,
               (ub4)strlen((char *)updstmt), (ub4)OCI_NTV_SYNTAX,
               (ub4)OCI_NO_SHARING);
```

The OCI_NO_SHARING flag has no effect if the process has not been initialized in the shared mode.

> **See Also:** OCIStmtPrepare() on page 15-94.

To detach a process from the shared memory subsystem, use the OCITerminate() call.

> **See Also:** OCITerminate() on page 15-31.

Using Environment Variables

The environment variables OCI_SHARED_MODE and OCI_NUM_SHARED_PROCS can be used to set OCI shared mode functionality. However, this is not the recommended method. This procedure lets you to quickly examine the benefits of using shared mode functionality in existing applications.

OCI_SHARED_MODE To initialize an OCI application to run in shared mode, set the environment variable OCI_SHARED_MODE before executing a OCI program. To set the variable in the C-shell under Solaris, for example, issue the command:

```
setenv OCI_SHARED_MODE number
```

where *number* is the size of the shared memory address space. For example:

```
setenv OCI_SHARED_MODE 20000000
```

If the shared subsystem is not already running, setting this variable launches the subsystem by creating a shared memory address space with the size specified. The size of the shared memory required is determined by the nature of the application and depends on the size and type of the SQL statement and the underlying table(s) that it accesses.

OCI_NUM_SHARED_PROCS

To set the maximum number of processes that can connect to the shared subsystem, set the environment variable ORA_OCI_NUM_SHARED_PROCS. To set this variable, issue the command:

```
setenv OCI_NUM_SHARED_PROCS number
```

where *number* is the maximum number of processes. For example:

```
setenv OCI_NUM_SHARED_PROCS   20
```

ORA_OCI_NUM_SHARED_PROCS is an initialization parameter for starting the shared subsystem. It has no effect if the shared subsystem is already running.

Allocating Handles and Descriptors

Oracle provides OCI functions to allocate and deallocate handles and descriptors. You must allocate handles using `OCIHandleAlloc()` before passing them into an OCI call, unless the OCI call, such as `OCIBindByPos()`, allocates the handles for you.

You can allocate the following types of handles with `OCIHandleAlloc()`:

- error handle.
- service context handle.
- statement handle.
- describe handle.
- server handle.
- user session handle.
- transaction handle.
- connection pool handle
- complex object retrieval handle.

- subscription handle.

- direct path context handle.

- direct path column array handle.

- direct path stream handle

Depending on the functionality of your application, it needs to allocate some or all of these handles.

> **See Also:** the description of `OCIHandleAlloc()` on page 15-41

Application Initialization, Connection, and Session Creation

An application must call OCIEnvCreate() to initialize the OCI environment handle.

Following this step, the application has two options for establishing a server connection and beginning a user session: Single User, Single Connection; or Multiple Sessions or Connections.

Note: `OCIEnvCreate()` should be used instead of the `OCIInitialize()` and `OCIEnvInit()` calls. `OCIInitialize()` and `OCIEnvInit()` calls are supported for backward compatibility.

Option 1: Single User, Single Connection

This option is the simplified logon method.

If an application maintains only a single user session per database connection at any time, the application can take advantage of the OCI's simplified logon procedure.

When an application calls `OCILogon()`, the OCI library initializes the service context handle that is passed to it and creates a connection to the specified server for the user whose username and password are passed to the function.

The following is an example of what a call to `OCILogon()` might look like:

```
OCILogon(envhp, errhp, &svchp, "scott", nameLen, "tiger",
     passwdLen, "oracledb", dbnameLen);
```

The parameters to this call include the service context handle (which are initialized), the username, the user's password, and the name of the database that are used to

establish the connection. The server and user session handles are also implicitly allocated by this function.

If an application uses this logon method, the service context, server, and user session handles will all be read-only, which means that the application cannot switch session or transaction by changing the appropriate attributes of the service context handle, using `OCIAttrSet()`.

An application that initializes its session and authorization using `OCILogon()` should terminate them using `OCILogoff()`.

Option 2: Multiple Sessions or Connections

This option uses explicit attach and begin session calls.

If an application needs to maintain multiple user sessions on a database connection, the application requires a different set of calls to set up the sessions and connections. This includes specific calls to attach to the server and begin sessions:

- `OCIServerAttach()` creates an access path to the data server for OCI operations.

- `OCISessionBegin()` establishes a session for a user against a particular server. This call is required for the user to be able to execute any operation on the server.

> **Note:** See "Non-Blocking Mode" on page 2-41 for information about specifying a blocking or non-blocking connection in the `OCIServerAttach()` call.

These calls set up an operational environment that lets you to execute SQL and PL/SQL statements against a database. The database must be up and running before the calls are made, or else they will fail.

> **See Also:** These calls are described in more detail in "Connect, Authorize, and Initialize Functions" on page 15-4. Refer to Chapter 9, "OCI Programming Advanced Topics", for more information about maintaining multiple sessions, transactions, and connections.

Example of Creating and initializing an OCI Environment

The following example demonstrates the use of creating and initializing an OCI environment. In the example, a server context is created and set in the service

handle. Then a user session handle is created and initialized using a database
username and password. For the sake of simplicity, error checking is not included.

```
#include <s.h>
#include <oci.h>
...
main()
{
...
OCIEnv *myenvhp;       /* the environment handle */
OCIServer *mysrvhp;    /* the server handle */
OCIError *myerrhp;     /* the error handle */
OCISession *myusrhp;   /* user session handle */
OCISvcCtx *mysvchp;    /* the  service handle */
...
/* initialize the mode to be the threaded and object environment */
(void) OCIEnvCreate(&myenvhp, OCI_THREADED|OCI_OBJECT, (dvoid *)0,
                               0, 0, 0, (size_t) 0, (dvoid **)0);

     /* allocate a server handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&mysrvhp,
      OCI_HTYPE_SERVER, 0, (dvoid **) 0);

     /* allocate an error handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&myerrhp,
      OCI_HTYPE_ERROR, 0, (dvoid **) 0);

     /* create a server context */
(void) OCIServerAttach (mysrvhp, myerrhp, (text *)"inst1_alias",
      strlen ("inst1_alias"), OCI_DEFAULT);

     /* allocate a service handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&mysvchp,
      OCI_HTYPE_SVCCTX, 0, (dvoid **) 0);

 /* set the server attribute in the service context handle*/
(void) OCIAttrSet ((dvoid *)mysvchp, OCI_HTYPE_SVCCTX,
       (dvoid *)mysrvhp, (ub4) 0, OCI_ATTR_SERVER, myerrhp);

     /* allocate a user session handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&myusrhp,
    OCI_HTYPE_SESSION, 0, (dvoid **) 0);

     /* set username attribute in user session handle */
 (void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
```

```
        (dvoid *)"scott", (ub4)strlen("scott"),
        OCI_ATTR_USERNAME, myerrhp);

      /* set password attribute in user session handle */
 (void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
        (dvoid *)"tiger", (ub4)strlen("tiger"),
        OCI_ATTR_PASSWORD, myerrhp);

 (void) OCISessionBegin ((dvoid *) mysvchp, myerrhp, myusrhp,
        OCI_CRED_RDBMS, OCI_DEFAULT);

    /* set the user session attribute in the service context handle*/
 (void) OCIAttrSet (  (dvoid *)mysvchp, OCI_HTYPE_SVCCTX,
        (dvoid *)myusrhp, (ub4) 0, OCI_ATTR_SESSION, myerrhp);
...
}
```

The demonstration program `cdemo81.c` in the `demo` directory illustrates this process, with error-checking.

Processing SQL Statements

For information about processing SQL statements, refer to Chapter 4, "Using SQL Statements in OCI".

Commit or Rollback

An application commits changes to the database by calling `OCITransCommit()`. This call takes a service context as one of its parameters. The transaction currently associated with the service context is the one whose changes are committed. This may be a transaction explicitly created by the application or the implicit transaction created when the application modifies the database.

> **Note:** Using the OCI_COMMIT_ON_SUCCESS mode of the `OCIExecute()` call, the application can selectively commit transactions at the end of each statement execution, saving an extra roundtrip.

If you want to roll back a transaction, use the `OCITransRollback()` call.

If an application disconnects from Oracle in some way other than a normal logoff (for example, losing a network connection), and OCITransCommit() has not been called, all active transactions are rolled back automatically.

> **See Also:** For more information about implicit transactions and transaction processing, see the section "Service Context and Associated Handles" on page 2-9, and the section "OCI Support for Transactions" on page 8-2

Terminating the Application

An OCI application should perform the following three steps before it terminates:

1. Delete the user session by calling OCISessionEnd() for each session.

2. Delete access to the data source(s) by calling OCIServerDetach() for each source.

3. Explicitly deallocate all handles by calling OCIHandleFree() for each handle.

4. Delete the environment handle, which deallocates all other handles associated with it.

> **Note:** When a parent OCI handle is freed, any child handles associated with it are freed automatically

The calls to OCIServerDetach() and OCISessionEnd() are not mandatory, but are recommended. If the application terminates, and OCITransCommit() (transaction commit) has not been called, any pending transactions are automatically rolled back

> **See Also:** For an example showing handles being freed at the end of an application, refer to the first sample program in Appendix B, "OCI Demonstration Programs"

> **Note:** If the application has used the simplified logon method of OCILogon(), then a call to OCILogoff() terminates the session, disconnects from the server, and frees the service context and associated handles. The application is still responsible for freeing other handles it has allocated.

Error Handling

OCI function calls have a set of return codes, listed in Table 2–3, "OCI Return Codes", which indicate the success or failure of the call, such as OCI_SUCCESS or OCI_ERROR, or provide other information that may be required by the application, such as OCI_NEED_DATA or OCI_STILL_EXECUTING. Most OCI calls return one of these codes.

> **See Also:**   For exceptions, see "Functions Returning Other Values" on page 2-34

Table 2–3   OCI Return Codes

OCI Return Code	Description
OCI_SUCCESS	The function completed successfully.
OCI_SUCCESS_WITH_INFO	The function completed successfully; a call to OCIErrorGet() returns additional diagnostic information. This may include warnings.
OCI_NO_DATA	The function completed, and there is no further data.
OCI_ERROR	The function failed; a call to OCIErrorGet() returns additional information.
OCI_INVALID_HANDLE	An invalid handle was passed as a parameter or a user callback is passed an invalid handle or invalid context. No further diagnostics are available.
OCI_NEED_DATA	The application must provide run-time data.
OCI_STILL_EXECUTING	The service context was established in non-blocking mode, and the current operation could not be completed immediately. The operation must be called again to complete. OCIErrorGet() returns ORA-03123 as the error code.
OCI_CONTINUE	This code is returned only from a callback function. It indicates that the callback function wants the OCI library to resume its normal processing.

If the return code indicates that an error has occurred, the application can retrieve Oracle-specific error codes and messages by calling OCIErrorGet(). One of the parameters to OCIErrorGet() is the error handle passed to the call that caused the error.

> **Note:** Multiple diagnostic records can be retrieved by calling
> `OCIErrorGet()` repeatedly until there are no more records
> (OCI_NO_DATA is returned). `OCIErrorGet()` returns at most a
> single diagnostic record at any time.

The following example code returns error information given an error handle and
the return code from an OCI function call. If the return code is OCI_ERROR, the
function prints out diagnostic information. OCI_SUCCESS results in no printout,
and other return codes print the return code information.

```
STATICF void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  ub4 buflen;
  ub4 errcode;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet (errhp, (ub4) 1, (text *) NULL, &errcode,
                   errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %s\n", errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
default:
```

```
    break;
  }
}
```

Return and Error Codes for Truncation and Null Data

In Table 2–4, Table 2–5, and Table 2–6, the OCI return code, Oracle error number, indicator variable, and column return code are specified when the data fetched is null or truncated.

> **See Also:** See "Indicator Variables" on page 2-36 for a discussion of indicator variables.

Table 2–4   Normal Data - Not Null and Not Truncated

	Indicator - not provided	Indicator - provided
Return code - not provided	OCI_SUCCESS error = 0	OCI_SUCCESS error = 0 indicator = 0
Return code - provided	OCI_SUCCESS error = 0 return code = 0	OCI_SUCCESS error = 0 indicator = 0 return code = 0

Table 2–5   Null Data

	Indicator - not provided	Indicator - provided
Return code - not provided	OCI_ERROR error = 1405	OCI_SUCCESS error = 0 indicator = -1
Return code - provided	OCI_ERROR error = 1405 return code = 1405	OCI_SUCCESS error = 0 indicator = -1 return code = 1405

Table 2–6   Truncated Data

	Indicator - not provided	Indicator - provided
Return code - not provided	OCI_ERROR error = 1406	OCI_ERROR error = 1406 indicator = data_len
Return code - provided	OCI_SUCCESS_WITH_INFO error = 24345 return code = 1405	OCI_SUCCESS_WITH_INFO error = 24345 indicator = data_len return code = 1406

In Table 2–6, data_len is the actual length of the data that has been truncated if this length is less than or equal to SB2MAXVAL. Otherwise, the indicator is set to -2.

Functions Returning Other Values

Some functions return values other than the OCI error codes listed in Table 2–3. When using these function be sure to take into account that they return a value directly from the function call, rather than through an OUT parameter. More detailed information about each function and its return values is listed in the reference chapters. Some examples of these functions are:

- OCICollMax()

- OCIRawPtr()

- OCIRawSize()

- OCIRefHexSize()

- OCIRefIsEqual()

- OCIRefIsNull()

- OCIStringPtr()

- OCIStringSize()

Additional Coding Guidelines

This section explains some additional factors to keep in mind when coding applications using the Oracle Call Interface.

Parameter Types

OCI functions take a variety of different types of parameters, including integers, handles, and character strings. Special considerations must be taken into account for some types of parameters, as described in the following sections.

> **See Also:** For more information about parameter datatypes and parameter passing conventions, refer to "Connect, Authorize, and Initialize Functions" on page 15-4.

Address Parameters

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. In all cases, you should pass your pointers carefully.

Integer Parameters

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.

Character String Parameters

Character strings are a special type of address parameter. This section 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 length of the string.

> **7.x Upgrade Note:** Unlike earlier versions of the OCI, you do not pass -1 for the string length parameter of a null-terminated string.

Inserting Nulls into a Column

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 Also:**   "Indicator Variables" on page 2-36

One other method to insert a NULL is to set the buffer length and maximum length parameters both to zero on a bind call.

Note: Following SQL92 requirements, Oracle 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.

Indicator Variables

Each bind and define OCI call has a parameter that lets 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.

The C languages does not have the concept of null values; therefore you associate indicator variables with input variables to specify whether the associated placeholder is a NULL. When data is passed to Oracle, the values of these indicator variables determine whether or not a NULL is assigned to a database field.

For output variables, indicator variables determine whether the value returned from Oracle is a NULL or a truncated value. In the case of a NULL fetch (on OCIStmtFetch()) or a truncation (on OCIStmtExecute() or OCIStmtFetch()), the OCI call returns OCI_SUCCESS. The corresponding indicator variable is set to the appropriate value, as listed in Table 2–8, "Output Indicator Values". If the application has a return code variable in the corresponding OCIDefineByPos() call, the OCI assigns a value of ORA-01405 (for NULL fetch) or ORA-01406 (for truncation) to the return code variable.

The datatype of indicator variables is **sb2**. In the case of arrays of indicator variables, the individual array elements should be of type **sb2**.

Input

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

Table 2–7   Input Indicator Values

Input Indicator Value	Action Taken by Oracle
-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.

Output

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

Table 2–8   Output Indicator Values

Output Indicator Value	Meaning
-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 in the sb2 indicator variable.
-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.

Indicator Variables for Named Data Types and REFs

Indicator variables for most new (after release 8.0) datatypes function as described above. The only exception is SQLT_NTY (a named datatype). Data of type SQLT_REF uses a standard scalar indicator, just like other variable types. For data of type SQLT_NTY, the indicator variable must be a pointer to an indicator structure.

When database types are translated into C struct representations using the Object Type Translator (OTT), a null indicator structure is generated for each object type. This structure includes an atomic null indicator, plus indicators for each object attribute.

See Also: ■

- ■ See the documentation for the OTT in Chapter 14, "The Object Type Translator (OTT)", and the section "Nullity" on page 10-30 of this manual for information about null indicator structures

- ■ See the descriptions of `OCIBindByName()` and `OCIBindByPos()` in"Bind, Define, and Describe Functions" on page 15-50, and the sections "Information for Named Datatype and REF Binds" on page 11-37, and "Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-39, for more information about setting indicator parameters for named datatypes and REFs

Cancelling Calls

On most platforms, you can cancel a long-running or repeated OCI call. You do this by entering the operating system's interrupt character (usually CTRL-C) from the keyboard.

> **Note:** This is not to be confused with cancelling a cursor, which is accomplished by calling `OCIStmtFetch()` with the *nrows* parameter set to zero

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.

Given a particular service context pointer or server context pointer, the `OCIBreak()` function performs an immediate (asynchronous) abort of any currently executing OCI function that is associated with the server. It is normally used to stop a long-running OCI call being processed on the server. The `OCIReset()` function is necessary to perform a protocol synchronization on a non-blocking connection after an OCI application aborts a function with `OCIBreak()`.

The status of potentially long-running calls can be monitored through the use of non-blocking calls. See the section "Non-Blocking Mode" on page 2-41 for more information.

Positioned Updates and Deletes

You can use the ROWID associated with a SELECT...FOR UPDATE OF... statement in a later UPDATE or DELETE statement. The ROWID is retrieved by calling OCIAttrGet() on the statement handle to retrieve the handle's OCI_ATTR_ROWID attribute.

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 attribute in the handle contains the row identifier of the selected row. You can retrieve the ROWID into a buffer in your program by calling OCIAttrGet() as follows:

```
OCIRowid *rowid;   /* the rowid in opaque format */
/* allocate descriptor with OCIDescriptorAlloc() */
err = OCIDescriptorAlloc ((dvoid *) envhp, (dvoid **) &rowid,
    (ub4) OCI_TYPE_ROWID, (size_t) 0, (dvoid **) 0));
err = OCIAttrGet ((dvoid*) mystmtp, OCI_HTYPE_STMT,
    (dvoid*) rowid, (ub4 *) 0, OCI_ATTR_ROWID, (OCIError *) myerrhp);
```

You can then use the saved ROWID in a DELETE or UPDATE statement. For example, if *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 *rowid* to the :2 placeholder. Be sure to use datatype code 104 (ROWID descriptor) when binding *rowid* to :2.

Using prefetching, an array of ROWIDs can be selected for use in subsequent batch updates.

> **See Also:** For more information on ROWIDs, see "UROWID" on page 3-6 and "DATE" on page 3-15.

Reserved Words

Some words are reserved by Oracle. That is, they have a special meaning to Oracle and cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes.

> **See Also:** To view the lists of the Oracle keywords or reserved words for SQL and PL/SQL, see *the Oracle9i SQL Reference and the PL/SQL User's Guide and Reference*

Oracle Reserved Namespaces

Table 2–9, "Oracle Reserved Namespaces" 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, do not use function names that begin with these characters. For example, the Oracle Net Transparent Network Service functions all begin with the characters NS, so you need to avoid naming functions that begin with NS.

Table 2–9   Oracle Reserved Namespaces

Namespace	Library
XA	external functions for XA applications only
SQ	external SQLLIB functions used by Oracle Precompiler and SQL*Module applications
O, OCI	external OCI functions internal OCI functions
UPI, KP	function names from the Oracle UPI layer
NA	Oracle Net Native Services Product
NC	Oracle Net Rpc Project
ND	Oracle Net Directory
NL	Oracle Net Network Library Layer
NM	Oracle Net Management Project
NR	Oracle Net Interchange
NS	Oracle Net Transparent Network Service
NT	Oracle Net Drivers
NZ	Oracle Net Security Service
OS	SQL*Net V1
TTC	Oracle Net Two Task
GEN, L, ORA	Core library functions
LI, LM, LX	function names from the Oracle Globalization Support layer
S	function names from system-dependent libraries

Table 2–9   Oracle Reserved Namespaces (Cont.)

Namespace	Library
KO	Kernel Objects

The list in Table 2–9, "Oracle Reserved Namespaces" is not a comprehensive list of all functions within the Oracle reserved namespaces. For a complete list of functions within a particular namespace, refer to the document that corresponds to the appropriate Oracle library.

Function Names

When creating a user function in an OCI program, do not start the function name with *OCI* to avoid possible conflicts with the OCI functions.

Application Linking

For information about application linking modes, including Oracle support for non-deferred linking and single task linking in various versions of the OCI, please see "" on page 1-24.

Non-Blocking Mode

The OCI provides the ability to establish a server connection in *blocking mode* or *non-blocking mode.* When a connection is made in blocking mode, an OCI call returns control to an OCI client application only when the call 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, and the call returns a value of OCI_STILL_EXECUTING.

In non-blocking mode, an application must test the return code of each OCI function to see if it returns OCI_STILL_EXECUTING. In this case, the OCI client can continue to process program logic while waiting to retry the OCI call to the server.

The non-blocking mode returns control to an OCI program once a call has been made 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, real-time applications, and in distributed environments.

The non-blocking 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 client application must check whether the

pending call has finished at the server by executing the call again *with the exact same parameters*.

> **Note:** While waiting to retry non-blocking OCI call, the application *may not* issue any other OCI calls, or an ORA-03124 error will occur. The only exceptions to this rule are OCIBreak() and OCIReset().
>
> See "Cancelling a Non-blocking Call" on page 2-42 for more information on these calls

Setting Blocking Modes

You can modify or check an application's blocking status by calling OCIAttrSet() to set the status or OCIAttrGet() to read the status on the server context handle with the *attrtype* parameter set to OCI_ATTR_NONBLOCKING_MODE.

> **See Also:** See OCI_ATTR_NONBLOCKING_MODE on page A-15.

> **Note:** Only functions that have server context or a service context handle as a parameter may return OCI_STILL_EXECUTING.

Cancelling a Non-blocking Call

You can cancel a long-running OCI call by using the OCIBreak() function. After issuing an OCIBreak() while an OCI call is in progress, you must issue an OCIReset() call to reset the asynchronous operation and protocol.

Non-blocking Example

The following code is an example of non-blocking mode.

```
int main (int argc, char **argv)
{
  sword retval;

  if (retval = InitOCIHandles()) /* initialize all handles */
  {
    printf ("Unable to allocate handles..\n");
    exit (EXIT_FAILURE);
```

```
  }

  if (retval = logon()) /* log on */
  {
    printf ("Unable to log on...\n");
    exit (EXIT_FAILURE);
  }
  if (retval = AllocStmtHandle ()) /* allocate statement handle */
  {
    printf ("Unable to allocate statement handle...\n");
    exit (EXIT_FAILURE);
  }
/* set non-blocking on */
  if (retval = OCIAttrSet ((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER,
                           (dvoid *) 0, (ub4) 0,
                           (ub4) OCI_ATTR_NONBLOCKING_MODE, errhp))
  {
    printf ("Unable to set non-blocking mode...\n");
    exit (EXIT_FAILURE);
  }

  while ((retval = OCIStmtExecute (svchp, stmhp, errhp, (ub4)0, (ub4)0,
                       (OCISnapshot *) 0, (OCISnapshot *)0,
                       OCI_DEFAULT)) == OCI_STILL_EXECUTING)
    printf (".");
  printf ("\n");

  if (retval != OCI_SUCCESS || retval != OCI_SUCCESS_WITH_INFO)
  {
    printf("Error in OCIStmtExecute...\n");
    exit (EXIT_FAILURE);
  }

  if (retval = logoff ()) /* log out */
  {
    printf ("Unable to logout ...\n");
    exit (EXIT_FAILURE);
  }

  cleanup();
  return (int)OCI_SUCCESS;
}
...
```

Using PL/SQL in an OCI Program

PL/SQL is Oracle's procedural extension to the SQL language. PL/SQL processes tasks that are more complicated than simple queries and SQL data manipulation language (DML) statements. PL/SQL lets you to group a number of constructs into a single block and execute them as a unit. These constructs include:

- 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
- Operate on objects in an Oracle8 server

Note: While the OCI can only directly process anonymous blocks, and not named packages or procedures, you can always put the package or procedure call within an anonymous block and process that block.

Note that all OUT variables have to be initialized to NULL (through an indicator of -1, or an actual length of 0) prior to executing a PL/SQL begin-end block in OCI.

OCI does not support the PL/SQL RECORD datatype.

Caution: When writing PL/SQL code, it is important to keep in mind that the parser treats everything that starts with "--" to a carriage return as a comment. So if comments are indicated on each line by "--", the C compiler can concatenate all lines in a PL/SQL block into a single line without putting a carriage return "\n" for each line. In this particular case, the parser fails to extract the PL/SQL code of a line if the previous line ends with a comment. To avoid the problem, the programmer should put "\n" after each "--" comment to make sure the comment ends there.

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

Globalization Support

OCI supports UTF-16 Unicode encoding formats in string-based function calls (for SQL statements, data, metadata such as user and password, object support, in error messages).

UTF-16 is a variable-width Unicode encoding built from 2-byte UCS2. Any character in any language can be represented in UTF-16. AL16UTF16 is the Oracle character set name for UTF-16. UTF-16 is a superset of UCS2.

ASCII and other native character sets are supported as well. Strings in ASCII follow so-called byte length semantics. Unicode strings have character length semantics.

OCI is the interface between users and servers, so describe, insert, update and fetch operations must be aware of codepoint-length semantics.

> **See Also:**
>
> - "Character Conversion Issues in Binding and Defining" on page 5-35 for more information about Unicode support in OCI calls
>
> - "Character Length Semantics Support in Describing" on page 6-21
>
> - For more information about Unicode support in OCI see the *Oracle9i Globalization and National Language Support Guide*

UTF-16 Environment Mode

OCI calls are constructed as hierarchical layers with inheritance between handles. Whether and how UTF-16 is used in OCI will be determined by setting up OCI environment handles through the mode parameter and using the concept of inheritance of handles. The way OCI handles are implemented allows lower level handles such as OCI statement handles to get the UTF-16 settings from their parents or to have OCI environment handles as default. Except for the environment, all handles are stateless for UTF-16 setting. In other words, they do not save UTF-16 information. So users should always rely on the parent environment handle in terms of UTF-16 setting.

Aside from `OCIEnvCreate()`, all these functions have one characteristic which is they take **text** * parameters which might point to UTF-16 buffers. However, some OCI functions, in order to simplify the OCI, require some parameters which are string primitives to be passed down as **dvoid** * followed by length parameters.

For instance, `OCIAttrGet()`, which retrieves handle attribute information, asks for attribute name returned out to be cast to **dvoid** * type, and `OCIDescribeAny()`, which describes existing schema objects, takes object name passed in as a **dvoid** * parameter. These **dvoid** * parameters can come from string buffers, such as names.

For **text** * parameters, their corresponding length values, no matter whether passed in or out, are always the number of bytes for the whole string, regardless of UTF-16 setting. In other words, the length should always be twice the number of UTF-16 characters in the string.

The only exception to this rule is the group of functions with existing UTF-16 support where the length parameters refer to number of characters.

Top-level Environment Creation

`OCIEnvCreate()` initializes an environment handle from which other OCI handles and structures get the fundamental information. A user can set UTF-16 encoding for the whole environment through this function call

Relational OCI Functions with Text Input

The following functions communicate with the server, but do not take mode parameters as they do not need to. They choose the mode setting embedded in the environment or statement handles that are passed down to them.

- `OCIStmtPrepare()`
- `OCIBindByName()`
- `OCIServerAttach()`
- `OCIPasswordChange()`
- `OCILogon()`
- `OCILobFileSetName()`
- `OCIAttrSet()`
- `OCIDescribeAny()`

If mixed programming for both UTF-16 and non-UTF-16 buffers is inevitable for an application, it is advised to create two separated environment handles each dealing with one encoding respectively, because most relational OCI and object OCI calls take only one environment handle at a time. For any functions taking more than text parameter, these string buffers should always be prepared and treated in the

same encoding. Mixed encoding for different string parameters in one function call is invalid.

Relational OCI Functions with Text Output

The same advice to users as for the functions with text input applies to these functions:

- `OCIStmtGetBindInfo()`
- `OCIErrorGet()`
- `OCIServerVersion()`
- `OCILobFileGetName()`
- `OCIAttrGet()`

OCI String Functions with UTF-16 Data

As a scalar type **OCIString**, the encoding of the buffer depends on the environment handle it belongs to. Being treated as a datatype SQLT_VST, **OCIString** behaves in the same way as normal string types for bind and define handles.

If the environment handle is created with OCI_UTF16 mode, the data in **OCIString** should be in UTF-16 encoding. Otherwise, the data is in NLS_LANG encoding. The default character set in the corresponding bind or define handles are OCI_UTF16ID, which means UTF-16. The size parameters are always in bytes. The related functions are:

- `OCIStringAssignText()` - can assign a UTF-16 text to **OCIString**
- `OCIStringResize()` - resizes the **OCIString** in bytes
- `OCIStringAllocSize()` - allocates **OCIString** in bytes
- `OCIStringPtr()` - returns a text pointer to an **OCIString** which can be in UTF-16
- `OCIStringSize()` - returns size in bytes

Character Length Semantics

OCI works as a translator between server and client, and passes around character information for constraint checking.

There are two kinds of character sets in the world, variable-width and fixed-width, as a single byte character set is just a special fixed-width character set where each byte stands for one character.

For fixed-width character sets, constraint checking is easier as number of bytes is simply equal to a multiple of number of characters. Therefore, no scanning through the whole string to find out number of characters is needed for fixed-width character sets. However, for variable-width ones, complete scanning is inevitable to find out the number of characters.

Character Set Support

See "Character Length Semantics Support in Describing" on page 6-21 and "Character Conversion Issues in Binding and Defining" on page 5-35 for a complete discussion.

3

Datatypes

This chapter provides a reference to Oracle external datatypes used by OCI
applications. It also provides a general discussion of Oracle datatypes, including
special datatypes new in the latest Oracle release. The information in this chapter is
useful for understanding the conversions between internal and external
representations that occur when you transfer data between your program and
Oracle. This chapter contains the following sections:

- Oracle Datatypes

- Internal Datatypes

- External Datatypes

- New External Datatypes

- Data Conversions

- Typecodes

- Definitions in oratypes.h

> **See Also:** For detailed information about Oracle internal
> datatypes, see the *Oracle9i SQL Reference*

Oracle Datatypes

One of the main functions of an OCI program is to communicate with a database through an Oracle server. The OCI application may retrieve data from database tables through SQL SELECT queries, or it may modify existing data in tables through INSERT, UPDATE, or DELETE statements.

Inside a database, values are stored in columns in tables. Internally, Oracle represents data in particular formats known as *internal datatypes.* Examples of internal datatypes include NUMBER, CHAR, and DATE.

In general, OCI applications do not work with internal datatype representations of data. OCI applications work with host language datatypes which are predefined by the language in which they are written. When data is transferred between an OCI client application and a database table, the OCI libraries convert the data between internal datatypes and *external datatypes.*

External datatypes are host language types that have been defined in the OCI header files. When an OCI application binds input variables, one of the bind parameters is an indication of the external datatype code (or *SQLT code)* of the variable. Similarly, when output variables are specified in a define call, the external representation of the retrieved data must be specified.

In some cases, external datatypes are similar to internal types. External types provide a convenience for the programmer by making it possible to work with host language types instead of proprietary data formats.

> **Note:** Even though some external types are similar to internal types, an OCI application never binds to internal datatypes. They are discussed here because it can be useful to understand how internal types can map to external types.

The OCI is capable of performing a wide range of datatype conversions when transferring data between Oracle and an OCI application. There are more OCI external datatypes than Oracle internal datatypes. In some cases a single external type maps to an internal type; in other cases multiple external types map to an single internal type.

The many-to-one mappings for some datatypes provide flexibility for the OCI programmer. 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 *dty* parameter in the OCIDefineByPos() call for the sal column. You also need to declare a string variable in your program and specify its address in the *valuep* 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 *valuep* 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. You must tell Oracle where the input or output variables are in your OCI program and their datatypes and lengths.

OCI also supports an additional set of OCI typecodes which are used by Oracle's type management system to represent datatypes of object type attributes. There is a set of predefined constants which can be used to represent these typecodes. The constants each contain the prefix OCI_TYPECODE.

In summary, the OCI programmer must be aware of the following different datatypes or data representations:

■ Internal Oracle datatypes, which are used by table columns in an Oracle database. These also include datatypes used by PL/SQL which are not used by Oracle columns (e.g., indexed table, boolean, record).

> **See Also:** "Internal Datatypes" on page 3-4

■ External OCI datatypes, which are used to specify host language representations of Oracle data.

> **See Also:** "External Datatypes" on page 3-7, and "Using External Datatype Codes" on page 3-4

■ OCI_TYPECODE values, which are used to Oracle to represent type information for object type attributes.

> **See Also:** "Typecodes" on page 3-31, and "Relationship Between SQLT and OCI_TYPECODE Values" on page 3-33

Information about a column's internal datatype is conveyed to your application in the form of an internal datatype code. Once your application knows what type of data will be returned, it can make appropriate decisions about how to convert and format the output data. The Oracle internal datatype codes are listed in the section "Internal Datatypes" on page 3-4.

> **See Also:** For detailed information about Oracle internal datatypes, see the *Oracle9i SQL Reference.* For information about describing select-list items in a query, see the section "Describing Select-List Items" on page 4-12.

Using External Datatype Codes

An external datatype code indicates to Oracle how a 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 in the OCIDefineByPos() 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 02-FEB-65 to a DATE column, specify the datatype as a character string 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.

> **See Also:** For a complete list of the external datatypes and datatype codes, see Table 3–2, "External Datatypes and Codes"

Internal Datatypes

The following table lists the Oracle internal (also known as *built-in*) datatypes, along with each type's maximum internal length and datatype code.

Table 3–1   Internal Oracle Datatypes

Internal Oracle Datatype	Maximum Internal Length	Datatype Code
VARCHAR2, NVARCHAR2	4000 bytes	1
NUMBER	21 bytes	2
LONG	2^31-1 bytes (2 gigabytes)	8
ROWID	10 bytes	11
DATE	7 bytes	12
RAW	2000 bytes	23
LONG RAW	2^31-1 bytes	24
CHAR, NCHAR	2000 bytes	96
User-defined type (object type, VARRAY, Nested Table)	N/A	108
REF	N/A	111
CLOB, NCLOB	4 gigabytes	112
BLOB	4 gigabytes	113
BFILE	4 gigabytes	114
TIMESTAMP	11 bytes	180
TIMESTAMP WITH TIME ZONE	13 bytes	181
INTERVAL YEAR TO MONTH	5 bytes	182
INTERVAL DAY TO SECOND	11 bytes	183
UROWID	3950 bytes	208
TIMESTAMP WITH LOCAL TIME ZONE	11 bytes	231

> **See Also:**   For more information about these built-in datatypes,
> see the *Oracle9i SQL Reference.* The following sections provide
> OCI-specific information about these datatypes.

LONG, RAW, LONG RAW, VARCHAR2

You can use the piecewise capabilities provided by `OCIBindByName()`,
`OCIBindByPos()`, `OCIDefineByPos()`, `OCIStmtGetPieceInfo()` and

`OCIStmtSetPieceInfo()` to perform inserts, updates or fetches involving column data of these types.

Character Strings and Byte Arrays

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

> **Note:** LOBs can contain characters and FILEs can contain binary data. They are handled differently than other types, so they are not included in this discussion. See Chapter 7, "LOB and FILE Operations", for more information about these data types.

`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.

UROWID

The Universal `ROWID` (`UROWID`) is a datatype that can store both logical and physical rowids of Oracle tables, and rowids of the foreign tables, such as DB2 tables accessed by a gateway. Logical rowids are primary key-based logical identifiers for the rows of Index-Organized Tables (IOTs).

To use columns of the `UROWID` datatype, the value of the `COMPATIBLE` initialization parameter must be set to **8.1** or higher.

The following host variables can be bound to Universal ROWIDs:

- SQLT_CHR (`VARCHAR2`)

- SQLT_VCS (VARCHAR)
- SQLT_STR (Null-Terminated string)
- SQLT_LVC (long varchar)
- SLQT_AFC (CHAR)
- SQLT_AVC (CHARZ)
- SQLT_VST (OCI String)
- SQLT_RDD (ROWID descriptor)

External Datatypes

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

Table 3–2   External Datatypes and Codes

EXTERNAL DATATYPE		TYPE OF PROGRAM	
NAME	CODE	VARIABLE	OCI DEFINED CONSTANT
VARCHAR2	1	char[n]	SQLT_CHR
NUMBER	2	unsigned char[21]	SQLT_NUM
8-bit signed INTEGER	3	signed char	SQLT_INT
16-bit signed INTEGER	3	signed short, signed int	SQLT_INT
32-bit signed INTEGER	3	signed int, signed long	SQLT_INT
FLOAT	4	float, double	SQLT_FLT
Null-terminated STRING	5	char[n+1]	SQLT_STR
VARNUM	6	char[22]	SQLT_VNU
LONG	8	char[n]	SQLT_LNG
VARCHAR	9	char[n+sizeof(short integer)]	SQLT_VCS
DATE	12	char[7]	SQLT_DAT
VARRAW	15	unsigned char[n+sizeof(short integer)]	SQLT_VBI
RAW	23	unsigned char[n]	SQLT_BIN

Table 3–2   External Datatypes and Codes (Cont.)

EXTERNAL DATATYPE		TYPE OF PROGRAM	
NAME	CODE	VARIABLE	OCI DEFINED CONSTANT
LONG RAW	24	unsigned char[n]	SQLT_LBI
UNSIGNED INT	68	unsigned	SQLT_UIN
LONG VARCHAR	94	char[n+sizeof(integer)]	SQLT_LVC
LONG VARRAW	95	unsigned char[n+sizeof(integer)]	SQLT_LVB
CHAR	96	char[n]	SQLT_AFC
CHARZ	97	char[n+1]	SQLT_AVC
ROWID descriptor	104	OCIRowid *	SQLT_RDD
NAMED DATA TYPE	108	struct	SQLT_NTY
REF	110	OCIRef	SQLT_REF
Character LOB descriptor	112	OCILobLocator (see note 3)	SQLT_CLOB
Binary LOB descriptor	113	OCILobLocator (see note 3)	SQLT_BLOB
Binary FILE descriptor	114	OCILobLocator	SQLT_FILE
OCI string type	155	OCIString	SQLT_VST (see note 2)
OCI date type	156	OCIDate *	SQLT_ODT (see note 2)
ANSI DATE descriptor	184	OCIDateTime *	SQLT_DATE
TIMESTAMP descriptor	187	OCIDateTime *	SQLT_TIMESTAMP
TIMESTAMP WITH TIME ZONE descriptor	188	OCIDateTime *	SQLT_TIMESTAMP_TZ
INTERVAL YEAR TO MONTH descriptor	189	OCIInterval *	SQLT_INTERVAL_YM

Table 3–2   External Datatypes and Codes (Cont.)

| EXTERNAL DATATYPE | | TYPE OF PROGRAM | |
NAME	CODE	VARIABLE	OCI DEFINED CONSTANT
INTERVAL DAY TO SECOND descriptor	190	OCIInterval *	SQLT_INTERVAL_DS
TIMESTAMP WITH LOCAL TIME ZONE descriptor	232	OCIDateTime *	SQLT_TIMESTAMP_LTZ

Notes:

(1) This type is valid only for version 7.x OCI calls. OCI release 8 or later applications should use the ROWID descriptor (type 104).

(2) For more information on the use of these datatypes, refer to Chapter 11, "Object-Relational Datatypes".

(3) In applications using datatype mappings generated by OTT, CLOBs may be mapped as OCIClobLocator, and BLOBs may be mapped as OCIBlobLocator. For more information, refer to Chapter 14, "The Object Type Translator (OTT)".

> **Note:**   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)

Each of the external datatypes is described below. Datatypes that are new as of release 8.0 or later are described in the section "New External Datatypes" on page 3-19.

The following three types are internal to PL/SQL and cannot be returned as values by OCI:

- Boolean, SQLT_BOL
- Indexed Table, SQLT_TAB
- Record, SQLT_REC

VARCHAR2

The VARCHAR2 datatype is a variable-length string of characters with a maximum length of 4000 bytes.

> **Note:** If you are using Oracle objects, you can work with a special **OCIString** external datatype using a set of predefined OCI functions. Refer to Chapter 11, "Object-Relational Datatypes" for more information about this datatype.

Input

The `value_sz` parameter determines the length in the `OCIBindByName()` or `OCIBindByPos()` call.

If the `value_sz` 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 `value_sz` 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 inserted into the database.

Output

Specify the desired length for the return value in the `value_sz` parameter of the `OCIDefineByPos()` call, or the `value_sz` parameter of `OCIBindByName()` or `OCIBindByPos()` for PL/SQL blocks. If zero is specified for the length, no data is returned.

If you omit the `rlenp` parameter of `OCIDefineByPos()`, returned values are blank-padded to the buffer length, and nulls are returned as a string of blank

characters. If `rlenp` is included, returned values are not blank-padded. Instead, their actual lengths are returned in the `rlenp` parameter.

To check if a null is returned or if character truncation has occurred, include an indicator parameter in the `OCIDefineByPos()` call. Oracle sets the indicator parameter to -1 when a null is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to. If you do not specify an indicator parameter and a null is selected, the fetch call returns the error code OCI_SUCCESS_WITH_INFO. Retrieving diagnostic information on the error will return ORA-1405.

> **See Also:**   "Indicator Variables" on page 2-36

You can also request output to a character string from an internal NUMBER datatype. 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.

NUMBER

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

> **Note:**   If you are using objects in an Oracle database server, you can work with a special **OCINumber** datatype using a set of predefined OCI functions. Refer to Chapter 11, "Object-Relational Datatypes" for more information about this datatype.

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 and it is cleared for negative numbers. The lower 7 bits represent the exponent, which is a base-100 digit with an offset of 65.

To calculate the decimal exponent, add 65 to the base-100 exponent and add another 128 if the number is positive. If the number is negative, you do the same, but subsequently the bits are inverted. For example, -5 has a base-100 exponent = 62 (0x3e). The decimal exponent is thus (~0x3e) -128 - 65 = 0xc1 -128 -65 = 193 -128 -65 = 0.

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 *dty* parameter of an OCIDefineByPos() 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 3-14 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 returned 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, Oracle returns an "overflow on conversion" error.

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.

The internal format of an Oracle number is decimal, and most floating-point implementations are binary; therefore Oracle can represent numbers with greater precision than floating-point representations.

> **Note:**   You may receive 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 for the operation.

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 language programs.

Input

The string length supplied in the OCIBindByName() or OCIBindByPos() 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, the OCI uses an implied maximum
string length of 4000.

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.

> **Note:**   Unlike earlier versions of the OCI, in release 8.0 or later,
> you cannot pass -1 for the string length parameter of a
> null-terminated string

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 send a VARNUM value to Oracle.

The following table shows several examples of the VARNUM values returned for numbers in an Oracle table.

Table 3–3   VARNUM Examples

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

LONG

The LONG datatype stores character strings longer than 4000 bytes. You can store up to two gigabytes ($2^{31}-1$ 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.

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.

Table 3–4   Format of the DATE Datatype

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

The century and year bytes (bytes 1 and 2) are in excess-100 notation. The first byte stores the value of the year, which is 1992, as an integer, divided by 100, giving 119 in excess-100 notation. The second byte stores year modulo 100, giving 192. 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 enter 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 DATE into character format, because the program usually deals with data in a character format, such as DD-MON-YY.

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.

> **Note:** If you are using objects in an Oracle database, you can work with a special **OCIDate** datatype using a set of predefined OCI functions.
>
> - Refer to Chapter 11, "Object-Relational Datatypes" for more information about this datatype.
>
> - For information about DATETIME and INTERVAL datatypes, refer to "Datetime and Interval Datatype Descriptors" on page 3-23.

RAW

The RAW datatype is used for binary data or byte strings that are not to be interpreted by Oracle, for example, to store graphics character sequences. The maximum length of a RAW column is 2000 bytes.

> **See Also:** *Oracle9i SQL Reference.*

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 OCIDefineByPos(), OCIBindByName(), OCIBindByPos(), OCIStmtGetPieceInfo(), and OCIStmtSetPieceInfo() to perform inserts, updates, or fetches involving RAW (or LONG RAW) columns.

> **Note:** If you are using objects in an Oracle database, you can work with a special **OCIRaw** datatype using a set of predefined OCI functions. Refer to Chapter 11, "Object-Relational Datatypes" for more information about this datatype.

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.

LONG RAW

The LONG RAW datatype is similar to the RAW datatype, except that it stores raw data with a length up to two gigabytes ($2^{31}-1$ 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, Oracle returns an "overflow on conversion" error.

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.

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.

CHAR

The CHAR datatype is a string of characters, with a maximum length of 2000. CHAR strings are compared using blank-padded comparison semantics

> **See Also:** *Oracle9i SQL Reference*

Input
The length is determined by the *value_sz* parameter in the OCIBindByName() or OCIBindByPos() call.

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

If the `value_sz` 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 for the `value_sz` 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 insert 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 `value_sz` parameter of the `OCIDefineByPos()` call. If zero is specified for the length, no data is returned.

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

To check whether a null is returned or if character truncation has occurred, include an indicator parameter or array of indicator parameters in the `OCIDefineByPos()` 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.

> **See Also:** "Indicator Variables" on page 2-36

You can also request output to a character string from an internal NUMBER datatype. 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.

New External Datatypes

The following new external datatypes were introduced with or after release 8.0. These datatypes are not supported when you connect to an Oracle release 7 server.

> **Note:** Both internal and external datatypes have Oracle-defined constant values, such as SQLT_NTY, SQLT_REF, corresponding to their datatype codes. Although the constants are not listed for all of the types in this chapter, they are used in this section when discussing new Oracle datatypes. The datatype constants are also used in other chapters of this guide when referring to these new types.

Named Data Types (Object, VARRAY, Nested Table)

Named data types are user-defined types which are specified with the CREATE TYPE command in SQL. Examples include object types, varrays, and nested tables. In the OCI, *named data type* refers to a host language representation of the type. The SQLT_NTY datatype code is used when binding or defining named data types.

In a C application, named data types are represented as C structs. These structs can be generated from types stored in the database by using the Object Type Translator. These types correspond to OCI_TYPECODE_OBJECT.

See Also:

- For more information about working with named data types in the OCI, refer to Part 2 of this guide.

- For information about how named data types are represented as C structs, refer to Chapter 14, "The Object Type Translator (OTT)".

REF

This is a reference to a named data type. The C language representation of a REF is a variable declared to be of type **OCIRef** *. The SQLT_REF datatype code is used when binding or defining REFs.

Access to REFs is only possible when an OCI application has been initialized in object mode. When REFs are retrieved from the server, they are stored in the client-side object cache.

To allocate a REF for use in your application, you should declare a variable to be a pointer to a REF, and then call OCIObjectNew(), passing OCI_TYPECODE_REF as the *typecode* parameter.

> **See Also:** For more information about working with REFs in the OCI, refer to Part II of this guide

ROWID Descriptor

The ROWID datatype identifies a particular row in a database table. ROWID can be a select-list item in a query, such as:

```
SELECT ROWID, ename, empno FROM emp
```

In this case, you can use the returned ROWID in further DELETE statements.

If you are performing a SELECT for UPDATE, the ROWID is implicitly returned. This ROWID can be read into a user-allocated ROWID descriptor using OCIAttrGet() on the statement handle and used in a subsequent UPDATE statement. The prefetch operation fetches all ROWIDs on a SELECT for UPDATE; use prefetching and then a single row fetch.

You access rowids through the use of a ROWID descriptor, which you can use as a bind or define variable.

> **See Also:** See the sections "Descriptors" on page 2-15 and "Positioned Updates and Deletes" on page 2-39 for more information about the use of the ROWID descriptor

LOB Descriptor

A LOB (Large Object) stores binary or character data up to 4 gigabytes in length. Binary data is stored in a BLOB (Binary LOB), and character data is stored in a CLOB (Character LOB) or NCLOB (National Character LOB).

LOB values may or may not be stored inline with other row data in the database. In either case, LOBs have the full transactional support of the database server. A database table stores a *LOB locator* which points to the LOB value which may be in a different storage space.

When an OCI application issues a SQL query which includes a LOB column or attribute in its select-list, fetching the result(s) of the query returns the locator, rather than the actual LOB value. In the OCI, the LOB locator maps to a variable of type **OCILobLocator**.

> **See Also:**
>
> - For more information about descriptors, including the LOB locator, see the section "Descriptors" on page 2-15
>
> - For more information about LOBs refer to the *Oracle9i SQL Reference* and the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

The OCI functions for LOBs take a LOB locator as one of their arguments. The OCI functions assume that the locator has already been created, whether or not the LOB to which it points contains data.

Bind and define operations are performed on the LOB locator, which is allocated with the OCIDescriptorAlloc() function.

The locator is always fetched first using SQL or OCIObjectPin(), and then operations are performed using the locator. The OCI functions never take the actual LOB value as a parameter.

> **See Also:** For more information about OCI LOB functions, see Chapter 7, "LOB and FILE Operations"

The datatype codes available for binding or defining LOBs are:

- SQLT_BLOB - a binary LOB data type.

- SQLT_CLOB - a character LOB data type.

The NCLOB is a special type of CLOB with the following requirements:

- To write into or read from an NCLOB, the user must set the character set form (*csfrm*) parameter to be SQLCS_NCHAR.

- The amount (*amtp*) parameter in calls involving CLOBs and NCLOBs is always interpreted in terms of characters, rather than bytes, for fixed-width character sets.

> **See Also:** "LOB and FILE Functions" on page 7-5

BFILE

The BFILE datatype provides access to file LOBs that are stored in file systems outside an Oracle database. Oracle currently only supports access to binary files, or BFILEs.

A BFILE column or attribute stores a file LOB locator, which serves as a pointer to a binary file on the server's file system. The locator maintains the directory alias and the filename.

Binary file LOBs do not participate in transactions. Rather, the underlying operating system provides file integrity and durability. The maximum file size supported is 4 gigabytes.

The database administrator must ensure that the file exists and that Oracle processes have operating system read permissions on the file.

The BFILE datatype allows read-only support of large binary files; you cannot modify a file through Oracle. Oracle provides APIs to access file data.

The datatype code available for binding or defining FILEs is:

- SQLT_BFILE - a binary FILE LOB data type

> **See Also:** For more information about directory aliases, refer to the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

BLOB

The BLOB datatype stores unstructured binary large objects. BLOBs can be thought of as bitstreams with no character set semantics. BLOBs can store up to four gigabytes of binary data.

BLOBs have full transactional support; changes made through the OCI participate fully in the transaction. The BLOB value manipulations can be committed or rolled back. You cannot save a BLOB locator in a variable in one transaction and then use it in another transaction or session.

CLOB

The CLOB datatype stores fixed- or varying-width character data. CLOBs can store up to 4 gigabytes of character data.

CLOBs have full transactional support; changes made through the OCI participate fully in the transaction. The CLOB value manipulations can be committed or rolled back. You cannot save a CLOB locator in a variable in one transaction and then use it in another transaction or session.

NCLOB. An NCLOB is a national character version of a CLOB. It stores fixed-width, single-byte or multibyte national character set (NCHAR) data, or varying-width character set data. NCLOBs can store up to 4 gigabytes of character text data.

NCLOBs have full transactional support; changes made through the OCI participate fully in the transaction. NCLOB value manipulations can be committed or rolled back. You cannot save a NCLOB locator in a variable in one transaction and then use it in another transaction or session.

You cannot create an object with NCLOB attributes, but you can specify NCLOB parameters in methods.

Datetime and Interval Datatype Descriptors

The datetime and interval datatype descriptors are briefly summarized here.

> **See Also:**   For more a more complete discussion, see *Oracle9i SQL Reference*

ANSI DATE

The ANSI DATE is based on the DATE, but contains no time portion. (Therefore, it also has no time zone.) ANSI DATE follows the ANSI specification for the DATE datatype. When assigning an ANSI DATE to a DATE or a timestamp datatype, the time portion of the Oracle DATE and the timestamp are set to zero. When assigning a DATE or a timestamp to an ANSI DATE, the time portion is ignored.

You are encouraged to instead use the TIMESTAMP datatype which contains both date and time.

TIMESTAMP

The TIMESTAMP datatype is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, plus the hour, minute, and second values. It has no time zone. The TIMESTAMP datatype has the form:

```
TIMESTAMP(fractional_seconds_precision)
```

where *fractional_seconds_precision* (which is optional) specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE (TSTZ) is a variant of TIMESTAMP that includes an explicit time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The TIMESTAMP WITH TIME ZONE datatype has the form:

```
TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE
```

where *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data.

TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ) is another variant of TIMESTAMP that includes a time zone displacement in its value. Storage is in the same format as for TIMESTAMP. This type differs from TIMESTAMP WITH TIME ZONE in that data stored in the database is normalized to the database time zone, and the time zone displacement is not stored as part of the column data. When users retrieve the data, Oracle returns it in the users' local session time zone.

The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The TIMESTAMP WITH LOCAL TIME ZONE datatype has the form:

```
TIMESTAMP(fractional_seconds_precision) WITH LOCAL TIME ZONE
```

where `fractional_seconds_precision` optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. The INTERVAL YEAR TO MONTH datatype has the form:

```
INTERVAL YEAR(year_precision) TO MONTH
```

where the optional `year_precision` is the number of digits in the YEAR datetime field. The default value of `year_precision` is 2.

INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds. The INTERVAL DAY TO SECOND datatype has the form:

```
INTERVAL DAY (day_precision) TO SECOND(fractional_seconds_precision)
```

where:

- `day_precision` is the number of digits in the DAY datetime field. It is optional. Accepted values are 0 to 9. The default is 2.

`fractional_seconds_precision` is the number of digits in the fractional part of the SECOND datetime field. It is optional. Accepted values are 0 to 9. The default is 6.

Avoiding Unexpected Results Using Datetime

> **Note:** To avoid unexpected results in your DML operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and SESSIONTIMEZONE. If the time zones have not been set manually, Oracle uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, Oracle uses UTC as the default value.

C Object-Relational Datatype Mappings

The OCI supports Oracle-defined C datatypes used to map user-defined datatypes to C representations (e.g. **OCINumber**, **OCIArray**). The OCI provides a set of calls to operate on these datatypes, and to use these datatypes in bind and define operations, in conjunction with OCI external datatype codes.

> **See Also:** For information on using these Oracle-defined C datatypes, refer to Chapter 11, "Object-Relational Datatypes"

Data Conversions

Table 3–5 and Table 3–6 show the supported conversions from internal datatypes to external datatypes, and from external datatypes into internal column representations, for all datatypes available through release 7.3. Information about data conversions for data types newer than release 7.3 is listed here:

- REFs stored in the database are converted to SQLT_REF on output.

- SQLT_REF is converted to the internal representation of REFs on input.

- Named Data Types stored in the database can be converted to SQLT_NTY (and represented by a C struct in the application) on output.

- SQLT_NTY (represented by a C struct in an application) is converted to the internal representation of the corresponding type on input.

LOBs are shown in a separate table that follows, because of the width limitation.

> **See Also:** For information about **OCIString**, **OCINumber**, and other new datatypes, refer to Chapter 11, "Object-Relational Datatypes"

Table 3–5   Data Conversions

EXTERNAL DATATYPES	INTERNAL DATATYPES								
	VARCHAR2	NUMBER	LONG	ROWID	UROWID	DATE	RAW	LONG RAW	CHAR
VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3)	
NUMBER	I/O(4)	I/O	I						I/O(4)
INTEGER	I/O(4)	I/O	I						I/O(4)
FLOAT	I/O(4)	I/O	I						I/O(4)

Table 3–5   Data Conversions (Cont.)

EXTERNAL DATATYPES	INTERNAL DATATYPES								
	VARCHAR2	NUMBER	LONG	ROWID	UROWID	DATE	RAW	LONG RAW	CHAR
STRING	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3, 5)	I/O
VARNUM	I/O(4)	I/O	I						I/O(4)
DECIMAL	I/O(4)	I/O	I						I/O(4)
LONG	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3, 5)	I/O
VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3, 5)	I/O
DATE	I/O		I			I/O			I/O
VARRAW	I/O(6)		I(5, 6)				I/O	I/O	I/O(6)
RAW	I/O(6)		I(5, 6)				I/O	I/O	I/O(6)
LONG RAW	O(6)		I(5, 6)				I/O	I/O	O(6)
UNSIGNED	I/O(4)	I/O	I						I/O(4)
LONG VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3, 5)	I/O
LONG VARRAW	I/O(6)		I(5, 6)				I/O	I/O	I/O(6)
CHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I(3)	I/O

Table 3–5   Data Conversions (Cont.)

EXTERNAL DATATYPES	INTERNAL DATATYPES								
	VARCHAR2	NUMBER	LONG	ROWID	UROWID	DATE	RAW	LONG RAW	CHAR
CHARZ	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I(3)	I/O
ROWID descriptor	I(1)			I/O	I/O				I(1)

Notes:

(1) For input, host string must be in Oracle ROWID/UROWID format.

   On output, column value is returned in Oracle ROWID/UROWID format.

(2) For input, host string must be in the Oracle DATE character format.

   On output, column value is returned in Oracle DATE format.

(3) For input, host string must be in hex format.

   On output, column value is returned in hex 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.

   On output, column value must be in hex format.

Legend:

I = Conversion valid for input only

O = Conversion valid for output only

I/O = Conversion valid for input or output

Data Conversions for LOB Datatype Descriptors

Table 3–6   Data Conversions for LOBs

EXTERNAL DATATYPES	INTERNAL DATATYPES	
	CLOB	BLOB
VARCHAR	I/O	
CHAR	I/O	
LONG	I/O	
LONG VARCHAR	I/O	
RAW		I/O
VARRAW		I/O
LONG RAW		I/O

Table 3–6   Data Conversions for LOBs

	INTERNAL DATATYPES	
EXTERNAL DATATYPES	**CLOB**	**BLOB**
LONG VARRAW		I/O

Data Conversions for Datetime and Interval Datatypes

You can also use one of the character data types for the host variable used in a fetch or insert operation from or to a datetime or interval column. Oracle will do the conversion between the character data type and datetime/interval data type for you.

Table 3–7   Data Conversion for Datetime and Interval Types

External Types	**Internal Types**						
	VARCHAR, CHAR	**DATE**	**TS**	**TSTZ**	**TSLTZ**	**INTERVAL YEAR TO MONTH**	**NTERVAL DAY TO SECOND**
VARCHAR2, CHAR	I/O	I/O	I/O	I/O	I/O	I/O	I/O
DATE	I/O	I/O	I/O	I/O	I/O	-	-
OCI DATE	I/O	I/O	I/O	I/O	I/O	-	-
ANSI DATE	I/O	I/O	I/O	I/O	I/O	-	-
TIMESTAMP (TS)	I/O	I/O	I/O	I/O	I/O	-	-
TIMESTAMP WITH TIME ZONE (TSTZ)	I/O	I/O	I/O	I/O	I/O	-	-
TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)	I/O	I/O	I/O	I/O	I/O		
INTERVAL YEAR TO MONTH	I/O	-	-	-	-	I/O	-
INTERVAL DAY TO SECOND	I/O	-	-	-	-	-	I/O

> **Note:** When assigning a source with time zone to a target without a time zone, the time zone portion of the source is ignored. On assigning a source without a time zone to a target with a time zone, the time zone of the target is set to the session's default time zone

(0) When assigning an Oracle DATE to a TIMESTAMP, the TIME portion of the DATE is copied over to the TIMESTAMP. When assigning a TIMESTAMP to Oracle DATE, the TIME portion of the result DATE is set to zero. This is done to encourage migration of Oracle DATE to ANSI compliant DATETIME data types

(1) When assigning an ANSI DATE to an Oracle DATE or a TIMESTAMP, the TIME portion of the Oracle DATE and the TIMESTAMP are set to zero. When assigning an Oracle DATE or a TIMESTAMP to an ANSI DATE, the TIME portion is ignored

(2) When assigning a DATETIME to a character string, the DATETIME is converted using the session's default DATETIME format. When assigning a character string to a DATETIME, the string must contain a valid DATETIME value based on the session's default DATETIME format

(3) When assigning a character string to an INTERVAL, the character string must be a valid INTERVAL character format.

Data Conversion Notes for Datetime and Interval Types

(1) When converting from TSLTZ to CHAR, DATE, TIMESTAMP and TSTZ, the value will be adjusted to the session time zone.

(2) When converting from CHAR, DATE, and TIMESTAMP to TSLTZ, the session time zone will be stored in memory.

(3) When assigning TSLTZ to ANSI DATE, the time portion will be zero.

(4) When converting from TSTZ, the time zone which the time stamp is in will be stored in memory.

(5) When assigning a character string to an interval, the character string must be a valid interval character format.

Datetime and Date Migration Rules

OCI has full forward and backward compatibility between a client application and the database server as far as the datetime and date columns are concerned.

Pre-9.0 Client with 9.0 or Later Server

The only datetime datatype available to a pre-9.0 application is the DATE datatype, SQLT_DAT. When a pre-9.0 client that defined a buffer as SQLT_DAT, tries to

obtain data from a TSLTZ column, then only the date portion of the value will be returned to the client.

Pre-9.0 Server with 9.0 or Later Client

In this case the new client might have a bind or define buffer of type SQLT_TIMESTAMP_LTZ. The following compatibilities are maintained in this case.

If any client application tries to insert a SQLT_TIMESTAMP_LTZ (or any of the new datetime datatypes) into a DATE column, an error will be issued since there is potential data loss in this situation.

When a client has an OUT bind or a define buffer that is of datatype SQLT_TIMESTAMP_LTZ and the underlying server side SQL buffer or column is of DATE type, then the session time zone is assigned.

Typecodes

There is a unique typecode associated with each Oracle type, whether scalar, collection, reference, or object type. This typecode identifies the type, and is used by Oracle to manage information about object type attributes. This typecode system is designed to be generic and extensible, and is not tied to a direct one-to-one mapping to Oracle datatypes. Consider the following SQL statements:

```
CREATE TYPE my_type AS OBJECT
( attr1    NUMBER,
  attr2    INTEGER,
  attr3    SMALLINT);

CREATE TABLE my_table AS TABLE OF my_type;
```

These statements create an object type and an object table. When it is created, my_table will have three columns, all of which are of Oracle NUMBER type, because SMALLINT and INTEGER map internally to NUMBER. The internal representation of the attributes of my_type, however, maintains the distinction between the datatypes of the three attributes: attr1 is OCI_TYPECODE_NUMBER, attr2 is OCI_TYPECODE_INTEGER, and attr3 is OCI_TYPECODE_SMALLINT. If an application describes my_type, these typecodes are returned.

OCITypeCode is the C datatype of the typecode. The typecode is used by some OCI functions, like OCIObjectNew() (where it helps determine what type of object is created). It is also returned as the value of some attributes when an object is

described; e.g., querying the OCI_ATTR_TYPECODE attribute of a type returns an **OCITypeCode** value.

Table 3–8 lists the possible values for an **OCITypeCode**. There is a value corresponding to each Oracle datatype.

Table 3–8   OCITypeCode Values

Value	Datatype
OCI_TYPECODE_REF	REF
OCI_TYPECODE_DATE	DATE
OCI_TYPECODE_TIMESTAMP	TIMESTAMP
OCI_TYPECODE_TIMESTAMP_TZ	TIMESTAMP WITH TIME ZONE
OCI_TYPECODE_TIMESTAMP_LTZ	TIMESTAMP WITH LOCAL TIME ZONE
OCI_TYPECODE_INTERVAL_YM	INTERVAL YEAR TO MONTH
OCI_TYPECODE_INTERVAL_DS	INTERVAL DAY TO SECOND
OCI_TYPECODE_REAL	single-precision real
OCI_TYPECODE_DOUBLE	double-precision real
OCI_TYPECODE_FLOAT	floating-point
OCI_TYPECODE_NUMBER	Oracle number
OCI_TYPECODE_DECIMAL	decimal
OCI_TYPECODE_OCTET	octet
OCI_TYPECODE_INTEGER	integer
OCI_TYPECODE_SMALLINT	smallint
OCI_TYPECODE_RAW	RAW
OCI_TYPECODE_VARCHAR2	variable string ANSI SQL, i.e., VARCHAR2
OCI_TYPECODE_VARCHAR	variable string Oracle SQL, i.e., VARCHAR
OCI_TYPECODE_CHAR	fixed-length string inside SQL, i.e. SQL CHAR
OCI_TYPECODE_VARRAY	variable-length array (varray)
OCI_TYPECODE_TABLE	multiset
OCI_TYPECODE_CLOB	character large object (CLOB)
OCI_TYPECODE_BLOB	binary large object (BLOB)

Table 3–8   OCITypeCode Values (Cont.)

Value	Datatype
OCI_TYPECODE_BFILE	binary large object file (BFILE)
OCI_TYPECODE_OBJECT	named object type
OCI_TYPECODE_NAMEDCOLLECTION	Domain (named primitive type)

Relationship Between SQLT and OCI_TYPECODE Values

Oracle recognizes two different sets of datatype code values. One set is distinguished by the SQLT_ prefix, the other by the OCI_TYPECODE_ prefix.

The SQLT typecodes are used by OCI to specify a datatype in a bind or define operation. In this way, the SQL typecodes help to control data conversions between Oracle and OCI client applications. The OCI_TYPECODE types are used by Oracle's type system to reference or describe predefined types when manipulating or creating user-defined types.

In many cases there are direct mappings between SQLT and OCI_TYPECODE values. In other cases, however, there is not a direct one-to-one mapping. For example OCI_TYPECODE_SIGNED16, OCI_TYPECODE_SIGNED32, OCI_TYPECODE_INTEGER, OCI_TYPECODE_OCTET, and OCI_TYPECODE_SMALLINT are all mapped to the SQLT_INT type.

The following table illustrates the mappings between SQLT and OCI_TYPECODE types.

Table 3–9   OCI_TYPECODE to SQLT Mappings

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
BFILE	OCI_TYPECODE_BFILE	SQLT_BFILE
BLOB	OCI_TYPECODE_BLOB	SQLT_BLOB
CHAR	OCI_TYPECODE_CHAR (n)	SQLT_AFC(n) [note 1]
CLOB	OCI_TYPECODE_CLOB	SQLT_CLOB
COLLECTION	OCI_TYPECODE_NAMEDCOLLECTION	SQLT_NCO
DATE	OCI_TYPECODE_DATE	SQLT_DAT
TIMESTAMP	OCI_TYPECODE_TIMESTAMP	SQLT_TIMESTAMP
TIMESTAMP WITH TIME ZONE	OCI_TYPECODE_TIMESTAMP_TZ	SQLT_TIMESTAMP_TZ

Table 3–9 OCI_TYPECODE to SQLT Mappings (Cont.)

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
TIMESTAMP WITH LOCAL TIME ZONE	OCI_TYPECODE_TIMESTAMP_LTZ	SQLT_TIMESTAMP_LTZ
INTERVAL YEAR TO MONTH	OCI_TYPECODE_INTERVAL_YM	SQLT_INTERVAL_YM
INTERVAL DAY TO SECOND	OCI_TYPECODE_INTERVAL_DS	SQLT_INTERVAL_DS
FLOAT	OCI_TYPECODE_FLOAT (b)	SQLT_FLT (8) [note 2]
DECIMAL	OCI_TYPECODE_DECIMAL (p)	SQLT_NUM (p, 0) [note 3]
DOUBLE	OCI_TYPECODE_DOUBLE	SQLT_FLT (8)
INTEGER	OCI_TYPECODE_INTEGER	SQLT_INT (i) [note 4]
NUMBER	OCI_TYPECODE_NUMBER (p, s)	SQLT_NUM (p, s) [note 5]
OCTET	OCI_TYPECODE_OCTET	SQLT_INT (1)
POINTER	OCI_TYPECODE_PTR	<NONE>
RAW	OCI_TYPECODE_RAW	SQLT_LVB
REAL	OCI_TYPECODE_REAL	SQLT_FLT (4)
REF	OCI_TYPECODE_REF	SQLT_REF
OBJECT	OCI_TYPECODE_OBJECT	SQLT_NTY
SIGNED(8)	OCI_TYPECODE_SIGNED8	SQLT_INT (1)
SIGNED(16)	OCI_TYPECODE_SIGNED16	SQLT_INT (2)
SIGNED(32)	OCI_TYPECODE_SIGNED32	SQLT_INT (4)
SMALLINT	OCI_TYPECODE_SMALLINT	SQLT_INT (i) [note 4]
TABLE [note 6]	OCI_TYPECODE_TABLE	<NONE>
TABLE (Indexed table)	OCI_TYPECODE_ITABLE	SQLT_TAB
UNSIGNED(8)	OCI_TYPECODE_UNSIGNED8	SQLT_UIN (1)
UNSIGNED(16)	OCI_TYPECODE_UNSIGNED16	SQLT_UIN (2)
UNSIGNED(32)	OCI_TYPECODE_UNSIGNED32	SQLT_UIN (4)
VARRAY [note 6]	OCI_TYPECODE_VARRAY	<NONE>

Table 3–9   OCI_TYPECODE to SQLT Mappings (Cont.)

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
VARCHAR	OCI_TYPECODE_VARCHAR (n)	SQLT_CHR (n) [note 1]
VARCHAR2	OCI_TYPECODE_VARCHAR2 (n)	SQLT_VCS (n) [note 1]

Notes:

1. n is the size of the string in bytes

2. These are floating point numbers, the precision is given in terms of binary digits. b is the precision of the number in binary digits.

3. This is equivalent to a NUMBER with no decimal places.

4. i is the size of the number in bytes, set as part of an OCI call.

5. p is the precision of the number in decimal digits; s is the scale of the number in decimal digits.

6. Can only be part of a named collection type.

Definitions in oratypes.h

Throughout this guide you will see references to datatypes like **ub2** or **sb4**, or to constants like UB4MAXVAL. These types are defined in the oratypes.h header file, an example of which is included here. The exact contents may vary according to the platform you are using.

> **Note:**   The use of the datatypes in oratypes.h is the only supported means of supplying parameters to the OCI.

```
#ifndef ORATYPES
# define ORATYPES
# define SX_ORACLE
# define SX3_ORACLE


#ifndef ORASTDDEF
# include <stddef.h>
# define ORASTDDEF
#endif

#ifndef ORALIMITS
# include <limits.h>
```

```
# define ORALIMITS
#endif


#ifndef TRUE
# define TRUE  1
# define FALSE 0
#endif


#ifdef lint
# ifndef mips
#  define signed
# endif
#endif

#ifdef ENCORE_88K
# ifndef signed
#  define signed
# endif
#endif

#if defined(SYSV_386) || defined(SUN_OS)
# ifdef signed
#  undef signed
# endif
# define signed
#endif




#ifndef lint
typedef unsigned char  ub1;
typedef   signed char  sb1;
#else
#define ub1 unsigned char
#define sb1 signed char
#endif

#define UB1MAXVAL ((ub1)UCHAR_MAX)
#define UB1MINVAL ((ub1)        0)
#define SB1MAXVAL ((sb1)SCHAR_MAX)
#define SB1MINVAL ((sb1)SCHAR_MIN)
```

```
#define MINUB1MAXVAL ((ub1)  255)
#define MAXUB1MINVAL ((ub1)    0)
#define MINSB1MAXVAL ((sb1)  127)
#define MAXSB1MINVAL ((sb1) -127)




#ifndef lint
typedef unsigned short    ub2;
typedef   signed short    sb2;
#else
#define ub2  unsigned short
#define sb2  signed short
#endif

#define UB2MAXVAL ((ub2)USHRT_MAX)
#define UB2MINVAL ((ub2)        0)
#define SB2MAXVAL ((sb2) SHRT_MAX)
#define SB2MINVAL ((sb2) SHRT_MIN)
#define MINUB2MAXVAL ((ub2) 65535)
#define MAXUB2MINVAL ((ub2)     0)
#define MINSB2MAXVAL ((sb2) 32767)
#define MAXSB2MINVAL ((sb2)-32767)




#ifndef lint
typedef unsigned int  ub4;
typedef   signed int  sb4;
#else
#define eb4 int
#define ub4 unsigned int
#define sb4 signed int
#endif

#define UB4MAXVAL ((ub4)UINT_MAX)
#define UB4MINVAL ((ub4)        0)
#define SB4MAXVAL ((sb4) INT_MAX)
#define SB4MINVAL ((sb4) INT_MIN)
#define MINUB4MAXVAL ((ub4) 4294967295)
#define MAXUB4MINVAL ((ub4)          0)
#define MINSB4MAXVAL ((sb4) 2147483647)
#define MAXSB4MINVAL ((sb4)-2147483647)
```

```
#define UB1BITS          CHAR_BIT
#define UB1MASK          ((1 << ((uword)CHAR_BIT)) - 1)



typedef ub1   bitvec;
#define BITVEC(n) (((n)+(UB1BITS-1))>>3)



#ifdef lint
# define oratext unsigned char
#else
  typedef unsigned char oratext;
#endif


#ifndef lint
typedef        ub4      duword;
typedef        sb4      dsword;
typedef        dsword   dword;

#else
#define duword ub4
#define dsword sb4
#define dword   dsword
#endif

#define  DUWORDMAXVAL       UB4MAXVAL
#define  DUWORDMINVAL       UB4MINVAL
#define  DSWORDMAXVAL       SB4MAXVAL
#define  DSWORDMINVAL       SB4MINVAL
#define  MINDUWORDMAXVAL    MINUB4MAXVAL
#define  MAXDUWORDMINVAL    MAXUB4MINVAL
#define  MINDSWORDMAXVAL    MINSB4MAXVAL
#define  MAXDSWORDMINVAL    MAXSB4MINVAL
#define  DEWORDMAXVAL       EB4MAXVAL
#define  DEWORDMINVAL       EB4MINVAL
#define  MINDEWORDMAXVAL    MINEB4MAXVAL
#define  MAXDEWORDMINVAL    MAXEB4MINVAL
#define  DWORDMAXVAL        DSWORDMAXVAL
#define  DWORDMINVAL        DSWORDMINVAL
```

```
#ifndef lint
typedef ub4 dsize_t;
# else
# define dsize_t ub4
#endif

# define DSIZE_TMAXVAL UB4MAXVAL
# define MINDSIZE_TMAXVAL (dsize_t)65535


#ifndef lint
typedef sb4 dboolean;
# else
# define dboolean sb4
#endif




#ifndef lint
typedef ub4 dptr_t;
#else
#define dptr_t ub4
#endif




#ifndef lint
typedef         char    eb1;
typedef         short   eb2;
typedef         int     eb4;
typedef         eb4     deword;
#else
# define        eb1     char
# define        eb2     short
# define        eb4     int
# define        deword  eb4
#endif
```

```
#define EB1MAXVAL      ((eb1)SCHAR_MAX)
#define EB1MINVAL      ((eb1)         0)
#define MINEB1MAXVAL   ((eb1)  127)
#define MAXEB1MINVAL   ((eb1)    0)
#define EB2MAXVAL      ((eb2) SHRT_MAX)
#define EB2MINVAL      ((eb2)         0)
#define MINEB2MAXVAL   ((eb2) 32767)
#define MAXEB2MINVAL   ((eb2)    0)
#define EB4MAXVAL      ((eb4) INT_MAX)
#define EB4MINVAL      ((eb4)         0)
#define MINEB4MAXVAL   ((eb4) 2147483647)
#define MAXEB4MINVAL   ((eb4)          0)



#ifndef lint
typedef        sb1   b1;
#else
#define        b1 sb1
#endif
#define  B1MAXVAL  SB1MAXVAL
#define  B1MINVAL  SB1MINVAL

#ifndef lint
typedef        sb2       b2;
#else
#define        b2 sb2
#endif
#define  B2MAXVAL  SB2MAXVAL
#define  B2MINVAL  SB2MINVAL

#ifndef lint
typedef        sb4       b4;
#else
#define        b4 sb4
#endif
# define  B4MAXVAL  SB4MAXVAL
# define  B4MINVAL  SB4MINVAL



#ifndef uiXT
typedef   ub1       BITS8;
typedef   ub2       BITS16;
typedef   ub4       BITS32;
```

```
#endif

#if !defined(LUSEMFC)
# ifdef lint
#  define text unsigned char
#  define OraText oratext
# else
   typedef oratext text;
   typedef oratext OraText;
# endif
#endif

#define  M_IDEN    30

#ifdef AIXRIOS
# define SLMXFNMLEN 256
#else
# define SLMXFNMLEN 512
#endif

#ifndef lint
typedef          int eword;
typedef unsigned int uword;
typedef   signed int sword;
#else
#define eword int
#define uword unsigned int
#define sword signed int
#endif

#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) 2147483647)
#define  MAXEWORDMINVAL  ((eword)       0)
#define  MINUWORDMAXVAL  ((uword)  4294967295)
#define  MAXUWORDMINVAL  ((uword)          0)
#define  MINSWORDMAXVAL  ((sword)  2147483647)
#define  MAXSWORDMINVAL  ((sword) -2147483647)
```

```
#ifndef lint
typedef unsigned long  ubig_ora;
typedef   signed long  sbig_ora;
#else
#define ubig_ora unsigned long
#define sbig_ora signed long
#endif

#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)

#define UBIGORABITS      (UB1BITS * sizeof(ubig_ora))


#ifndef lint
#if (__STDC__ != 1)
# define SLU8NATIVE
# define SLS8NATIVE
#endif
#endif

#ifdef SLU8NATIVE

#ifdef SS_64BIT_SERVER
# ifndef lint
   typedef unsigned long ub8;
# else
#  define ub8 unsigned long
# endif
#else
# ifndef lint
   typedef unsigned long long ub8;
# else
#  define ub8 unsigned long long
# endif
#endif

#define UB8ZERO      ((ub8)0)
```

```
#define UB8MINVAL    ((ub8)0)
#define UB8MAXVAL    ((ub8)18446744073709551615)

#define MAXUB8MINVAL ((ub8)0)
#define MINUB8MAXVAL ((ub8)18446744073709551615)

#endif


#ifdef SLS8NATIVE

#ifdef SS_64BIT_SERVER
# ifndef lint
   typedef signed long sb8;
# else
#  define sb8 signed long
# endif
#else
# ifndef lint
   typedef signed long long sb8;
# else
#  define sb8 signed long long
# endif
#endif

#define SB8ZERO      ((sb8)0)

#define SB8MINVAL    ((sb8)-9223372036854775808)
#define SB8MAXVAL    ((sb8) 9223372036854775807)

#define MAXSB8MINVAL ((sb8)-9223372036854775807)
#define MINSB8MAXVAL ((sb8) 9223372036854775807)

#endif



#undef CONST

#ifdef _olint
# define CONST const
#else
#if defined(PMAX) && defined(__STDC__)
#   define CONST const
#else
```

```
# ifdef M88OPEN
#  define CONST const
# else
#  if defined(SEQ_PSX) && defined(__STDC__)
#    define CONST const
#  else
#    ifdef A_OSF
#      if defined(__STDC__)
#        define CONST const
#      else
#        define CONST
#      endif
#    else
#      define CONST
#    endif
#  endif
# endif
#endif
#endif



#ifdef lint
# define dvoid void
#else

# ifdef UTS2
#  define dvoid char
# else
# define dvoid void
# endif

#endif


typedef void (*lgenfp_t)( void );



#ifndef ORASYS_TYPES
# include <sys/types.h>
# define ORASYS_TYPES
#endif

#ifndef boolean
```

```
#ifndef lint
typedef int boolean;
#else
#define boolean int
#endif
#endif

#ifndef SIZE_TMAXVAL
# define SIZE_TMAXVAL UBIG_ORAMAXVAL
#endif

#ifndef MINSIZE_TMAXVAL
# define MINSIZE_TMAXVAL (size_t)4294967295
#endif


#if !defined(MOTIF) && !defined(LISPL)  && !defined(__cplusplus) && \
 !defined(LUSEMFC)
typedef  OraText *string;
#endif

#ifndef lint
typedef unsigned short  utext;
#else
#define utext  unsigned short
#endif


#endif
```

4

Using SQL Statements in OCI

This chapter discusses the concepts and steps involved in processing SQL statements with the Oracle Call Interface. The following topics are covered in this chapter:

- Overview of SQL Statement Processing
- Processing SQL Statements
- Preparing Statements
- What is Binding?
- Executing Statements
- Describing Select-List Items
- What is Defining?
- Fetching Results
- Scrollable Cursors

Overview of SQL Statement Processing

Chapter 2, "OCI Programming Basics" discussed the basic steps involved in any OCI application. This chapter presents a more detailed look at the specific tasks involved in processing SQL statements in an OCI program.

Processing SQL Statements

One of the most common tasks of an OCI program is to accept and process SQL statements. This section outlines the specific steps involved in processing SQL.

Once you have allocated the necessary handles and attached to a server, the basic steps in processing a SQL statement are the following, as illustrated in Figure 4–1, "Steps In Processing SQL Statements":

1. **Prepare**. Define an application request using `OCIStmtPrepare()`.

2. **Bind**. For DML statements and queries with input variables, perform one or more bind calls using

   - `OCIBindByPos()`,
   - `OCIBindByName()`,
   - `OCIBindObject()`,
   - `OCIBindDynamic()`,
   - or `OCIBindArrayOfStruct()`

   to bind the address of each input variable (or PL/SQL output variable) or array to each placeholder in the statement.

3. **Execute**. Call `OCIStmtExecute()` to execute the statement. For DDL statements, no further steps are necessary.

4. **Describe**. Describe the select-list items, if necessary, using `OCIParamGet()` and `OCIAttrGet()`. 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.

5. **Define**. For queries, perform one or more define calls to `OCIDefineByPos()`, `OCIDefineObject()`, `OCIDefineDynamic()`, or `OCIDefineArrayOfStruct()` to define an output variable for each select-list item in the SQL statement. Note that you do not use a define call to define the output variables in an anonymous PL/SQL block. You have done this when you have bound the data.

6. Fetch. For queries, call `OCIStmtFetch()` to fetch the results of the query.

Following these steps, the application can free allocated handles and then detach from the server, or it may process additional statements.

> **7.x Upgrade Note:** OCI programs no longer require an explicit parse step. If a statement must be parsed, that step takes place upon execution. This means that 8.0 or later applications must issue an execute command for both DML and DDL statements.

Figure 4–1   Steps In Processing SQL Statements


```
Prepare              OCIStmtPrepare()
Statement

Bind                 OCIBindByName() or OCIBindByPos()
Placeholders*        OCIBindObject()
                     OCIBindArrayOfStruct()
                     OCIBindDynamic()

Execute              OCIStmtExecute()
Statement

Describe             OCIParamGet()
Select-list Items*   OCIAttrGet()

Define               OCIDefineByPos()
Output Variables*    OCIDefineObject()
                     OCIDefineArrayOfStruct()
                     OCIDefineDynamic()

Fetch and            OCIStmtFetch()
Process Data*
```
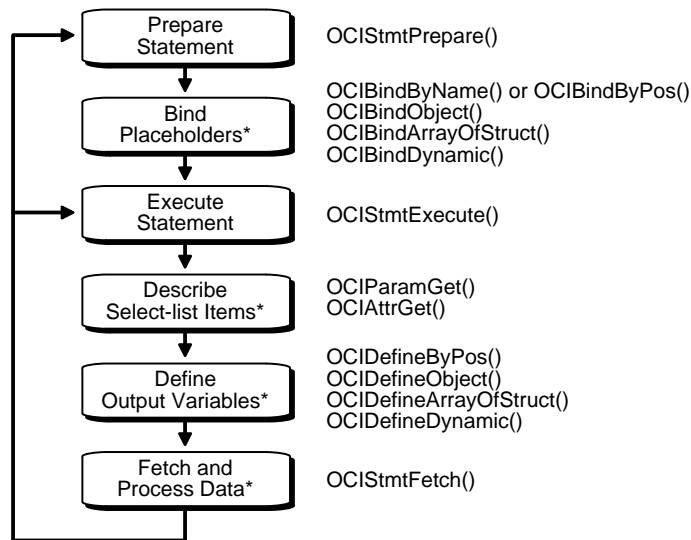
* These steps performed
  if necessary

For each of the steps in the diagram, the corresponding OCI function calls are listed. In some cases multiple calls may be required.

Each step above is described in detail in the following sections.

> **Note:** Some variation in the order of steps is possible. For example, it is possible to do the define step before the execute if the datatypes and lengths of returned values are known at compile time. Also, as indicated by the asterisks (*), some steps may not be required by your application.

Additional steps beyond those listed above may be required if your application needs to do the following:

- initiate and manage multiple transactions

- manage multiple threads of execution

- perform piecewise inserts, updates, or fetches

    See Also:

    - These topics are described in Chapter 9, "OCI Programming Advanced Topics".

    - For information on using OCI shared mode functionality, refer to "Shared Data Mode" on page 2-22.

Preparing Statements

SQL and PL/SQL statements need to be prepared for execution by using the statement prepare call and bind calls (if necessary). In this phase, the application specifies a SQL or PL/SQL statement and binds associated placeholders in the statement to data for execution. The client-side library allocates storage to maintain the statement prepared for execution.

An application requests a SQL or PL/SQL statement to be prepared for execution using the `OCIStmtPrepare()` call and passing it a previously allocated statement handle. This is a completely local call, requiring no round trip to the server. No association is made at this point between the statement and a particular server.

Following the request call, an application can call `OCIAttrGet()` on the statement handle, passing OCI_ATTR_STMT_TYPE to the *attrtype* parameter, to determine what type of SQL statement was prepared. The possible attribute values, and corresponding statement types are listed in Table 4–1.

Table 4–1   OCI_ATTR_STMT_TYPE Values and Statement Types

Attribute Value	Statement Type
OCI_STMT_SELECT	SELECT statement
OCI_STMT_UPDATE	UPDATE statement
OCI_STMT_DELETE	DELETE statement
OCI_STMT_INSERT	INSERT statement
OCI_STMT_CREATE	CREATE statement
OCI_STMT_DROP	DROP statement
OCI_STMT_ALTER	ALTER statement
OCI_STMT_BEGIN	BEGIN... (PL/SQL)
OCI_STMT_DECLARE	DECLARE... (PL/SQL)

See Also:

- For more information on the specifics of using PL/SQL in an OCI application, see the section "Using PL/SQL in an OCI Program" on page 2-44

- See the OCIStmtPrepare() call

Using Prepared Statements on Multiple Servers

A prepared application request can be executed on multiple servers at run time by reassociating the statement handle with the respective service context handles for the servers. All information cached about the current service context and statement handle association is lost when a new association is made.

For example, consider an application such as a network manager, which manages multiple servers. In many cases, it is likely that the same SELECT statement will need to be executed against multiple servers to retrieve information for display. The OCI allows the network manager application to prepare a SELECT statement once and execute it against multiple servers. It must fetch all of the required rows from each server prior to reassociating the prepared statement with the next server.

> **Note:** If a prepared statement must be reexecuted frequently on the same server, it is efficient to prepare a new statement for another service context.

What is Binding?

Most DML statements, and some queries (such as those with a WHERE clause), require a program to pass data 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 following SQL statement, which adds an employee to a database contains several literals, such as 'BESTRY' and 2365:

```
INSERT INTO emp VALUES
    (2365, 'BESTRY', 'PROGRAMMER', 2000, 20)
```

Coding a statement like this into an application would severely limit its usefulness. You would need to change the statement and recompile the program each time you add a new employee to the database. To make the program more flexible, you can write the program so that a user can supply input data at run time.

When you prepare a SQL statement or PL/SQL block that contains input data to be supplied at run time, placeholders in the SQL statement or PL/SQL block mark where data must be supplied. For example, the following SQL statement contains five placeholders, indicated by the leading colons (:ename), that show where input data must be supplied by the program.

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

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. In PL/SQL, placeholders can also be used for output variables.

Placeholders cannot be used to represent other Oracle objects such as tables. For example, the following is *not* a valid use of the emp placeholder:

```
INSERT INTO :emp VALUES
    (12345, 'OERTEL', 'WRITER', 50000, 30)
```

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. When the statement executes, Oracle gets the data that your program placed in the input, or bind, variables and passes it to the server with the SQL statement.

> **See Also:** For detailed information about implementing bind operations, refer to Chapter 5, "Binding and Defining"

Executing Statements

An OCI application executes prepared statements individually using
`OCIStmtExecute()`.

> **See Also:** `OCIStmtExecute()` for a syntax description

When an OCI application executes a query, it receives data from Oracle that
matches the query specifications. Within the database, the data is stored in
Oracle-defined formats. When the results are returned, an OCI application can
request that data be converted to a particular host language format, and stored in a
particular output variable or buffer.

For each item in the select-list of a query, the OCI application must define an output
variable to receive the results of the query. The define step indicates the address of
the buffer and the type of the data to be retrieved.

> **Note:** If output variables are defined for a SELECT statement
> before a call to `OCIStmtExecute()`, the number of rows specified
> by the *iters* parameter are fetched directly into the defined
> output buffers and additional rows equivalent to the prefetch count
> are prefetched. If there are no additional rows, then the fetch is
> complete without calling `OCIStmtFetch()`.

For non-queries, the number of times the statement is executed during array
operations is equal to *iters - rowoff*, where *rowoff* is the offset in the bound
array, and is also a parameter of the `OCIStmtExecute()` call. For example, if an
array of 10 items is bound to a placeholder for an INSERT statement, and *iters* is
set to 10, all 10 items will be inserted in a single execute call when *rowoff* is zero. If
rowoff is set to 2, only 8 items will be inserted.

> **See Also:** "What is Defining?" on page 4-15 for more information
> about defining output variables

Execution Snapshots

The `OCIStmtExecute()` call provides the ability to ensure that multiple service
contexts operate on the same consistent snapshot of the database's committed data.
This is achieved by taking the contents of the *snap_out* parameter of one
`OCIStmtExecute()` call and passing that value in the *snap_in* parameter of the
next `OCIStmtExecute()` call.

> **Note:** Uncommitted data in one service context is *not* visible to another context, even when using the same snapshot

The datatype of both the *snap_out* and *snap_in* parameter is **OCISnapshot**, an OCI snapshot descriptor. This descriptor is allocated with the OCIDescAlloc() function.

> **See Also:** For more information about descriptors, see the section "Descriptors" on page 2-15

It is not necessary to specify a *snapshot* when calling OCIStmtExecute(). The following sample code shows a basic execution in which the *snapshot* parameters are passed as NULL.

```
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (OCISnapshot *)NULL, (OCISnapshot *) NULL, OCI_DEFAULT))
```

> **Note:** The checkerr() function evaluates the return code from an OCI application. The code for the function is listed in the section "Error Handling" on page 2-31.

Execution Modes

You can specify several modes for the OCIStmtExecute() call:

- OCI_DEFAULT. Calling OCIStmtExecute() in this mode executes the statement. It also implicitly returns describe information about the select-list.

- OCI_DESCRIBE_ONLY. This mode is for users who wish to describe a query prior to execution. Calling OCIStmtExecute() in this mode does not execute the statement, but it does return the select-list description.

- OCI_COMMIT_ON_SUCCESS - When a statement is executed in this mode, the current transaction is committed after execution, if execution completes successfully.

- OCI_EXACT_FETCH - Used when the application knows in advance exactly how many rows it will be fetching.

- OCI_BATCH_ERRORS - See "Batch Error Mode for OCIStmtExecute()" on page 4-9, for information about this mode.

Batch Error Mode for **OCIStmtExecute()**

The OCI provides the ability to perform array DML operations. For example, an application can process an array of INSERT, UPDATE, or DELETE statements with a single statement execution. If one of the operations fails due to an error from the server, such as a unique constraint violation, the array operation aborts and the OCI returns an error. Any rows remaining in the array are ignored. The application must then reexecute the remainder of the array, and go through the whole process again if it encounters more errors, which makes additional roundtrips.

To facilitate processing of array DML operations, the OCI provides the *batch error mode* (also called the *enhanced DML array* feature). This mode, which is specified in the OCIStmtExecute() call, simplifies DML array processing in the event of one or more errors. In this mode, the OCI attempts to INSERT, UPDATE, or DELETE all rows, and collects (*batches*) information about any errors which occurred. The application can then retrieve this error information and reexecute any DML operations which failed during the first call.

> **Note:** This function is only available to applications linked with the 8.1 or later OCI libraries running against a release 8.1 or later server. Applications must also be recoded to account for the new program logic described in this section.

In this way, all DML operations in the array are attempted in the first call, and any failed operations can be reissued in a second call.

This mode is used as follows:

1. The user specifies OCI_BATCH_ERRORS as the *mode* parameter of the OCIStmtExecute() call.

2. After performing an array DML operation with OCIStmtExecute(), the application can retrieve the number of errors encountered during the operation by calling OCIAttrGet() on the statement handle to retrieve the OCI_ATTR_NUM_DML_ERRORS attribute. For example:

```
ub4   num_errs;
OCIAttrGet(stmtp, OCI_HTYPE_STMT, &num_err, 0, OCI_ATTR_NUM_DML_ERRORS, errhp);
```

3. The list of errors hangs off an error handle.

   The application extracts each error, along with its row information, from the error handle which was passed to the OCIStmtExecute() call using OCIParamGet(). In order to retrieve the information, the application must

allocate an additional new error handle for the `OCIParamGet()` call. This new error handle is populated with the batched error information. The application obtains the syntax of each error with `OCIErrorGet()`, and the row offset (into the DML array) at which the error occurred by calling `OCIAttrGet()` on the new error handle.

For example, once the `num_errs` amount has been retrieved, the application can issue the following calls:

```
...
OCIError errhndl;
for (i=0; i<num_errs; i++)
{
    OCIParamGet(errhp, OCI_HTYPE_ERROR, errhp, &errhndl, i);
    OCIAttrGet(errhndl, OCI_HTYPE_ERROR, &row_offset, 0,
        OCI_ATTR_DML_ROW_OFFSET, errhp);
    OCIErrorGet(..., errhndl, ...);
...
```

Following this, the application can correct the bind information for the appropriate entry in the array using the diagnostic information retrieved from the batched error. Once the appropriate bind buffers are corrected or updated, the application can reexecute the associated DML statements.

Because the application cannot know at compile time which rows in the first execution will cause errors, the binds for the subsequent DML should be done dynamically by passing in the appropriate buffers at runtime. The user can reuse the bind buffers used in the array binds done on the first DML operation.

Example of Batch Error Mode

The following code shows an example of how this execution mode might be used. In this example assume that we have an application which inserts five rows (with two columns, of types `NUMBER` and `CHAR`) into a table. Furthermore, let us assume only two rows (say, 1 and 3) are successfully inserted in the initial DML operation. The user then proceeds to correct the data (wrong data was being inserted the first time) and to issue an update with the corrected data. The user uses statement handles `stmtp1` and `stmtp2` to issue the `INSERT` and `UPDATE` respectively.

```
...
OCIBind *bindp1[2], *bindp2[2];
ub4 num_errs, row_off[MAXROWS], number[MAXROWS] = {1,2,3,4,5};
char grade[MAXROWS] = {'A','B','C','D','E'};
...
/* Array bind all the positions */
```

```
OCIBindByPos (stmtp1,&bindp1[0],errhp,1,(dvoid *)&number[0],
     sizeof(number[0]),SQLT_NUM,(dvoid *)0, (ub2 *)0,(ub2 *)0,
          0,(ub4 *)0,OCI_DEFAULT);
OCIBindByPos (stmtp1,&bindp1[1],errhp,2,(dvoid *)&grade[0],
     sizeof(grade[0],SQLT_CHR,(dvoid *)0, (ub2 *)0,(ub2 *)0,0,
          (ub4 *)0,OCI_DEFAULT);
/* execute the array INSERT */
OCIStmtExecute (svchp,stmtp1,errhp,5,0,0,0,OCI_BATCH_ERRORS);
/* get the number of errors */
OCIAttrGet (stmtp1, OCI_HTYPE_STMT, &num_errs, 0,
          OCI_ATTR_NUM_DML_ERRORS, errhp);
if (num_errs) {
   /* The user can do one of two things: 1) Allocate as many  */
   /* error handles as number of errors and free all handles  */
   /* at a later time; or 2) Allocate one err handle and reuse */
   /* the same handle for all the errors */
   OCIError *errhndl[num_errs];
   for (i = 0; i < num_errs; i++) {
      OCIParamGet(errhp, OCI_HTYPE_ERROR, errhp, &errhndl[i], i);
      OCIAttrGet (errhndl[i], OCI_HTYPE_ERROR, &row_off[i], 0,
                OCI_ATTR_DML_ROW_OFFSET, errhp);
      /* get server diagnostics */
      OCIErrorGet (..., errhndl[i], ...);
      }
   }
/* make corrections to bind data */
OCIBindByPos (stmtp2,&bindp2[0],errhp,1,(dvoid *)0,0,SQLT_NUM,
     (dvoid *)0, (ub2 *)0,(ub2 *)0,0,(ub4 *)0,OCI_DATA_AT_EXEC);
OCIBindByPos (stmtp2,&bindp2[1],errhp,2,(dvoid *)0,0,SQLT_DAT,
     (dvoid *)0, (ub2 *)0,(ub2 *)0,0,(ub4 *)0,OCI_DATA_AT_EXEC);
/* register the callback for each bind handle */
OCIBindDynamic (bindp2[0],errhp,row_off,my_callback,0,0);
OCIBindDynamic (bindp2[1],errhp,row_off,my_callback,0,0);
/* execute the UPDATE statement */
OCIStmtExecute (svchp,stmtp2,errhp,2,0,0,0,OCI_BATCH_ERRORS);
...
```

In this example, `OCIBindDynamic()` is used with a callback because the user does
not know at compile time what rows will return with errors. With a callback, you
can simply pass the erroneous row numbers, stored in *row_off*, through the
callback context and send only those rows that need to be updated or corrected. The
same bind buffers can be shared between the INSERT and the UPDATE executes.

Describing Select-List Items

If your OCI application is processing a query, you may need to obtain more information about the items in the select-list. This is particularly true for dynamic queries whose contents are not known until run time. In this case, the program may need to obtain information about the datatypes and column lengths of the select-list items. This information is necessary to define output variables that will receive query results.

For example, a user might enter a query such as

```
SELECT * FROM employees
```

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

There are two types of describes available: implicit and explicit. An *implicit describe* is one which does not require any special calls to retrieve describe information from the server although special calls *are* necessary to access the information. An *explicit describe* is one which requires the application to call a particular function to bring the describe information from the server.

An application may describe a select-list (query) either implicitly or explicitly. Other schema elements must be described explicitly.

An implicit describe allows an application to obtain select-list information as an attribute of the statement handle *after a statement has been executed* without making a specific describe call. It is called *implicit*, because no describe call is required. The describe information comes *free* with the execute.

You can describe a query explicitly prior to execution. To do this, specify OCI_DESCRIBE_ONLY as the mode of OCIStmtExecute(). Calling OCIStmtExecute() in this mode does not execute the statement, but it does return the select-list description. For performance reasons, however, it is recommended that applications take advantage of the implicit describe that comes *free* with a standard statement execution.

An explicit describe with the OCIDescribeAny() call obtains information about schema objects rather than select-lists.

In all cases, the specific information about columns and datatypes is retrieved by reading handle attributes.

> **See Also:** For information about using OCIDescribeAny() to obtain metadata pertaining to schema objects, refer to Chapter 6, "Describing Schema Metadata"

Implicit Describe

After a SQL statement is executed, information about the select-list is available as an attribute of the statement handle. No explicit describe call is needed.

To retrieve information about select-list items from the statement handle, the application must call `OCIParamGet()` once for each position in the select-list to allocate a parameter descriptor for that position. Select-list positions are 1-based, meaning that the first item in the select-list is considered to be position number 1.

To retrieve information about multiple select-list items, an application can call `OCIParamGet()` with the *pos* parameter set to 1 the first time, and then iterate the value of *pos* and repeat the `OCIParamGet()` call until OCI_ERROR with ORA-24334 is returned. An application could also specify any position *n* to get a column at random.

Once a parameter descriptor has been allocated for a position in the select-list, the application can retrieve specific information by calling `OCIAttrGet()` on the parameter descriptor. Information available from the parameter descriptor includes the datatype and maximum size of the parameter.

The following sample code shows a loop that retrieves the column names and data types corresponding to a query following query execution. The query was associated with the statement handle by a prior call to `OCIStmtPrepare()`.

```
OCIParam    *mypard;
ub4         counter;
ub2         dtype;
text        *col_name;
ub4         col_name_len;
sb4         parm_status;

...

/* Request a parameter descriptor for position 1 in the select-list */
counter = 1;
parm_status = OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
                (ub4) counter);

/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */
while (parm_status==OCI_SUCCESS) {

/* Retrieve the data type attribute */
checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                (dvoid*) &dtype,(ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
```

```
                            (OCIError *) errhp  ));

/* Retrieve the column name attribute */
checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
        (dvoid**) &col_name,(ub4 *) &col_name_len, (ub4) OCI_ATTR_NAME,
        (OCIError *) errhp ));

printf("column=%s  datatype=%d\n\n", col_name, dtype);
fflush(stdout);

/* increment counter and get next descriptor, if there is one */
counter++;
parm_status = OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
              (ub4) counter);
}
```

> **Note:**   Error handling for the initial `OCIParamGet()` call is not
> included in this example. Ellipses (...) indicate portions of code that
> have been omitted for this example.

The `checkerr()` function is used for error handling. The complete listing can
be found in the first sample application in Appendix B, "OCI Demonstration
Programs".

The calls to `OCIAttrGet()` and `OCIParamGet()` are local calls that do not require
a network round trip, because all of the select-list information is cached on the
client side after the statement is executed.

> **See Also:**
>
> ■   See the descriptions of `OCIParamGet()` and `OCIAttrGet()`
>   for more information about these calls.
>
> ■   See the section "Parameter Attributes" on page 6-5 for a list of
>   the specific attributes of the parameter descriptor which may be
>   read by `OCIAttrGet()`.

Explicit Describe of Queries

You can describe a query explicitly prior to execution. To do this, specify
OCI_DESCRIBE_ONLY as the mode of `OCIStmtExecute()`. Calling

`OCIStmtExecute()` in this mode does not execute the statement, but it does return the select-list description.

> **Note:** To maximize performance, it is recommended that applications execute the statement in default mode and use the implicit describe which accompanies the execution.

The following short example demonstrates the use of this mechanism to perform an explicit describe of a select-list to return information about the columns in the select-list. This pseudo-code shows how to retrieve column information (for example, data type).

```
/* initialize svchp, stmhp, errhp, rowoff, iters, snap_in, snap_out */
/* set the execution mode to OCI_DESCRIBE_ONLY. Note that setting the mode to
OCI_DEFAULT does an implicit describe of the statement in addition to executing
the statement */

OCIParam *colhd;    /* column handle */
checkerr(errhp, OCIStmtExecute(svchp, stmhp, errhp, iters, rowoff,
        snap_in, snap_out, OCI_DESCRIBE_ONLY);

/* Get the number of columns in the query */
checkerr(errhp, OCIAttrGet(stmhp, OCI_HTYPE_STMT, &numcols,
                      0, OCI_ATTR_PARAM_COUNT, errh));

/* go through the column list and retrieve the data type of each column. We
start from pos = 1 */
for (i = 1; i <= numcols; i++)
{
  /* get parameter for column i */
  checkerr(errhp, OCIParamGet(stmhp, OCI_HTYPE_STMT, errh, &colhd, i));

  /* get data-type of column i */
  checkerr(errhp, OCIAttrGet(colhd, OCI_DTYPE_PARAM,
        &type[i-1], 0, OCI_ATTR_DATA_TYPE, errh));
}
```

What is Defining?

Query statements return data from the database to your application. When processing a query, you must define an output variable or an array of output variables for each item in the select-list from which you want to retrieve data. The

define step creates an association which determines where returned results are stored, and in what format.

For example, if your OCI statement processes the following statement:

```
SELECT name, ssn FROM employees
    WHERE empno = :empnum
```

you would normally need to define two output variables, one to receive the value returned from the name column, and one to receive the value returned from the ssn column.

> **See Also:** For information about implementing define operations, please refer to Chapter 5, "Binding and Defining"

Fetching Results

If an OCI application has processed a query, it is typically necessary to fetch the results with OCIStmtFetch() or with OCIStmtFetch2() after the statement has been executed. Oracle encourages the use of OCIStmtFetch2(), which supports **scrollable cursors**, and will be enhanced.

> **See Also:** For information about scrollable cursors, see "Scrollable Cursors" on page 4-18

Fetched data is retrieved into output variables that have been specified by define operations.

Note: If output variables are defined for a SELECT statement before a call to OCIStmtExecute(), the number of rows specified by the *iters* parameter is fetched directly into the defined output buffers

---

See Also:

- These statements fetch data associated with the sample code in the section "Steps Used in Defining" on page 5-20. Refer to that example for more information.

- For information about defining output variables, see the section "Defining" on page 5-19.

Fetching LOB Data

If LOB columns or attributes are part of a select-list, they can be returned as LOB locators or actual LOB values, depending on how the user has defined them. If LOB locators are fetched, then the application can perform further operations on these locators via the `OCILob*` interfaces.

> **See Also:** See Chapter 7, "LOB and FILE Operations", for more information about working with LOB locators in the OCI

Setting Prefetch Count

In order to minimize server round trips and maximize the performance of applications, the OCI can prefetch result set rows when executing a query. The OCI programmer can customize this prefetching by setting the OCI_ATTR_PREFETCH_ROWS or OCI_ATTR_PREFETCH_MEMORY attribute of the statement handle using the `OCIAttrSet()` function. The attributes are used as follows:

- OCI_ATTR_PREFETCH_ROWS sets the number of rows to be prefetched.

- OCI_ATTR_PREFETCH_MEMORY sets the memory allocated for rows to be prefetched. The application then fetches as many rows as will fit into that much memory.

When both of these attributes are set, the OCI prefetches rows up to the OCI_ATTR_PREFETCH_ROWS limit unless the OCI_ATTR_PREFETCH_MEMORY limit is reached, in which case the OCI returns as many rows as will fit in a buffer of size OCI_ATTR_PREFETCH_MEMORY.

By default, prefetching is turned on, and the OCI fetches an extra row all the time. To turn prefetching off, set both the OCI_ATTR_PREFETCH_ROWS and OCI_ATTR_PREFETCH_MEMORY attributes to zero.

> **Note:** Prefetching is not in effect if `LONG` columns are part of the query. Queries containing LOB columns *can* be prefetched, because the LOB locator, rather than the data, is returned by the query.

> **See Also:** For more information about these handle attributes, see the section "Statement Handle Attributes" on page A-24.

Scrollable Cursors

A cursor is a database query and its *result set*. Execution of a cursor puts the results of the query into a set of rows called the result set. The result set can be fetched either sequentially or non-sequentially. The latter case is known as a *scrollable cursor.*

A scrollable cursor provides support for forward and backward access into the result set from a given position, using either absolute or relative row number offsets into the result set.

Rows are numbered starting at one. For a scrollable cursor, you can fetch previously-fetched rows, the n-th row in the result set, or the n-th row from the current position. Client-side caching of either the partial or entire result set results in fewer calls to the server, thus improving performance.

Oracle does not support DML operations on scrollable cursors. A cursor cannot be made scrollable if the LONG datatype is part of the select list.

Moreover, fetches from a scrollable statement handle are based on the snapshot at execution time. The size of the client cache can be controlled by the existing OCI attributes OCI_ATTR_PREFETCH_ROWS and OCI_ATTR_PREFETCH_MEMORY.

Support for Scrollable Cursors in OCI

The `OCIStmtExecute()` call has an execution mode for scrollable cursors, OCI_STMT_SCROLLABLE_READONLY. The default for statement handles is non-scrollable, that is, forward sequential access only (where the mode is OCI_FETCH_NEXT). You must set this execution mode each time the statement handle is executed.

> **See Also:** `OCIStmtExecute()` for further information

The statement handle attribute OCI_ATTR_CURRENT_POSITION can be retrieved only using `OCIAttrGet()`. This attribute cannot be set by the application. This indicates the current position in the result set.

For non-scrollable cursors, OCI_ATTR_ROW_COUNT is the total number of rows fetched into user buffers with the `OCIStmtFetch2()` calls issued since this statement handle was executed. Since they are forward sequential only, this also represents the highest row number seen by the application.

For scrollable cursors, OCI_ATTR_ROW_COUNT will represent the maximum (absolute) row number fetched into the user buffers. Since the application can

arbitrarily position the fetches, this need not be the total number of rows fetched into the user's buffers since the (scrollable) statement was executed.

The attribute OCI_ATTR_ROWS_FETCHED on the statement handle, represents the number of rows that were successfully fetched into the user's buffers in the last fetch call or execute. It works for both scrollable and non-scrollable cursors.

Use the call OCIStmtFetch2(), which is replacing the call OCIStmtFetch(), which is retained for backward compatibility. You are encouraged to use OCIStmtFetch2() instead, for all new applications, even those not using scrollable cursors. This call also works for non-scrollable cursor, but then an error will be raised if any other orientation besides OCI_DEFAULT or OCI_FETCH_NEXT is passed.

> **See Also:** OCIStmtFetch2() for a complete description of how to make calls that work with scrollable cursors

Example of Access on a Scrollable Cursor

Assume result set is returned by the SQL query:

```
SELECT empno, ename FROM emp
```

and that the table EMP has 14 rows. One usage of scrollable cursors is illustrated in the following code snippet:

```
...
/* execute the scrollable cursor in the scrollable mode */
OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4)
                                  0,  (CONST OCISnapshot *)
                                  NULL, (OCISnapshot *) NULL,
                                  OCI_STMT_SCROLLABLE_READONLY );

/* Fetches rows with absolute row numbers 6, 7, 8. After this call,
 OCI_ATTR_CURRENT_POSITION = 8, OCI_ATTR_ROW_COUNT = 8 */
checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 3,
                             OCI_FETCH_ABSOLUTE, (sb4) 6, OCI_DEFAULT);

/* Fetches rows with absolute row numbers 6, 7, 8. After this call,
 OCI_ATTR_CURRENT_POSITION = 8, OCI_ATTR_ROW_COUNT = 8 */
checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 3,
                             OCI_FETCH_RELATIVE, (sb4) -2, OCI_DEFAULT);

/* Fetches rows with absolute row numbers 14. After this call,
 OCI_ATTR_CURRENT_POSITION = 14, OCI_ATTR_ROW_COUNT = 14 */
```

```
                checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 1,
                                        OCI_FETCH_LAST, (sb4) 0, OCI_DEFAULT);

                /* Fetches rows with absolute row number 1. After this call,
                 OCI_ATTR_CURRENT_POSITION = 1, OCI_ATTR_ROW_COUNT = 14 */
                checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 1,
                                        OCI_FETCH_FIRST, (sb4) 0, OCI_DEFAULT);

                /* Fetches rows with absolute row numbers 2, 3, 4. After this call,
                 OCI_ATTR_CURRENT_POSITION = 4, OCI_ATTR_ROW_COUNT = 14 */
                checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 3,
                                        OCI_FETCH_NEXT, (sb4) 0, OCI_DEFAULT);

                /* Fetches rows with absolute row numbers 3,4,5,6,7. After this call,
                 OCI_ATTR_CURRENT_POSITION = 7, OCI_ATTR_ROW_COUNT = 14. It is assumed
                the user's define memory is allocated. */
                checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 5,
                                        OCI_FETCH_PRIOR, (sb4) 0, OCI_DEFAULT);
                ...
                }
                checkprint (errhp, status)
                {
                 ub4 rows_fetched;
                 checkerr (errhp, status);
                 checkerr(errhp, OCIAttrGet((CONST void *) stmthp, OCI_HTYPE_STMT,
                          (void *) &rows_fetched, (uint *) 0,
                          OCI_ATTR_ROWS_FETCHED, errhp));
                }
                ...
```

5

Binding and Defining

This chapter revisits the basic concepts of binding and defining that were introduced in Chapter 2, "OCI Programming Basics", and provides more detailed information about the different types of binds and defines you can use in OCI applications. Additionally, this chapter discusses the use of arrays of structures, as well as other issues involved in binding, defining, and character conversions.

This chapter includes the following sections:

- Binding

- Advanced Bind Operations

- Defining

- Advanced Define Operations

- Binding and Defining Arrays of Structures

- DML with RETURNING Clause

- Character Conversion Issues in Binding and Defining

- PL/SQL REF CURSORs and Nested Tables

- Runtime Data Allocation and Piecewise Operations

Binding

Most DML statements, and some queries (such as those with a WHERE clause), require a program to pass data 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 following SQL statement, which adds an employee to a database contains several literals, such as 'BESTRY' and 2365:

```
INSERT INTO emp VALUES
    (2365, 'BESTRY', 'PROGRAMMER', 2000, 20)
```

Coding a statement like this into an application would severely limit its usefulness. You would need to change the statement and recompile the program each time you add a new employee to the database. To make the program more flexible, you can write the program so that a user can supply input data at runtime.

When you prepare 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 following SQL statement contains five placeholders, indicated by the leading colons (for example, :ename), that show where input data must be supplied by the program.

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

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. In PL/SQL, placeholders can also be used for output variables.

> **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. When the statement executes, Oracle gets the data that your program placed in the input, or bind, variables and passes it to the server with the SQL statement. Data does not have to be in a bind variable when you perform the bind step. At the bind step, you are only specifying the address, datatype, and length of the variable.

> **Note:** If program variables do not contain data at bind time, make sure they contain valid data when you execute the SQL statement or PL/SQL block using `OCIStmtExecute()`.

For example, given the INSERT statement

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

and the following variable declarations

```
text     *ename, *job;
sword    empno, sal, deptno;
```

the bind step makes an association between the placeholder name and the address of the program variables. The bind also indicates the datatype and length of the program variables, as illustrated in Figure 5–1.

> **See Also:** The code that implements this example is found in the section "Steps Used in Binding" on page 5-6.

Figure 5–1   Using OCIBindByName() to Associate Placeholders with Program Variables


```
INSERT INTO emp  (empno, ename, job, sal, deptno)
        VALUES  (:empno, :ename, :job, :sal, :deptno)
```

OCIBindByName ( )

Address	&empno	ename	job	&sal	&deptno
Data Type	integer	string	string	integer	integer
Length	sizeof(empno)	strlen(ename)+1	strlen(job)+1	sizeof(sal)	sizeof(deptno)
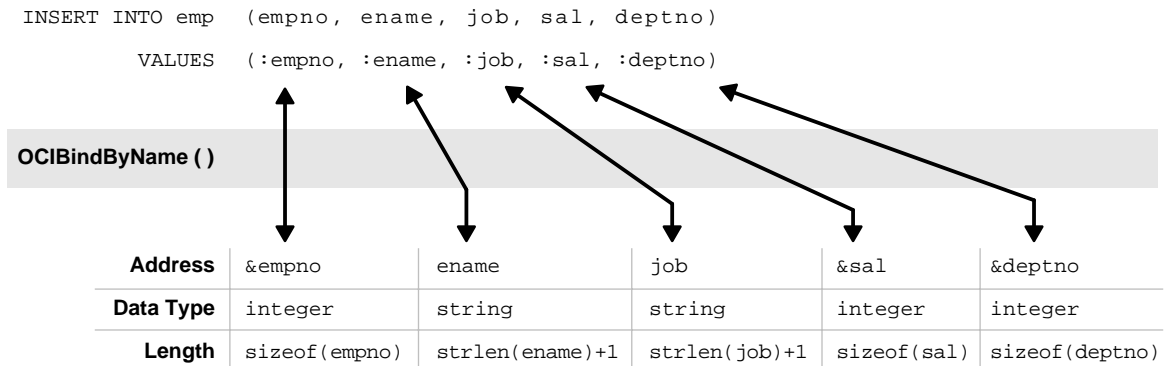
If you change only the value of a bind variable, it is not necessary to rebind in order to execute the statement again. The bind is a bind by reference, so as long as the address of the bind variable and bind handle remain valid, you can reexecute a statement that references the variable without rebinding.

> **Note:** At the interface level, all bind variables are considered at least IN and must be properly initialized. If the variable is a pure OUT bind variable, you can set the variable to zero. You can also provide a NULL indicator and set that indicator to -1 (NULL).

In the Oracle server, new datatypes have been implemented for named datatypes, REFs and LOBs, and they may be bound as placeholders in a SQL statement.

> **Note:** For opaque data types (descriptors or locators) whose sizes are not known to you, pass the address of the descriptor or locator pointer. Set the size parameter to the size of the appropriate data structure, (sizeof(*structure*))

Named Binds and Positional Binds

The SQL statement in the previous section is an example of a *named bind.* Each placeholder in the statement has a name associated with it, such as 'ename' or 'sal'. When this statement is prepared and the placeholders are associated with values in the application, the association is made by the name of the placeholder using the OCIBindByName() call with the name of the placeholder passed in the *placeholder* parameter.

A second type of bind is known as a *positional bind.* In a positional bind, the placeholders are referred to by their position in the statement rather than their names. For binding purposes, an association is made between an input value and the position of the placeholder, using the OCIBindByPos() call.

The example from the previous section could also be used for a positional bind:

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

The five placeholders would then each be bound by calling OCIBindByPos() and passing the position number of the placeholder in the *position* parameter. For example, the :empno placeholder would be bound by calling OCIBindByPos() with a position of 1, :ename with a position of 2, and so on.

In the case of a duplicate bind, only a single bind call may be necessary. Consider the following SQL statement, which queries the database for those employees whose commission and salary are both greater than a given amount:

```
SELECT empno FROM emp
```

```
WHERE sal > :some_value
AND comm > :some_value
```

An OCI application could complete the binds for this statement with a single call to
`OCIBindByName()` to bind the `:some_value` placeholder by name. In this case,
the second placeholder inherits the bind information from the first placeholder.

OCI Array Interface

You can pass data to Oracle in various ways. You can execute a SQL statement
repeatedly using the `OCIStmtExecute()` 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 and a single call to
`OCIStmtExecute()`. In this case you bind an array to an input placeholder, and
the entire array can be passed at the same time, under the control of the *iters*
parameter.

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

> **Note:**   When using the OCI array interface to perform inserts, row
> triggers in the database are fired as each row of the insert gets
> inserted.

Binding Placeholders in PL/SQL

You process a PL/SQL block by placing the block in a string variable, 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 `OCIBindByName()` or `OCIBindByPos()` to perform the basic bind binds. You
can use `OCIBindByName()` or `OCIBindByPos()` to bind host variables that are
either scalars or arrays.

The following short PL/SQL block contains two placeholders, which represent IN parameters to a procedure that updates an employee's salary, given the employee number and the new salary amount:

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

These placeholders can be bound to input variables in the same way as placeholders in a SQL statement.

When processing PL/SQL statements, output variables are also associated with program variables using bind calls.

For example, in a PL/SQL block such as

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

you would use `OCIBindByName()` to bind variables in place of the *:emp_name*, *:salary*, and *:commission* output placeholders, and in place of the input placeholder *:emp_number*.

7.x Upgrade Note: In the Oracle7 OCI, it was sufficient for applications to initialize only IN-bind buffers. In later releases, all buffers, even pure OUT buffers, must be initialized by setting the buffer length to zero in the bind call, or by setting the corresponding indicator to -1.

See Also: For more information about binding PL/SQL placeholders see "Information for Named Datatype and REF Binds" on page 11-37.

Steps Used in Binding

Binding placeholders is done in one or more steps. For a simple scalar or array bind, it is only necessary to specify an association between the placeholder and the data. This is done by using OCI bind by name (`OCIBindByName()`) or OCI bind by position (`OCIBindByPos()`) call.

> **See Also:** See the section "Named Binds and Positional Binds" on page 5-4 for information about the difference between these types of binds.

Once the bind is complete, the OCI library knows where to find the input data (or where to put PL/SQL output data) when the SQL statement is executed. As mentioned in the section "Binding" on page 5-2, program input data does not need to be in the program variable when it is bound to the placeholder, but the data must be there when the statement is executed.

The following code example shows handle allocation and binding for each of five placeholders in a SQL statement.

> **Note:** The checkerr() function evaluates the return code from an OCI application. The code for the function is listed in the section "Error Handling" on page 2-31.

```
...
/* The SQL statement, associated with stmthp (the statement handle)
by calling OCIStmtPrepare() */
text *insert = (text *) "INSERT INTO emp(empno, ename, job, sal, deptno)\
    VALUES (:empno, :ename, :job, :sal, :deptno)";
...

/*  Bind the placeholders in the SQL statement, one per bind handle. */
checkerr(errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":ENAME",
    strlen(":ENAME"), (ub1 *) ename, enamelen+1, STRING_TYPE, (dvoid *) 0,
    (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":JOB",
    strlen(":JOB"), (ub1 *) job, joblen+1, STRING_TYPE, (dvoid *)
    &job_ind, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":SAL",
    strlen(":SAL"), (ub1 *) &sal, (sword) sizeof(sal), INT_TYPE,
    (dvoid *) &sal_ind, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0,
    OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd4p, errhp, (text *) ":DEPTNO",
    strlen(":DEPTNO"), (ub1 *) &deptno,(sword) sizeof(deptno), INT_TYPE,
    (dvoid *) 0, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd5p, errhp, (text *) ":EMPNO",
    strlen(":EMPNO"), (ub1 *) &empno, (sword) sizeof(empno), INT_TYPE,
    (dvoid *) 0, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0,OCI_DEFAULT))
```

PL/SQL 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 shows how to embed a stored procedure call in an OCI application. For the sake of brevity, only the relevant portions of the program are reproduced here.

The program passes an employee number and a salary increase as inputs to a stored procedure called raise_salary, which takes these parameters:

```
raise_salary (employee_num IN, sal_increase IN, new_salary OUT);
```

This procedure raises a given employee's salary by a given amount. The increased salary which results is returned in the stored procedure's OUT variable new_salary, and the program displays this value.

```
/* Define PL/SQL statement to be used in program. */
text *give_raise = (text *) "BEGIN\
                  RAISE_SALARY(:emp_number,:sal_increase, :new_salary);\
                    END;";
OCIBind  *bnd1p = NULL;                       /* the first bind handle */
OCIBind  *bnd2p = NULL;                       /* the second bind handle */
OCIBind  *bnd3p = NULL;                        /* the third bind handle */

static void checkerr();
sb4 status;

main()
{
  sword    empno, raise, new_sal;
  dvoid    *tmp;
  OCISession *usrhp = (OCISession *)NULL;
...
/* attach to database server, and perform necessary initializations
and authorizations */
...
      /* allocate a statement handle */
  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
          OCI_HTYPE_STMT, 100, (dvoid **) &tmp));
```

```
      /* prepare the statement request, passing the PL/SQL text
         block as the statement to be prepared */
checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) give_raise, (ub4)
      strlen(give_raise), OCI_NTV_SYNTAX, OCI_DEFAULT));

      /* bind each of the placeholders to a program variable */
 checkerr( errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":emp_number",
            -1, (ub1 *) &empno,
           (sword) sizeof(empno), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

 checkerr( errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":sal_increase",
            -1, (ub1 *) &raise,
            (sword) sizeof(raise), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

      /* remember that PL/SQL OUT variable are bound, not defined */

checkerr( OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":new_salary",
            -1, (ub1 *) &new_sal,
            (sword) sizeof(new_sal), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

      /* prompt the user for input values */
printf("Enter the employee number: ");
scanf("%d", &empno);
      /* flush the input buffer */
myfflush();

printf("Enter employee's raise: ");
scanf("%d", &raise);
      /* flush the input buffer */
myfflush();

  /* execute PL/SQL block*/
  checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
      (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT));

  /* display the new salary, following the raise */
printf("The new salary is %d\n", new_sal);
}
```

The following is one possible sample output from this program. Before execution, the salary of employee 7954 is 2000.

```
Enter the employee number: 7954
Enter employee's raise: 1000
The new salary is 3000.
```

The previous section and example demonstrated how to perform a simple scalar bind. In that case, only a single bind call is necessary. In some cases, additional bind calls are necessary to define specific attributes for specific bind datatypes or execution modes. These more sophisticated bind operations are discussed in the following section.

Oracle also provides predefined C datatypes that map object attributes.

> **See Also:** Information about binding these datatypes, such as **OCIDate** and **OCINumber**, can be found in Chapter 12, "Direct Path Loading".

Advanced Bind Operations

The section "What is Binding?" on page 4-6 discussed how a basic bind operation is performed to create an association between a placeholder in a SQL statement and a program variable using OCIBindByName() or OCIBindByPos().

This section covers more advanced bind operations, including multi-step binds, and binds of named data types and REFs.

In certain cases, additional bind calls are necessary to define specific attributes for certain bind data types or certain execution modes.

The following sections describe these special cases, and the information about binding is summarized in Table 5–1, "Bind Information for Different Bind Types".

Named Data Type Binds

For information on binding named data types (objects),

> **See Also:** "Named Datatype Binds" on page 11-36.

Binding REFs

For information on this topic,

> **See Also:** "Binding REFs" on page 11-37.

Binding LOBs

There are two ways of binding LOBs:

- Bind the LOB locator, rather than the actual LOB values. In this case the LOB value is written or read by passing a LOB locator to the OCILob functions.

- Bind the LOB value directly, without using the LOB locator.

Both of these ways are discussed next.

Binding LOB Locators

Either a single locator or an array of locators can be bound in a single bind call. In each case, the application must pass *the address of a LOB locator* and not the locator itself. For example, if an application has prepared a SQL statement like

```
INSERT INTO some_table VALUES (:one_lob)
```

where `one_lob` is a bind variable corresponding to a LOB column, and has made the following declaration:

```
OCILobLocator * one_lob;
```

Then the following sequence of steps would be used to bind the placeholder, and execute the statement

```
/* initialize single locator */
one_lob = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
...
/* pass the address of the locator */
OCIBindByName(...,(dvoid *) &one_lob,... SQLT_CLOB, ...);
OCIStmtExecute(...,1,...)                /* 1 is the iters parameter */
```

> **Note:** In these examples, most parameters are omitted for simplicity.

You could also do an array insert using the same SQL INSERT statement. In this case, the application would include the following code:

```
OCILobLocator * lob_array[10];
...
for (i=0; i<10, i++)
    lob_array[i] = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
                                    /* initialize array of locators */
```

```
...
OCIBindByName(...,(dvoid *) lob_array,...);
OCIBindArrayOfStruct(...);
OCIStmtExecute(...,10,...);                  /* 10 is the iters parameter */
```

Note that you must allocate descriptors with the `OCIDescriptorAlloc()` routine before they can be used. In the case of an array of locators, you must initialize each array element using `OCIDescriptorAlloc()`. Use OCI_DTYPE_LOB as the *type* parameter when allocating `BLOBs`, `CLOBs`, and `NCLOBs`. Use OCI_DTYPE_FILE when allocating `BFILEs`

Binding LOB Data

Oracle allows nonzero binds for `INSERTs` and `UPDATEs` of any size LOB. So you can bind up to 4 gigabytes of data into a LOB column using `OCIBindByPos()`, `OCIBindByName()`, and PL/SQL binds. Because you can have multiple LOBs in a row, you can bind up to 4 gigabytes of data for each one of those LOBs in the same `INSERT` or `UPDATE` statement.

The bind of more than 4 kilobytes of data to a LOB column uses space from the temporary tablespace. Users of this features should make sure that their temporary tablespace is big enough to hold at least the amount of data equal to the sum of all the bind lengths for LOBs. If your temporary tablespace is extendable, it will be extended automatically after the existing space is fully consumed. Use the command:

```
"CREATE TABLESPACE ... AUTOEXTENT ON ... TEMPORARY ...;"
```

to create an extendable temporary tablespace.

Restrictions on LOB Binds

- If a table has both `LONG` and LOB columns, then you can have binds of greater than 4 kilobytes for either the `LONG` column or the LOB columns, but not both in the same statement.

- You cannot bind data of any size to LOB attributes in object-relational datatypes.For LOB attributes, you need to insert an empty LOB locator and then modify the contents of the LOB using `OCILob*()` functions.

- In an `INSERT AS SELECT` operation, Oracle does not allow binding of any length data to LOB columns.

- Oracle doesn't do any implicit conversion such as `HEX` to `RAW` or `RAW` to `HEX` for data of size more than 4000 bytes. The following PL/SQL code illustrates this:

```
create table t (c1 clob, c2 blob);
declare
  text   varchar(32767);
  binbuf raw(32767);
begin
  text := lpad ('a', 12000, 'a');
  binbuf := utl_raw.cast_to_raw(text);

  -- The following works ...
  insert into t values (text, binbuf);

  -- The following won't work because Oracle won't do implicit
  -- hex to raw conversion.
  insert into t (c2) values (text);

  -- The following won't work because Oracle won't do implicit
  -- raw to hex conversion.
  insert into t (c1) values (binbuf);

  -- The following won't work because we can't combine the
  -- utl_raw.cast_to_raw() operator with the >4k bind.
  insert into t (c2) values (utl_raw.cast_to_raw(text));

end;
/
```

- If you bind more than 4000 bytes of data to a BLOB or a CLOB, and the data is filtered by a SQL operator, then Oracle will limit the size of the result to at most 4000 bytes.

  For example:

```
create table t (c1 clob, c2 blob);
-- The following command inserts only 4000 bytes because the result of
-- LPAD is limited to 4000 bytes
insert into t(c1) values (lpad('a', 5000, 'a'));

-- The following command inserts only 2000 bytes because the result of
-- LPAD is limited to 4000 bytes, and the implicit hex to raw conversion
-- converts it to 2000 bytes of RAW data.
insert into t(c2) values (lpad('a', 5000, 'a'));
```

Examples of Binding LOBs

Consider the following SQL statements which will be used in the examples that follow:

```
CREATE TABLE foo( a INTEGER );
CREATE TYPE lob_typ( A1 CLOB );
CREATE TABLE lob_long_tab (C1 CLOB, C2 CLOB, CT3 lob_typ, L LONG);
```

Example1: Binding LOBs

```
void insert()                  /* A function in an OCI program */
{
   /* The following is allowed */
   ub1 buffer[8000];
   text *insert_sql = "INSERT INTO lob_long_tab (C1, C2, L)
                       VALUES (:1, :2, :3)";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 2000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Example2: Binding LOBs

```
void insert()
{
   /* The following is allowed */
   ub1 buffer[8000];
   text *insert_sql = "INSERT INTO lob_long_tab (C1, L)
                       VALUES (:1, :2)";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
                SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Example3: Binding LOBs

```
void insert()
{
   /* The following is allowed, no matter how many rows it updates */
   ub1 buffer[8000];
   text *insert_sql = (text *)"UPDATE lob_long_tab SET
                              C1 = :1, C2=:2, L=:3";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
               SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
               SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 2000,
               SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Example4: Binding LOBs

```
void insert()
{
   /* The following is allowed, no matter how many rows it updates */
   ub1 buffer[8000];
   text *insert_sql = (text *)"UPDATE lob_long_tab SET
                              C1 = :1, C2=:2, L=:3";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
               SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 2000,
               SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 8000,
               SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Example5: Binding LOBs

```
void insert()
{
   /* Piecewise, callback and array insert/update operations similar to
    * the allowed regular insert/update operations are also allowed */
}
```

Example6: Binding LOBs

```
void insert()
{
   /* The following is NOT allowed because we try to insert >4000 bytes
    * to both LOB and LONG columns */
   ub1 buffer[8000];
   text *insert_sql = (text *)"INSERT INTO lob_long_tab (C1, L)
                              VALUES (:1, :2)";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Example7: Binding LOBs

```
void insert()
{
   /* The following is NOT allowed because we try to insert data into
    * LOB attributes */
   ub1 buffer[8000];
   text *insert_sql = (text *)"INSERT INTO lob_long_tab (CT3)
                              VALUES (lob_typ(:1))";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
              SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Example8: Binding LOBs

```
void insert()
{
   /* The following is NOT allowed because we try to do insert as
    * select character data into LOB column */
   ub1 buffer[8000];
   text *insert_sql = (text *)"INSERT INTO lob_long_tab (C1) SELECT
                              :1 from FOO";
   OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
```

```
                    SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Other Disallowed Operations

Other update operations similar to the disallowed insert operations are also not allowed. Piecewise and callback INSERT or UPDATE operations similar to the disallowed regular INSERT or UPDATE operations are also not allowed.

> **See Also:** For more information about the OCILob functions, refer to Chapter 7, "LOB and FILE Operations".

Binding FILEs

When using a FILE locator as a bind variable for an INSERT or UPDATE statement, you must first initialize the locator with a directory alias and filename (using OCILobFileSetName()) before issuing the INSERT or UPDATE statement.

Binding in OCI_DATA_AT_EXEC Mode

If the *mode* parameter in a call to OCIBindByName() or OCIBindByPos() is set to OCI_DATA_AT_EXEC, an additional call to OCIBindDynamic() is necessary if the application will use the callback method for providing data at runtime. The call to OCIBindDynamic() sets up the callback routines, if necessary, for indicating the data or piece that is being provided.

If the OCI_DATA_AT_EXEC mode is chosen, but the standard OCI piecewise polling method will be used instead of callbacks, the call to OCIBindDynamic() is not necessary.

When binding RETURN clause variables, an application must use OCI_DATA_AT_EXEC mode, and it must provide callbacks.

> **See Also:** For more information about piecewise operations, please refer to the section "Runtime Data Allocation and Piecewise Operations" on page 5-44.

Binding Ref Cursor Variables

Ref Cursors are bound to a statement handle with a bind datatype of SQLT_RSET.

> **See Also:** "PL/SQL REF CURSORs and Nested Tables" on page 5-43

Summary of Bind Information

The following table summarizes the bind calls necessary for different types of binds. For each type, the table lists the bind datatype (passed in the `dty` parameter of `OCIBindByName()` or `OCIBindByPos()`), and notes about the bind

Table 5–1   Bind Information for Different Bind Types

Type of Bind	Bind Datatype	Notes
Scalar	any scalar datatype	Bind a single scalar using `OCIBindByName()` or `OCIBindByPos()`.
Array of Scalars	any scalar datatype	Bind an array of scalars using `OCIBindByName()` or `OCIBindByPos()`.
Named Data Type	SQLT_NTY	Two bind calls are required: ■ `OCIBindByName()` or `OCIBindByPos()` ■ `OCIBindObject()`
REF	SQLT_REF	Two bind calls are required: ■ `OCIBindByName()` or `OCIBindByPos()` ■ `OCIBindObject()`
LOB BFILE	SQLT_BLOB SQLT_CLOB	Allocate the LOB locator using `OCIDescriptorAlloc()`, and then bind its address (**OCILobLocator** **) with `OCIBindByName()` or `OCIBindByPos()`, using one of the LOB datatypes.
Array of Structures or Static Arrays	varies	Two bind calls are required: ■ `OCIBindByName()` or `OCIBindByPos()` ■ `OCIBindArrayOfStruct()`
Piecewise Insert	varies	`OCIBindByName()` or `OCIBindByPos()` is required. The application may also need to call `OCIBindDynamic()` to register piecewise callbacks.
REF CURSOR variables	SQLT_RSET	Allocate a statement handle, **OCIStmt**, and then bind its address (**OCIStmt** **) using the SQLT_RSET datatype.

See Also: For more information about datatypes and datatype codes, see Chapter 3, "Datatypes".

Defining

Query statements return data from the database to your application. When processing a query, you must define an output variable or an array of output variables for each item in the select-list from which you want to retrieve data. The define step creates an association that determines where returned results are stored, and in what format.

For example, if your OCI statement processes the following statement:

```
SELECT name, ssn FROM employees
    WHERE empno = :empnum
```

you would normally need to define two output variables, one to receive the value returned from the name column, and one to receive the value returned from the ssn column.

> **Note:**   If you were only interested in retrieving values from the name column, you would not need to define an output variable for ssn.

If the SELECT statement being processed might return more than a single value for a query, the output variables you define may be arrays instead of scalar values.

Depending on the application, the define step can take place before or after the execute. If the datatypes of select-list items are known when the application is coded, the define can take place before the statement is executed. If your application is processing dynamic SQL statements—statements entered by you at runtime— or statements that do not have a clearly defined select-list, such as

```
SELECT * FROM employees
```

the application must execute the statement and retrieve describe information before defining output variables.

> **See Also:**   See the section "Describing Select-List Items" on page 4-12 for more information.

The OCI processes the define call locally, on the client side. In addition to indicating the location of buffers where results should be stored, the define step also determines what type of data conversions, if any, will take place when data is returned to the application.

> **Note:** Output buffers must be 2-byte aligned.

The *dty* parameter of the OCIDefineByPos() call specifies the datatype of the output variable. The OCI is capable of a wide range of data conversions when data is fetched into the output variable. For example, internal data in Oracle DATE format can be automatically converted to a string datatype on output.

> **See Also:** For more information about datatypes and conversions, refer to Chapter 3, "Datatypes".

Steps Used in Defining

Defining output variables is done in one or more steps. A basic define is accomplished with the OCI define by position call, OCIDefineByPos(). This step creates an association between a select-list item and an output variable. Additional define calls may be necessary for certain datatypes or fetch modes.

Once the define step is complete, the OCI library knows where to put retrieved data after fetching it from the database.

> **Note:** You can make your define calls again to redefine the output variables without having to reprepare or reexecute the SQL statement.

The following example code shows a scalar output variable being defined following an execute and a describe.

```
/* The following statement was prepared, and associated with statement
   handle stmthp1.

   SELECT dname FROM dept WHERE deptno = :dept_input

   The input placeholder was bound earlier, and the data comes from the
   user input below */

printf("Enter employee dept: ");
     scanf("%d", &deptno);
     myfflush();

/* Execute the statement. If OCIStmtExecute() returns OCI_NO_DATA, meaning that
no data matches the query, then the department number is invalid. */
```

```
        if ((status = OCIStmtExecute(svchp, stmthp1, errhp, 0, 0, 0, 0,
              OCI_DEFAULT))
            && (status != OCI_NO_DATA))
        {
          checkerr(errhp, status);
          do_exit(EXIT_FAILURE);
        }
        if (status == OCI_NO_DATA) {
          printf("The dept you entered doesn't exist.\n");
          return 0;
          }
/* The next two statements describe the select-list item, dname, and
   return its length */
checkerr(errhp, OCIParamGet(stmthp1, errhp, &parmdp, (ub4) 1));
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &deptlen, (ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
        (OCIError *) errhp  ));

/* Use the retrieved length of dname to allocate an output buffer, and
   then define the output variable. If the define call returns an error,
   exit the application */
  dept = (text *) malloc((int) deptlen + 1);
  if (status = OCIDefineByPos(stmthp1, &defnp, errhp,
            1, (ub1 *) dept, deptlen+1,
            SQLT_STRING, (dvoid *) 0,
            (ub2 *) 0, OCI_DEFAULT))
  {
    checkerr(errhp, status);
    do_exit(EXIT_FAILURE);
  }
```

> **See Also:** For an explanation of the describe step, see the section

Advanced Defines

In some cases the define step requires more than just a call to OCIDefineByPos().
There are additional calls that define the attributes of an array fetch
(OCIDefineArrayOfStruct()) or a named data type fetch
(OCIDefineObject()). For example, to fetch multiple rows with a column of
named data types, all three calls must be invoked for the column; but to fetch
multiple rows of scalar columns, OCIDefineArrayOfStruct() and
OCIDefineByPos() are sufficient.

> **See Also:** These more sophisticated define operations are covered in the section "Advanced Define Operations" on page 5-22.

Oracle also provides pre-defined C datatypes that map object type attributes.

> **See Also:** Information about defining these datatypes (for example, **OCIDate**, **OCINumber**) can be found in Chapter 11, "Object-Relational Datatypes"

Advanced Define Operations

The section "What is Defining?" on page 4-15 discussed how a basic define operation is performed to create an association between a SQL select-list item and an output buffer in an application.

This section covers more advanced defined operations, including multi-step defines, and defines of named data types and REFs.

In some cases the define step requires more than just a call to `OCIDefineByPos()`. There are additional calls that define the attributes of an array fetch (`OCIDefineArrayOfStruct()`) or a named data type fetch (`OCIDefineObject()`). For example, to fetch multiple rows with a column of named data types, all the three calls must be invoked for the column; but to fetch multiple rows of scalar columns only `OCIDefineArrayOfStruct()` and `OCIDefineByPos()` are sufficient.

The following sections discuss specific information pertaining to different types of defines.

Defining Named Data Type Output Variables

For information on defining named data type (object) output variables, refer to "Defining Named Datatype Output Variables" on page 11-38.

Defining REF Output Variables

For information on defining REF output variables, refer to "Defining REF Output Variables" on page 11-39.

Defining LOB Output Variables

There are two ways of defining LOBs:

■ Define as a LOB locator, rather than the actual LOB values. In this case the LOB value is written or read by passing a LOB locator to the OCILob functions.

■ Define as a LOB value directly, without using the LOB locator.

Both of these ways are discussed next.

Defining LOB Locators

Either a single locator or an array of locators can be defined in a single define call. In each case, the application must pass *the address of a LOB locator* and not the locator itself. For example, if an application has prepared a SQL statement like:

```
SELECT lob1 FROM some_table;
```

where `lob1` is the LOB column and `one_lob` is a define variable corresponding to a LOB column with the following declaration:

```
OCILobLocator * one_lob;
```

Then the following sequence of steps would be used to bind the placeholder, and execute the statement

```
/* initialize single locator */
one_lob = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
...
/* pass the address of the locator */
OCIDefineByPos(... 1, ...,(dvoid *) &one_lob,... SQLT_CLOB, ...);
OCIStmtExecute(...,1,...)                    /* 1 is the iters parameter */
```

> **Note:** In these examples, most parameters are omitted for simplicity.

You could also do an array select using the same SQL SELECT statement. In this case, the application would include the following code:

```
OCILobLocator * lob_array[10];
...
for (i=0; i<10, i++)
   lob_array[i] = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
                                    /* initialize array of locators */
...
OCIDefineByPos(...,1, (dvoid *) lob_array,... SQLT_CLOB, ...);
OCIDefineArrayOfStruct(...);
```

```
OCIStmtExecute(...,10,...);                    /* 10 is the iters parameter */
```

Note that you must allocate descriptors with the OCIDescriptorAlloc() routine before they can be used. In the case of an array of locators, you must initialize each array element using OCIDescriptorAlloc(). Use OCI_DTYPE_LOB as the *type* parameter when allocating BLOBs, CLOBs, and NCLOBs. Use OCI_DTYPE_FILE when allocating BFILEs

Defining LOB Data

Oracle allows nonzero defines for SELECTs of any size LOB. So you can select up to 4 gigabytes of data from a LOB column using OCIDefineByPos(), and PL/SQL defines. Because you can have multiple LOBs in a row, you can select up to 4 gigabytes of data from each one of those LOBs in the same SELECT statement.

Examples of Defining LOBs

Consider the following SQL statements which will be used in the examples that follow:

```
CREATE TABLE lob_tab (C1 CLOB, C2 CLOB);
```

Example1: Defining LOBs

```
void select_define_before_execute()      /* A function in an OCI program */
{
   /* The following is allowed */
   ub1 buffer1[8000];
   ub1 buffer2[8000];
   text *select_sql = "SELECT c1, c2 FROM lob_tab";
   OCIStmtPrepare(stmthp, errhp, select_sql, strlen((char*)select_sql),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[0], errhp, 1, (dvoid *)buffer1, 8000,
                SQLT_LNG, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (dvoid *)buffer2, 8000,
                SQLT_LNG, 0, 0, 0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

Example2: Defining LOBs

```
void select_execute_before_define()
{
   /* The following is allowed */
```

```
    ub1 buffer1[8000];
    ub1 buffer2[8000];
    text *select_sql = "SELECT c1, c2 FROM lob_tab";
    OCIStmtPrepare(stmthp, errhp, select_sql, strlen((char*)select_sql),
                   (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIStmtExecute(svchp, stmthp, errhp, 0, 0, OCI_DEFAULT);
    OCIDefineByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer1, 8000,
                   SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIDefineByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer2, 8000,
                   SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);
}
```

Example3: Defining LOBs

```
void select()
{
    /* Piecewise, callback and array select operations similar to
     * the allowed regular select operations are also allowed */
}
```

Defining PL/SQL Output Variables

You do not use the define calls to define output variables for select-list items in a SQL SELECT statement in a PL/SQL block. You must use OCI bind calls instead.

> **See Also:** See the section "Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-39 for more information about defining PL/SQL output variables.

Defining For a Piecewise Fetch

When performing a piecewise fetch, an initial call to OCIDefineByPos() is required. An additional call to OCIDefineDynamic() is necessary if the application will use callbacks rather than the standard polling mechanism for fetching data.

> **See Also:** See the section "Runtime Data Allocation and Piecewise Operations" on page 5-44 for more information.

Binding and Defining Arrays of Structures

When using arrays of structures, an initial call to `OCIDefineByPos()` is required. An additional call to `OCIDefineArrayOfStruct()` is necessary to set up additional parameters, including the skip parameter necessary for arrays of structures operations.

The arrays of structures functionality of OCI can simplify the processing of multi-row, multi-column operations. The OCI programmer can create a structure of related scalar data items and then fetch values from the database into an array of these structures or insert values into the database from an array of these structures.

For example, an application may need to fetch multiple rows of data from three columns named NAME, AGE, and SALARY. The OCI application could include the definition of a structure containing separate fields to hold the NAME, AGE and SALARY data from one row in the database table. The application would then fetch data 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 `OCIDefineArrayOfStruct()` and `OCIBindArrayOfStruct()` calls, specifies where fetched data will be stored, or where inserted or updated data will be found.

Figure 5–2, "Fetching Data Into an Array of Structures" is a graphical representation of this process. In the figure, an application fetches various fields from a database row into a single structure in an array of structures. Each column being fetched corresponds to one of the fields in the structure.

Figure 5–2   Fetching Data Into an Array of Structures
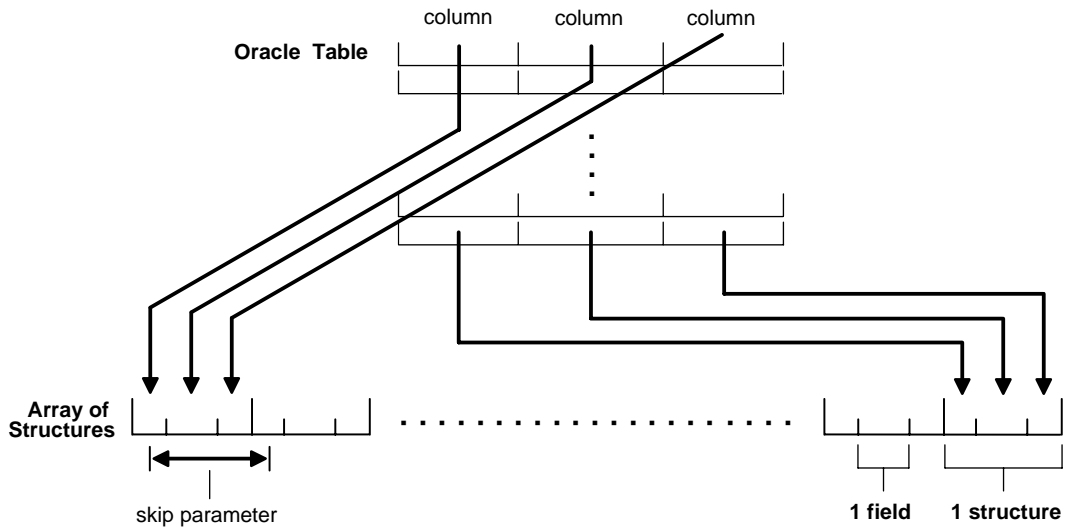
Skip Parameters

When you split column data 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 they are binding or defining. 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.

The figure below demonstrates how a skip parameter is determined. In this case the skip parameter is the sum of the sizes of the fields `field1, field2,` and `field3`, which is 8 bytes. This equals the size of one structure.

Figure 5–3   Determining Skip Parameters

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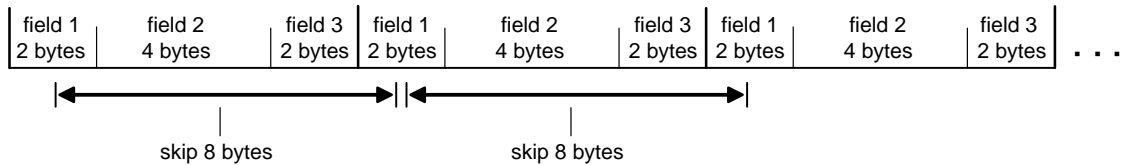
On some systems it may be necessary to set the skip parameter to be `sizeof(`*one_array_element*`)` rather than `sizeof(`*struct*`)`. This is because some compilers may insert padding into a structure. For example, consider an array of C structures consisting of two fields, a **ub4** and a **ub1**.

```
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, `skip_parameter` will be set to five bytes by this statement. In this case, use the following statement to get the correct value for the skip parameter:

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

Skip Parameters for Standard Arrays

The ability to work with arrays of structures is an extension of the functionality for binding and defining arrays of program variables. Programmers can also work with standard arrays (as opposed to arrays of structures). 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 OCI calls must be used when performing operations involving arrays of structures: `OCIBindArrayOfStruct()` (for binding fields in arrays of structures for input variables) and `OCIDefineArrayOfStruct()` (for defining arrays of structures for output variables).

> **Note:** When binding or defining for arrays of structures, multiple calls are required. A call to `OCIBindByName()` or `OCIBindByPos()` must proceed a call to `OCIBindArrayOfStruct()`, and a call to `OCIDefineByPos()` must proceed a call to `OCIDefineArrayOfStruct()`.

> **See Also:** See the descriptions of `OCIBindArrayOfStruct()` and `OCIDefineArrayOfStruct()` for syntax and parameter descriptions.

Arrays of Structures and Indicator Variables

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 `OCIBindArrayOfStruct()` or `OCIDefineArrayOfStruct()`.

You can set up arrays of structures of program values and indicator variables in many ways. For example, consider an application that fetches data from three database columns into an array of structures containing three fields. You can set up 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.

> **Note:** A one-to-one relationship between the fields in an indicator struct and the number of select-list items is not necessary.

See Also: See "Indicator Variables" on page 2-36 for more
information about indicator variables.

DML with RETURNING Clause

The OCI supports the use of the RETURNING clause with SQL INSERT, UPDATE,
and DELETE statements. This section outlines the rules an OCI application must
follow to correctly implement DML statements with the RETURNING clause.

Note: For more information about the use of the RETURNING
clause with INSERT, UPDATE, or DELETE statements, please refer to
the descriptions of those commands in the *Oracle9i SQL Reference*.

See Also: For a complete code example, see the demonstration
programs included with your Oracle installation. For additional
information, refer to Appendix B, "OCI Demonstration Programs"

Using DML with RETURNING Clause

Using the RETURNING clause with a DML statement nonzero you to essentially
combine two SQL statements into one, possibly saving you a server round trip. This
is accomplished by adding an extra clause to the traditional UPDATE, INSERT, and
DELETE statements. The extra clause effectively adds a query to the DML statement.

In the OCI, the values are returned to the application through the use of OUT bind
variables. The rules for binding these variables are described in the next section. In
the following examples, the bind variables are indicated by the preceding colon,
such as :out1. These examples assume the existence of a table called table1,
which contains three columns: col1, col2, and col3.

For example, the following statement inserts new values into the database and then
retrieves the column values of the affected row from the database, allowing your
application to work with inserted rows.

```
INSERT INTO table1 VALUES (:1, :2, :3,)
    RETURNING col1, col2, col3
    INTO :out1, :out2, :out3
```

The next example uses the UPDATE statement. This statement updates the values of
all columns whose col1 value falls within a certain range, and then returns the
affected rows to the application, allowing the application to see which rows were
modified.

```
UPDATE table1 SET col1 = col1 + :1, col2 = :2, col3 = :3
    WHERE col1 >= :low AND col1 <= :high
    RETURNING col1, col2, col3
    INTO :out1, :out2, :out3
```

The following DELETE statement deletes the rows whose col1 value falls within a certain range, and then returns the data from those rows so that the application can check them.

```
DELETE FROM table1 WHERE col1 >= :low AND col2 <= :high
    RETURNING col1, col2, col3
    INTO :out1, :out2, :out3
```

Note that in both the UPDATE and DELETE examples there is the possibility that the statement will affect multiple rows in the table. Additionally, a DML statement could be executed multiple times in a single OCIExecute() statement. Because of this possibility for multiple returning values, an OCI application may not know how much data will be returned at runtime. As a result, the variables corresponding to the RETURNING...INTO placeholders must be bound in OCI_DATA_AT_EXEC mode. It is an additional requirement that the application must define its own dynamic data handling callbacks rather than using the OCI_DATA_AT_EXEC polling mechanism.

> **Note:** Even if the application can be sure that it will only get a single value back in the RETURNING clause, it must still bind in OCI_DATA_AT_EXEC mode and use callbacks.

The returning clause can be particularly useful when working with LOBs. Normally, an application must insert an empty LOB locator into the database, and then SELECT it back out again to operate on it. Using the RETURNING clause, the application can combine these two steps into a single statement:

```
INSERT INTO some_table VALUES (:in_locator)
    RETURNING lob_column
    INTO :out_locator
```

Binding RETURNING...INTO variables

An OCI application implements the placeholders in the RETURNING clause as pure OUT bind variables. However, all binds in the RETURNING clause are initially IN

and must be properly initialized. To provide a valid value, you can provide a NULL indicator and set that indicator to -1 (NULL).

An application must adhere to the following rules when working with bind variables in a RETURNING clause:

1. Bind RETURNING clause placeholders in OCI_DATA_AT_EXEC mode using OCIBindByName() or OCIBindByPos(), followed by a call to OCIBindDynamic() for each placeholder.

   Note: The OCI only supports the callback mechanism for RETURNING clause binds. The polling mechanism is not supported.

2. When binding RETURNING clause placeholders, you must supply a valid out bind function as the *ocbfp* parameter of the OCIBindDynamic() call. This function must provide storage to hold the returned data.

3. The *icbfp* parameter of OCIBindDynamic() call should provide a "dummy" function which returns NULL values when called.

4. The *piecep* parameter of OCIBindDynamic() must be set to OCI_ONE_PIECE.

5. No duplicate binds are allowed in a DML statement with a RETURNING clause, such as no duplication between bind variables in the DML section and the RETURNING section of the statement.

Error Handling

The out bind function provided to OCIBindDynamic() must be prepared to receive partial results of a statement in the event of an error. For example, if the application has issued a DML statement which should be executed 10 times, and an error occurs during the fifth iteration, the server will still return the data from iterations 1 through 4. The callback function would still be called to receive data for the first four iterations.

DML with RETURNING REF...INTO clause

The RETURNING clause can also be used to return a REF to an object which is being inserted into or updated in the database. The following SQL statement shows how this could be used.

```
UPDATE EXTADDR E SET E.ZIP = '12345', E.STATE='AZ'
    WHERE E.STATE = 'CA' AND E.ZIP='95117'
    RETURNING REF(E), ZIP
    INTO :addref, :zip
```

This statement updates several attributes of an object in an object table and then returns a REF to the object (along with the scalar ZIP code) in the RETURNING clause.

Binding the REF output variable in an OCI application requires three steps:

1. The initial bind information is set using OCIBindByName()

2. Additional bind information for the REF (including the TDO) is set with OCIBindObject()

3. A call to OCIBindDynamic()

The following pseudocode shows a function which performs the binds necessary for the above example.

```
sword bind_output(stmthp, bndhp, errhp)
OCIStmt *stmthp;
OCIBind *bndhp[];
OCIError *errhp;
{
  ub4 i;
                                  /* get TDO for BindObject call */
  if (OCITypeByName(envhp, errhp, svchp, (CONST text *) 0,
                  (ub4) 0, (CONST text *) "ADDRESS_OBJECT",
                  (ub4) strlen((CONST char *) "ADDRESS_OBJECT"),
                  (CONST text *) 0, (ub4) 0,
                   OCI_DURATION_SESSION, OCI_TYPEGET_HEADER, &addrtdo))
  {
    return OCI_ERROR;
  }

                      /* initial bind call for both variables */
  if (OCIBindByName(stmthp, &bndhp[2], errhp,
                      (text *) ":addref", (sb4) strlen((char *) ":addref"),
                      (dvoid *) 0, (sb4) sizeof(OCIRef *), SQLT_REF,
                      (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                      (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
  ||  OCIBindByName(stmthp, &bndhp[3], errhp,
                      (text *) ":zip", (sb4) strlen((char *) ":zip"),
                      (dvoid *) 0, (sb4) MAXZIPLEN, SQLT_CHR,
                      (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                      (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC))
  {
    return OCI_ERROR;
  }
```

```
                                      /* object bind for REF variable */
if (OCIBindObject(bndhp[2], errhp, (OCIType *) addrtdo,
        (dvoid **) &addrref[0], (ub4 *) 0, (dvoid **) 0, (ub4 *) 0))
{
  return OCI_ERROR;
}


for (i = 0; i < MAXCOLS; i++)
  pos[i] = i;
                    /* dynamic binds for both RETURNING variables */
if (OCIBindDynamic(bndhp[2], errhp, (dvoid *) &pos[0], cbf_no_data,
                   (dvoid *) &pos[0], cbf_get_data)
|| OCIBindDynamic(bndhp[3], errhp, (dvoid *) &pos[1], cbf_no_data,
                   (dvoid *) &pos[1], cbf_get_data))
{
  return OCI_ERROR;
}

return OCI_SUCCESS;
}
```

Additional Notes About Callbacks

When a callback function is called, the OCI_ATTR_ROWS_RETURNED attribute of the bind handle tells the application the number of rows being returned in that particular iteration. Thus, when the callback is called the first time in a particular iteration (that is, index=0), you can allocate space for all the rows which will be returned for that bind variable. When the callback is called subsequently (with index>0) within the same iteration, you can merely increment the buffer pointer to the correct memory within the allocated space to retrieve the data.

Array Interface for DML RETURNING Statements

OCI provides additional functionality for single-row DML operations and array DML operations in which each iteration returns more than one row. To take advantage of this feature, the client application must specify an OUT buffer in the bind call which is at least as big as the iteration count specified in the OCIStmtExecute() call. This is in addition to the method by which bind buffers are provided through callbacks.

When the statement executes, if any of the iterations returns more than one row, then the application receives an OCI_SUCCESS_WITH_INFO return code. In this

case, the DML operation is successfully completed. At this point the application may choose to roll back the transaction or ignore the warning.

Character Conversion Issues in Binding and Defining

This section discusses issues involving character conversions between the client and the server.

Choosing Character Set

Oracle provides support for character data in the database, and OCI provides support for binding and defining character data. If a database column containing character data is defined to be an NCHAR/NVARCHAR2 column, then a bind or define involving that column must take into account special considerations for dealing with character set specifications.

These considerations are necessary in case the width of the client character set is different from that on the server, and also for proper character conversion between the client and server. During conversion of data between different character sets, the size of the data may grow or shrink as much as fourfold. Care must be taken to insure that buffers provided to hold the data are of sufficient size.

In some cases, it may also be easier for an application to deal with NCHAR/NVARCHAR2 data in terms of numbers of characters, rather than numbers of bytes (which is the usual case).

Character Set Form and ID

Each OCI bind and define handle has OCI_ATTR_CHARSET_FORM and OCI_ATTR_CHARSET_ID attributes associated with it. An application can set these attributes with the OCIAttrSet() call in order to specify the character form and character set ID of the bind/define buffer.

The *form* attribute (OCI_ATTR_CHARSET_FORM) indicates the character set the client buffer is in, for binds, and the character set in which to store fetched data, for defines. It has two possible values:

- SQLCS_IMPLICIT - indicates database character set ID

- SQLCS_NCHAR - indicates national character set ID

The default value is SQLCS_IMPLICIT, which implies the database character set for the bind or define buffer and the character data in the buffer is converted to the server database character set. SQLCS_NCHAR implies national character set ID for

the bind or define buffer and the client buffer data is converted to the server national character set.

If the character set ID attribute (OCI_ATTR_CHARSET_ID) is not specified, then the default value of the database or national character set ID of the client is used, depending on the value of *form*. They are the values specified in the NLS_LANG and NLS_NCHAR environment variables.

> **Note:**
>
> - No matter what values are assigned to the character set ID and form of the client-side bind buffer, the data is converted and inserted into the database according to the server's database character set ID or national character set ID.
>
> - OCI_ATTR_CHARSET_ID must never be set to zero.

> **See Also:** For more information about NCHAR data, refer to the *Oracle9i Database Reference.*

Implicit Conversion Between CHAR and NCHAR

As the result of implicit conversion between database character sets and national character sets, OCI can support cross binding and cross defining between CHAR and NCHAR. For example, even though the OCI_ATTR_CHARSET_FORM attribute is set to be SQLCS_NCHAR, OCI enables converting data to the database character set if the data is inserted into a CHAR column.

Using OCI_ATTR_MAXDATA_SIZE Attribute

Every bind handle has a OCI_ATTR_MAXDATA_SIZE attribute. This attribute specifies the number of bytes to be allocated on the server to accommodate the client-side bind data after any necessary character set conversions.

> **Note:** Character set conversions performed when data is sent to the server may result in the data expanding or contracting, so its size on the client may not be the same as its size on the server.

An application will typically set OCI_ATTR_MAXDATA_SIZE to the maximum size of the column or the size of the PL/SQL variable, depending on how it is used.

Oracle issues an error if OCI_ATTR_MAXDATA_SIZE is not a large enough value to accommodate the data after conversion, and the operation will fail.

The following scenarios demonstrate some examples of the use of the OCI_ATTR_MAXDATA_SIZE attribute:

- **Scenario 1: CHAR (source data) -> non-CHAR (destination column)**

  In this case there are implicit bind conversions taking place on the data. The recommended value of OCI_ATTR_MAXDATA_SIZE in this case would be the size of the source buffer multiplied by the worst-case expansion between the client and server character sets.

- **Scenario 2: CHAR (source data) -> CHAR (destination column) or non-`CHAR` (source data) -> `CHAR` (destination column)**

  In either of these cases, the recommended value of OCI_ATTR_MAXDATA_SIZE is the size of the column.

- **Scenario 3: CHAR (source data) -> PL/SQL variable**

  In this case, the recommended value of OCI_ATTR_MAXDATA_SIZE is the size of the PL/SQL variable.

Using OCI_ATTR_MAXCHAR_SIZE Attribute

Bind and define handles have an attribute, OCI_ATTR_MAXCHAR_SIZE, associated with them. An application can use this attribute to work with data in terms of number of characters, rather than number of bytes.

For binds, the OCI_ATTR_MAXCHAR_SIZE attribute sets the number of characters that an application reserves on the server to store the data being bound. This works together with the OCI_ATTR_MAXDATA_SIZE attribute, and the nonzero minimum of their nderived byte length is used.

For example, if OCI_ATTR_MAXDATA_SIZE is set to 100, and OCI_ATTR_MAXCHAR_SIZE is set to 0, then the maximum possible size of the data on the server after conversion is 100 bytes. However, if OCI_ATTR_MAXDATA_SIZE is set to 300, and OCI_ATTR_MAXCHAR_SIZE is set to a nonzero value, such as 100, then if the character set has 2 bytes/character, the maximum possible allocated size is 200 bytes, which is the minimum of 300 bytes and 2*100 bytes.

For defines, the OCI_ATTR_MAXCHAR_SIZE attribute specifies the maximum number of characters that the client application allows in the return buffer. Its

derived byte length overrides the `maxlength` parameter specified in the `OCIDefineByPos()` call.

> **Note:** Regardless of the value of the attribute OCI_ATTR_MAXCHAR_SIZE, the buffer lengths specified in a bind or define call are always considered to be in terms of number of bytes. The actual length values sent and received by you are also in bytes in this case.

Buffer Expansion During Binding

Update or insert operations are done through variable binding. When binding variables, specify OCI_ATTR_MAXCHAR_SIZE and/or OCI_ATTR_MAXDATA_SIZE in the bind handle to specify character and byte constraints to be used while inserting data in the server.

These attributes are defined as:

- OCI_ATTR_MAXCHAR_SIZE sets the maximum number of allowed characters in the buffer on the server side.

- OCI_ATTR_MAXDATA_SIZE sets the maximum number of allowed bytes in the buffer on the server side.

If neither of these two attributes is set, OCI expands the buffer using its best estimates.

In Binds

If the underlying column was created using character length semantics, then it is preferable to specify the constraint using OCI_ATTR_MAXCHAR_SIZE. In this case, as long as the actual buffer contains less characters that specified in OCI_ATTR_MAXCHAR_SIZE, no constraints are violated at OCI level.

If the underlying column was created using byte length semantics, then use OCI_ATTR_MAXDATA_SIZE in the bind handle to specify the byte constraint on the server. If you also specify an OCI_ATTR_MAXCHAR_SIZE value, then this constraint is additionally imposed when allocating the receiving buffer on the server side.

Dynamic SQL

For dynamic SQL, you can use the explicit describe to get OCI_ATTR_DATA_SIZE and or OCI_ATTR_CHAR_SIZE in parameter handles as a guide for setting

OCI_ATTR_MAXDATA_SIZE and OCI_ATTR_MAXCHAR_SIZE attributes in bind handles. Furthermore, it is always a safer practice to specify OCI_ATTR_MAXDATA_SIZE and OCI_ATTR_MAXCHAR_SIZE to be no more than the actual column width in bytes, and characters.

Buffer Expansion During Inserts

Consider the following scenario where you should avoid unexpected behavior caused by buffer expansion during inserts.

If the database column has character length semantics and the user tries to insert data into that column using `OCIBindByPos()` or `OCIBindByName()` and sets only the OCI_ATTR_MAXCHAR_SIZE to, say 3000 bytes. The database character set is UTF8 and the client character set is ASCII. Then, in this case although 3000 characters will fit in a buffer of size 3000 bytes for the client, on the server side it might expand to more than 4000 bytes. In this case, unless the underlying column is a LONG or a LOB type, the server will return an error. You can get around this problem by specifying, additionally, the OCI_ATTR_MAXDATA_SIZE to be 4000, to guarantee that the data will never exceed 4000 bytes.

Constraint Checking During Defining

To select from data columns into client buffers, OCI uses defined variables. You can set an OCI_ATTR_MAXCHAR_SIZE value on the define buffer to impose an additional character length constraint on the define buffer. There is no OCI_ATTR_MAXDATA_SIZE attribute for define handles since the buffer size in bytes serves as the limit on byte length. So, normally, the define buffer size provided in the `OCIDefineByPos()` call can be used as the byte constraint.

Dynamic SQL Selects

When sizing buffers for dynamic SQL, always use the OCI_ATTR_DATA_SIZE value in the implicit describe to size your buffer to avoid truncation. If the database column is created using character length semantics, which is known through OCI_ATTR_CHAR_USED attribute, then you can use the OCI_ATTR_MAXCHAR_SIZE value to set an additional constraint on the define buffer. In this case no more than the number of OCI_ATTR_MAXCHAR_SIZE characters will be put in the buffer.

Return Lengths

The following length values are always in bytes irrespective of the character length semantics of the database:

- The value returned in the $alen$, or the actual length field in binds and defines.

- The value that appears in the length prefixed in special datatypes like VARCHAR and LONG VARCHAR

- The value of the indicator variable in case of truncation.

The only exception to this rule is for string buffers in OCI_UTF16ID. If you specify the character set ID as OCI_UTF16ID for a bind/define handle, then the above lengths are in UTF-16 units.

Note that the buffer sizes in the bind and define calls and the piece sizes in the OCIGetPieceInfo() and OCISetPieceInfo() and the callbacks are always in bytes.

General Compatibility Issues for Character Length Semantics

- For a release 9.0 or later client talking to an 8.1 or earlier server, OCI_ATTR_MAXCHAR_SIZE is not understood by the server, so this value will be ignored. If you specify only this value, OCI will derive the corresponding OCI_ATTR_MAXDATA_SIZE value based on the maximum bytes per character for the client-side character set.

- For an 8.1 or earlier client talking to a 9.0 or later server, the client will never be able to specify an OCI_ATTR_MAXCHAR_SIZE value, so the server will consider the client always expecting byte length semantics. This is similar to the situation when the client specifies only OCI_ATTR_MAXDATA_SIZE.

So in both cases, the server and client can exchange information in an appropriate manner.

Code Examples for Binding and Defining with Character Conversion

The following two examples illustrate concepts discussed above.

Code Example for Inserting and Selecting Using OCI_ATTR_MAXCHAR_SIZE

When a column is created by specifying a number N of characters, the actual allocation in the data base will consider the worst scenario in the following table. The real bytes allocated will be a multiple of N, say M times N. Currently, M is three as the maximum bytes per character in UTF8.

For example, in the following table EMP, ENAME column is defined as 30 characters and ADDRESS is defined as 80 characters. Then the corresponding byte lengths in database are M*30 or 3*30=90, and M*80 or 3*80=240 respectively.

```
...
utext ename[31], address[81];
/* E' <= 30+ 1, D' <= 80+ 1, considering null-termination */
sb2 ename_max_chars = EC=20, address_max_chars = ED=60;
 /* EC <= (E' - 1), ED <= (D' - 1) */
sb2 ename_max_bytes = EB=80, address_max_bytes = DB=200;
 /* EB <= M * EC, DB <= M * DC */
text *insstmt = (text *)"INSERT INTO EMP(ENAME, ADDRESS) VALUES (:ENAME,
:ADDRESS)";
text *selstmt = (text *)"SELECT ENAME, ADDRESS FROM EMP";
...
/* Inserting Column Data */
OCIStmtPrepare(stmthp1, errhp, insstmt, (ub4)strlen((char *)insstmt),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCIBindByName(stmthp1, &bndlp, errhp, (text *)":ENAME",
    (sb4)strlen((char *)":ENAME",
    (dvoid *)ename, sizeof(ename), SQLT_STR, (dvoid *)&insname_ind,
    (ub2 *)alenp, (ub2 *)rcodep, (ub4)maxarr_len, (ub4 *)curelep, OCI_DEFAULT);
/* either */
OCIAttrSet((dvoid *)bnd1p, (ub4)OCI_HTYPE_BIND, (dvoid *)&ename_max_bytes,
    (ub4)0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
/* or */
OCIAttrSet((dvoid *)bnd1p, (ub4)OCI_HTYPE_BIND, (dvoid *)&ename_max_chars,
    (ub4)0, (ub4)OCI_ATTR_MAXCHAR_SIZE, errhp);
...
/* Retrieving Column Data */
OCIStmtPrepare(stmthp2, errhp, selstmt, strlen((char *)selstmt),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCIDefineByPos(stmthp2, &dfn1p, errhp, (ub4)1, (dvoid *)ename,
   (sb4)sizeof (ename),
   SQLT_STR, (dvoid *)&selname_ind, (ub2 *)alenp, (ub2 *)rcodep,
   (ub4)OCI_DEFAULT);
/* if not called, byte semantics is by default */
OCIAttrSet((dvoid *)bnd1p, (ub4)OCI_HTYPE_DEFINE, (dvoid *)&ename_max_chars,
   (ub4)0,
   (ub4)OCI_ATTR_MAXCHAR_SIZE, errhp);
...
```

Code Example for UTF-16 Binding and Defining

The character set ID in bind and define of the CHAR/VARCHAR2 or
NCHAR/NVARCHAR variant handles can be set to specify that all data passed by
the corresponding bind and define calls is assumed to be in UTF-16 (Unicode)
encoding. To specify UTF-16, set OCI_ATTR_CHARSET_ID = OCI_UTF16ID.

See Also: For more information, see the bind attribute "OCI_ATTR_CHARSET_ID" on page A-32 and the define attribute "OCI_ATTR_CHARSET_ID" on page A-35

OCI provides a typedef called **utext** to facilitate binding and defining of UTF-16 data. The internal representation of **utext** is a 16-bit unsigned integer (**ub2**). Platforms where the encoding scheme of the **wchar_t** datatype conforms to UTF-16 (unsigned 16-bit value) can easily convert **utext** to the **wchar_t** datatype using cast operators.

Even for UTF-16 data, the buffer size in bind and define calls is assumed to be in bytes. Users should use the new **utext** datatype as the buffer for input/output data.

The following pseudocode illustrates a bind and define for UTF-16 data:

```
...
OCIStmt  *stmthp1, *stmthp2;
OCIDefine *dfn1p, *dfnp2;
OCIBind *bnd1p, *bnd2p;
text *insstmt=
      (text *) "INSERT INTO EMP(ENAME, ADDRESS) VALUES (:ename, :address)";
text *selname =
      (text *) "SELECT ENAME, ADDRESS FROM EMP";
utext ename[21];   /* Name -    UTF-16 */
utext address[51]; /* Address - UTF-16 */
ub2 csid = OCI_UTF16ID;
sb2 ename_col_len = 20;
sb2 address_col_len = 50;
...
/* Inserting UTF-16 data */
OCIStmtPrepare (stmthp1, errhp, insstmt, (ub4)strlen ((char *)insstmt),
                (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));
OCIBindByName (stmthp1, &bnd1p, errhp, (text*)":ENAME",
               (sb4)strlen((char *)":ENAME"),
               (dvoid *) ename, sizeof(ename), SQLT_STR,
               (dvoid *)&insname_ind, (ub2 *) 0, (ub2 *) 0, (ub4) 0,
               (ub4 *)0, OCI_DEFAULT);
OCIAttrSet ((dvoid *) bnd1p, (ub4) OCI_HTYPE_BIND, (dvoid *) &csid,
            (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((dvoid *) bnd1p, (ub4) OCI_HTYPE_BIND, (dvoid *) &ename_col_len,
           (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving UTF-16 data */
OCIStmtPrepare (stmthp2, errhp, selname, strlen((char *) selname),
                (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
```

```
OCIDefineByPos (stmthp2, &dfn1p, errhp, (ub4)1, (dvoid *)ename,
               (sb4)sizeof(ename), SQLT_STR,
               (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAttrSet ((dvoid *) dfn1p, (ub4) OCI_HTYPE_DEFINE, (dvoid *) &csid,
            (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
...
```

PL/SQL REF CURSORs and Nested Tables

The OCI provides the ability to bind and define PL/SQL REF CURSORs and nested tables. An application can use a statement handle to bind and define these types of variables. As an example, consider this PL/SQL block:

```
static const text *plsql_block = (text *)
    "begin \
        OPEN :cursor1 FOR SELECT empno, ename, job, mgr, sal, deptno \
               FROM emp_rc WHERE job=:job ORDER BY empno; \
        OPEN :cursor2 FOR SELECT * FROM dept_rc ORDER BY deptno; \
    end;";
```

An application would allocate a statement handle for binding, by calling `OCIHandleAlloc()`, and then bind the `:cursor1` placeholder to the statement handle, as in the following code, where `:cursor1` is bound to stm2p. Note that the handle allocation code is not included here.

```
err = OCIStmtPrepare (stm1p, errhp, (text *) nst_tab, strlen(nst_tab),
                        OCI_NTV_SYNTAX, OCI_DEFAULT);
...
err = OCIBindByName (stm1p, (OCIBind **) bndp, errhp,
               (text *)":cursor1", (sb4)strlen((char *)":cursor1"),
               (dvoid *)&stm2p, (sb4) 0,  SQLT_RSET, (dvoid *)0,
                 (ub2 *)0, (ub2 *)0, (ub4)0, (ub4 *)0,   (ub4)OCI_DEFAULT);
```

In this code, stm1p is the statement handle for the PL/SQL block, while stm2p is the statement handle which is bound as a REF CURSOR for later data retrieval. A value of SQLT_RSET is passed for the `dty` parameter.

As another example, consider the following:

```
static const text *nst_tab = (text *)
    "SELECT ename, CURSOR(SELECT dname, loc FROM  dept_rc) \
        FROM emp_rc WHERE ename = 'LOCKE'";
```

In this case the second position is a nested table, which an OCI application can define as a statement handle as follows. Note that the handle allocation code is not included here.

```
err = OCIStmtPrepare (stm1p, errhp, (text *) nst_tab, strlen(nst_tab),
                      OCI_NTV_SYNTAX, OCI_DEFAULT);
...
err = OCIDefineByPos (stm1p, (OCIDefine **) dfn2p, errhp, (ub4)2,
                      (dvoid *)&stm2p,
                      (sb4)0, SQLT_RSET, (dvoid *)0, (ub2 *)0,
                      (ub2 *)0, (ub4)OCI_DEFAULT);
```

After execution, when you fetch a row into `stm2p` it then becomes a valid statement handle.

Note: If you have retrieved multiple REF CURSORs, you must take care when fetching them into `stm2p`. If you fetch the first one, you can then perform fetches on it to retrieve its data. However, once you fetch the second REF CURSOR into `stm2p`, you no longer have access to the data from the first REF CURSOR.

Runtime Data Allocation and Piecewise Operations

You can use the OCI to perform piecewise inserts and updates, and fetches of data. You can also use the OCI to provide data dynamically in the case of array inserts or updates, instead of providing a static array of bind values. You can insert or retrieve a very large column as a series of chunks of smaller size, minimizing client-side memory requirements.

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.

The piecewise functionality of OCI can be particularly useful when you are performing operations on extremely large blocks of string or binary data (for example, operations involving database columns that store `CLOB`, `BLOB`, `LONG`, `RAW`, or `LONG RAW` data).

Valid Datatypes for Piecewise Operations

Only some datatypes can be manipulated in pieces. OCI applications can perform piecewise fetches, inserts, or updates of all the following datatypes:

- `VARCHAR2`

- STRING

- LONG

- LONG RAW

- RAW

- CLOB

- BLOB

Some LOB/FILE operations also provide piecewise semantics for reading or writing data.

Another way of using this feature for *all* datatypes is to provide data dynamically for array inserts or updates. Note, however, that the callbacks should always specify OCI_ONE_PIECE for the `piecep` parameter of the callback for datatypes that do not support piecewise operations.

> **See Also:** See the descriptions of OCILobWrite() on page 16-61 and OCILobRead() on page 16-54 for more information about these operations. For information about streaming using callbacks with OCILobWrite() and OCILobRead(), see "LOB Read and Write Callbacks" on page 7-14.

Binding and Defining for LOBs

The following functions accept SQLT_CHR (VARCHAR2), SQLT_LNG (LONG), and SQLT_CLOB (CLOB) for CLOB columns. They also accept SQLT_LBI (LONG RAW), and SQLT_BIN (RAW) and SQLT_BLOB (BLOB) datatypes for BLOB columns:

- OCIDefineByPos()

- OCIBindByName()

- OCIBindByPos()

For example, if the datatype is specified as SQLT_CHR for a CLOB column, then OCIStmtFetch() will fetch the CLOB data. However, if the datatype is specified as SQLT_CLOB for a CLOB column, then OCIStmtFetch() works as before release 9.0.1. It will return the CLOB locator and you can call OCILobRead() to read the CLOB data.

Types of Piecewise Operations

Figure 5–4, "Piecewise Insert of a Column" shows a single column being inserted piecewise into a database table through a series of insert operations (i1, i2, i3...in). In this example the inserted pieces are of varying sizes.

Figure 5–4   Piecewise Insert of a Column
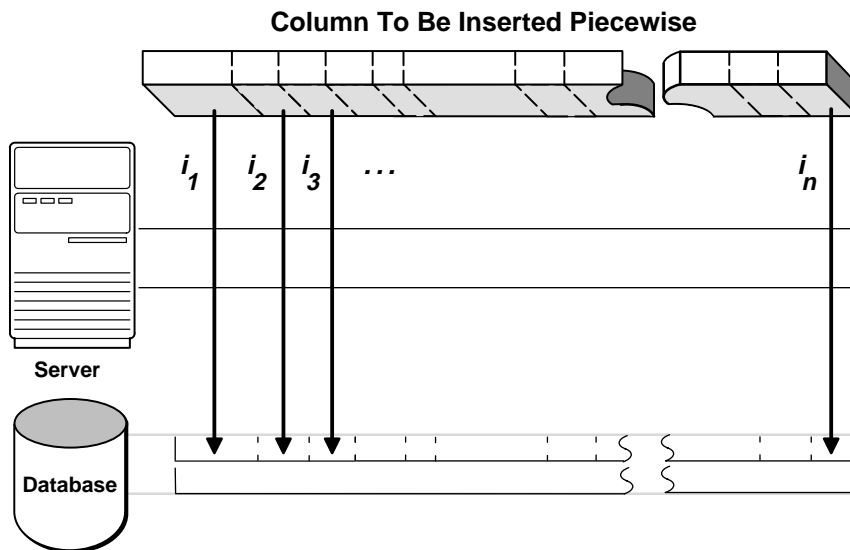
You can perform piecewise operations in two ways:

- Use calls provided in the OCI library to execute piecewise operations under a polling paradigm, as in release 7.3.

- Employ user-defined callback functions to provide the necessary information and data blocks.

When you set the `mode` parameter of an `OCIBindByPos()` or `OCIBindByName()` call to OCI_DATA_AT_EXEC, this indicates that an OCI application will be providing data for an `INSERT` or `UPDATE` dynamically at runtime.

Similarly, when you set the `mode` parameter of an `OCIDefineByPos()` call to OCI_DYNAMIC_FETCH, this indicates that an application will dynamically provide allocation space for receiving data at the time of the fetch.

In each case, you can provide the run-time information for the INSERT, UPDATE, or FETCH in one of two ways: through callback functions, or by using piecewise operations. If callbacks are desired, an additional bind or define call is necessary to register the callbacks.

The following sections give specific information about run-time data allocation and piecewise operations for inserts, updates, and fetches.

> **Note:** In addition to SQL statements, piecewise operations are also valid for PL/SQL blocks.

Providing INSERT or UPDATE Data at Runtime

When you specify the OCI_DATA_AT_EXEC mode in a call to OCIBindByPos() or OCIBindByName(), the *value_sz* parameter defines the total size of the data that can be provided at runtime. The application must be ready to provide to the OCI library the run-time IN data buffers on demand as many times as is necessary to complete the operation. When the allocated buffers are not required any more, they should be freed by the client.

Runtime data is provided in one of the two ways:

- You can define a callback using the OCIBindDynamic() function which when called at runtime returns a piece or the whole data.

- If no callbacks are defined, the call to OCIStmtExecute() to process the SQL statement returns the OCI_NEED_DATA error code. The client application then provides the IN/OUT data buffer or piece using the OCIStmtSetPieceInfo() call. OCIStmtGetPieceInfo() provides information about which bind and which piece are being used.

Performing a Piecewise Insert or Update

Once the OCI environment has been initialized, and a database connection and session have been established, a piecewise insert begins with calls to prepare a SQL or PL/SQL statement and to bind input values. Piecewise operations using standard OCI calls, rather than user-defined callbacks, do not require a call to OCIBindDynamic().

> **Note:** Additional bind variables in the statement that are not part of piecewise operations may require additional bind calls, depending on their datatypes.

Following the statement preparation and bind, the application performs a series of calls to `OCIStmtExecute()`, `OCIStmtGetPieceInfo()` and `OCIStmtSetPieceInfo()` to complete the piecewise operation. Each call to `OCIStmtExecute()` returns a value that determines 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.

Keep in mind that the insert buffer can be of arbitrary size and is provided 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 `OCIStmtSetPieceInfo()` call.

> **Note:** 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 that preceded it. For example, if a data value 10,050,036 bytes long is inserted in chunks of 500 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 `OCIStmtSetPieceInfo()` call.

The following steps outline the procedure involved in performing a piecewise insert or update. The procedure is illustrated in Figure 5–5, "Steps for Performing Piecewise Insert" on page 5-50.

Step 1. Initialize the OCI environment, allocate the necessary handles, connect to a server, authorize a user, and prepare a statement request. These steps are described in the section "OCI Programming Steps" on page 2-20.

Step 2. Bind a placeholder using `OCIBindByName()` or `OCIBindByPos()`. At this point you do not need to specify the actual size of the pieces you will use, but you must provide the total size of the data that can be provided at runtime.

7.x Upgrade Note: The context pointer that was formerly part of the `obindps()` and `ogetpi()` routines does not exist in release 8.0 and later. Clients wishing to provide their own context can use the callback method.

Step 3. Call `OCIStmtExecute()` for the first time. At this point no data is actually inserted, and the OCI_NEED_DATA error code is returned to the application.

If any other value is returned, it indicates that an error occurred.

Step 4. Call `OCIStmtGetPieceInfo()` to retrieve information about the piece that needs to be inserted. The parameters of `OCIStmtGetPieceInfo()` include a pointer that returns a value indicating whether the required piece is the first piece (OCI_FIRST_PIECE) or a subsequent piece (OCI_NEXT_PIECE).

Step 5. The application populates a buffer with the piece of data to be inserted and calls `OCIStmtSetPieceInfo()`. The parameters passed to `OCIStmtSetPieceInfo()` 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 `OCIStmtExecute()` again. If OCI_LAST_PIECE was indicated in Step 5 and `OCIStmtExecute()` returns OCI_SUCCESS, all pieces were inserted successfully. If `OCIStmtExecute()` returns OCI_NEED_DATA, go back to Step 3 for the next insert. If `OCIStmtExecute()` returns any other value, an error occurred.

The piecewise operation is complete when the final piece has been successfully inserted. This is indicated by the OCI_SUCCESS return value from the final `OCIStmtExecute()` call.

Figure 5–5   Steps for Performing Piecewise Insert
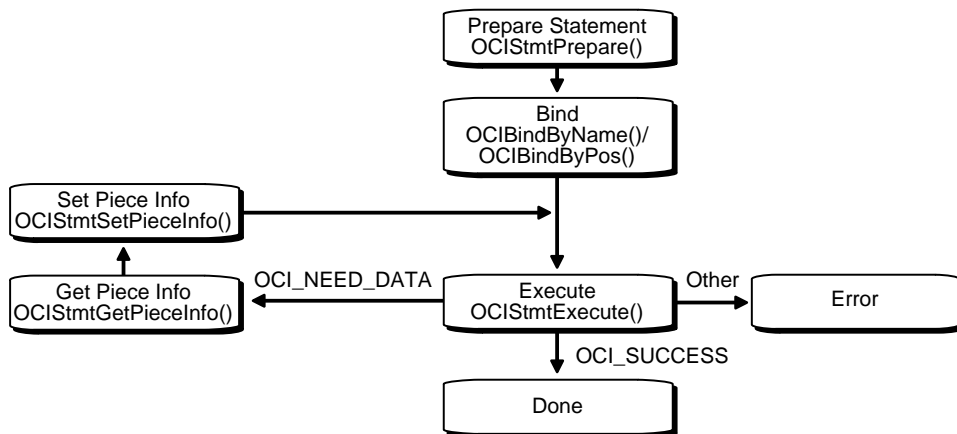
Piecewise updates are performed in a similar manner. In a piecewise update operation the insert buffer is populated with the data that is being updated, and OCIStmtExecute() is called to execute the update.

> **Note:** For additional important information about piecewise operations, see the section "Additional Information About Piecewise Operations with No Callbacks" on page 5-53

Piecewise Operations With PL/SQL

An OCI application can perform piecewise operations with PL/SQL for IN, OUT, and IN/OUT bind variables in a method similar to that outlined above. Keep in mind that all placeholders in PL/SQL statements are bound, rather than defined. The call to OCIBindDynamic() specifies the appropriate callbacks for OUT or IN/OUT parameters.

Providing FETCH Information at Runtime

When a call is made to OCIDefineByPos() with the *mode* parameter set to OCI_DYNAMIC_FETCH, an application can specify information about the data buffer at the time of fetch. You also may need to call OCIDefineDynamic() to set up the callback function that will be invoked to get information about your data buffer.

Run-time data is provided in one of the two ways:

- You can define a callback using the `OCIDefineDynamic()` call. The *value_sz* parameter defines the maximum size of the data that will be provided at runtime. When the client library needs a buffer to return the fetched data, the callback will be invoked to provide a run-time buffer into which a piece or the whole data will be returned.

- If no callbacks are defined, the OCI_NEED_DATA error code is returned and the OUT data buffer or piece can then be provided by the client application using `OCIStmtSetPieceInfo()` call. The `OCIStmtGetPieceInfo()` call provides Information about which define and which piece are involved.

  See Also: For information about which datatypes are valid for piecewise operations, refer to the section "Valid Datatypes for Piecewise Operations" on page 5-44.

Performing a Piecewise Fetch

Once the OCI environment has been initialized, and a database connection and session have been established, a piecewise fetch begins with calls to prepare a SQL or PL/SQL statement and to define output variables. Piecewise operations using standard OCI calls, rather than user-defined callbacks, do not require a call to `OCIDefineDynamic()`.

Following the statement preparation and define, the application performs a series of calls to `OCIStmtFetch()`, `OCIStmtGetPieceInfo()`, and `OCIStmtSetPieceInfo()` to complete the piecewise operation. Each call to `OCIStmtFetch()` returns a value that determines 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 that piece into a buffer. When the last piece has been fetched, the operation is complete.

Keep in mind that the fetch buffer can be of arbitrary size. In addition, each fetched piece does not need to be of the same size. The only requirement is that the size of the final fetch must be exactly the size of the last remaining piece. The size of each piece to be fetched is established by each `OCIStmtSetPieceInfo()` call.

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

> **See Also:** This process is illustrated in Figure 5–6, "Steps for Performing Piecewise Fetch".

Step 1. Initialize the OCI environment, allocate necessary handles, connect to a database, authorize a user, prepare a statement, and execute the statement. These steps are described in "OCI Programming Steps" on page 2-20.

Step 2. Define an output variable using `OCIDefineByPos()`, with *mode* set to OCI_DYNAMIC_FETCH. At this point you do not need to specify the actual size of the pieces you will use, but you must provide the total size of the data that will be fetched at runtime.

7.x Upgrade Note: The context pointer that was part of the `odefinps()` and `ogetpi()` routines does not exist in release 8.x or later. Clients wishing to provide their own context can use the callback method.

Step 3. Call `OCIStmtFetch()` for the first time. At this point no data is actually retrieved, and the OCI_NEED_DATA error code is returned to the application.

If any other value is returned, an error occurred.

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

Step 5. Call `OCIStmtSetPieceInfo()` to specify the buffer into which you wish to fetch the piece.

Step 6. Call `OCIStmtFetch()` again to retrieve the actual piece. If `OCIStmtFetch()` returns OCI_SUCCESS, all the pieces have been fetched successfully. If `OCIStmtFetch()` returns OCI_NEED_DATA, return to Step 4 to process the next piece. If any other value is returned, an error occurred.

The piecewise fetch is complete when the final `OCIStmtFetch()` call returns a value of OCI_SUCCESS.

Figure 5–6   Steps for Performing Piecewise Fetch
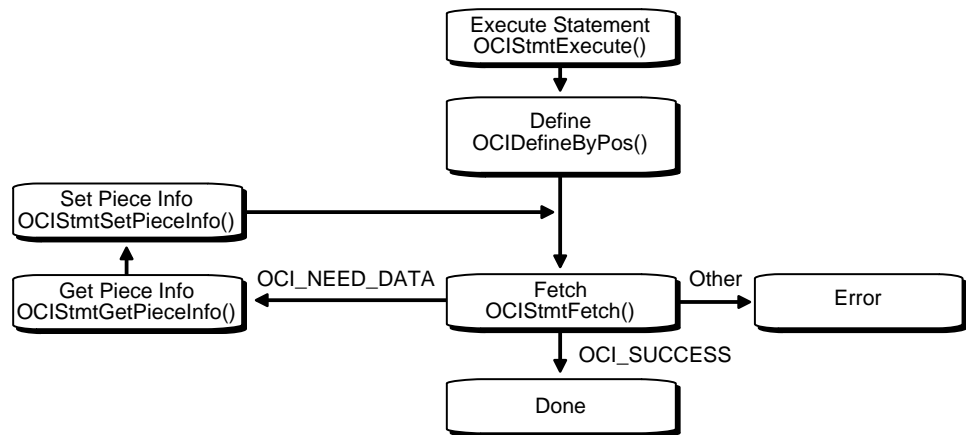
Additional Information About Piecewise Operations with No Callbacks

In both the piecewise fetch and insert, it is important to understand the sequence of calls necessary for the operation to complete successfully. In particular, keep in mind that for a piecewise insert you must call `OCIStmtExecute()` one time more than the number of pieces to be inserted (if callbacks are not used). This is because the first time `OCIStmtExecute()` is called, it merely returns a value indicating that the first piece to be inserted is required. As a result, if you are inserting *n* pieces, you must call `OCIStmtExecute()` a total of *n+1* times.

Similarly, when performing a piecewise fetch, you must call `OCIStmtFetch()` once more than the number of pieces to be fetched.

Users who are binding to PL/SQL index-by tables can retrieve a pointer to the current index of the table during the `OCIStmtGetPieceInfo()` calls.

6

Describing Schema Metadata

This chapter discusses the use of the `OCIDescribeAny()` function to obtain information about schema elements. The following topics are covered in this chapter:

- Describing Schema Metadata
- Using OCIDescribeAny()
- Examples Using OCIDescribeAny()

Describing Schema Metadata

This chapter discusses the use of the OCIDescribeAny() function to describe schema objects.

See Also:

- For information about describing select-list items, refer to the section "Describing Select-List Items" on page 4-12

- For additional information about the OCIDescribeAny() call and its parameters, refer to the function description on page 15-78

Using OCIDescribeAny()

The OCIDescribeAny() function enables you to perform an explicit describe of one of the following schema objects, and their subschema objects:

- tables and views

- synonyms

- procedures

- functions

- packages

- sequences

- collections

- types

- schemas

- databases

Information about other schema elements (procedure/function arguments, columns, type attributes, and type methods) is available through a describe of one of the above schema objects or an explicit describe of the subschema object.

When an application describes a table, it can then retrieve information about that table's columns. Additionally, OCIDescribeAny() can directly describe subschema objects such as columns of a table, packages of a function, or fields of a type if the user knows the name of the subschema object.

The OCIDescribeAny() call requires a describe handle as one of its parameters. The describe handle must have been previously allocated with a call to OCIHandleAlloc(). After the call to OCIDescribeAny(), an application can retrieve information about the described object from the describe handle.

The information returned by OCIDescribeAny() is organized hierarchically like a tree. For example, Figure 6–1 shows how the description of a certain table might be organized.

Figure 6–1 OCIDescribeAny() Table Description
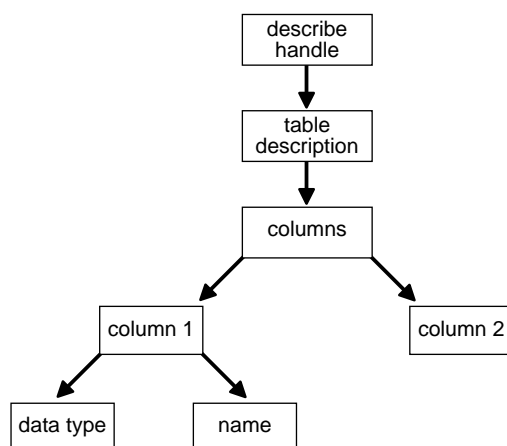
The describe handle returned by OCIDescribeAny() has an attribute, OCI_ATTR_PARAM, that points to such a description tree. Each node of the tree has attributes associated with the node and attributes (which are like recursive describe handles) that point to subtrees containing more information. If all the attributes are homogenous, as in case of elements of a list, such as a column list, then we refer to them as *parameters*. In this chapter, the terms *handle* and *parameter* are used interchangeably. The attributes associated with any node are returned by OCIAttrGet(), and the parameters are returned by OCIParamGet().

For example, an OCIAttrGet() on the describe handle for the table can return a handle to the column-list information. An application can then use OCIParamGet() to retrieve the handle to the column description of a particular column in the column-list. The handle to the column descriptor can be passed to OCIAttrGet() to get further information about the column, such as the name and data type (as illustrated by following the left-hand side of the above figure).

No subsequent `OCIAttrGet()` or `OCIParamGet()` call requires extra round trips, as all the description is cached on the client side by `OCIDescribeAny()`.

Restrictions

The `OCIDescribeAny()` call limits information returned to the basic information and stops expanding a node if it amounts to another describe. For example, if a table column is of an object type, then the OCI does not return a subtree describing the type since this information can be obtained by another describe.

Notes on Types and Attributes

When performing describe operations, you should be aware of the following notes.

Note on Datatype Codes

> **See Also:** For more information about typecodes, such as the OCI_TYPCODE values returned in the OCI_ATTR_TYPECODE attribute and the SQLT typecodes returned in the OCI_ATTR_DATA_TYPE attribute, refer to the section "Typecodes" on page 3-31

OCI_ATTR_TYPECODE returns typecodes which represent the types supplied by the user when a new type is created using the CREATE TYPE statement. These typecodes are of the enumerated type **OCITypeCode**, and are represented by OCI_TYPECODE constants. Internal PL/SQL types (boolean, indexed table) are not supported.

OCI_ATTR_DATA_TYPE returns typecodes which represent the datatypes stored in database columns. These are similar to the describe values returned by previous versions of Oracle. These values are represented by SQLT constants (**ub2** values). BOOLEAN types return SQLT_BOL.

Note on Describing Types

In order to describe type objects, it is necessary to initialize the OCI process in object mode:

```
/* Initialize the OCI Process */
  if (OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,
                    (dvoid * (*)(dvoid *, size_t)) 0,
                    (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                    (void (*)(dvoid *, dvoid *)) 0 ))
```

```
{ (void) printf("FAILED: OCIInitialize()\n");
  return OCI_ERROR; }
```

> **See Also:** For more information on this function, refer to the description of `OCIInitialize()` on page 15-14

Note on Implicit and Explicit Describes

The column attribute OCI_ATTR_PRECISION can be returned using an implicit describe with `OCIStmtExecute()` and an explicit describe with `OCIDescribeAny()`. When using an implicit describe, the precision should be set to **sb2**. When using an explicit describe, the precision should be set to ub1 for a placeholder. This is necessary to match the datatype of precision in the dictionary.

Note on **OCI_ATTR_LIST_ARGUMENTS**

The OCI_ATTR_LIST_ARGUMENTS attribute for type methods represents *second-level* arguments for the method.

For example, given the following record `my_type` and the procedure `my_proc` which takes an argument of type `my_type`:

```
my_type record(a number, b char)
my_proc (my_input my_type)
```

the OCI_ATTR_LIST_ARGUMENTS attribute would apply to arguments `a` and `b` of the `my_type` record.

Parameter Attributes

A parameter is returned by `OCIParamGet()`. Parameters can describe different types of objects or information. Parameters have attributes depending on the type of description they contain and these are the type-specific attributes. This section describes the attributes and handles that belong to different parameters.

The following table lists the attributes that belong to all parameters:

Table 6–1 Attributes Belonging to All Parameters

Attribute	Description	Attribute Datatype
OCI_ATTR_NUM_PARAMS	The number of parameters	ub2
OCI_ATTR_OBJ_ID	Object or schema Id	ub4
OCI_ATTR_OBJ_NAME	Database name or object name in a schema	text*

Table 6–1 Attributes Belonging to All Parameters (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJ_SCHEMA	Schema name where the object is located	text*
OCI_ATTR_PTYPE	Type of information described by the parameter. Possible values are:	ub1
	OCI_PTYPE_TABLE - table	
	OCI_PTYPE_VIEW - view	
	OCI_PTYPE_PROC - procedure	
	OCI_PTYPE_FUNC - function	
	OCI_PTYPE_PKG - package	
	OCI_PTYPE_TYPE - type	
	OCI_PTYPE_TYPE_ATTR - attribute of a type	
	OCI_PTYPE_TYPE_COLL - collection type information	
	OCI_PTYPE_TYPE_METHOD - a method of a type	
	OCI_PTYPE_SYN - synonym	
	OCI_PTYPE_SEQ - sequence	
	OCI_PTYPE_COL - column of a table or view	
	OCI_PTYPE_ARG - argument of a function or procedure	
	OCI_PTYPE_TYPE_ARG - argument of a type method	
	OCI_PTYPE_TYPE_RESULT - the results of a method	
	OCI_PTYPE_LIST - column list for tables and views, argument list for functions and procedures, or subprogram list for packages.	
	OCI_PTYPE_SCHEMA - schema	
	OCI_PTYPE_DATABASE- database	
OCI_ATTR_TIMESTAMP	The timestamp of the object this description is based on (in Oracle date format)	ub1 *

The subsections that follow list the attributes and handles specific to different types of parameters.

Types OCI_PTYPE_TABLE or OCI_PTYPE_VIEW

When a parameter is for a table or view (type OCI_PTYPE_TABLE or OCI_PTYPE_VIEW), it has the following type specific attributes:

Table 6–2   Attributes Belonging to Tables or Views

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJID	Object id	ub4
OCI_ATTR_NUM_COLS	Number of columns	ub2
OCI_ATTR_LIST_COLUMNS	Column list (type OCI_PTYPE_LIST)	dvoid *
OCI_ATTR_REF_TDO	REF to the TDO of the base type in case of extent tables	OCIRef*
OCI_ATTR_IS_TEMPORARY	Is the table temporary?	ub1
OCI_ATTR_IS_TYPED	Is the table typed?	ub1
OCI_ATTR_DURATION	Duration of a temporary table. Values can be: OCI_DURATION_SESSION - session OCI_DURATION_TRANS - transaction OCI_DURATION_NULL -table not temporary	OCIDuration

The following are additional attributes which belong to tables:

Table 6–3   Attributes Specific to Tables

Attribute	Description	Attribute Datatype
OCI_ATTR_DBA	Data block address of the segment header	ub4
OCI_ATTR_TABLESPACE	Tablespace the table resides in	word
OCI_ATTR_CLUSTERED	Is the table clustered?	ub1
OCI_ATTR_PARTITIONED	Is the table partitioned?	ub1
OCI_ATTR_INDEX_ONLY	Is the table index-only?	ub1

Procedure/Function/Subprogram Attributes

When a parameter is for a procedure or function (type OCI_PTYPE_PROC or OCI_PTYPE_FUNC), it has the following type specific attributes:

Table 6–4   Attribute Belonging to Procedures or Functions

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_ARGUMENTS	Argument list. See "List Attributes" on page 6-18.	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	Is the procedure or function invoker-rights?	ub1

The following attributes are defined only for package subprograms:

Table 6–5   Attributes Specific to Package Subprograms

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Name of the procedure or function	text *
OCI_ATTR_OVERLOAD_ID	Overloading ID number (relevant in case the procedure or function is part of a package and is overloaded). Values returned may be different from direct query of a PL/SQL function or procedure.	ub2

Package Attributes

When a parameter is for a package (type OCI_PTYPE_PKG), it has the following type specific attributes:

Table 6–6   Attributes Belonging to Packages

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_SUBPROGRAMS	Subprogram list. See "List Attributes" on page 6-18.	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	Is the package invoker-rights?	ub1

Type Attributes

When a parameter is for a type (type OCI_PTYPE_TYPE), it has the attributes listed in Table 6–7. These attributes are only valid if the application initialized the OCI process in OCI_OBJECT mode in a call to OCIInitialize().

Table 6–7 Attributes Belonging to Types

Attribute	Description	Attribute Datatype
OCI_ATTR_REF_TDO	Returns the in-memory REF of the type descriptor object for the type, if the column type is an object type. If space has not been reserved for the OCIRef, then it is allocated implicitly in the cache. The caller can then pin the TDO with `OCIObjectPin()`.	OCIRef *
OCI_ATTR_TYPECODE	Typecode. See "Note on Datatype Codes" on page 6-4. Currently can be only OCI_TYPECODE_OBJECT or OCI_TYPECODE_NAMEDCOLLECTION.	OCITypeCode
OCI_ATTR_COLLECTION_TYPECODE	Typecode of collection if type is collection; invalid otherwise. See "Note on Datatype Codes" on page 6-4. Currently can be only OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE. Error is returned if this attribute is queried for non-collection type.	OCITypeCode
OCI_ATTR_IS_INCOMPLETE_TYPE	Is this an incomplete type?	ub1
OCI_ATTR_IS_SYSTEM_TYPE	Is this a system type?	ub1
OCI_ATTR_IS_PREDEFINED_TYPE	Is this a predefined type?	ub1
OCI_ATTR_IS_TRANSIENT_TYPE	Is this a transient type?	ub1
OCI_ATTR_IS_SYSTEM_ GENERATED_TYPE	Is this a system-generated type?	ub1
OCI_ATTR_HAS_NESTED_TABLE	Does this type contain a nested table attribute?	ub1
OCI_ATTR_HAS_LOB	Does this type contain a LOB attribute?	ub1
OCI_ATTR_HAS_FILE	Does this type contain a FILE attribute?	ub1
OCI_ATTR_COLLECTION_ELEMENT	Handle to collection element. See "Collection Attributes" on page 6-12.	dvoid *
OCI_ATTR_NUM_TYPE_ATTRS	Number of type attributes	ub2
OCI_ATTR_LIST_TYPE_ATTRS	List of type attributes. See "List Attributes" on page 6-18.	dvoid *
OCI_ATTR_NUM_TYPE_METHODS	Number of type methods	ub2

Table 6–7   Attributes Belonging to Types (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_TYPE_METHODS	List of type methods. See "List Attributes" on page 6-18.	dvoid *
OCI_ATTR_MAP_METHOD	Map method of type. See "Type Method Attributes" on page 6-11.	dvoid *
OCI_ATTR_ORDER_METHOD	Order method of type. See "Type Method Attributes" on page 6-11.	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	Is the type invoker-rights?	ub1
OCI_ATTR_NAME	A pointer to a string which is the type attribute name	text *
OCI_ATTR_SCHEMA_NAME	A string with the schema name under which the type has been created	text *
OCI_ATTR_IS_FINAL_TYPE	Is this a final type?	ub1
OCI_ATTR_IS_INSTANTIABLE_TYPE	Is this an instantiable type?	ub1
OCI_ATTR_IS_SUBTYPE	Is this a subtype?	ub1
OCI_ATTR_SUPERTYPE_SCHEMA_NAME	Name of the schema containing the supertype	text *
OCI_ATTR_SUPERTYPE_NAME	Name of the supertype	text *

Type Attribute Attributes

When a parameter is for an attribute of a type (type OCI_PTYPE_TYPE_ATTR), it has the attributes listed in Table 6–8.

Table 6–8   Attributes Belonging to Type Attributes

Attribute	Description	Attribute Datatype
OCI_ATTR_DATA_SIZE	The maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERs.	ub4
OCI_ATTR_TYPECODE	Typecode. See "Note on Datatype Codes" on page 6-4.	OCITypeCode
OCI_ATTR_DATA_TYPE	The data type of the type attribute. See "Note on Datatype Codes" on page 6-4.	ub2
OCI_ATTR_NAME	A pointer to a string which is the type attribute name	text *

Table 6–8   Attributes Belonging to Type Attributes (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_PRECISION	The precision of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	ub1
OCI_ATTR_SCALE	The scale of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	sb1
OCI_ATTR_TYPE_NAME	A string which is the type name. The returned value will contain the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, the name of the named data type's type is returned. If the data type is SQLT_REF, the type name of the named data type pointed to by the REF is returned	text *
OCI_ATTR_SCHEMA_NAME	A string with the schema name under which the type has been created	text *
OCI_ATTR_REF_TDO	Returns the in-memory REF of the TDO for the type, if the column type is an object type. If space has not been reserved for the OCIRef, then it is allocated implicitly in the cache. The caller can then pin the TDO with OCIObjectPin().	OCIRef *
OCI_ATTR_CHARSET_ID	The character set id, if the type attribute is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	The character set form, if the type attribute is of a string/character type	ub1
OCI_ATTR_FSPRECISION	The fractional seconds precision of a datetime or interval.	ub1
OCI_ATTR_LFPRECISION	The leading field precision of an interval.	ub1

Type Method Attributes

When a parameter is for a method of a type (type OCI_PTYPE_TYPE_METHOD), it has the attributes listed in Table 6–9.

Table 6–9   Attributes Belonging to Type Methods

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Name of method (procedure or function)	text *
OCI_ATTR_ENCAPSULATION	Encapsulation level of the method (either OCI_TYPEENCAP_PRIVATE or OCI_TYPEENCAP_PUBLIC)	OCITypeEncap
OCI_ATTR_LIST_ARGUMENTS	Argument list. See "Note on OCI_ATTR_LIST_ARGUMENTS" on page 6-5, and "List Attributes" on page 6-18.	dvoid *
OCI_ATTR_IS_CONSTRUCTOR	Is method a constructor?	ub1
OCI_ATTR_IS_DESTRUCTOR	Is method a destructor?	ub1
OCI_ATTR_IS_OPERATOR	Is method an operator?	ub1
OCI_ATTR_IS_SELFISH	Is method selfish?	ub1
OCI_ATTR_IS_MAP	Is method a map method?	ub1
OCI_ATTR_IS_ORDER	Is method an order method?	ub1
OCI_ATTR_IS_RNDS	Is "Read No Data State" set for method?	ub1
OCI_ATTR_IS_RNPS	Is "Read No Process State" set for method?	ub1
OCI_ATTR_IS_WNDS	Is "Write No Data State" set for method?	ub1
OCI_ATTR_IS_WNPS	Is "Write No Process State" set for method?	ub1
OCI_ATTR_IS_FINAL_METHOD	Is this a final method?	ub1
OCI_ATTR_IS_INSTANTIABLE_METHOD	Is this an instantiable method?	ub1
OCI_ATTR_IS_OVERRIDING_METHOD	Is this an overriding method?	ub1

Collection Attributes

When a parameter is for a collection type (type OCI_PTYPE_COLL), it has the attributes listed in Table 6–10.

Table 6–10   Attributes Belonging to Collection Types

Attribute	Description	Attribute Datatype
OCI_ATTR_DATA_SIZE	The maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERs.	ub2
OCI_ATTR_TYPECODE	Typecode. See "Note on Datatype Codes" on page 6-4.	OCITypeCode
OCI_ATTR_DATA_TYPE	The data type of the type attribute. See "Note on Datatype Codes" on page 6-4.	ub2
OCI_ATTR_NUM_ELEMENTS	The number of elements in an array. It is only valid for collections that are arrays	ub4
OCI_ATTR_NAME	A pointer to a string which is the type attribute name	text *
OCI_ATTR_PRECISION	The precision of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	ub1
OCI_ATTR_SCALE	The scale of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	sb1
OCI_ATTR_TYPE_NAME	A string which is the type name. The returned value will contain the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, the name of the named data type's type is returned. If the data type is SQLT_REF, the type name of the named data type pointed to by the REF is returned	text *
OCI_ATTR_SCHEMA_NAME	A string with the schema name under which the type has been created	text *

Table 6–10   Attributes Belonging to Collection Types (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_REF_TDO	Returns the in-memory REF of the TDO for the type, if the column type is an object type. If space has not been reserved for the OCIRef, then it is allocated implicitly in the cache. The caller can then pin the TDO with `OCIObjectPin()`.	OCIRef *
OCI_ATTR_CHARSET_ID	The character set id, if the type attribute is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	The character set form, if the type attribute is of a string/character type	ub1

Synonym Attributes

When a parameter is for a synonym (type OCI_PTYPE_SYN), it has the attributes listed in Table 6–11.

Table 6–11   Attributes Belonging to Synonyms

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJID	Object id	ub4
OCI_ATTR_SCHEMA_NAME	A null-terminated string containing the schema name of the synonym translation	text *
OCI_ATTR_NAME	A null-terminated string containing the object name of the synonym translation	text *
OCI_ATTR_LINK	A null-terminated string containing the database link name of the synonym translation	text *

Sequence Attributes

When a parameter is for a sequence (type OCI_PTYPE_SEQ), it has the attributes listed in Table 6–12.

Table 6–12   Attributes Belonging to Sequences

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJID	Object id	ub4
OCI_ATTR_MIN	Minimum value (in Oracle number format)	ub1 *
OCI_ATTR_MAX	Maximum value (in Oracle number format)	ub1 *

Table 6–12   Attributes Belonging to Sequences (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_INCR	Increment (in Oracle number format)	ub1 *
OCI_ATTR_CACHE	Number of sequence numbers cached; zero if the sequence is not a cached sequence (in Oracle number format)	ub1 *
OCI_ATTR_ORDER	Whether the sequence is ordered	ub1
OCI_ATTR_HW_MARK	High-water mark (in Oracle number format)	ub1 *

Column Attributes

When a parameter is for a column of a table or view (type OCI_PTYPE_COL), it has the attributes listed in Table 6–13.

Table 6–13   Attributes Belonging to Columns of Tables or Views

Attribute	Description	Attribute Datatype
OCI_ATTR_CHAR_USED	Returns the type of length semantics of the column. 0 means byte-length semantics and 1 means character-length semantics. See "Character Length Semantics Support in Describing" on page 6-21 .	ub4
OCI_ATTR_CHAR_SIZE	Returns the column character length which is the number of characters allowed in the column. It is the counterpart of OCI_ATTR_DATA_SIZE which gets the byte length. See "Character Length Semantics Support in Describing" on page 6-21 .	ub1
OCI_ATTR_DATA_SIZE	The maximum size of the column. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERs.	ub2
OCI_ATTR_DATA_TYPE	The data type of the column. See "Note on Datatype Codes" on page 6-4.	ub2
OCI_ATTR_NAME	A pointer to a string which is the column name	text *
OCI_ATTR_PRECISION	The precision of numeric columns. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	ub1 for explicit describe sb2 for implicit describe

Table 6–13   Attributes Belonging to Columns of Tables or Views (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_SCALE	The scale of numeric columns. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	sb1
OCI_ATTR_IS_NULL	Returns 0 if null values are not permitted for the column	ub1
OCI_ATTR_TYPE_NAME	Returns a string which is the type name. The returned value will contain the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, the name of the named data type's type is returned. If the data type is SQLT_REF, the type name of the named data type pointed to by the REF is returned	text *
OCI_ATTR_SCHEMA_NAME	Returns a string with the schema name under which the type has been created	text *
OCI_ATTR_REF_TDO	The REF of the TDO for the type, if the column type is an object type	OCIRef *
OCI_ATTR_CHARSET_ID	The character set id, if the column is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	The character set form, if the column is of a string/character type	ub1

Argument/Result Attributes

When a parameter is for an argument of a procedure/function (type OCI_PTYPE_ARG), for a type method argument (type OCI_PTYPE_TYPE_ARG) or for method results (type OCI_PTYPE_TYPE_RESULT), it has the attributes listed in Table 6–14.

Table 6–14   Attributes Belonging to Arguments/Results

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Returns a pointer to a string which is the argument name	text *
OCI_ATTR_POSITION	The position of the argument in the argument list. Always returns zero.	ub2
OCI_ATTR_TYPECODE	Typecode. See "Note on Datatype Codes" on page 6-4.	OCITypeCode
OCI_ATTR_DATA_TYPE	The data type of the argument. See "Note on Datatype Codes" on page 6-4.	ub2
OCI_ATTR_DATA_SIZE	The size of the data type of the argument. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERs.	ub2
OCI_ATTR_PRECISION	The precision of numeric arguments. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	ub1
OCI_ATTR_SCALE	The scale of numeric arguments. If the precision is nonzero and scale is -127, then it is a FLOAT, else it is a NUMBER(precision, scale). For the case when precision is 0, NUMBER(precision, scale) can be represented simply as NUMBER.	sb1
OCI_ATTR_LEVEL	The data type levels. This attribute always returns zero.	ub2
OCI_ATTR_HAS_DEFAULT	Indicates whether an argument has a default	ub1
OCI_ATTR_LIST_ARGUMENTS	The list of arguments at the next level (when the argument is of a record or table type).	dvoid *
OCI_ATTR_IOMODE	Indicates the argument mode: 0 is IN (OCI_TYPEPARAM_IN), 1 is OUT (OCI_TYPEPARAM_OUT), 2 is IN/OUT (OCI_TYPEPARAM_INOUT)	OCITypeParamMode
OCI_ATTR_RADIX	Returns a radix (if number type)	ub1

Table 6–14   Attributes Belonging to Arguments/Results (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_IS_NULL	Returns 0 if null values are not permitted for the column	ub1
OCI_ATTR_TYPE_NAME	Returns a string which is the type name, or the package name in the case of package local types. The returned value will contain the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, the name of the named data type's type is returned. If the data type is SQLT_REF, the type name of the named datatype pointed to by the REF is returned.	text *
OCI_ATTR_SCHEMA_NAME	For SQLT_NTY or SQLT_REF, returns a string with the schema name under which the type was created, or under which the package was created in the case of package local types	text *
OCI_ATTR_SUB_NAME	For SQLT_NTY or SQLT_REF, returns a string with the type name, in the case of package local types	text *
OCI_ATTR_LINK	For SQLT_NTY or SQLT_REF, returns a string with the database link name of the database on which the type exists. This can happen only in the case of package local types, when the package is remote.	text *
OCI_ATTR_REF_TDO	Returns the REF of the TDO for the type, if the argument type is an object	OCIRef *
OCI_ATTR_CHARSET_ID	Returns the character set ID if the argument is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	Returns the character set form if the argument is of a string/character type	ub1

List Attributes

When a parameter is for a list of columns, arguments, or subprograms (type OCI_PTYPE_LIST), it has the following type specific attributes and handles (parameters):

The list has an OCI_ATTR_LIST_TYPE attribute which designates the list type. The possible values and their lower bounds when traversing the list are:

Table 6–15   List Attributes

List Attribute	Description	Lower Bound
OCI_LTYPE_COLUMN	Column list	1
OCI_LTYPE_ARG_PROC	Procedure argument list	1
OCI_LTYPE_ARG_FUNC	Function argument list	0
OCI_LTYPE_SUBPRG	Subprogram list	0
OCI_LTYPE_TYPE_ATTR	Type attribute list	1
OCI_LTYPE_TYPE_METHOD	Type method list	1
OCI_LTYPE_TYPE_ARG_PROC	Type method without result argument list	0
OCI_LTYPE_TYPE_ARG_FUNC	Type method without result argument list	1
OCI_LTYPE_SCH_OBJ	Object list within a schema	0
OCI_LTYPE_DB_SCH	Schema list within a database	0

- The list has an OCI_ATTR_NUM_PARAMS attribute, which tells the number of elements in the list.

- Each list has LowerBound .. OCI_ATTR_NUM_PARAMS parameters. LowerBound is the value in the Lower Bound column of Table 6–15, "List Attributes". In the case of a function argument list, position 0 has a parameter for the return value (type OCI_PTYPE_ARG).

Schema Attributes

When a parameter is for a schema type (type OCI_PTYPE_SCHEMA), it has the attributes listed in Table 6–16:

Table 6–16   Attributes Specific to Schemas

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_OBJECTS	List of objects in the schema	text*

Database Attributes

When a parameter is for a database type (type OCI_PTYPE_DATABASE), it has the attributes listed in Table 6–17:

Table 6–17   Attributes Specific to Databases

Attribute	Description	Attribute Datatype
OCI_ATTR_VERSION	Database version	text*
OCI_ATTR_CHARSET_ID	Database character set Id from the server handle	ub2
OCI_ATTR_NCHARSET_ID	Database character set Id from the server handle	ub2
OCI_ATTR_LIST_SCHEMAS	List of schemas (type OCI_PTYPE_SCHEMA) in the database	OCI_PTYPE_LIST
OCI_ATTR_MAX_PROC_LEN	Maximum length of a procedure name	ub4
OCI_ATTR_MAX_COLUMN_LEN	Maximum length of a column name	ub4
OCI_ATTR_CURSOR_COMMIT_BEHAVIOR	How a COMMIT operation affects cursors and prepared statements in the database. Values are: OCI_CURSOR_OPEN - preserve cursor state as before the commit operation OCI_CURSOR_CLOSED - cursors are closed on COMMIT, but the application can still reexecute the statement without re-preparing it	ub1
OCI_ATTR_MAX_CATALOG_NAMELEN	Maximum length of a catalog (database) name	ub1
OCI_ATTR_CATALOG_LOCATION	Position of the catalog in a qualified table. Values are OCI_CL_START and OCI_CL_END	ub1
OCI_ATTR_SAVEPOINT_SUPPORT	Does database support savepoints? Values are OCI_SP_SUPPORTED and OCI_SP_UNSUPPORTED	ub1
OCI_ATTR_NOWAIT_SUPPORT	Does database support the nowait clause? Values are OCI_NW_SUPPORTED and OCI_NW_UNSUPPORTED	ub1
OCI_ATTR_AUTOCOMMIT_DDL	Is autocommit mode required for DDL statements? Values are OCI_AC_DDL and OCI_NO_AC_DDL	ub1
OCI_ATTR_LOCKING_MODE	Locking mode for the database. Values are OCI_LOCK_IMMEDIATE and OCI_LOCK_DELAYED	ub1

Character Length Semantics Support in Describing

Starting with release Oracle9*i*, query and column information are supported with character length semantics.

These attributes of describe handles support character length semantics:

OCI_ATTR_CHAR_SIZE, gets the column character length which is the number of characters allowed in the column. It is the counterpart of OCI_ATTR_DATA_SIZE which gets the byte length;

OCI_ATTR_CHAR_USED, gets the type of length semantics of the column. 0 means byte-length semantics and 1 means character length semantics.

An application may describe a select-list (query) either implicitly or explicitly through `OCIStmtExecute()`. Other schema elements must be described explicitly through `OCIDescribeAny()`.

Implicit Describing

If the database column was created using character length semantics, then the implicit describe information will contain both the character length and the byte length and a flag saying how the database column was created. That is, OCI_ATTR_CHAR_SIZE is the character length of the column or expression. The OCI_ATTR_CHAR_USED flag is a 1 in this case, indicating that the column or expression was created with the character length semantics.

The OCI_ATTR_DATA_SIZE value will be always large enough to hold all the data, as many as OCI_ATTR_CHAR_SIZE number of characters. The OCI_ATTR_DATA_SIZE will be usually set to the OCI_ATTR_CHAR_SIZE times the client's max bytes per character value.

If the database column was created with byte length semantics, then the implicit describe will behave exactly as it does before release 9.0. That is, the OCI_ATTR_DATA_SIZE value returned will be the column's byte length times the maximum conversion ratio between the client and server's character set, that is, column byte length divided by the server's max bytes per character multiplied by the client's max bytes per character. The OCI_ATTR_CHAR_USED value is 0 and the OCI_ATTR_CHAR_SIZE value will be set to the same value as OCI_ATTR_DATA_SIZE.

Explicit Describing

Explicit describes of tables will have an attribute OCI_ATTR_DATA_SIZE that gets the column's size in bytes (as it appears in the server), the length in characters in

OCI_ATTR_CHAR_SIZE, and a flag OCI_ATTR_CHAR_USED that indicates how the column was created.

When inserting, if the OCI_ATTR_CHAR_USED flag is set you may set the OCI_ATTR_MAXCHAR_SIZE (to be explained in binds) in the bind handle to the value returned by OCI_ATTR_CHAR_SIZE in the parameter handle from the explicit describe. This will assure you that you never violate the size constraint for the column.

> **See Also:** *"In Binds" on page 5-38*

Another way to do explicit describe is through function OCIDescribeAny(), which is a generic describe call for existing schema objects: tables, views, synonyms, procedures, functions, packages, sequences, types, schemas, and databases. This call also describes subschema objects, such as a column in a table. This call populates the describe handle with the object-specific attributes which can be obtained through an OCIAttrGet() call.

Calling OCIAttrGet() on the parameter descriptor returns the specific attributes of a stored procedure or function parameter, or a table column descriptor. These subsequent calls do not need an extra round trip to the server because the entire schema object description is cached on the client side by OCIDescribeAny().

When an application describes a table, it can then retrieve information about that table's columns. Additionally, OCIDescribeAny() can directly describe subschema objects such as columns of a table, packages of a function, or fields of a type if the user knows the name of the subschema object. In all cases, the specific information about columns and datatypes is retrieved by reading handle attributes.

After a SQL statement is executed, information about the select-list is available as an attribute of the statement handle. No explicit describe call is needed. To retrieve information about select-list items from the statement handle, the application must call OCIParamGet() once for each position in the select-list to allocate a parameter descriptor for that position.

Once a parameter descriptor has been allocated for a position in the select-list, the application can retrieve specific information by calling OCIAttrGet() on the parameter descriptor. Information available from the parameter descriptor includes the datatype and maximum size of the parameter.

Client and Server Compatibility Issues for Describing

When an Oracle9*i* or later client talks to an Oracle8*i* or earlier server, it will behave as if the database is only using byte length semantics;

When an Oracle8*i* or earlier client talks to a Oracle9*i* or later server, the attributes
OCI_ATTR_CHAR_SIZE and OCI_ATTR_CHAR_USED are not available on the
client side.

In either case, the character length semantics cannot be described when either the
server or client has an Oracle8*i* or earlier software release.

Examples Using OCIDescribeAny()

The following examples demonstrate the use of OCIDescribeAny() for describing
different types of schema objects. For a more detailed code sample, see the
demonstration program cdemodsa.c included with your Oracle installation.

> **See Also:** For additional information on the demonstration
> programs, see Appendix B, "OCI Demonstration Programs"

Retrieving Column Data Types For a Table

This example illustrates the use of an explicit describe. Let us take an example
application, which needs to retrieve the column datatypes for a table. The following
code fragment shows how an application would be able to use the describe
interface:

```
...
text objptr[] = "tablename";
OCIParam       *mypard;
ub4            counter;
ub2            dtype;
text           *col_name;
ub4            counter, col_name_len, char_semantics, col_width;
ub4 objp_len = strlen("tablename");
OCIParam *parmh;         /* parameter handle */
OCIParam *collsthd;      /* handle to list of columns */
OCIParam *colhd;         /* column handle */
OCIDescribe *dschp;      /* describe handle */

...
/* get the describe handle for the table */
if (OCIDescribeAny(svch, errh, objptr, objp_len, OCI_OTYPE_NAME, 0,
    OCI_PTYPE_TABLE, dschp))
   return error;
/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
```

```
        return error;
    /* The type information of the object, in this case, OCI_PTYPE_TABLE,
    is obtained from the parameter descriptor returned by the OCIAttrGet(). */
    /* get the number of columns in the table */
    if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &numcols, 0,
         OCI_ATTR_NUM_COLS, errh))
         return error;
    /* get the handle to the column list of the table */
    if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &collsthd, 0,
         OCI_ATTR_LIST_COLUMNS, errh)==OCI_NO_DATA)
       return error;
    /* go through the column list and retrieve the data-type of each column,
    and then recursively describe column types. */

    for (i = 1; i <= numcols; i++)
    {
        /* get parameter for column i */
        if (OCIParamGet(collsthd, OCI_DTYPE_PARAM, errh, &colhd, i))
            return error;
        /* for example, get data type for ith column */
        if (OCIAttrGet(colhd, OCI_DTYPE_PARAM, &datatype[i-1], 0,
           OCI_ATTR_DATA_TYPE, errh))
            return error;
    /* Retrieve the length semantics for the column */
    OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                  (dvoid*) &char_semantics,(ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
                  (OCIError *) errhp  );
    if (char_semantics)
        /* Retrieve the column width in characters */
        OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                      (dvoid*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
                      (OCIError *) errhp  );
    else
        /* Retrieve the column width in bytes */
        OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                      (dvoid*) &col_width,(ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
                      (OCIError *) errhp  );
    }
    ...
```

Describing the Stored Procedure

Let us consider a stored procedure or a function.The difference between a
procedure and a function is that the latter has a return type at position 0 in the

argument list, while the former has no argument associated with position 0 in the argument list. The steps required to describe type methods (also divided into functions and procedures) are identical to that of regular PL/SQL functions and procedures. Note that procedures and functions can take default types of objects as arguments. Let us consider the following procedure:

```
P1 (arg1 emp.sal%type, arg2 emp%rowtype)
```

Furthermore, let us assume that each row in emp table has two columns: name(VARCHAR2(20)), and sal(NUMBER). Thus, in the argument list for P1, we have two arguments, *arg1* and *arg2*, at positions 1 and 2 respectively at level 0, and arguments name and sal at positions 1and 2 respectively at level 1. Description of P1 returns the number of arguments as two while returning the higher level (> 0) arguments as attributes of the 0 zero level arguments.

The following code fragment explains the description of P1.

```
...
text objptr[] = "P1";       /* procedure name */
ub4 objp_len = strlen("P1");
OCIParam *parmh;            /* parameter handle */
OCIParam *arglst;          /* list of args */
OCIParam *arg;             /* argument handle */
OCIDescribe *dschp;     /* describe handle */

ub2 numargs, pos, level;
text *name;
ub4 namelen;

...
/* get the describe handle for the table */
if (OCIDescribeAny(svch, errh, objptr, objp_len, OCI_OTYPE_NAME, 0,
    OCI_PTYPE_PROC, dschp))
    return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
    return error;

/* Get the number of arguments and the arg list */
if (OCIAttrGet (parmh, OCI_DTYPE_PARAM, &arglst,
0, OCI_ATTR_LIST_ARGUMENTS, errh))
    return error;
if (OCIAttrGet (parmh, OCI_DTYPE_PARAM, &numargs, 0,
```

```
        OCI_ATTR_NUM_PARAMS, errh))
        return error;

/* For a procedure, we begin with i = 1; for a
function, we begin with i = 0. */

for (i = 1; i < numargs; i++) {
OCIParamGet (arglst, OCI_DTYPE_PARAM, errh, &arg, i);
OCIAttrGet (arg, OCI_DTYPE_PARAM, &name, &namelen, OCI_ATTR_NAME,
        errh);
...
/* to print the attributes of the argument of type record
(arguments at the next level), traverse the argument list */

OCIAttrGet (arg, OCI_DTYPE_PARAM, &arglst1, 0,
OCI_ATTR_LIST_ARGUMENTS, erh);

/* check if the current argument is a record. For arg1 in P1
arglst1 is NULL. */

  if (arglst1) {
    OCIAttrGet (arg, OCI_DTYPE_PARAM, &numargs1,0, OCI_ATTR_NUM_PARAMS,
        errh);

/* Note that for both functions and procedures,the next higher level
arguments start from index 1. For arg2 in P1, the number of arguments at
the level 1 would be 2 */

    for (i = 1; i < numargs1, i++) {
      OCIParamGet (arglst1, OCI_DTYPE_PARAM, errh, &arg1, i);
      OCIAttrGet (arg1, OCI_DTYPE_PARAM, &name1, &namelen1,
        OCI_ATTR_NAME, errh);
    ...
    }
  }
}
...
```

Retrieving Attributes of an Object Type

This example illustrates the use of an explicit describe on a named object type. We illustrate how you can describe an object by its name or by its object reference (**OCIRef**). The following code fragment attempts to retrieve the datatype value of each of the object type's attributes.

See Also: It is very similar to the first example in the section "Retrieving Column Data Types For a Table" on page 6-23

```
...
text type_name[] = "typename";
ub4 type_name_len = strlen("typename");
OCIRef *type_ref = ...;
un4 numattrs;
OCIDescribe *dschp;      /* describe handle */
OCIParam *parmh;         /* parameter handle */
OCIParam *attrlsthd;     /* handle to list of attrs */
OCIParam *attrhd;        /* attribute handle */
...
/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                   (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
    return error;

/* get the describe handle for the type */
if (describe_by_name)
  if (OCIDescribeAny(svch, errh, (dvoid*)type_name, type_name_len,
        OCI_OTYPE_NAME, 0, OCI_PTYPE_TYPE, dschp))
        return error;
else
  if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF,
        0, OCI_PTYPE_TYPE, dschp))
          return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
      return error;

/* The type information of the object, in this case, OCI_PTYPE_TYPE, is
obtained from the parameter descriptor returned by the OCIAttrGet */

/* get the number of attributes in the type */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &numattrs, 0,
    OCI_ATTR_NUM_TYPE_ATTRS, errh))
      return error;

/* get the handle to the attribute list of the type */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, (dvoid *)&attrlsthd, 0,
      OCI_ATTR_LIST_TYPE_ATTRS, errh)==OCI_NO_DATA)
```

```
  return error;

/* go through the attribute list and retrieve the data-type of each attribute,
and then recursively describe attribute types. */

for (i = 1; i <= numattrs; i++)
{
/* get parameter for attribute i */
if (OCIParamGet(attrlsthd, OCI_DTYPE_PARAM, errh, &attrhd, i))
      return error;

/* for example, get data type and typecode for attribute; note that
OCI_ATTR_DATA_TYPE returns the SQLT code, while OCI_ATTR_TYPECODE returns the
Oracle Type System typecode. */
if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,&datatype[i-1], 0,
     OCI_ATTR_DATA_TYPE,errh))
    return error;
/* for example, get data type for attribute*/
if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,&typecode[i-1], 0,
     OCI_ATTR_TYPECODE, errh))
    return error;

/* if attribute is an object type, recursively describe it */
if (typecode[i-1] == OCI_TYPECODE_OBJECT)
{
OCIRef *attr_type_ref;
OCIDescribe *nested_dschp;

/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh,(dvoid**)&dschp,
(ub4)OCI_HTYPE_DESCRIBE,(size_t)0, (dvoid **)0))
return error;

if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,
        &attr_type_ref, 0, OCI_ATTR_REF_TDO,errh))
    return error;
    OCIDescribeAny(svch, errh,(dvoid*)attr_type_ref, 0,
        OCI_OTYPE_REF, 0, OCI_PTYPE_TYPE, nested_dschp);
/* go on describing the type... */
   }
}
```

Retrieving the Collection Element's Data Type of a Named Collection Type

This example illustrates the use of an explicit describe on a named collection type. We illustrate how you can describe an object by its name or by its object reference (**OCIRef**). The following code fragment attempts to retrieve the data type value of each of the object type's attributes.

> **See Also:** It is very similar to the first example in section "Retrieving Column Data Types For a Table" on page 6-23

```
...
text type_name[] = "typename";
ub4 type_name_len = strlen("typename");
OCIRef *type_ref = ...;
un4 numattrs;
OCIDescribe *dschp;      /* describe handle */
OCIParam *parmh;         /* parameter handle */
OCIParam *attrlsthd;     /* handle to list of attrs */
OCIParam *attrhd;        /* attribute handle */
...
/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                   (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
  return error;

/* get the describe handle for the type */
if (describe_by_name)
  if (OCIDescribeAny(svch, errh, (dvoid*)type_name, type_name_len,
      OCI_OTYPE_NAME, 0, OCI_PTYPE_TYPE, dschp))
      return error;
else
  if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF, 0,
      OCI_PTYPE_TYPE, dschp))
      return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
    return error;

/* get the Oracle Type System type code of the type to determine that this is a
collection type */
if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,&typecode, 0, OCI_ATTR_TYPECODE,
    errh))
```

```
      return error;

/* if typecode is OCI_TYPECODE_NAMEDCOLLECTION,
  proceed to describe collection element */
if (typecode == OCI_TYPECODE_NAMEDCOLLECTION)
{
   /* get the collection's type: ie, OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE */

if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, (dvoid *)&collection_typecode, 0,
OCI_ATTR_COLLECTION_TYPECODE, errh))
    return error;

/* get the collection element; you MUST use this to further retrieve information
about the collection's element */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &collection_element_parmh, 0,
OCI_ATTR_COLLECTION_ELEMENT, errh))
    return error;

/* get the number of elements if collection is a VARRAY; not valid for nested
tables */
if (collection_typecode == OCI_TYPECODE_VARRAY)
    if OCIAttrGet(collection_element_parmh, OCI_DTYPE_PARAM,
      (dvoid *)&num_elements, 0, OCI_ATTR_NUM_ELEMENTS, errh))
      return error;

/* now use the collection_element parameter handle to retrieve information about
the collection element */
if OCIAttrGet(collection_element_parmh, OCI_DTYPE_PARAM,
    (dvoid *)&element_typecode, 0, OCI_ATTR_TYPECODE, errh))
    return error;

/* do the same to describe additional collection element information; this is
very similar to describing type attributes */
...
```

Describing with Character Length Semantics

The following sample code shows a loop that retrieves the column names and data
types corresponding to a query following query execution. The query was
associated with the statement handle by a prior call to OCIStmtPrepare().

```
...
OCIParam    *mypard;
ub4         counter;
ub2         dtype;
```

```
text          *col_name;
ub4           counter, col_name_len, char_semantics, col_width;
sb4           parm_status;
...
/* Request a parameter descriptor for position 1 in the select-list */
counter = 1;
parm_status = OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
            (ub4) counter);
/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */
while (parm_status == OCI_SUCCESS) {
   /* Retrieve the data type attribute */
   checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
           (dvoid*) &dtype,(ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
           (OCIError *) errhp  ));
   /* Retrieve the column name attribute */
   checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
           (dvoid**) &col_name, (ub4 *) &col_char_len, (ub4) OCI_ATTR_NAME,
           (OCIError *) errhp ));
   /* Retrieve the length semantics for the column */
   checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
           (dvoid*) &char_semantics,(ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
           (OCIError *) errhp  ));
   if (char_semantics)
       /* Retrieve the column width in characters */
       checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
               (dvoid*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
               (OCIError *) errhp  ));
   else
       /* Retrieve the column width in bytes */
       checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
               (dvoid*) &col_width,(ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
               (OCIError *) errhp  ));
   printf("column=%s  datatype=%d, char semantics=%d, width=%d\n\n", col_name,
               dtype, char_semantics, col_width);
   fflush(stdout);
   /* increment counter and get next descriptor, if there is one */
   counter++;
   parm_status = OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
                 (ub4) counter);
   }
...
```

7

LOB and FILE Operations

The following topics are covered in this chapter:

- OCI and LOBs
- Creating and Modifying Internal LOBs
- Creating and Modifying Internal LOBs
- Associating a FILE in a Table with an OS File
- LOB Attributes of an Object
- Array Interface For LOBs
- LOB and FILE Functions
- LOB Read and Write Callbacks
- Temporary LOB Support

OCI and LOBs

The Oracle OCI includes a set of functions for performing operations on large objects (LOBs) in a database. Internal LOBs (BLOBs, CLOBs, NCLOBs) are stored in the database tablespaces in a way that optimizes space and provides efficient access. These LOBs have the full transactional support of the database server. External LOBs (FILEs) are large data objects stored in the server's operating system files outside the database tablespaces.

The OCI also provides support for temporary LOBs, which can be used like local variables for operating on LOB data.

The maximum length of a LOB/FILE is 4 gigabytes. FILE functionality is read-only. Oracle currently supports only binary files (BFILEs).

> **See Also:**
>
> - For code samples showing the use of LOB operations, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".
>
> - Customers who are interested in using the dbms_lob package to work with LOBs should refer to *Oracle9i Supplied PL/SQL Packages and Types Reference.* For general information about LOBs and the LOB interfaces available, see the *Oracle9i Application Developer's Guide - Large Objects (LOBs).*
>
> - For information about temporary LOBs, refer to "Temporary LOB Support" on page 7-18.

Creating and Modifying Internal LOBs

You create a new internal LOB by initializing a new LOB locator using OCIDescriptorAlloc(), calling OCIAttrSet() to set it to empty (using the OCI_ATTR_LOBEMPTY attribute), and then binding the locator to a placeholder in an INSERT statement. Doing so inserts the empty locator into a table with a LOB column or attribute. You can then SELECT...FOR UPDATE this row to get the locator, and then write to it using one of the OCI LOB functions.

> **Note:** Whenever you want to modify a LOB column or attribute (write, copy, trim, and so forth), you must lock the row containing the LOB. One way to do this is to use a SELECT...FOR UPDATE statement to select the locator before performing the operation.

For any LOB write command to be successful, a transaction must be open. This means that if you commit a transaction before writing the data, then you must lock the row again (by reissuing the SELECT...FOR UPDATE, for example), because the commit closes the transaction.

For information on creating internal LOBs using EMPTY_BLOB() and EMPTY_CLOB() instead of OCIAttrSet() see the second document listed next:

> **See Also:** ■ For information about binding LOB locators to placeholders, and using them in INSERT statements, refer to the section "Binding LOBs" on page 5-11
>
> ■ For information about LOB reads and writes from within a trigger, see *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

Associating a FILE in a Table with an OS File

The BFILENAME() function can be used in an INSERT statement to associate an external server-side (OS) file with a BFILE column/attribute in a table. Using BFILENAME() in an UPDATE statement associates the BFILE column or attribute with a different OS file. OCILobFileSetName() can also be used to associate a FILE in a table with an OS file. BFILENAME() is usually used in an INSERT or UPDATE without bind variables and OCILobFileSetName() is used for bind variables.

> **See Also:** For more information, see OCILobFileSetName() on page 16-33. For more information about the BFILENAME() function, please refer to the *Oracle9i Application Developer's Guide - Large Objects (LOBs).*

LOB Attributes of an Object

An OCI application can use OCIObjectNew() to create a persistent or transient object with a LOB attribute.

Writing to a LOB Attribute of an Object

It is possible to use the OCI to create a new persistent object with a LOB attribute and write to that LOB attribute. The application would follow these steps:

1. Call `OCIObjectNew()` to create a persistent object with a LOB attribute.

2. Mark the object as dirty.

3. Flush the object, thereby inserting a row into the table

4. Repin the latest version of the object (or refresh the object), thereby retrieving the object from the database and acquiring a valid locator for the LOB

5. Call `OCILobWrite()` using the LOB locator in the object to write the data.

> **See Also:** For more information about object operations, such as marking, flushing, and refreshing, refer to Chapter 10, "OCI Object-Relational Programming" and the following chapters

Transient Objects with LOB Attributes

An application can call `OCIObjectNew()` and create a transient object with an internal LOB (`BLOB`, `CLOB`, `NCLOB`) attribute. However, you cannot perform any operations (e.g., read or write) on the LOB attribute because transient LOBs are not currently supported. Calling `OCIObjectNew()` to create a transient internal LOB type will not fail, but the application cannot use any LOB operations with the transient LOB.

An application can, however, create a transient object with a FILE attribute and use the FILE attribute to read data from the file stored in the server's file system. The application can also call `OCIObjectNew()` to create a transient FILE and use that FILE to read from the server's file.

Array Interface For LOBs

It is possible to use the OCI's array interface with LOBs, just as with any other datatype. Note, however, that you must do the following to allocate the descriptors:

```
/* First create an array of OCILocator pointers: */
OCILobLocator *lobp[10];

for (i=0; i < 10; i++)
{ OCIDescriptorAlloc (...,&lobp[i],...);

/* Then bind the descriptor as follows */
```

```
    OCIBindByPos(... &lobp[i], ...);
}
```

LOB and FILE Functions

The functions in Table 7–1 are available to operate on LOBs and FILEs.

> **See Also:**   More detailed information about each function is found
> in "LOB Functions" on page 16-4. These LOB/FILE calls are not
> valid when an application is connected to an Oracle release 7
> server.

In all LOB operations that involve offsets into the data, the offset begins at 1. For
LOB operations, such as `OCILobCopy()`, `OCILobErase()`,
`OCILobLoadFromFile()`, and `OCILobTrim()`, the *amount* parameter is in
characters for `CLOB`s and `NCLOB`s, regardless of the client-side character set. These
LOB operations refer to the amount of LOB data on the server. When the client-side
character set is varying width, the following general rules apply to the *amount* and
offset parameters in LOB calls:

- *amount*- When the amount parameter refers to the server-side LOB, the amount
  is in characters. When the amount parameter refers to the client-side buffer, the
  amount is in bytes.

- *offset* - Regardless of whether the client-side character set is varying-width,
  the offset parameter is always in characters for `CLOB`s/`NCLOB`s and in bytes for
  `BLOB`s/`BFILE`s.

Exceptions to these general rules are noted in the description of the specific LOB
call.

> **See Also:**   For more information about FILEs, refer to the
> description of `BFILE`s in the *Oracle9i Application Developer's Guide -
> Large Objects (LOBs)*

Table 7–1   OCI LOB and FILE Functions

Function	Restrictions	Purpose
OCILobAppend()	Internal LOBs only	Appends data from one internal LOB onto another internal LOB. The source and the destination LOBs must already exist. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes) or to try to append from a NULL LOB.
OCILobAssign()		Assigns one LOB/FILE locator to another. This function cannot be used for temporary LOBs; use OCILobLocatorAssign().
OCILobCharSetForm()		Gets the character set form of a CLOB/NCLOB.
OCILobCharSetId()		Gets the character set ID of a CLOB/NCLOB.
OCILobClose()		Closes an opened LOB or BFILE.

Table 7–1   OCI LOB and FILE Functions (Cont.)

Function	Restrictions	Purpose
OCILobCopy()	Internal LOBs only	This function copies a portion of an internal LOB into another internal LOB. The source and destination LOBs must already exist. If data already exists at the destination's start position, it is overwritten with the source data. If the destination's start position is beyond the end of the current value, zero-byte fillers (BLOBs) or spaces (CLOBs/NCLOBs) are placed in the LOB from the end of the destination value to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes). LOB copy operations must be performed on LOBS of the same type; i.e., one CLOB can be copied to another CLOB, and one BLOB can be copied to another BLOB, but a CLOB cannot be copied to a BLOB, and vice versa.
OCILobCreateTemporary()		Creates a temporary LOB.
OCILobDisableBuffering()	Internal LOBs only	Disables LOB buffering for a given internal locator.
OCILobEnableBuffering()	Internal LOBs only	Enables LOB buffering for a given internal locator.

Table 7–1   OCI LOB and FILE Functions (Cont.)

Function	Restrictions	Purpose
`OCILobErase()`	Internal LOBs only	Erases a specified portion of the internal LOB value starting at a specified offset. The actual number of characters/bytes erased is returned. The actual number of characters/bytes and the requested number of characters/bytes will differ if the end of the LOB data is reached before erasing the requested number of characters/bytes. If the LOB is NULL, this routine shows that 0 characters/bytes were erased.
`OCILobFileClose()`, `OCILobFileCloseAll()`		Closes a previously opened FILE, or all open FILEs. It is an error if this function is called for an internal LOB. No error is returned if the FILE exists but is not opened.
`OCILobFileExists()`		Tests to see if a FILE exists on the server.
`OCILobFileGetName()`		Gets the name and the directory alias of a FILE.
`OCILobFileIsOpen()`		Tests to see if a FILE has been opened with the input locator.
`OCILobFileOpen()`		Opens a FILE. The FILE can be opened for read-only access. It is an error if this call is made on an internal LOB.
`OCILobFileSetName()`		Sets the name and the directory alias of a FILE.
`OCILobFlushBuffer()`	Internal LOBs only	Flushes the LOB buffer.
`OCILobFreeTemporary()`		Frees the temporary LOB value.
`OCILobGetChunkSize()`		Gets the usable LOB chunk size.

Table 7–1   OCI LOB and FILE Functions (Cont.)

Function	Restrictions	Purpose
OCILobGetLength()		This function gets the length of a LOB/FILE. If the LOB/FILE is NULL, the length is undefined. Empty internal LOBs have a length of zero. Regardless of whether the client-side character set is varying-width, the output length is in characters for CLOBs/NCLOBs and in bytes for BLOBs/BFILEs.
OCILobIsEqual()		Tests to see if two LOB/FILE locators are equal. Two locators are equal if and only if they both refer to the same LOB/FILE value.
OCILobIsOpen()		Tests whether the LOB is open.
OCILobIsTemporary()		Tests whether it is a temporary LOB.
OCILobLoadFromFile()		Populates all or part of a LOB with data from a FILE.
OCILobLocatorAssign()		Assigns a LOB/FILE locator to another LOB/FILE locator.
OCILobLocatorIsInit()		Tests to see if a LOB/FILE locator is initialized.
OCILobOpen()		Opens a LOB or BFILE.

Table 7–1   OCI LOB and FILE Functions (Cont.)

Function	Restrictions	Purpose
OCILobRead()		This function reads a portion of the LOB/FILE value into a buffer. It is an error to try to read from a NULL LOB/FILE. If the client-side character set is varying-width, then for CLOBs and NCLOBs, the input amount is in characters and the output amount is in bytes. The input amount refers to the number of characters to read from the server-side CLOB/NCLOB. The output amount indicates how many bytes were read into the buffer `bufp`. When using polling mode, note the value of the `amtp` parameter after each `OCILobRead()` call to see how many bytes were read into the buffer because the buffer may not be entirely full. When using callbacks, the `len` parameter, which is input to the callback, indicates how many bytes are filled in the buffer. Be sure to check the `len` parameter during the callback processing because the entire buffer may not be filled with data.
OCILobTrim()	Internal LOBs only	This function truncates a LOB, trimming the LOB value to a specified smaller length.
OCILobWrite()	Internal LOBs only	This function writes data from a buffer into an internal LOB. If data already exists in the LOB, it is overwritten with the data stored in the buffer. If the client-side character set is varying-width, then for CLOBs and NCLOBs, the input amount is in bytes and the output amount is in characters. The input amount refers to the number of bytes of data that should be written to the LOB. The output amount refers to the number of characters written into the server-side CLOB/NCLOB.

Table 7–1   OCI LOB and FILE Functions (Cont.)

Function	Restrictions	Purpose
OCILobWriteAppend()		Writes data starting at the end of the LOB.

Functions for Improving LOB Read/Write Performance

Using OCILobGetChunkSize()

You can take advantage of the OCILobGetChunkSize() call to improve the performance of LOB read and write operations. OCILobGetChunkSize() returns the usable chunk size in bytes for BLOBs and in characters for CLOBs and NCLOBs. When a read or write is done using data whose size is a multiple of the usable chunk size and starts on a chunk boundary, performance improves. You can specify the chunk size for a LOB column when creating a table that contains the LOB.

Calling OCILobGetChunkSize() returns the usable chunk size of the LOB, and an application can batch a series of write operations until an entire chunk can be written, rather than issuing multiple LOB write calls that operate on the same chunk.

To read through the end of a LOB, call OCILobRead() with an amount of 4 gigabytes. This avoids the round-trip involved with first calling OCILobGetLength() because OCILobRead() with an amount of 4 gigabytes reads until the end of the LOB is reached.

> **Note:** For LOBs which store varying width characters, OCILobGetChunkSize() returns the number of Unicode characters that fit in a LOB chunk.

Using OCILobWriteAppend()

The OCI provides a shortcut to make it more efficient to write data to the end of a LOB. The OCILobWriteAppend() enables an application to append data to the end of a LOB without first requiring a call to OCILobGetLength() to determine the starting point for an OCILobWrite() operation. OCILobWriteAppend() takes care of both steps.

LOB Buffering Functions

The Oracle OCI provides several calls for controlling LOB buffering for small reads and writes of internal LOB values:

- `OCILobEnableBuffering()`
- `OCILobDisableBuffering()`
- `OCILobFlushBuffer()`

These functions provide performance improvements by allowing applications using internal LOBs (`BLOB`, `CLOB`, `NCLOB`) to buffer small reads and writes of LOBs in client-side buffers. This reduces the number of network roundtrips and LOB versions, thereby improving LOB performance significantly for small reads and writes.

> **See Also:**
>
> - For more information on LOB buffering, refer to the chapter on LOBs in the *Oracle9i Application Developer's Guide - Large Objects (LOBs)*
> - For a code sample showing the use of LOB buffering, see the demonstration programs included with your Oracle installation. Refer to Appendix B, "OCI Demonstration Programs".

Functions for Opening and Closing LOBs

The OCI provides functions to explicitly open (`OCILobOpen()`) and close (`OCILobClose()`) a LOB, and also to test whether a particular LOB is already open (`OCILobIsOpen()`). These functions allow an application to mark the beginning and end of a series of LOB operations so that specific processing (e.g., updating indexes, etc.) can be performed when a LOB is closed.

> **Note:** For internal LOBs, the concept of openness is associated with a LOB and not its locator. The locator does not store any information about whether the LOB to which it refers is open. It is possible for more than one locator to point to the same open LOB. However, for `BFILES`, openness is associated with a specific locator. Hence, more than one open can be performed on the same `BFILE` using different locators.

If an application does not wrap LOB operations between a set of `OCILobOpen()` and `OCILobClose()` calls, then each modification to the LOB implicitly opens and closes the LOB, thereby firing any triggers associated with changes to the LOB.

> **Note:** If LOB operations are not wrapped inside open and close calls, any extensible indexes on the LOB are updated as LOB modifications are made, and thus are always valid and may be used at any time. If the LOB is modified between a set of `OCILobOpen()` and `OCILobClose()` calls, triggers are not fired for individual LOB modifications. Triggers are only fired after the `OCILobClose()` call, so indexes are not updated until after the close call and thus are not valid in between the open and close calls. `OCILobIsOpen()` can be used with internal and external LOBs (`BFILEs`).

It is an error to commit the transaction before closing all opened LOBs that were opened by the transaction. When the error is returned, the LOB is no longer marked as open, but the transaction is successfully committed. Hence, all the changes made to the LOB and non-LOB data in the transaction are committed, but the domain and functional indexing are not updated. If this happens, please rebuild your functional and domain indexes on the LOB column.

A LOB opened when there is no transaction must be closed before the end of session. If there are LOBs open at the end of session, the LOB is no longer marked as open and the domain and functional indexing will not be updated. If this happens, please rebuild your functional and domain indexes on the LOB column.

Restrictions

The LOB opening and closing mechanism has the following restrictions:

1. An application must close all previously opened LOBs before committing a transaction. Failing to do so will result in an error. If a transaction is rolled back, all open LOBs are discarded along with the changes made (the LOBs are not closed), so associated triggers are not fired.

2. While there is no limit to the number of open internal LOBs, there is a limit on the number of open files. Refer to SESSION_MAX_OPEN_FILES parameter in *Oracle9i Database Reference*. Note that assigning an already opened locator to another locator does not count as opening a new LOB.

3. It is an error to open or close the same internal LOB twice within the same transaction, either with different locators or the same locator.

4. It is an error to close a LOB that has not been opened.

> **Note:** The definition of a *transaction* within which an open LOB value must be closed is one of the following:
>
> - between SET TRANSACTION and COMMIT
> - between DATA MODIFYING DML or SELECT ... FOR UPDATE and COMMIT.
> - within an autonomous transaction block

LOB Open/Close Examples

> **See Also:** For examples of the use of the OCILobOpen() and OCILobCLose() calls, see the list of online demonstration programs in Appendix B, "OCI Demonstration Programs"

Server Roundtrips for LOB Functions

> **See Also:** For a table showing the number of server roundtrips required for individual OCI LOB functions, refer to Appendix C, "OCI Function Server Round-trips"

LOB Read and Write Callbacks

The OCI LOB read and write functions provide the ability to define callback functions which can be used to provide data to be written or handle data that was read. This allows the client application to perform optional processing on the data. One example usage of this would be to use the callbacks to implement a compression algorithm for writing the data and a decompression algorithm for reading it.

> **Note:** The LOB read/write streaming callbacks provides a fast method for reading/writing large amounts of LOB data.

The following sections describe the use of callbacks in more detail.

The Callback Interface for Streaming

Your application can use user-defined read and write callback functions to insert data into or retrieve data from a LOB. This provides an alternative to the polling method for streaming data into a LOB and retrieving data from a LOB. The user-defined callbacks have a specific prototype which is described below. These functions are implemented by you and registered with OCI through the `OCILobRead()` and `OCILobWrite()` calls. The callback functions are called by OCI whenever required.

Reading LOBs using Callbacks

The user-defined read callback function is registered through the `OCILobRead()` function. The callback function should have the following prototype:

CallbackFunctionName (dvoid *ctxp, CONST dvoid *bufp, ub4 bufl, ub1 piece)

The first parameter, *ctxp*, is the context of the callback that is passed to OCI in the `OCILobRead()` function call. When the callback function is called, the information provided by you in *ctxp* is passed back to you (the OCI does not use this information on the way IN). The *bufp* parameter is the pointer to the storage where the LOB data is returned and *bufl* is the length of this buffer. It tells you how much data has been read into the buffer provided by you.

If the buffer length provided by you in the original `OCILobRead()` call is insufficient to store all the data returned by the server, then the user-defined callback is called. In this case the *piece* parameter indicates to you whether the information returned in the buffer in the first, next or last piece.

The following is a code fragment of a typical way to implement read callback functions. Assume here that *lobl* is a valid locator that has been previously selected, *svchp* is a valid service handle and *errhp* is a valid error handle.

```
...
ub4   offset = 1;
ub4   loblen = 0;
ub1   bufp[MAXBUFLEN];
ub4   amtp = 0;
sword retval;
amtp = 4294967295;              /* 4 gigabytes minus 1 */
if (retval = OCILobRead(svchp, errhp, lobl, &amtp, offset, (dvoid *) bufp,
        (ub4) MAXBUFLEN, (dvoid *) bufp, cbk_read_lob,
        (ub2) 0, (ub1) SQLCS_IMPLICIT))
   {
     (void) printf("ERROR: OCILobRead() LOB.\n");
```

```
        report_error();
    }
...
sb4 cbk_read_lob(ctxp, bufxp, len, piece)
dvoid *ctxp;
CONST dvoid *bufxp;
ub4 len;
ub1 piece;
{
static ub4 piece_count = 0;
piece_count++;
switch (piece)
{
  case OCI_LAST_PIECE:
    /* process buffer bufxp */
    --- buffer processing code goes here ---
    (void) printf("callback read the %d th piece\n\n", piece_count);
    piece_count = 0;
    break;
  case OCI_FIRST_PIECE:
  case OCI_NEXT_PIECE:
    /* process buffer bufxp */
    --- buffer processing code goes here ---
    (void) printf("callback read the %d th piece\n", piece_count);
    break;
  default:
    (void) printf("callback read error: unkown piece = %d.\n", piece);
  return OCI_ERROR;
  }
  return OCI_CONTINUE;
}
```

In the above example the use- defined function `cbk_read_lob()` is repeatedly called until all the LOB data has been read by you.

> **See Also:** For an example of the use of `OCILobRead()` using polling and callbacks, see the list of online demonstration programs in Appendix B, "OCI Demonstration Programs"

Writing LOBs using Callbacks

Similar to read callbacks, the user-defined write callback function is registered through the `OCILobWrite()` function. The callback function should have the following prototype:

```
CallbackFunctionName ( dvoid *ctxp, dvoid *bufp, ub4 *bufl, ub1 *piecep)
```

The first parameter, *ctxp*, is the context of the callback that is passed to OCI in the `OCILobWrite()` function call. The information provided by you in *ctxp*, is passed back to you when the callback function is called by the OCI (the OCI does not use this information on the way IN). The *bufp* parameter is the pointer to a storage area You provide this pointer in the call to `OCILobWrite()`. After inserting the data provided in the call to `OCILobWrite()` if there is more to write, then the user-defined callback is called. In the callback you should provide the data to insert in the storage indicated by *bufp* and also specify the length in *bufl*. You should also indicate whether it is the next (OCI_NEXT_PIECE) or the last (OCI_LAST_PIECE) piece using the *piecep* parameter. Note that you are completely responsible for the storage pointer the application provides and should make sure that it does not write more than the allocated size of the storage.

The following is a code fragment of a typical way to implement write callback functions.

Assume here that *lobl* is a valid locator that has been locked for updating, *svchp* is a valid service handle and *errhp* is a valid error handle

```
...
ub4    offset = 1;
ub1    bufp[MAXBUFLEN];
ub4    amtp = MAXBUFLEN * 20;
ub4    nbytes = MAXBUFLEN;
/* Fill bufp with some data */
-- code to fill bufp with data goes here. nbytes should reflect the size and
should be less than or equal to MAXBUFLEN --
if (retval = OCILobWrite(svchp, errhp, lobl, &amtp, offset, (dvoid*)
        bufp,(ub4)nbytes, OCI_FIRST_PIECE, (dvoid *)0, cbk_write_lob,
        (ub2) 0, (ub1) SQLCS_IMPLICIT))
  {
    (void) printf("ERROR: OCILobWrite().\n");
    report_error();
    return;
  }
 ...
sb4 cbk_write_lob(ctxp, bufxp, lenp, piecep)
dvoid *ctxp;
dvoid *bufxp;
ub4 *lenp;
ub1 *piecep;
{
  /* Fill bufxp with  data */
```

```
  -- code to fill bufxp with data goes here. *lenp should reflect the size
  and should be less than or equal to MAXBUFLEN --
  if (this is the last data buffer)
    *piecep = OCI_LAST_PIECE;
  else
    *piecep = OCI_NEXT_PIECE;;
  return OCI_CONTINUE;
}
```

In the above example, the user-defined function `cbk_write_lob()` is repeatedly called until you indicate that the application is providing the last piece using the *piecep* parameter.

> **See Also:** For an example of the use of `OCILobWrite()` using polling and callbacks, see the list of online demonstration programs in Appendix B, "OCI Demonstration Programs"

Temporary LOB Support

The OCI provides functions for creating and freeing temporary LOBs, `OCILobCreateTemporary()` and `OCILobFreeTemporary()`, plus a function for querying whether or not a given LOB is a temporary LOB, `OCILobIsTemporary()`.

Temporary LOBs are not permanently stored in the database, but can act like local variables for operating on LOB data. OCI functions which operate on standard (persistent) LOBs can also be used on temporary LOBs.

As with standard LOBs, all functions operate on the locator for the temporary LOB, and the actual LOB data is accessed through the locator.

Temporary LOB locators can be used as arguments to the following types of SQL statements:

- UPDATE -Tthe temporary LOB locator can be used as a value in a WHERE clause when testing for nullity or as a parameter to a function. The locator can also be used in a SET clause.

- DELETE - The temporary LOB locator can be used in a WHERE clause when testing for nullness or as a parameter to a function.

- SELECT - The temporary LOB locator can be used in a WHERE clause when testing for nullity or as a parameter to a function. The temporary LOB can also be used as a return variable in a SELECT...INTO statement when selecting the return value of a function.

- **Note:** If you select a permanent locator into a temporary locator, the temporary locator is overwritten with the permanent locator. In this case the temporary LOB is not implicitly freed. You must explicitly free the temporary LOB before the SELECT...INTO. If the temporary LOB is not freed explicitly, it will not be freed until the end of its duration. Unless you have another temporary locator pointing to the same LOB, you will no longer have a locator pointing to the temporary LOB, because the original locator was overwritten by the SELECT...INTO.

Creating and Freeing Temporary LOBs

You create a temporary LOB with the OCILobCreateTemporary() function. The parameters passed to this function include a value for the duration of the LOB. The default duration is for the length of the current session. At the end of the duration all temporary LOBs are deleted. Users can reclaim temporary LOB space by explicitly freeing the temporary LOB with the OCILobFreeTemporary() function. A temporary LOB is empty when it is created.

When creating a temporary LOB, you can also specify whether or not the temporary LOB is read into the server's buffer cache.

To make a temporary LOB permanent, the application can use OCILobCopy() to copy the data from the temporary LOB into a permanent one. The application can also use the temporary LOB in the VALUES clause of an INSERT statement, use the temporary LOB as the source of the assignment in an UPDATE statement, or assign the temporary LOB to a persistent LOB attribute and the flush the object.

Temporary LOBs can be modified with the same functions which are used for standard LOBs.

Temporary LOB Durations

The OCI supports several predefined durations for temporary LOBs and a set of functions that the application can use to define application-specific durations. The predefined durations are:

1. call (OCI_DURATION_CALL), only on the server side

2. session (OCI_DURATION_SESSION)

The *session duration* expires when the containing session/connection ends. The *call duration* expires at the end of the current OCI call.

When running in object mode, a you can also define application-specific durations. An *application-specific duration*, also referred to as a *user duration,* is defined by

specifying the start of a duration using the `OCIDurationBegin()` function and the end of the duration using the `OCIDurationEnd()` function.

> **Note:** User-defined durations are only available if an application has been initialized in object mode.

Each application-specific duration has a duration identifier that is returned by `OCIDurationBegin()` and is guaranteed to be unique until `OCIDurationEnd()` is called on the duration. An application-specific duration can be as long as, but not longer, than a session duration.

At the end of a duration, all temporary LOBs associated with that duration are freed. However, the descriptor associated with the temporary LOB must be freed explicitly with the `OCIDescriptorFree()` call.

User-defined durations can be nested—one duration can be defined as a *child duration* of another user duration. It is possible for a parent duration to have child durations which, in turn, have their own child durations.

> **Note:** When a duration is started with `OCIDurationBegin()`, one of the parameters is the identifier of a parent duration. When a parent duration is ended, all child durations are also ended. For more information, see `OCIDurationBegin()` on page 16-7.

Temporary LOB Example

The following code example shows how temporary LOBs might be used:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>

/* Function Prototype */
static void checkerr (/*_ OCIError *errhp, sword status _*/);
sb4 select_and_createtemp (OCILobLocator *lob_loc,
                           OCIError     *errhp,
                           OCISvcCtx    *svchp,
                           OCIStmt      *stmthp,
                           OCIEnv       *envhp);

/* This function reads in a single video Frame from the Multimedia_tab table.
```

Then it creates a temporary lob. The temporary LOB which is created is read
through the CACHE, and is automatically cleaned up at the end of the user's
session, if it is not explicitly freed sooner. This function returns OCI_SUCCESS
if it completes successfully or OCI_ERROR if it fails. */

```
sb4 select_and_createtemp (OCILobLocator *lob_loc,
                           OCIError      *errhp,
                           OCISvcCtx     *svchp,
                           OCIStmt       *stmthp,
                           OCIEnv        *envhp)
{
  OCIDefine     *defnp1;
  OCIBind       *bndhp;
  text          *sqlstmt;
  int rowind =1;
  ub4 loblen = 0;
  OCILobLocator *tblob;
  printf ("in select_and_createtemp \n");
    if(OCIDescriptorAlloc((dvoid*)envhp, (dvoid **)&tblob,
                                    (ub4)OCI_DTYPE_LOB, (size_t)0,
                                    (dvoid**)0))
  {
    printf("failed in OCIDescriptor Alloc in select_and_createtemp \n");
    return OCI_ERROR;
  }
  /* arbitrarily select where Clip_ID =1 */
  sqlstmt = (text *)"SELECT Frame FROM Multimedia_tab WHERE Clip_ID = 1 FOR
UPDATE";
  if (OCIStmtPrepare(stmthp, errhp, sqlstmt,
                     (ub4) strlen((char *)sqlstmt),
                     (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
  {
      (void) printf("FAILED: OCIStmtPrepare() sqlstmt\n");
      return OCI_ERROR;
  }
  /* Define for BLOB */
  if (OCIDefineByPos(stmthp,
                     &defnp1,
                     errhp,
                     (ub4) 1,
                     (dvoid *) &lob_loc,
                     (sb4)0,
                     (ub2) SQLT_BLOB,
                     (dvoid *) 0,
                     (ub2 *) 0,
```

```
                             (ub2 *) 0,
                             (ub4) OCI_DEFAULT))
   {
     (void) printf("FAILED: Select locator: OCIDefineByPos()\n");
     return OCI_ERROR;
   }
   /* Execute the select and fetch one row */
   if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                      (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                      (ub4) OCI_DEFAULT))
   {
     (void) printf("FAILED: OCIStmtExecute() sqlstmt\n");
     return OCI_ERROR;
   }
   if(OCILobCreateTemporary(svchp,
                                 errhp,
                                 tblob,
                                 (ub2)0,
                                 SQLCS_IMPLICIT,
                                 OCI_TEMP_BLOB,
                                 OCI_ATTR_NOCACHE,
                                 OCI_DURATION_SESSION))
   {
 (void) printf("FAILED: CreateTemporary() \n");
     return OCI_ERROR;
   }
   if (OCILobGetLength(svchp, errhp, lob_loc, &loblen) != OCI_SUCCESS)
   {
     printf("OCILobGetLength FAILED\n");
     return OCI_ERROR;
   }
   if (OCILobCopy(svchp, errhp, tblob,lob_loc,(ub4)loblen, (ub4) 1,
                 (ub4) 1))
   {
     printf( "OCILobCopy FAILED \n");
   }
   if(OCILobFreeTemporary(svchp,errhp,tblob))
   {
     printf ("FAILED: OCILobFreeTemporary call \n");
     return OCI_ERROR;
   }
      return OCI_SUCCESS;
}
int main(char *argv, int argc)
{
```

```
/* OCI Handles */

OCIEnv        *envhp;
OCIServer     *srvhp;
OCISvcCtx     *svchp;
OCIError      *errhp;
OCISession    *authp;
OCIStmt       *stmthp;
OCILobLocator *clob, *blob;
OCILobLocator *lob_loc;
int type =1;
/* Initialize and Logon */
(void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                     (dvoid * (*)(dvoid *, size_t)) 0,
                     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                     (void (*)(dvoid *, dvoid *)) 0 );
(void) OCIEnvInit( (OCIEnv **) &envhp,
                   OCI_DEFAULT, (size_t) 0,
                   (dvoid **) 0 );
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                        (size_t) 0, (dvoid **) 0);
/* server contexts */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                        (size_t) 0, (dvoid **) 0);
 /* service context */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                        (size_t) 0, (dvoid **) 0);
  /* attach to Oracle */
(void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);
  /* set attribute server context in the service context */
(void) OCIAttrSet ((dvoid *) svchp, OCI_HTYPE_SVCCTX,
                   (dvoid *)srvhp, (ub4) 0,
                   OCI_ATTR_SERVER, (OCIError *) errhp);
 (void) OCIHandleAlloc((dvoid *) envhp,
                       (dvoid **)&authp, (ub4) OCI_HTYPE_SESSION,
                       (size_t) 0, (dvoid **) 0);
 (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                   (dvoid *) "scott", (ub4)5,
                   (ub4) OCI_ATTR_USERNAME, errhp);
(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                   (dvoid *) "tiger", (ub4) 5,
                   (ub4) OCI_ATTR_PASSWORD, errhp);
/* Begin a User Session */
checkerr(errhp, OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,
                                  (ub4) OCI_DEFAULT));
```

```
                      (void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                                   (dvoid *) authp, (ub4) 0,
                                   (ub4) OCI_ATTR_SESSION, errhp);
                  /* ------- Done loggin in --------------------------------*/
                   /* allocate a statement handle */
                  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                                                 OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
                  checkerr(errhp, OCIDescriptorAlloc((dvoid *)envhp, (dvoid **) &lob_loc,
                                                     (ub4) OCI_DTYPE_LOB,
                                                     (size_t) 0, (dvoid **) 0));
                  /* Subroutine calls begin here */
                  printf("calling select_and_createtemp\n");
                  select_and_createtemp (lob_loc, errhp, svchp,stmthp,envhp);

                  return 0;
                }
                void checkerr(errhp, status)
                OCIError *errhp;
                sword status;
                {
                  text errbuf[512];
                  sb4 errcode = 0;
                  switch (status)
                  {
                  case OCI_SUCCESS:
                    break;
                  case OCI_SUCCESS_WITH_INFO:
                    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
                    break;
                  case OCI_NEED_DATA:
                    (void) printf("Error - OCI_NEED_DATA\n");
                    break;
                  case OCI_NO_DATA:
                    (void) printf("Error - OCI_NODATA\n");
                    break;
                  case OCI_ERROR:
                    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                                       errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
                    (void) printf("Error - %.*s\n", 512, errbuf);
                    break;
                  case OCI_INVALID_HANDLE:
                    (void) printf("Error - OCI_INVALID_HANDLE\n");
                    break;
                  case OCI_STILL_EXECUTING:
                    (void) printf("Error - OCI_STILL_EXECUTE\n");
```

```
       break;
     case OCI_CONTINUE:
       (void) printf("Error - OCI_CONTINUE\n");
       break;
     default:
       break;
     }
}
```

8

Managing Scalable Platforms

The following topics are covered in this chapter:

- OCI Support for Transactions
- Password and Session Management
- Middle-tier Applications
- Externally Initialized Context
- Oracle9i Application Server Cache

OCI Support for Transactions

OCI has a set of API calls to support operations on both local and global transactions. These calls include object support, so that if an OCI application is running in object mode, the commit and rollback calls will synchronize the object cache with the state of the transaction.

The functions listed below perform transaction operations. Each call takes a service context handle that should be initialized with the proper server context and user session handle. The transaction handle is the third element of the service context; it stores specific information related to a transaction. When a SQL statement is prepared, it is associated with a particular service context. When the statement is executed, its effects (query, fetch, insert) become part of the transaction that is currently associated with the service context.

- `OCITransStart()` - marks the start of a transaction
- `OCITransDetach()` - detaches a transaction
- `OCITransCommit()` - commits a transaction
- `OCITransRollback()` - rolls back a transaction
- `OCITransPrepare()` - prepares a transaction to be committed in a distributed processing environment
- `OCITransMultiPrepare()` - prepares a transaction with multiple branches in a single call.
- `OCITransForget()` - causes the server to forget a heuristically completed global transaction.

Depending on the level of transactional complexity in your application, you may need all or only a few of these calls. The following section discusses this in more detail.

> **See Also:** For more specific information about these calls, refer to the function descriptions in "Transaction Functions" on page 16-142

Levels of Transactional Complexity

The OCI supports several levels of transaction complexity. Each level is described in one of the following sections.

- Simple Local Transactions
- Serializable or Read-Only Local Transactions

- Global Transactions

Simple Local Transactions

Many applications work with only simple local transactions. In these applications, an implicit transaction is created when the application makes database changes. The only transaction-specific calls needed by such applications are:

- `OCITransCommit()` - to commit the transaction
- `OCITransRollback()` - to roll back the transaction

As soon as one transaction has been committed or rolled back, the next modification to the database creates a new implicit transaction for the application.

Only one implicit transaction can be active at any time on a service context. Attributes of the implicit transaction are opaque to the user.

If an application creates multiple sessions, each one can have an implicit transaction associated with it.

> **See Also:** For sample code showing the use of simple local transactions, refer to the example for `OCITransCommit()` on page 16-143

Serializable or Read-Only Local Transactions

Applications requiring serializable or read-only transactions require an additional OCI call beyond those needed by applications operating on simple local transactions. To initiate a serializable or read-only transactions, the application must create the transaction by calling `OCITransStart()` to start the transaction.

The call to `OCITransStart()` should specify OCI_TRANS_SERIALIZABLE or OCI_TRANS_READONLY, as appropriate, for the *flags* parameter. If no flag is specified, the default value is OCI_TRANS_READWRITE for a standard read-write transaction.

Specifying the read-only option in the `OCITransStart()` call saves the application from performing a server round trip to execute a SET TRANSACTION READ ONLY statement.

Global Transactions

Global transactions are necessary only in more sophisticated transaction-processing applications.

> **Note:** Users not operating in distributed or global transaction environments may skip this section

This section has some background about global transactions, and then gives specific information about using OCI calls to process global transactions.

Transaction Identifiers Three-tiered applications such as transaction processing (TP) monitors create and manage global transactions. They supply a *global transaction identifier* (XID), which a server then associates with a local transaction.

A global transaction has one or more *branches*. Each branch is identified by an XID. The XID consists of a *global transaction identifier* (gtrid) and a *branch qualifier* (bqual). This structure is based on the standard XA specification.

For example, the following is the structure for one possible XID of 1234:

Component	Value
gtrid	12
bqual	34
gtrid+bqual=XID	1234

> **See Also:** For more information about transaction identifiers, see *Oracle9i Distributed Database Systems*

The transaction identifier used by OCI transaction calls is set in the OCI_ATTR_XID attribute of the transaction handle, using `OCIAttrSet()`. Alternately, the transaction can be identified by a name set in the OCI_ATTR_TRANS_NAME attribute.

Transaction Branches Within a single global transaction, Oracle supports both tightly coupled and loosely coupled relationships between a pair of branches.

- Tightly coupled branches are different branches that share the same local transaction. In this case, the *gtrid* references a unique local transaction, and multiple branches point to that same transaction. The owner of the transaction is the branch that was created first.

- Loosely coupled branches are different branches that use different local transactions. In this case the *gtrid* and *bqual* together map to a unique local transaction. Each branch points to a different transaction.

The *flags* parameter of OCITransStart() allows applications to pass OCI_TRANS_TIGHT or OCI_TRANS_LOOSE to specify the type of coupling.

A session corresponds to a user session, created with OCISessionBegin().

The following figure illustrates tightly coupled branches within an application. In the figure, S1 and S2, are sessions, B1 and B2 are branches, and T is a transaction. In this first example, the XIDs of the two branches would share the same *gtrid*, because they are operating on the same transaction, but they would have a different *bqual*, because they are separate branches

Figure 8–1   Multiple Tightly Coupled Branches
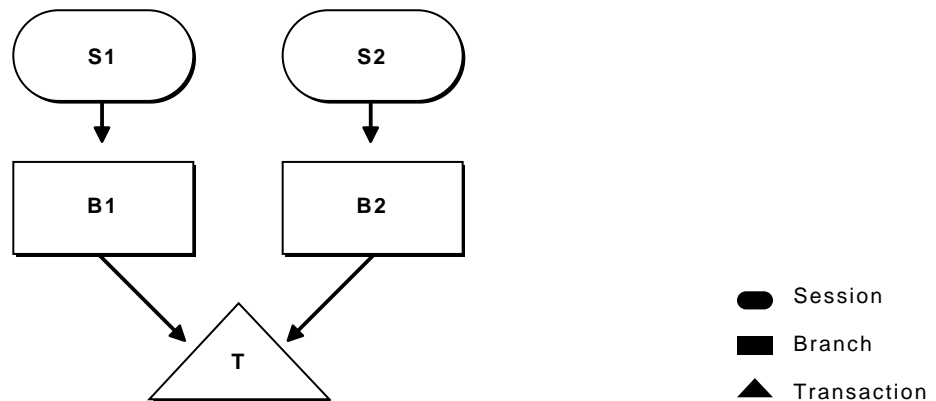
It is also possible for a single session to operate on different branches. In this case, illustrated in the next figure, *gtrid* component of the XIDs would be different, because they are separate global transactions

Figure 8–2   Session Operating on Multiple Branches
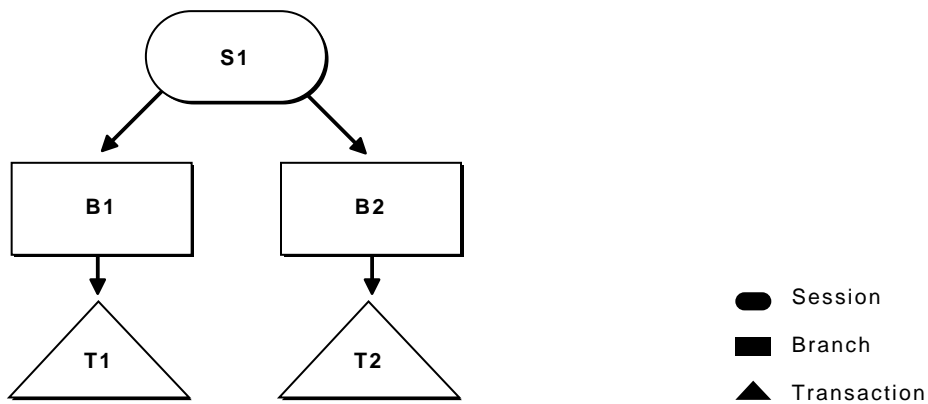
> **See Also:**   For sample code demonstrating this scenario, refer to
> the examples for `OCITransStart()` on page 16-151.

It is possible for a single session to operate on multiple branches that share the
same transaction, but this scenario does not have much practical value.

> **See Also:**   Sample code demonstrating this scenario can be found
> in the examples for `OCITransStart()` on page 16-151.

The following figure illustrates loosely coupled branches:

Figure 8–3   Loosely Coupled Branches
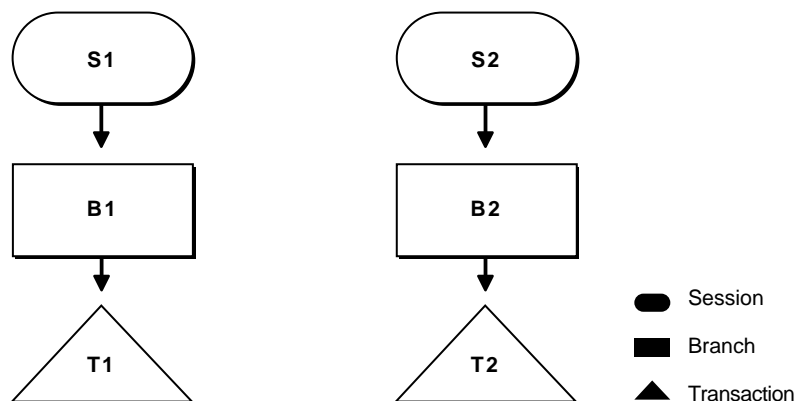
Branch States Transaction branches are classified into two states: *active branches* and *inactive branches.*

A branch is active if a server process is executing requests on the branch. A branch is inactive if no server processes are executing requests in the branch. In this case no session is the parent of the branch, and the branch becomes owned by the PMON process in the server.

Detaching and Resuming Branches A branch becomes inactive when an OCI application detaches it, using the OCITransDetach() call. The branch can be made active again by resuming it with a call to OCITransStart() with the *flags* parameter set to OCI_TRANS_RESUME.

When an application detaches a branch with OCITransDetach(), it utilizes the value specified in the *timeout* parameter of the OCITransStart() call that created the branch. The *timeout* specifies the number of seconds the transaction can remain dormant as a child of PMON before being deleted.

When an application wants to resume a branch, it calls OCITransStart(), specifying the XID of the branch as an attribute of the transaction handle, OCI_TRANS_RESUME for the *flags* parameter, and a different *timeout* parameter. This *timeout* value for this call specifies the length of time that the session will wait for the branch to become available if it is currently in use by another process. If no other processes are accessing the branch, it can be resumed immediately.

> **Note:** A transaction can be resumed by a different process than the one that detached it, as long as that process has the same authorization as the one that detached the transaction.

Setting Client Database Name The server handle has OCI_ATTR_EXTERNAL_NAME and OCI_ATTR_INTERNAL_NAME attributes associated with it. These attributes set the client database name that will be recorded when performing global transactions. The name can be used by the DBA to track transactions that may be pending in a prepared state due to failures.

> **Caution:** An OCI application should set these attributes, using OCIAttrSet(), before logging on and using global transactions.

One-Phase Versus Two-Phase Commit Global transactions may be committed in one or two phases. The simplest situation is when a single transaction is operating against a single database. In this case, the application can perform a one-phase commit of the transaction, by calling OCITransCommit(), because the default value of the call is for one-phase commit.

The situation is more complicated if the application is processing transactions against multiple databases or multiple Oracle servers. In this case, a two-phase commit is necessary. A two-phase commit consists of these steps:

1. **Prepare** - The application issues a prepare call, OCITransPrepare() against each transaction. The transaction returns a value indicating whether it is able to commit its current work (OCI_SUCCESS) or not (OCI_ERROR).

2. **Commit** - If each prepare call returns a value of OCI_SUCCESS, the application can issue a commit call, OCITransCommit() to each transaction. The *flags* parameter of the commit call must be explicitly set to OCI_TRANS_TWOPHASE for the appropriate behavior. The default for this call is for a one-phase commit.

> **Note:** The prepare call can also return OCI_SUCCESS_WITH_INFO if a transaction needs to indicate that it is read-only, so that a commit is neither appropriate nor necessary.

An additional call, `OCITransForget()` indicates that a database should forget a heuristically completed transaction. This call is for situations in which a problem has occurred that requires that a two-phase commit be aborted. When a server receives a `OCITransForget()` call, it *forgets* all information about the transaction.

> **See Also:** For more information about two-phase commit, refer to *Oracle9i Distributed Database Systems*

Preparing Multiple Branches in a Single Message There are times when multiple applications will be using different branches of a global transaction against the same Oracle database. Before such a transaction can be committed, all branches must be prepared.

Most often, the applications using the branches are responsible for preparing their own branches. However, some architectures turn this responsibility over to an external transaction service. This external transaction service must then prepare each branch of the global transaction. Using the traditional `OCITransPrepare()` call, will then be very expensive, as each branch must be individually prepared. The number of messages sent from the client to the server can be greatly reduced by using the `OCITransMultiPrepare()` call. This call will prepare multiple branches involved in the same global transaction in one round trip.

Transaction Examples

Here is how to use the transaction OCI calls:

The following tables show series of OCI calls and other actions, along with their resulting behavior. For the sake of simplicity, not all parameters to these calls are listed; rather, the flow of calls which is being demonstrated.

The **OCI Action** column indicates what the OCI application is doing, or what call it is making. The **XID** column lists the transaction identifier, when necessary. The **Flags** column lists the value(s) passed in the *flags* parameter. The **Result** column describes the result of the call.

Update Successfully, One-Phase Commit

Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW	Starts new read-write transaction
2	SQL UPDATE			Update rows

Step	OCI Action	XID	Flags	Result
3	OCITransCommit			Commit succeeds

Start a Transaction, Detach, Resume, Prepare, Two-Phase Commit

Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW	Starts new read-only transaction
2	SQL UPDATE			Update rows
3	OCITransDetach			Transaction is detached
4	OCITransStart	1234	OCI_TRANS_RESUME	Transaction is resumed
5	SQL UPDATE			
6	OCITransPrepare			Transaction prepared for two-phase commit
7	OCITransCommit		OCI_TRANS_TWOPHASE	Transaction is committed.

Note: In step 4, above, the transaction could have been resumed by a different process, as long as it had the same authorization.

Read-Only Update Fails

Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW | OCI_TRANS_READONLY	Starts new read-only transaction
2	SQL UPDATE			Update fails, because transaction is read-only
3	OCITransCommit			Commit has no effect

Start a Read-Only Transaction, Select and Commit

Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW | OCI_TRANS_READONLY	Starts new read-only transaction
2	SQL SELECT			Query database
3	OCITransCommit			No effect — transaction is read-only, no changes made

Related Initialization Parameters

Two initialization parameters relate to the use of global transaction branches and migratable open connections:

- TRANSACTIONS - This parameter specifies the maximum number of global transaction branches in the entire system. In contrast, the number of branches on a single global transaction is 8.

- OPEN_LINKS_PER_INSTANCE - This parameter specifies the maximum number of migratable open connections. Migratable open connections are used by global transactions so that connections are cached after a transaction is committed. This is different from the OPEN_LINKS parameter, which is the number of connections from a session (and is not applicable to applications that use global transactions).

Password and Session Management

The OCI has the ability to authenticate and maintain multiple users in an OCI application. There is also an OCI call which allows the application to update a user

password. This is particularly helpful if an expired password message is returned by an authentication attempt.

Authentication Management

The `OCISessionBegin()` call is used to authenticate a user against the server set in the service context handle. `OCISessionBegin()` must be called for any given server handle before requests can be made against it. Also, `OCISessionBegin()` only supports authenticating the user for access to the Oracle server specified by the server handle in the service context that is used for the `OCISessionBegin()` call. In other words, after `OCIServerAttach()` is called to initialize a server handle, `OCISessionBegin()` must be called to authenticate the user for that given server identified by the server handle.

When `OCISessionBegin()` is called for the first time for a given server handle, the user session may not be created in migratable mode (OCI_MIGRATE). After `OCISessionBegin()` has been called for a server handle, the application may call `OCISessionBegin()` again to initialize another user session handle with different or the same credentials and different or the same operation modes. If an application wants to authenticate a user in OCI_MIGRATE mode, the service handle must already be associated with a non-migratable user handle. The userid of that user handle becomes the ownership ID of the migratable user session. Every migratable session must have a non-migratable parent session.

If the OCI_MIGRATE mode is not specified, then the user session context can only be used with the same server handle that was used with the `OCISessionBegin()`. If OCI_MIGRATE mode is specified, then the user authentication may be set with different server handles. However, the user session context may only be used with server handles which resolve to the same database instance. Security checking is done during session switching.

A migratable session is allowed to switch to a different server handle only if the ownership ID of the session matches the userid of a non-migratable session currently connected to that same server.

OCI_SYSDBA, OCI_SYSOPER, and OCI_PRELIM_AUTH may only be used with a primary user session context.

A migratable session can be switched, or migrated, to a server handle within a given environment represented by a environment handle. It can also be migrated, or cloned, to a server handle in another environment in the same process or in a different process in a different mode. To perform this migration, or cloning, you need to do the following:

1. Extract the session Id from the session handle using OCI_ATTR_MIGSESSION. This is an array of bytes. It should not be modified by the caller.

   See Also: OCI_ATTR_MIGSESSION on page A-19

2. Transport this session Id to any other process by any means.

3. In the new environment, create a session handle and set the session Id using OCI_ATTR_MIGSESSION.

4. Execute `OCISessionBegin()`. The resulting session handle is a fully-authenticated session handle.

To provide credentials for a call to `OCISessionBegin()`, one of two methods are supported. The first is to provide a valid username and password pair for database authentication in the user session handle passed to `OCISessionBegin()`. This involves using `OCIAttrSet()` to set the OCI_ATTR_USERNAME and OCI_ATTR_PASSWORD attributes on the user session handle. Then `OCISessionBegin()` is called with OCI_CRED_RDBMS.

> **Note:** When the user session handle is terminated using `OCISessionEnd()`, the username and password attributes are changed and thus cannot be re-used in a future call to `OCISessionBegin()`. They must be reset to new values before the next `OCISessionBegin()` call.

The second type of credentials supported are external credentials. No attributes need to be set on the user session handle before calling `OCISessionBegin()`. The credential type is OCI_CRED_EXT. If values have been set for OCI_ATTR_USERNAME and OCI_ATTR_PASSWORD, then these are ignored if OCI_CRED_EXT is used.

Password Management

OCI has the `OCIPasswordChange()` call to allow an OCI application to modify a user's database password as necessary. This is particularly useful if a call to `OCISessionBegin()` returns an error message or warning indicating that a user's password has expired.

Applications can also use `OCIPasswordChange()` to establish a user authentication context, as well as to change password, if appropriate flags are set. If `OCIPasswordChange()` is called with an uninitialized service context, it

establishes a service context and authenticates the user's account using the old password, and then changes the password to the new password. If the OCI_AUTH flag is set, it leaves the user session initialized. Otherwise, the user session is cleared.

If the service context passed to `OCIPasswordChange()` is already initialized, then `OCIPasswordChange()` authenticates the given account using the old password and changes the password to the new password. In this case, no matter how the flag is set, the user session remains initialized.

Session Management

Applications, such as transaction servers, that perform active user load balancing by multiplexing user sessions over a few server connections must group these connections into a server group. Oracle uses the server groups to identify these connections so that sessions can be managed effectively and securely.

The attribute OCI_ATTR_SERVER_GROUP must be defined for a server context to specify the server group name. For example:

```
OCIAttrSet ((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER, (dvoid *) group_name,
            (ub4) strlen ((CONST char *)  group_name,
            (ub4) OCI_ATTR_SERVER_GROUP, errhp);
```

The server group name is an alpha-numeric string not exceeding 30 characters. OCI_ATTR_SERVER_GROUP attribute must be set in the server context prior to creating the first non-migratable session using that context. After the session is created successfully and the connection is established to the server, the server group name cannot be changed.

> **See Also:** OCI_ATTR_SERVER_GROUP on page A-16

All migratable sessions created on servers within a server group can only migrate to other servers in the same server group. Servers that terminate will get removed from the server group. New servers may be created within an existing server group at any time.

Server groups are optional. If no server group is specified, the server will get created in a server group called DEFAULT.

The owner of the first non-migratable session created in the first server in a server group other than DEFAULT establishes ownership of the server group. All subsequent non-migratable sessions for any server in this server group must be created by the same user as the owner of the server group.

The server group feature is useful when dedicated servers are used. It has no effect for shared servers. In case of shared servers, all shared servers will effectively belong to the server group DEFAULT.

Middle-tier Applications

A middle-tier application receives requests from browser clients and decides whether to access a database to generate an HTML page to return. Applications can have multiple user sessions within a single database session. These "lightweight sessions" allow each user to be authenticated, without the overhead of a separate database connection, and preserve the identity of the real user through the middle tier.

As long as the client authenticates itself with the middle tier and the middle tier authenticates itself with the database, and the middle tier is authorized to act on behalf of the client by the administrator, client identities can be maintained all the way into the database without compromising the security of the client.

The design of a secure three-tiered architecture is developed around a set of three trust zones. The first trust zone, the client trust zone, is Web clients connecting to a Web/application server. The clients can be authenticated by the middle tier using any means: password, cryptographic token, etc. This method may be entirely different from the method used to establish the other trust zones.

The next trust zone is the application server trust region. The data server verifies the identity of the application server and trusts it to pass the correct identity of the client. The third trust zone is the data server interaction with the authorization server to obtain the roles assigned to the client and the application server.

The application server creates a primary session for itself once it connects to a server. It authenticates itself in the normal manner to the database creating the application server trust zone. The application server identity is now well known and trusted to the data server.

When the application verifies the identity of a client connecting to the application server, it creates the first trust zone. The application server now needs a session handle for the client so that it can service client requests. The middle-tier process allocates a session handle and then sets the following attributes of the client using OCIAttrSet():

OCI_ATTR_USERNAME to set the database user name of the client.

OCI_ATTR_PROXY_CREDENTIALS to indicate the authenticated application making the proxy request.

If the application server wants to specify a list of roles that are to be activated after it connects as the client, it can call `OCIAttrSet()` with the attribute OCI_ATTR_INITIAL_CLIENT_ROLES and an array of strings that contains the list of roles prior to `OCISessionBegin()`. Then the role establishment as well as the verification of the proxy capability happens *in one round trip.* The `OCISessionBegin()` call will fail if the application server is not allowed to proxy on behalf of the client by the administrator or if the application server is not allowed to activate the specified roles.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals,* the chapter on Establishing Security Policies

Attributes for Middle-tier Applications

The following attributes allow you to specify the external name and initial privileges of a client. The new types of credentials are used by applications as alternative means of identifying or authenticating the client.

OCI_CRED_PROXY

Use OCI_CRED_PROXY as the type of credentials when an application server starts a session on behalf of a client, rather than OCI_CRED_RDBMS (database username and password required) or OCI_CRED_EXT (externally provided credentials).

OCI_ATTR_PROXY_CREDENTIALS

Use this attribute to specify the credentials of the application server for use in the authentication of the client. You pass the session handle of the application server:

```
OCIAttrSet(OCISession *session_handle,
           OCI_HTYPE_SESSION,
           OCISession *application_server_session_handle,
           (ub4) 0,
           OCI_ATTR_PROXY_CREDENTIALS,
           OCIError *error_handle);
```

OCI_ATTR_DISTINGUISHED_NAME

Your applications can use the distinguished name contained within a X.509 certificate as the login name of the client, rather than the database user name.

To pass over the distinguished name of the client, the middle-tier server would call `OCIAttrSet()`, passing OCI_ATTR_DISTINGUISHED_NAME:

```
OCIAttrSet(OCISession *session_handle,
```

```
    OCI_HTYPE_SESSION,
    lxstp *distinguished_name,
    (ub4) 0,
    OCI_ATTR_DISTINGUISHED_NAME,
    OCIError *error_handle);
```

OCI_ATTR_CERTIFICATE

This method is similar to that of distinguished name above. However, instead of the distinguished name, the entire X.509 certificate is passed by the middle-tier server to the database and the database retrieves the distinguished name from it.

To pass over the entire certificate, the middle tier would call OCIAttrSet(), passing OCI_ATTR_CERTIFICATE:

```
OCIAttrSet(OCISession *session_handle,
    OCI_HTYPE_SESSION,
    ub1 *certificate,
    ub4 certificate_length,
    OCI_ATTR_CERTIFICATE,
    OCIError *error_handle);
```

If the certificate type is to be passed over with the certificate, the middle tier would call OCIAttrSet(), passing OCI_ATTR_CERTIFICATE_TYPE:

```
OCIAttrSet(OCISession *session_handle,
    OCI_HTYPE_SESSION,
    ub1 *certificate_type,
    ub4 certificate_type_length,
    OCI_ATTR_CERTIFICATE_TYPE,
    OCIError *error_handle);
```

If the type is not specified, then the server will use its default certificate type of X.509.

OCI_ATTR_INITIAL_CLIENT_ROLES

Use the OCI_ATTR_INITIAL_CLIENT_ROLES attribute to specify a role or roles that the client is to initially possess when the application server connects to the Oracle server on its behalf. To enable a set of roles, the function OCIAttrSet() is called with the attribute, an array of null-terminated strings and the number of strings in the array:

```
OCIAttrSet(OCISession *session_handle,
        OCI_HTYPE_SESSION,
        text ** role_array,
```

```
                    ub4 number_of_strings,
                    OCI_ATTR_INITIAL_CLIENT_ROLES,
                    OCIError *error_handle);
```

OCI_ATTR_CLIENT_IDENTIFIER

To support the Global Application Context, the client can set the user identifier in
the session handle at any time in the session. Use the attribute
OCI_ATTR_CLIENT_IDENTIFIER in the call to OCIAttrSet() to do this. Then,
on the next request to the server, the information is propagated and stored in the
server session. For example:

```
OCIAttrSet(sesssion,
    OCI_HTYPE_SESSION,
    (dvoid *) "appuser1",
    (ub4)strlen("appuser1"),
    OCI_ATTR_CLIENT_IDENTIFIER,
    OCIError *error_handle);
```

Middle-tier Example

As an example, the code might look like the following:

```
...
*OCIEnv *environment_handle;
OCIServer *data_server_handle;
OCIError *error_handle;
OCISvcCtx *application_server_service_handle;
text *client_roles[2];
OCISession *first_client_session_handle, second_client_session_handle;
...
/*
** General initialization and allocation of contexts.
*/

(void) OCIInitialize((ub4) OCI_DEFAULT,
    (dvoid *)0,
    (dvoid * (*)(dvoid *, size_t)) 0,
    (dvoid * (*)(dvoid *, dvoid *, size_t))0,
    (void (*)(dvoid *, dvoid *)) 0 );
(void) OCIEnvInit( (OCIEnv **) &environment_handle, OCI_DEFAULT, (size_t) 0,
    (dvoid **) 0 );
(void) OCIHandleAlloc( (dvoid *) environment_handle, (dvoid **) &error_handle,
    OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0);
/*
```

```
** Allocate and initialize the server and service contexts used by the
** application server.
*/

(void) OCIHandleAlloc( (dvoid *) environment_handle,
     (dvoid **)&data_server_handle, OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0);
(void) OCIHandleAlloc( (dvoid *) environment_handle, (dvoid **)
     &application_server_service_handle, OCI_HTYPE_SVCCTX, (size_t) 0,
     (dvoid **) 0);
(void) OCIAttrSet((dvoid *) application_server_service_handle,
     OCI_HTYPE_SVCCTX, (dvoid *) data_server_handle, (ub4) 0, OCI_ATTR_SERVER,
     error_handle);
/*
** Authenticate the application server. In this case, external authentication is
** being used.
*/

(void) OCIHandleAlloc((dvoid *) environment_handle,
     (dvoid **)&application_server_session_handle, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0, (dvoid **) 0);
checkerr(error_handle, OCISessionBegin(application_server_service_handle,
     error_handle, application_server_session_handle, OCI_CRED_EXT,
     OCI_DEFAULT));
/*
** Authenticate the first client ** Note that no password is specified by the
** application server for the client as it is trusted.
*/

(void) OCIHandleAlloc((dvoid *) environment_handle,
     (dvoid **)&first_client_session_handle, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0,(dvoid **) 0);
(void) OCIAttrSet((dvoid *) first_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (dvoid *) "jeff", (ub4) strlen("jeff"),
     OCI_ATTR_USERNAME, error_handle);
/*
** In place of specifying a password, pass the session handle of the application
** server instead.
*/

(void) OCIAttrSet((dvoid *) first_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (dvoid *) application_server_session_handle,
     (ub4) 0, OCI_ATTR_PROXY_CREDENTIALS, error_handle);
(void) OCIAttrSet((dvoid *) first_client_session_handle,
     (ub4) OCI_HTYPE_SESSION, (dvoid *) "jeff@VeryBigBank.com",
     (ub4) strlen("jeff@VeryBigBank.com"), OCI_ATTR_EXTERNAL_NAME,
```

```
              error_handle);
          /*
          ** Establish the roles that the application server can use as the client.
          */

          client_roles[0] = (text *) "TELLER"; client_roles[1] = (text *) "SUPERVISOR";
          (void) OCIAttrSet((dvoid *) first_client_session_handle,
              OCI_ATTR_INITIAL_CLIENT_ROLES, error_handle);
          checkerr(error_handle, OCISessionBegin(application_server_service_handle,
              error_handle, first_client_session_handle, OCI_CRED_PROXY, OCI_DEFAULT));
          /*
          ** To start a session as another client, the application server would do the
          ** following. It should be
          ** noted this code is unchanged from the current way of doing session switching.
          */

          (void) OCIHandleAlloc((dvoid *) environment_handle,
              (dvoid **)&second_client_session_handle, (ub4) OCI_HTYPE_SESSION,
              (size_t) 0, (dvoid **) 0);
          (void) OCIAttrSet((dvoid *) second_client_session_handle,
              (ub4) OCI_HTYPE_SESSION, (dvoid *) "mutt", (ub4) strlen("mutt"),
              OCI_ATTR_USERNAME, error_handle);
          (void) OCIAttrSet((dvoid *) second_client_session_handle,
              (ub4) OCI_HTYPE_SESSION, (dvoid *) application_server_session_handle,
              (ub4) 0, OCI_ATTR_PROXY_CREDENTIALS, error_handle);
          (void) OCIAttrSet((dvoid *) second_client_session_handle,
              (ub4) OCI_HTYPE_SESSION, (dvoid *) "mutt@VeryBigBank.com",
              (ub4) strlen("mutt@VeryBigBank.com"), OCI_ATTR_EXTERNAL_NAME,
              error_handle);
          /*
          ** Note that the application server has not specified any initial roles to have
          ** as the second client.
          */

          checkerr(error_handle, OCISessionBegin(application_server_service_handle,
              error_handle, second_client_session_handle, OCI_CRED_PROXY, OCI_DEFAULT));
          /*
          ** To switch to the first user, the application server would apply the session
          ** handle obtained by the first
          ** OCISessionBegin() call. This is the same as is currently done.
          */

          (void) OCIAttrSet((dvoid *)application_server_service_handle,
              (ub4) OCI_HTYPE_SVCCTX, (dvoid *)first_client_session_handle,
              (ub4)0, (ub4)OCI_ATTR_SESSION, error_handle);
```

```
/*
** After doing some operations, the application server might want to switch to
** the second client. That
** would be done by the following call:
*/

(void) OCIAttrSet((dvoid *)application_server_service_handle,
    (ub4) OCI_HTYPE_SVCCTX,
    (dvoid *)second_client_session_handle, (ub4)0, (ub4)OCI_ATTR_SESSION,
    error_handle);
/*
** and then do operations as that client
*/
...
```

Attributes for Authentication

A middle-tier can ask the database server to authenticate a client on its behalf by validating the password of the client rather than doing the authentication itself.

For more complete information on this topic:

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals,* the chapter on Establishing Security Policies

While it may appear that this is essentially the same as a client/data server connection, it is different. The client does not have to have Oracle software installed on the client's system to be able to perform database operations.

To pass over the password of the client, the application server supplies `OCIAttrSet()` with the following data, using the existing attribute OCI_ATTR_PASSWORD:

```
OCIAttrSet(OCISession *session_handle,
    OCI_HTYPE_SESSION,
    lxstp *password,
    (ub4) 0,
    OCI_ATTR_PASSWORD,
    OCIError *error_handle);
```

Externally Initialized Context

An externally initialized context is an application context whose attributes can be initialized from OCI. Use the SQL statement CREATE CONTEXT to create a context namespace in the server with the option INITIALIZED EXTERNALLY.

Then, you can initialize an OCI interface when establishing a session using OCIAttrSet() and OCISessionBegin(). After the session is established, you can issue commands to write to any attributes inside the namespace only with the PL/SQL package designated in the CREATE CONTEXT statement.

You are able to send default values along with other session attributes via the OCISessionBegin() call, thus reducing server round trips. For a definition and further discussion of application contexts:

> **See Also:**
>
> - *Oracle9i Application Developer's Guide - Fundamentals,* the chapter on Establishing Security
>
> - *Oracle9i SQL Reference,* the CREATE CONTEXT statement and the SYS_CONTEXT function

OCI Attributes for Externally Initialized Context

The client application you develop can set application contexts explicitly in the session handle, before authentication, using the following attributes in OCI functions:

OCI_ATTR_APPCTX_SIZE

Use this to initialize the context array size with the desired number of context attributes in the OCIAttrSet() call. (The argument *size* is a **ub4** datatype.)

```
OCIAttrSet(session, OCI_HTYPE_SESSION,
          (dvoid *)&size, (ub4)0, OCI_ATTR_APPCTX_SIZE, error_handle);
```

OCI_ATTR_APPCTX_LIST

Use this attribute to get a handle on the application context list descriptor for the session in the OCIAttrGet() call. (Where the parameter *ctxl_desc* must be of datatype **OCIParam ***).

```
OCIAttrGet(session, OCI_HTYPE_SESSION,
          (dvoid *)&ctxl_desc, (ub4)0, OCI_ATTR_APPCTX_LIST, error_handle);
```

Then use the application context list descriptor to obtain an individual descriptor for the i-th application context in a call to `OCIParamGet()`:

```
OCIParamGet(ctxl_desc, OCI_DTYPE_PARAM, error_handle,(dvoid **)&ctx_desc, i);
```

Where *ctx_desc* is of **OCIParam** * datatype.

Session Handle Attributes Used to Set an Externally Initialized Context

Set the appropriate values of the application context using these three attributes:

- OCI_ATTR_APPCTX_NAME to set the namespace of the context, which must be a valid SQL identifier.

- OCI_ATTR_APPCTX_ATTR to set an attribute name in the given context, a non-case sensitive string of up to 30 bytes.

- OCI_ATTR_APPCTX_VALUE to set the value of an attribute in the given context.

Each namespace can have many attributes, each of which has one value. Here are the three calls you can use to set them:

```
OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
     (dvoid *)ctx_name, sizeof(ctx_name), OCI_ATTR_APPCTX_NAME, error_handle);

OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
    (dvoid *)attr_name, sizeof(attr_name), OCI_ATTR_APPCTX_ATTR, error_handle);

OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
     (dvoid *)value, sizeof(value), OCI_ATTR_APPCTX_VALUE, error_handle);
```

Note that only character type is supported, because application context operations are based on the VARCHAR2 datatype.

> **See Also:** "User Session Handle Attributes" on page A-17 for more information about the attributes discussed in this section

Using OCISessionBegin() with an Externally initialized Context

When you then call `OCISessionBegin()`, the context set in the session handle will be pushed to the server. After this call, no more contexts will be propagated to the server session.

Here is a code example to illustrate use of these calls and attributes:

```
#include <stdio.h>
```

```
                #include <stdlib.h>
                #include <string.h>
                #include <oci.h>

                static text *username = (text *) "SCOTT";
                static text *password = (text *) "TIGER";

                static OCIEnv *envhp;
                static OCIError *errhp;

                int main(/*_ int argc, char *argv[] _*/);

                static sword status;

                int main(argc, argv)
                int argc;
                char *argv[];
                {

                  OCISession *authp = (OCISession *) 0;
                  OCIServer *srvhp;
                  OCISvcCtx *svchp;
                  OCIDefine *defnp = (OCIDefine *) 0;
                  dvoid      *parmdp;
                  ub4         ctxsize;
                  OCIParam  *ctxldesc;
                  OCIParam  *ctxedesc;

                  (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                                       (dvoid * (*)(dvoid *, size_t)) 0,
                                       (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                                       (void (*)(dvoid *, dvoid *)) 0 );

                  (void) OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0,
                                     (dvoid **) 0 );

                  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                                     (size_t) 0, (dvoid **) 0);

                  /* server contexts */
                  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                                     (size_t) 0, (dvoid **) 0);

                  (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                                     (size_t) 0, (dvoid **) 0);
```

```
   (void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);

   /* set attribute server context in the service context */
   (void) OCIAttrSet( (dvoid *) svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp,
                     (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp);

   (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&authp,
                         (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);

/****************************************/

   /* set app ctx size to 2 because we want to set up 2 application contexts */

   ctxsize = 2;

   /* set up app ctx buffer */
   (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                    (dvoid *) &ctxsize, (ub4) 0,
                    (ub4) OCI_ATTR_APPCTX_SIZE, errhp);

   /* retrieve the list descriptor */
   (void) OCIAttrGet((dvoid *)authp, (ub4) OCI_HTYPE_SESSION,
                     (dvoid *)&ctxldesc, 0, OCI_ATTR_APPCTX_LIST, errhp );

   /* retrieve the 1st ctx element descriptor */
   (void) OCIParamGet(ctxldesc, OCI_DTYPE_PARAM, errhp, (dvoid**)&ctxedesc, 1);

   (void) OCIAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                    (dvoid *) "HR", (ub4) strlen((char *)"HR"),
                    (ub4) OCI_ATTR_APPCTX_NAME, errhp);

   (void) OCIAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                    (dvoid *) "ATTR1", (ub4) strlen((char *)"ATTR1"),
                    (ub4) OCI_ATTR_APPCTX_ATTR, errhp);

   (void) OCIAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                    (dvoid *) "VALUE1", (ub4) strlen((char *)"VALUE1"),
                    (ub4) OCI_ATTR_APPCTX_VALUE, errhp);

   /* set second context */
   (void) OCIParamGet(ctxldesc, OCI_DTYPE_PARAM, errhp, (dvoid**)&ctxedesc, 2);

   (void) OCIAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                    (dvoid *) "HR", (ub4) strlen((char *)"HR"),
```

```
                              (ub4) OCI_ATTR_APPCTX_NAME, errhp);

       (void) OCIAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                     (dvoid *) "ATTR2", (ub4) strlen((char *)"ATTR2"),
                     (ub4) OCI_ATTR_APPCTX_ATTR, errhp);

       (void) OCIAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                     (dvoid *) "VALUE2", (ub4) strlen((char *)"VALUE2"),
                     (ub4) OCI_ATTR_APPCTX_VALUE, errhp);

   /**************************************/
     (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                     (dvoid *) username, (ub4) strlen((char *)username),
                     (ub4) OCI_ATTR_USERNAME, errhp);

     (void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                     (dvoid *) password, (ub4) strlen((char *)password),
                     (ub4) OCI_ATTR_PASSWORD, errhp);

     OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,(ub4) OCI_DEFAULT);

   }
   ...
```

Oracle9*i* Application Server Cache

Oracle9*i* Application Server Cache improves the scalability and performance of applications that access Oracle databases by caching frequently used data on a middle-tier system. With Oracle9*i* Application Server Cache, your applications can process several times as many requests as their original capacity. In addition, you do not need to modify your existing applications to use this product. Oracle9*i* Application Server Cache is transparent to your end users.

> **See Also:** *Oracle9i Application Server Cache Concepts and Administration Guide,* on the iAS Documentation CD, or at the web site:
>
> http://www.oracle.com/
>
> Click on Oracle Technology Network, Documentation, Oracle9*i* Application Server, and Oracle9*i* Application Server Documentation.

9

OCI Programming Advanced Topics

This chapter introduces advanced programming topics, including the following:

- Thread Safety
- The OCIThread Package
- Connection Pooling
- User-Defined Callback Functions
- Application Failover Callbacks
- OCI and Advanced Queuing
- Publish-Subscribe Notification

Thread Safety

The thread safety feature of the Oracle database server and OCI libraries allows developers to use the OCI in a multithreaded environment. With thread safety, OCI code can be reentrant, with multiple threads of a user program making OCI calls without side effects from one thread to another.

> **Note:**   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 multithreaded applications.

Advantages of OCI Thread Safety

The implementation of thread safety in the Oracle Call Interface has 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 multithreaded programs do not pay a performance penalty for using thread-safe OCI calls.

- 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.

Thread Safety and Three-Tier Architectures

In addition to client-server applications, where the client can be a multithreaded program, a typical use of multithreaded applications is in three-tier (also called client-agent-server) architectures. In this architecture the client is concerned only 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 multithreaded application server, with each thread

serving a client application. In an Oracle environment this application server is an OCI or precompiler program.

Basic Concepts of Multithreaded Development

Threads are lightweight processes that 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.

Once spawned, threads run asynchronously to one another. They can access common data elements and make OCI calls in any order. Because of this shared access to data elements, a mechanism is required to maintain the integrity of data being accessed by multiple threads.

The mechanism to manage data access takes the form of *mutexes* (mutual exclusivity locks), that ensure that no conflicts arise between multiple threads that are accessing shared resources within an application. In OCI, mutexes are granted on a per-environment-handle basis.

Implementing Thread Safety

In order to take advantage of thread safety, an application must be running on a thread-safe platform. Then the application must tell the OCI layer that the application is running in multithreaded mode, by specifying OCI_THREADED for the *mode* parameter of the opening call to OCIEnvCreate().

Once OCIEnvCreate() is called with OCI_THREADED, all subsequent calls to OCIEnvCreate() must also be made with OCI_THREADED.

> **Note:** Applications running on non-thread-safe platforms should not pass a value of OCI_THREADED to OCIInitialize() or OCIEnvCreate().

If an application is single-threaded, whether or not the platform is thread safe, the application should pass a value of OCI_DEFAULT to OCIInitialize() or OCIEnvCreate(). Single-threaded applications which run in OCI_THREADED mode may incur performance hits.

If a multithreaded application is running on a thread-safe platform, the OCI library will manage mutexing for the application on a per-environment-handle basis. If the

application programmer desires, this application can override this feature and maintain its own mutexing scheme. This is done by specifying a value of OCI_NO_MUTEX to the OCIEnvCreate() call.

The following three scenarios are possible, depending on how many connections exist per environment handle, and how many threads will be spawned per connection.

1. If an application has multiple environment handles, but each only has one thread (one session exists per environment handle), no mutexing is required.

2. If an application running in OCI_THREADED mode maintains one or more environment handles, each of which has multiple connections, it also has the following options:

   ■ Pass a value of OCI_NO_MUTEX for the *mode* of OCIEnvCreate(). In this case the application must mutex OCI calls by made on the same environment handle itself. This has the advantage that the mutexing scheme can be optimized based on the application design. The programmer must also insure that only one OCI call is in process on the environment handle connection at any given time.

   ■ Pass a value of OCI_DEFAULT to OCIEnvCreate(). In this case, the OCI library automatically gets a mutex on every OCI call on the same environment handle.

Note: The bulk of processing of an OCI call happens on the server, so if two threads using OCI calls go to the same connection, then one them could be blocked while the other finishes processing at the server.

Mixing 7.x and Later Release OCI Calls

If an application is mixing later release and 7.x OCI calls, and the application has been initialized as thread safe (with the appropriate calls of the later release), it is not necessary to call opinit() to achieve thread safety. The application will get 7.x behavior on any subsequent 7.x function calls.

Multithreading Example

See cdemothr.c in the demo directory for an example of a multithreading application.

The OCIThread Package

The OCIThread package provides a number of commonly used threading primitives for use by Oracle customers. It offers a portable interface to threading capabilities native to various platforms. It does not implement threading on platforms that do not have native threading capability.

OCIThread does not provide a portable implementation of multithreaded facilities. It only serves as a set of portable covers for native multithreaded facilities. Therefore, platforms that do not have native support for multithreading will only be able to support a limited implementation of OCIThread. As a result, products that rely on all of OCIThread's functionality will not port to all platforms. Products that must port to all platforms must use only a subset of OCIThread's functionality. This issue is discussed further in later sections of this document.

The OCIThread API is split into three main parts. Each part is described briefly here. The following subsections describe each in greater detail.

> **See Also:**
>
> - See "Using the OCIThread Package" on page 9-11 for important additional information
>
> - Detailed descriptions of OCIThread functions, including syntax, parameters lists, and other comments can be found in "Thread Management Functions" on page 16-114

- Initialization and Termination

  These calls deal with the initialization and termination of OCIThread. Initialization of OCIThread initializes the OCIThread context that is a member of the OCI environment or user session handle. This context is required for other OCIThread calls.

- Passive Threading Primitives

  The *passive* threading primitives include primitives to manipulate mutual exclusion (mutex) locks, thread ID's, and thread-specific data keys.

  The reason that these primitives are described as passive is that while their specifications allow for the existence of multiple threads, they do not require it. This means that it is possible for these primitives to be implemented according to specification in both single-threaded and multithreaded environments.

As a result, `OCIThread` clients that use only these primitives will not require the existence of multiple threads in order to work correctly, i.e., they will be able to work in single-threaded environments without branching code.

■ Active Threading Primitives

Active threading primitives include primitives dealing with the creation, termination, and other manipulation of threads.

The reason that these primitives are described as *active* is that they can only be used in true multithreaded environments. Their specifications explicitly require that it be possible to have multiple threads. If you need to determine at runtime whether or not you are in a multithreaded environment, call `OCIThreadIsMulti()` before calling an `OCIThread` active primitive.

Initialization and Termination

The types and functions described in this section are associated with the initialization and termination of the `OCIThread` package. `OCIThread` must be properly initialized before any of its functionality can be used. `OCIThread`'s process initialization function, `OCIThreadProcessInit()`, must be called with care, as described below.

The observed behavior of the initialization and termination functions is the same regardless of whether `OCIThread` is in single-threaded or multithreaded environment.

OCIThread Context

Most calls to `OCIThread` functions take the OCI environment or user session handle as a parameter. The `OCIThread` context is part of the OCI environment or user session handle and it must be initialized by calling `OCIThreadInit()`. Termination of the `OCIThread` context occurs by calling `OCIThreadTerm()`.

> **Note:** The `OCIThread` context is an opaque data structure. Do not attempt to examine the contents of the context.

The following functions are used to implement thread initialization and termination. Detailed descriptions of each function can be found in "Thread Management Functions" on page 16-114.

Function	Purpose
`OCIThreadProcessInit()`	Performs OCIThread process initialization.
`OCIThreadInit()`	Initializes OCIThread context.
`OCIThreadTerm()`	Terminates the OCIThread layer and frees context memory.
`OCIThreadIsMulti()`	Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment.

Passive Threading Primitives

The passive threading primitives deal with the manipulation of mutex, thread ID's, and thread-specific data. Since the specifications of these primitives do not require the existence of multiple threads, they can be used both on multithreaded and single-threaded platforms.

OCIThreadMutex

The type **OCIThreadMutex** is used to represent a mutual exclusion lock (mutex). A mutex is typically used for one of two purposes:

- to ensure that only one thread accesses a given set of data at a time

- to ensure that only one thread executes a given critical section of code at a time

Mutex pointers can be declared as parts of client structures or as stand-alone variables. Before they can be used, they must be initialized using `OCIThreadMutexInit()`. Once they are no longer needed, they must be destroyed using `OCIThreadMutexDestroy()`. A mutex pointer must not be used after it is destroyed.

A thread can acquire a mutex by using `OCIThreadMutexAcquire()`. This ensures that only one thread at a time is allowed to hold a given mutex. A thread that holds a mutex can release it by calling `OCIThreadMutexRelease()`.

OCIThreadKey

The type **OCIThreadKey** can be thought of as a process-wide variable that has a thread-specific value. What this means is that all the threads in a process can use any given key. However, each thread can examine or modify that key independently of the other threads. The value that a thread sees when it examines

the key will always be the same as the value that it last set for the key. It will not see any values set for the key by the other threads.

The type of the value held by a key is a **dvoid** * generic pointer.

Keys can be created using `OCIThreadKeyInit()`. When a key is created, its value is initialized to `NULL` for all threads.

A thread can set a key's value using `OCIThreadKeySet()`. A thread can get a key's value using `OCIThreadKeyGet()`.

The `OCIThread` key functions will save and retrieve data specific to the thread. When clients maintain a pool of threads and assign the threads to different tasks, it may not be appropriate for a task to use `OCIThread` key functions to save data associated with it.

Here is a scenario of how things can fail: A thread is assigned to execute the initialization of a task. During the initialization, the task stored some data related to it in the thread using `OCIThread` key functions.

After the initialization, the thread is returned back to the threads pool. Later, the threads pool manager assigned another thread to perform some operations on the task, and the task needs to retrieve the data it stored earlier in initialization. Since the task is running in another thread, it will not be able to retrieve the same data. Applications that use thread pools should be aware of this and be cautious when using `OCIThread` key functions.

OCIThreadKeyDestFunc

OCIThreadKeyDestFunc is the type of a pointer to a key's destructor routine. Keys can be associated with a destructor routine when they are created (see `OCIThreadKeyInit()`).

A key's destructor routine will be called whenever a thread that has a non-`NULL` value for the key terminates.

The destructor routine returns nothing and takes one parameter. The parameter will be the value that was set for key when the thread terminated.

The destructor routine is guaranteed to be called on a thread's value in the key after the termination of the thread and before process termination. No more precise guarantee can be made about the timing of the destructor routine call; thus no code in the process may assume any post-condition of the destructor routine. In particular, the destructor is not guaranteed to execute before a join call on the terminated thread returns.

OCIThreadId

OCIThreadId is the type that will be used to identify a thread. At any given time, no two threads will ever have the same **OCIThreadId**. However, **OCIThreadId** values can be recycled; i.e., once a thread dies, a new thread may be created that has the same **OCIThreadId** as the one that died. In particular, the thread ID must uniquely identify a thread T within a process, and it must be consistent and valid in all threads U of the process for which it can be guaranteed that T is running concurrently with U. The thread ID for a thread T must be retrievable within thread T. This will be done using OCIThreadIdGet().

The **OCIThreadId** type supports the concept of a NULL thread ID: the NULL thread ID will never be the same as the ID of an actual thread.

Passive Threading Functions

The following functions are used to manipulate mutexes, thread keys and thread IDs.

> **See Also:** Complete descriptions of each function can be found in

Function	Purpose
OCIThreadMutexInit()	Allocates and initializes a mutex.
OCIThreadMutexDestroy()	Destroys and deallocates a mutex.
OCIThreadMutexAcquire()	Acquires a mutex for the thread in which it is called.
OCIThreadMutexRelease()	Releases a mutex.
OCIThreadKeyInit()	Allocates and initializes a key.
OCIThreadKeyDestroy()	Destroys and deallocates a key.
OCIThreadKeyGet()	Gets the calling thread's current value for a key.
OCIThreadKeySet()	Sets the calling thread's value for a key.
OCIThreadIdInit()	Allocates and initializes a thread ID.
OCIThreadIdDestroy()	Destroys and deallocates a thread ID.
OCIThreadIdSet()	Sets on thread ID to another.
OCIThreadIdSetNull()	Nulls a thread ID.

Function	Purpose
OCIThreadIdGet()	Retrieves a thread ID for the thread in which it is called.
OCIThreadIdSame()	Determines if two thread IDs represent the same thread.
OCIThreadIdNull()	Determines if a thread ID is NULL.

Active Threading Primitives

The active threading primitives deal with the manipulation of actual threads. Because the specifications of most of these primitives require that it be possible to have multiple threads, they work correctly only in the enabled OCIThread; In the disabled OCIThread, they always return failure. The exception is OCIThreadHandleGet(); it may be called in a single-threaded environment, in which case it has no effect.

Active primitives should only be called by code running in a multithreaded environment. You can call OCIThreadIsMulti() to determine whether the environment is multithreaded or single-threaded.

OCIThreadHandle

Type **OCIThreadHandle** is used to manipulate a thread in the active primitives: OCIThreadJoin() and OCIThreadClose(). A thread handle opened by OCIThreadCreate() must be closed in a matching call to OCIThreadClose(). A thread handle is invalid after the call to OCIThreadClose().

The distinction between a thread ID and a thread handle in OCIThread usage follows the distinction between the thread ID and the thread handle on Windows NT. On many platforms, the underlying native types are the same.

Active Threading Functions

The following functions are used to implement active threading.

> **See Also:** Complete descriptions of the functions are available in "Thread Management Functions" on page 16-114

Function	Purpose
OCIThreadHndInit()	Allocates and initializes a thread handle.

Function	Purpose
`OCIThreadHndDestroy()`	Destroys and deallocates a thread handle.
`OCIThreadCreate()`	Creates a new thread.
`OCIThreadJoin()`	Allows the calling thread to join with another.
`OCIThreadClose()`	Closes a thread handle.
`OCIThreadHandleGet()`	Retrieves a thread handle.

Using the OCIThread Package

This section summarizes some of the more important details relating to the use of OCIThread.

Process initialization
`OCIThread` only requires that the process initialization function (`OCIThreadProcessInit()`) be called when `OCIThread` is being used in a multithreaded application. Failing to call `OCIThreadProcessInit()` in a single-threaded application is not an error.

OCIThread initialization
Separate calls to `OCIThreadInit()` will all return the same `OCIThread` context.

Also, remember that each call to `OCIThreadInit()` must eventually be matched by a call to `OCIThreadTerm()`.

Active versus Passive Threading Primitives
`OCIThread` client code written without using any active primitives can be compiled and used without modifications on both single-threaded and multithreaded platforms.

`OCIThread` client code written using active primitives will only work correctly on multithreaded platforms. In order to write a version of the same application to run on single-threaded platform, it is necessary to branch your code, whether by branching versions of the source file or by branching at runtime with the `OCIThreadIsMulti()` call.

Example Using OCIThread

The following code sample illustrates the use of OCIThread.

> **See Also:** For a listing of the complete demonstration programs,
> see Appendix B, "OCI Demonstration Programs"

```
static OCIEnv *envhp;
static OCIError *errhp;
void parent(argc, argv)
sb4 argc;
text **argv;
{
  OCIThreadId *tidArr[5];
  OCIThreadHandle *tHndArr[5];
  ub4 i;
  OCIThreadKey *key;
  (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                       (dvoid * (*)(dvoid *, size_t)) 0,
                       (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                       (void (*)(dvoid *, dvoid *)) 0 );
  (void) OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0,
                     (dvoid **) 0 );
   (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp,
                          OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0);
  OCIThreadProcessInit();
  OCIThreadInit(envhp, errhp);
  OCIThreadKeyInit(envhp, errhp, &key, (OCIThreadKeyDestFunc) NULL);
  for (i=0; i<5; i++)
    {
      OCIThreadIdInit(envhp, errhp, &(tidArr[i]));
      OCIThreadHndInit(envhp, errhp, &(tHndArr[i]));
    }
  for (i=0; i<5; i++)
    OCIThreadCreate(envhp, errhp, child, (dvoid *)key,
                             tidArr[i], tHndArr[i]);
  for (i=0; i<5; i++)
    {
      OCIThreadJoin(envhp, errhp, tHndArr[i]);
      OCIThreadClose(envhp, errhp, tHndArr[i]);
    }
  for (i=0; i<5; i++)
    {
       OCIThreadIdDestroy(envhp, errhp, &(tidArr[i]));
       OCIThreadHndDestroy(envhp, errhp, &(tHndArr[i]));
    }
  OCIThreadKeyDestroy(envhp, errhp, &key);
```

```
      OCIThreadTerm(envhp, errhp);
}
void child(arg)
dvoid *arg;
{
  OCIThreadKey *key = (OCIThreadKey *)arg;
  OCIThreadId *tid;
  dvoid *keyval;
  OCIThreadIdInit(envhp, errhp, &tid);
  OCIThreadIdGet(envhp, errhp, tid);
   if (OCIThreadKeySet(envhp, errhp, key, (dvoid *)tid) != OCI_SUCCESS)
      printf("Could not set value for key\n");
   if (OCIThreadKeyGet(envhp, errhp, key, &keyval) !=OCI_SUCCESS)
      printf("Could not retrieve value for key\n");
   if (keyval != (dvoid *)tid)
     printf("Incorrect value from key after setting it\n");
  /* we must destroy thread id */
  OCIThreadIdDestroy(envhp, errhp, &tid);
}
```

Connection Pooling

Connection pooling is the use of a group (the pool) of reusable physical connections by several sessions, in order to balance loads. The management of the pool is done by OCI, not the application. Applications that can use connection pooling include middle-tier applications for web application servers and e-mail servers.

A sample usage of this feature would be in a web application server connected to a back-end Oracle database. Suppose that a web application server gets several concurrent requests for data from the database server. Typically, it would have to explicitly manage the connections to the database. However, by using this functionality, it can leave that task to OCI. The application can create a pool (or a set of pools) in each environment during initialization.

OCI Connection Pooling Concepts

Oracle has several transaction monitor capabilities such as the fine grained management of database sessions and connections. This is done by separating the notion of database sessions (user handles) from connections (server handles). Using these OCI calls for session switching and session migration, it is possible for an application server or transaction monitor to multiplex several sessions over fewer physical connections, thus achieving a high degree of scalability by pooling of connections and back-end Oracle server processes.

Connection pooling further simplifies the session and connection separation by hiding the management of the physical connection pool from the end user, who has only to create the necessary database sessions. The connection pool itself is normally configured with a shared pool of physical connections, translating to a back-end server pool containing an identical number of dedicated server processes.

The number of physical connections is less than the number of database sessions in use by the application.The number of physical connections and back-end server processes are also reduced by using connection pooling. Thus many more database sessions can be multiplexed.

Similarities and Differences from Shared Server

Connection pooling on the middle-tier is similar to what shared server offers on the back end. Connection pooling makes a dedicated server instance behave like a shared server instance by managing the session multiplexing logic on the middle tier.

Thus, the pooling of dedicated server processes including incoming connections into the dedicated server processes is controlled by the connection pool on the

middle tier. The main difference between connection pooling and a shared server is that in the latter case, the connection from the client is normally to a dispatcher in the database instance. The dispatcher is responsible for directing the client request to an appropriate shared server. On the other hand, the physical connection from the connection pool is established directly from the middle-tier to the dedicated server process in the back-end server pool.

Connection pooling is beneficial only if the middle tier itself is multithreaded. Each thread could maintain a session to the database. The actual connections to the database are maintained by the connection pool and these connections (including the pool of dedicated database server processes) are shared among all the threads in the middle tier.

Stateless Sessions Versus Stateful Sessions

What connection pooling offers is stateless connections and stateful sessions. Users who need to work with a stateless session behavior can choose to pool the created OCI sessions by sharing service contexts (among multiple threads) on top of the existing connection pool. This can be done at application level by keeping a list of available service contexts and another list of service contexts that are in use in a serially accessed and modified data structure available to all threads.

Multiple Connection Pools

This advanced concept can be used for different database connections. Multiple connection pools can also be used when different priorities are assigned to users. Different service level guarantees can be implemented using connection pooling.

The following figure illustrates connection pooling, as it is described above:

Figure 9–1   OCI Connection Pooling
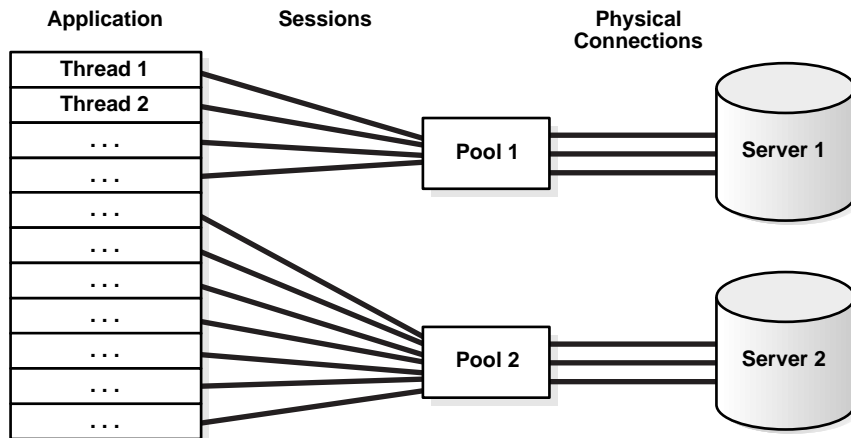
OCI Calls for Connection Pooling

The steps in using connection pooling in your application are:

Allocate the Pool Handle

Connection pooling requires that the pool handle OCI_HTYPE_CPOOL be allocated by OCIHandleAlloc(). Multiple pools can be created for a given environment handle.

For a single connection pool, here is an allocation example:

```
OCICPool *poolhp;
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &poolhp, OCI_HTYPE_CPOOL,
                      (size_t) 0, (dvoid **) 0));
```

Create the Connection Pool

The function OCIConnectionPoolCreate() initializes the connection pool handle. It has these IN parameters:

- *connMin*, the minimum number of connections to be opened when the pool is created.

- *connIncr*, the incremental number of connections to be opened when all the connections are busy and a call needs a connection. This increment is used only

when the total number of open connections is less than the maximum number of connections that can be opened in that pool.

- *connMax*, the maximum number of connections that can be opened in the pool. When the maximum number of connections are open in the pool, and all the connections are busy, if a call needs a connection, it will wait till it gets one. However, if the OCI_ATTR_CONN_NOWAIT attribute is set for the pool, an error is returned.

- A *poolUsername* and a *poolPasswd* to allow user sessions to transparently migrate between connections in the pool.

- In addition, an attribute OCI_ATTR_CONN_TIMEOUT can be set to time out the connections in the pool. Connections idle for more than this time are terminated periodically, to maintain an optimum number of open connections. If this attribute is not set, then the connections are never timed out.

All the above attributes can be configured dynamically. So the application has the flexibility of reading the current load (number of open connections and number of busy connections) and tuning these attributes appropriately.

If the pool attributes (*connMax, connMin, connIncr*) are to be changed dynamically, `OCIConnectionPoolCreate()` must be called with *mode* set to OCI_CPOOL_REINITIALIZE.

The OUT parameters *poolName* and *poolNameLen* will contain values to be used in subsequent `OCIServerAttach()` and `OCILogon2()` calls in place of the database name and the database name length arguments.

There is no limit on the number of pools that can be created by an application. Middle tier applications can take advantage of this feature and create multiple pools to connect to the same server or to different servers, to balance the load based on the specific needs of the application.

Here is an example of this call:

```
OCIConnectionPoolCreate((OCIEnv *)envhp,
                (OCIError *)errhp, (OCICPool *)poolhp,
                &poolName, &poolNameLen,
                (text *)database,strlen(database),
                (ub4) conMin, (ub4) conMax, (ub4) conIncr,
                (text *)pooluser,strlen(pooluser),
                (text *)poolpasswd,strlen(poolpasswd),
                OCI_DEFAULT));
```

Logon to the Database

Log on to the database for each thread using the `OCILogon2()` call, giving the user name and password, with parameter *mode* set to OCI_CPOOL. This gives a set of service context handles attached to the pool, which can be used by the threads to process the database calls.

The `OCIServerAttach()` call can be used instead, with *mode* set to OCI_CPOOL.

If the `OCILogon2()` call is made with a `null` password, OCI assumes that the logon is for a proxy user. It implicitly creates a proxy connection in such a case, using the pool user to authenticate the proxy user.

Here is an example using `OCILogon2()`:

```
for (i = 0; i < MAXTHREADS; ++i)
  {
    OCILogon2(envhp, errhp, &svchp[i],
                 "scott", 5,
                 "tiger", 5,
                 poolName, poolNameLen,
                 OCI_CPOOL));
  }
```

SGA Limitation in Connection Pooling

With OCI_CPOOL mode (connection pooling), the session memory (UGA) in the back-end database will come out of the SGA. This may require some SGA tuning on the back-end database to have a larger SGA if your application consumes more session memory than the SGA can accommodate. The memory tuning requirements for the back-end database will be similar to configuring the LARGE POOL in case of a shared server back end except that the instance is still in dedicated mode.

> **See Also:** For more information, see the section on configuring Shared Server in the *Oracle9i Database Performance Guide and Reference*

If you are still running into the SGA limitation, you should consider:

- Reducing the session memory consumption by having fewer open statements per session, or

- reducing the number of sessions in the back end by pooling sessions on the mid-tier or otherwise, or

- turning off connection pooling.

The application should avoid using dedicated database links on the back end with connection pooling.

If the back end is a dedicated server, effective connection pooling will not be possible because sessions using dedicated database links will be tied to a physical connection rendering that same connection unusable by other sessions. If your application uses dedicated database links and you do not see effective sharing of back-end processes among your sessions, you should consider using shared database links.

> **See Also:** For more information about distributed databases, see the section on shared database links in *Oracle9i Database Administrator's Guide*

Logoff from the Database

To logoff, use `OCILogoff()` or delete access to the data source by the use of `OCIServerDetach()`.

Destroy Connection Pool

Use `OCIConnectionPoolDestroy()` to destroy the connection pool.

Free the Pool Handle

The pool handle is freed using `OCIHandleFree()`.

These last three actions are illustrated in this code fragment:

```
for (i = 0; i < MAXTHREADS; ++i)
 {
   checkerr(errhp, OCILogoff((dvoid *) svchp[i], errhp));
 }
 checkerr(errhp, OCIConnectionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
 checkerr(errhp, OCIHandleFree((dvoid *)poolhp, OCI_HTYPE_CPOOL));
```

> **See Also:**
>
> - *"Connection Pool Handle Attributes"* on page A-21 for more information about the connection pooling attributes
>
> - `OCIConnectionPoolCreate()`, `OCILogon2()`, and `OCIConnectionPoolDestroy()` for more information about these functions

Alternative Interface for Connection Pooling

Another interface can be used if the application needs to set any special attributes on the user session handle and server handle. For such a requirement, applications need to do this:

Allocate all the handles (connection pool handle, server handles, session handles and service context handles).

Create the connection pool.

Call `OCIServerAttach()` with *mode* as OCI_CPOOL.

Call `OCISessionBegin()` with *mode* as OCI_DEFAULT or OCI_MIGRATE. The OCI_MIGRATE flag will be set internally in any case.

Credentials can be set to OCI_CRED_RDBMS or OCI_CRED_PROXY. If the credentials are set to OCI_CRED_PROXY, only username needs to be set on the session handle. (no explicit primary session needs to be created and OCI_ATTR_MIGSESSION need not be set).

Example of Connection Pooling

Here is an example of connection pooling:

```
#include <oci.h>

#define MAXTHREAD 10

static OCIError   *errhp;
static OCIEnv     *envhp;
static OCICPool   *poolhp;

static int employeeNum[MAXTHREAD];

static OraText *poolName;
static sb4 poolNameLen;
static text *database = (text *)"";
static text *username =(text *)"SCOTT";
static text *password =(text *)"TIGER";
static text *appusername =(text *)"APPUSER";
static text *apppassword =(text *)"APPPASSWD";

static ub4 conMin = 2;
static ub4 conMax = 5;
static ub4 conIncr = 1;
```

```
static void checkerr (OCIError *errhp, sword status);
static void threadFunction (dvoid *arg);

int main (void)
{
  int i = 0;

  OCIEnvCreate (&envhp, OCI_THREADED, (dvoid *)0,  (dvoid * (*)()) 0,
    (dvoid * (*)()) 0, (dvoid (*)()) 0, 0, (dvoid *)0);

  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                    (size_t) 0, (dvoid **) 0);


  (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &poolhp, OCI_HTYPE_CPOOL,
                        (size_t) 0, (dvoid **) 0);

  /* CREATE THE CONNECTION POOL */
  checkerr (errhp, OCIConnectionPoolCreate(envhp,
                    errhp,poolhp, &poolName, &poolNameLen,
                    database,strlen(database),
                    conMin, conMax, conIncr,
                    appusername,strlen(appusername),
                    apppassword,strlen(apppassword),OCI_DEFAULT));

  /* Multiple threads using the connection pool */
  {
    OCIThreadId      *thrid[MAXTHREAD];
    OCIThreadHandle *thrhp[MAXTHREAD];

    OCIThreadProcessInit ();
    checkerr (errhp, OCIThreadInit (envhp, errhp));
    for (i = 0; i < MAXTHREAD; ++i)
    {
      checkerr (errhp, OCIThreadIdInit (envhp, errhp, &thrid[i]));
      checkerr (errhp, OCIThreadHndInit (envhp, errhp, &thrhp[i]));
    }
    for (i = 0; i < MAXTHREAD; ++i)
    {
      employeeNum[i]=i;
      checkerr (errhp, OCIThreadCreate (envhp, errhp, threadFunction,
        (dvoid *) &employeeNum[i], thrid[i], thrhp[i]));
    }
    for (i = 0; i < MAXTHREAD; ++i)
    {
```

```
        checkerr (errhp, OCIThreadJoin (envhp, errhp, thrhp[i]));
        checkerr (errhp, OCIThreadClose (envhp, errhp, thrhp[i]));
        checkerr (errhp, OCIThreadIdDestroy (envhp, errhp, &(thrid[i])));
        checkerr (errhp, OCIThreadHndDestroy (envhp, errhp, &(thrhp[i])));
      }
      checkerr (errhp, OCIThreadTerm (envhp, errhp));
    } /* ALL THE THREADS ARE COMPLETE */

    checkerr(errhp, OCIConnectionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
    checkerr(errhp, OCIHandleFree((dvoid *)poolhp, OCI_HTYPE_CPOOL));
    checkerr(errhp, OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR));
} /* end of main () */

static void threadFunction (dvoid *arg)
{
    int empno = *(int *)arg;
    OCISvcCtx *svchp = (OCISvcCtx *) arg;
    text insertst1[256];
    OCIStmt *stmthp = (OCIStmt *)0;

    checkerr(errhp,OCILogon2(envhp, errhp, &svchp,
                           (CONST OraText *)username, strlen(username),
                           (CONST OraText *)password, strlen(password),
                           (CONST OraText *)poolName, poolNameLen,
                           OCI_CPOOL));

    sprintf(insertst1,"INSERT INTO emp(empno, ename, job, sal, deptno) values\
            (%d,'abc','MANAGER',122,20)",empno);

    OCIHandleAlloc(envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, (size_t)0,
                   (dvoid **)0);

    checkerr(errhp, OCIStmtPrepare (stmthp, errhp, (text *)insertst1,
            (ub4)strlen(insertst1), OCI_NTV_SYNTAX, OCI_DEFAULT));

    checkerr(errhp, OCIStmtExecute (svchp, stmthp, errhp, (ub4)1, (ub4)0,
            (OCISnapshot *)0, (OCISnapshot *)0, OCI_DEFAULT ));

    checkerr(errhp, OCITransCommit(svchp,errhp,(ub4)0));

    checkerr(errhp, OCIHandleFree((dvoid *) stmthp, OCI_HTYPE_STMT));
    checkerr(errhp, OCILogoff((dvoid *) svchp, errhp));
} /* end of threadFunction (dvoid *) */

void checkerr(errhp, status)
```

```
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                        errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}
```

User-Defined Callback Functions

The Oracle Call Interface has the capability to execute user-specific code in addition to OCI calls. This functionality can be used for:

- Adding tracing and performance measurement code to enable users to tune their applications.

- Performing pre- or post-processing code for specific OCI calls.

- Accessing other data sources with OCI by using the native OCI interface for Oracle databases and directing the OCI calls to use user callbacks for non-Oracle data sources.

The OCI callback feature has been added by providing support for calling user code before or after executing the OCI calls. Functionality has also been provided to allow the user-defined code to be executed instead of executing the OCI code.

The user callback code can also be registered dynamically without modifying the source code of the application. The dynamic registration is implemented by loading up to five user-created dynamically linked libraries after the initialization of the environment handle during the `OCIEnvCreate()` call. These user-created libraries (such as Dynamic Link Libraries (DLLs) on NT, or shared libraries on Solaris) register the user callbacks for the selected OCI calls transparently to the application.

Sample Application

For a listing of the complete demonstration programs that illustrate the OCI user callback feature, see Appendix B, "OCI Demonstration Programs".

Registering User Callbacks

An application can register user callback libraries with the `OCIUserCallbackRegister()` function. Callbacks are registered in the context of the environment handle. An application can retrieve information about callbacks registered with a handle with the `OCIUserCallbackGet()` function.

> **See Also:** For detailed descriptions of these functions and their parameters, refer to the descriptions of `OCIUserCallbackGet()` and `OCIUserCallbackRegister()`

A user-defined callback is a subroutine that is registered against an OCI call and an environment handle. It can be specified to be either an entry callback, a replacement callback, or an exit callback.

- If it is an entry callback, it is called when the program enters the OCI function.

- Replacement callbacks are executed after entry callbacks. If the replacement callback returns a value of OCI_CONTINUE, then a subsequent replacement callback or the normal OCI-specific code is executed. If a replacement callback returns anything other than OCI_CONTINUE, subsequent replacement callbacks and the OCI code does not execute.

- After a replacement callback returns something other than OCI_CONTINUE, or an OCI function successfully executes, program control transfers to the exit callback (if one is registered).

If a replacement or exit callback returns anything other than OCI_CONTINUE, then the return code from the callback is returned from the associated OCI call.

A user callback can return OCI_INVALID_HANDLE when either an invalid handle or an invalid context is passed to it.

> **Note:**   If any callback returns anything other than
> OCI_CONTINUE, then that return code is passed to the subsequent
> callbacks. If a replacement or exit callback returns a return code
> other than OCI_CONTINUE, then the final (not OCI_CONTINUE)
> return code is returned from the OCI call.

OCIUserCallbackRegister

A user callback is registered using the `OCIUserCallbackRegister()` call.

> **See Also:**   See `OCIUserCallbackRegister()` on page 16-173
> for the syntax of this call.

Currently, `OCIUserCallbackRegister()` is only registered on the environment handle. The user's callback function of typedef *OCIUserCallback* is registered along with its context for the OCI call identified by the OCI function code, *fcode*. The type of the callback, whether entry, replacement, or exit, is specified by the *when* parameter.

For example, the *stmtprep_entry_dyncbk_fn* entry callback function and its context *dynamic_context*, are registered against the environment handle *hndlp* for the OCIStmtPrepare() call by calling the OCIUserCallbackRegister() function with the following parameters.

```
OCIUserCallbackRegister( hndlp,
                         OCI_HTYPE_ENV,
```

```
                                 errh,
                                 stmtprep_entry_dyncbk_fn,
                                 dynamic_context,
                                 OCI_FNCODE_STMTPREPARE,
                                 OCI_UCBTYPE_ENTRY
                                 (OCIUcb*) NULL);
```

User Callback Function

The user callback function has to follow the following syntax:

```
typedef sword (*OCIUserCallback)
       (dvoid *ctxp,      /* context for the user callback*/
        dvoid *hndlp,     /* handle for the callback, env handle for now */
        ub4 type,         /* type of handlp, OCI_HTYPE_ENV for this release */
        ub4 fcode,        /* function code of the OCI call */
        ub1 when,         /* type of the callback, entry or exit */
        sword returnCode, /* OCI return code */
        ub4 *errnop,      /* Oracle error number */
        va_list arglist); /* parameters of the oci call */
```

In addition to the parameters described in the `OCIUserCallbackRegister()` call, the callback is called with the return code, *errnop*, and all the parameters of the original OCI as declared by the prototype definition.

The return code is always passed in as OCI_SUCCESS and **errnop* is always passed in as 0 for the first entry callback. Note that **errnop* refers to the content of *errnop* because *errnop* is an IN/OUT parameter.

If the callback does not want to change the OCI return code, then it must return OCI_CONTINUE, and the value returned in **errnop* is ignored. If on the other hand, the callback returns any other return code than OCI_CONTINUE, the last returned return code becomes the return code for the call. At the this point, the value of **errnop* returned is set in the error handle, or in the environment handle if the error information is returned in the environment handle because of the absence of the error handle for certain OCI calls such as `OCIHandleAlloc()`.

For replacement callbacks, the *returnCode* is the non-OCI_CONTINUE return code from the previous callback or OCI call and **errnop* is the value of the error number being returned in the error handle. This allows the subsequent callback to change the return code or error information if needed.

The processing of replacement callbacks is different in that if it returns anything other than OCI_CONTINUE, then subsequent replacement callbacks and OCI code is bypassed and processing jumps to the exit callbacks.

Note that if the replacement callbacks return OCI_CONTINUE to allow processing of OCI code, then the return code from entry callbacks is ignored.

All the original parameters of the OCI call are passed to the callback as variable parameters and the callback must retrieve them using the *va_arg* macros. The callback demonstration programs provide examples.

> **See Also:** See Appendix B, "OCI Demonstration Programs"

A null value can be registered to de-register a callback. That is, if the value of the callback (*OCIUserCallback*) is NULL in the OCIUserCallbackRegister() call, then the user callback is de-registered.

When using the thread-safe mode, the OCI program acquires all mutexes before calling the user callbacks.

UserCallback Control Flow

This pseudocode describes the overall processing of a typical OCI call:

```
OCIXyzCall()
{
 Acquire mutexes on handles;
 retCode = OCI_SUCCESS;
 errno = 0;
 for all ENTRY callbacks do
  {

     EntryretCode = (*entryCallback)(..., retcode, &errno, ...);
     if (retCode != OCI_CONTINUE)
      {
         set errno in error handle or environment handle;
         retCode = EntryretCode;
      }
  }
  for all REPLACEMENT callbacks do
  {
   retCode = (*replacementCallback) (..., retcode, &errno, ...);
   if (retCode != OCI_CONTINUE)
      {
       set errno in error handle or environment handle
       goto executeEXITCallback;
      }
  }
```

```
            retCode = return code for XyzCall; /* normal processing of OCI call */

            errno = error number from error handle or env handle;

      executeExitCallback:
        for all EXIT callbacks do
        {
            exitRetCode = (*exitCallback)(..., retCode, &errno,...);
            if (exitRetCode != OCI_CONTINUE)
            {
                set errno in error handle or environment handle;
                retCode = exitRetCode;
            }
        }
         release mutexes;
         return retCode
      }
```

UserCallback for OCIErrorGet()

If the callbacks are a total replacement of the OCI code, then they usually maintain their own error information in the call context and use that to return error information in *bufp* and *errnop* parameters of the replacement callback of the `OCIErrorGet()` call.

If on the other hand, the callbacks are either partially overriding OCI code, or just doing some other post processing, then they can use the exit callback to modify the error text and *errnop* parameters of the `OCIErrorGet()` by their own error message and error number. Note that the **errnop* passed into the exit callback is the error number in the error or the environment handle.

Errors from Entry Callbacks

If an entry callback wants to return an error to the caller of the OCI call, then it must register a replacement or exit callback. This is because if the OCI code is executed, then the error code from the entry callback is ignored. Therefore the entry callback should pass the error to the replacement or exit callback through its own context.

Dynamic Callback Registrations

Because user callbacks are expected to be used for monitoring OCI behavior or to access other data sources, it is desirable that the registration of the callbacks be done transparently and non-intrusively. This is accomplished by loading user-created dynamically linked libraries at OCI initialization time. These dynamically linked

libraries are called *packages*. The user-created packages register the user callbacks for the selected OCI calls. These callbacks can further register or de-register user callbacks as needed when receiving control at runtime.

A makefile (`ociucb.mk` on Solaris) is provided with the OCI demonstration programs to create the package. The exact naming and location of this package is operating system dependent. The source code for the package must provide code for special callbacks that are called at OCI initialization and environment creation times.

The loading of the package is controlled by setting an operating system environment variable, ORA_OCI_UCBPKG. This variable names the packages in a generic way. The packages must be located in the $ORACLE_HOME/lib directory.

Loading Multiple Packages

The ORA_OCI_UCBPKG variable can contain a semicolon separated list of package names. The packages are loaded in the order they are specified in the list.

For example, previously one specified the package as:

```
setenv ORA_OCI_UCBPKG mypkg
```

Now, you can still specify the package as above, but in addition multiple packages can be specified as:

```
setenv ORA_OCI_UCBPKG "mypkg;yourpkg;oraclepkg;sunpkg;msoftpkg"
```

All these packages are loaded in order. That is, `mypkg` is loaded first and `msoftpkg` is loaded last.

A maximum of five packages can be specified.

> **Note:** The sample makefile `ociucb.mk` creates `ociucb.so.1.0` on a Solaris system or `ociucb.dll` on an NT system. To load the `ociucb` package, the environmental variable ORA_OCI_UCBPKG must be set to `ociucb`. On Solaris, if the package name ends with `.so`, `OCIInitialize()` fails. The package name must end with `.so.1.0`.
>
> For further details about creating the dynamic link libraries, read the makefiles provided in the demo directory for your platform. For further information on user-defined callbacks, see your platform-specific documentation on compiling and linking applications.

Package Format

Previously a package had to specify the source code for the `OCIEnvCallback()` function. Now the `OCIEnvCallback()` function is obsolete. Instead, the package source must provide two functions. The first function has to be named as *packagename* suffixed with the word *Init*. For example, if the package is named foo, then the source file (for example, but not necessarily, foo.c) should contain a `fooInit()` function with a call to `OCISharedLibInit()` function specified exactly as:

```
sword fooInit(metaCtx, libCtx, argfmt, argc, argv)
      dvoid *        metaCtx;        /* The metacontext */
      dvoid *        libCtx;         /* The context for this package. */
      ub4            argfmt;         /* package argument format */
      sword          argc;           /* package arg count*/
      dvoid *        argv[];         /* package arguments */
{
  return  (OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv,
                        fooEnvCallback));
}
```

The last parameter of the `OCISharedLibInit()` function, `fooEnvCallback()`, in this case, is the name of the second function. It can be named anything, but by convention it can be named *packagename* suffixed with the word *EnvCallback*.

This function is a replacement for `OCIEnvCallback()`. Now all the dynamic user callbacks must be registered in this function. The function must be of type **OCIEnvCallbackType**, which is specified as:

```
typedef sword (*OCIEnvCallbackType)(OCIEnv *env, ub4 mode,
```

```
                                        size_t xtramem_sz, dvoid *usrmemp,
                                        OCIUcb *ucbDesc);
```

When an environment handle is created, then this callback function is called at the very end. The env parameter is the newly created environment handle.

The *mode*, *xtramem_sz*, and *usrmemp* are the parameters passed to the OCIEnvCreate() call. The last parameter, *ucbDesc*, is a descriptor that is passed to the package. The package uses this descriptor to register the user callbacks as described later.

A sample ociucb.c file is provided in the demo directory. The makefile ociucb.mk is also provided (on Solaris) in the demo directory to create the package. Please note that this may be different on other platforms. The demo directory also contains full user callback demo programs (cdemoucb.c, cdemoucbl.c,) illustrating this.

User Callback Chaining

User callbacks can both be registered statically in the application itself or dynamically at runtime in the DLLs. A mechanism is needed to allow the application to override a previously registered callback and then later invoke the overridden one in the newly registered callback to preserve the behavior intended by the dynamic registrations. This can result in chaining of user callbacks.

For this purpose, the OCIUserCallbackGet() function is provided to find out which function and context is registered for an OCI call.

> **See Also:** See OCIUserCallbackGet() on page 16-171 for the syntax of this call

Accessing Other Data Sources Through OCI

Because Oracle is the predominant database accessed, applications can take advantage of the OCI interface to access non-Oracle data by using the user callbacks to access them. This allows an application written in OCI to access Oracle data without any performance penalty. To access non-Oracle data sources, drivers can be written that access the non-Oracle data in user callbacks. Because OCI provides a very rich interface, there is usually a straightforward mapping of OCI calls to most data sources. This solution is better than writing applications for other middle layers such as ODBC that introduce performance penalties for all data sources. Using OCI does not incur any penalty for the common case of accessing Oracle data sources, and incurs the same penalty that ODBC does for non-Oracle data sources.

Restrictions on Callback Functions

There are certain restrictions on the usage of callback functions, including
`OCIEnvCallback()`:

- A callback cannot call other OCI functions except
  `OCIUserCallbackRegister()`, `OCIUserCallbackGet()`,
  `OCIHandleAlloc()`, `OCIHandleFree()`. Even for these functions, if they are
  called in a user callback, then callbacks on them are not called to avoid
  recursion. For example, if `OCIHandleFree()` is called in the callback for
  `OCILogoff()`, then the callback for `OCIHandleFree()` is disabled during the
  execution of the callback for `OCILogoff()`.

- A callback cannot modify OCI data structures such as the environment or error
  handles.

- A callback cannot be registered for `OCIUserCallbackRegister()` call itself,
  or for any of the following:

  - `OCIUserCallbackGet()`

  - `OCIEnvCreate()`

  - `OCIInitialize()`

  - `OCIEnvInit()`

Example of OCI Callbacks

For example, lets suppose that there are five packages each registering entry,
replacement, and exit callbacks for OCIStmtPrepare call. That is, the
ORA_OCI_UCBPKG variable is set as:

```
setenv ORA_OCI_UCBPKG "pkg1;pkg2;pkg3;pkg4;pkg5"
```

In each package pkgN (where N can be 1 through 5), the `pkgNInit()` and
`PkgNEnvCallback()` functions are specified as:

```
pkgNInit(metaCtx, libCtx, argfmt, argc, argv)
{
  return OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv, pkgNEnvCallback);
}
```

The `pkgNEnvCallback()` function registers the entry, replacement, and exit
callbacks as:

```
 pkgNEnvCallback(env, mode, xtramemsz, usrmemp, ucbDesc)
{
```

```
OCIHandleAlloc((dvoid *)env, (dvoid **)&errh, OCI_HTYPE_ERROR, (size_t) 0,
      (dvoid **)NULL);

OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_entry_callback_fn,
      pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_ENTRY, ucbDesc);

OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_replace_callback_fn,
      pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_REPLACE, ucbDesc);

OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_exit_callback_fn,
      pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_EXIT, ucbDesc);

return OCI_CONTINUE;
}
```

Finally, in the source code for the application, user callbacks can be registered with
the NULL *ucbDesc* as:

```
OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_entry_callback_fn,
      pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_ENTRY, (OCIUcb *)NULL);

OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_replace_callback_fn,
      pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_REPLACE, (OCIUcb *)NULL);

OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_exit_callback_fn,
      pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_EXIT, (OCIUcb *)NULL);
```

When the `OCIStmtPrepare()` call is executed, the callbacks are called in the
following order:

```
static_entry_callback_fn()
pkg1_entry_callback_fn()
pkg2_entry_callback_fn()
pkg3_entry_callback_fn()
pkg4_entry_callback_fn()
pkg5_entry_callback_fn()

static_replace_callback_fn()
 pkg1_replace_callback_fn()
  pkg2_replace_callback_fn()
   pkg3_replace_callback_fn()
    pkg4_replace_callback_fn()
     pkg5_replace_callback_fn()

      OCI code for OCIStmtPrepare call
```

```
pkg5_exit_callback_fn()
pkg4_exit_callback_fn()
pkg3_exit_callback_fn()
pkg2_exit_callback_fn()
pkg1_exit_callback_fn() 9

static_exit_callback_fn()
```

> **Note:** The exit callbacks are called in the reverse order of the entry and replacement callbacks

The entry and exit callbacks can return any return code and the processing continues to the next callback. However, if the replacement callback returns anything other than OCI_CONTINUE, then the next callback (or OCI code if it is the last replacement callback) in the chain is bypassed and processing jumps to the exit callback. For example, if pkg3_replace_callback_fn() returned OCI_SUCCESS, then pkg4_replace_callback_fn(), pkg5_replace_callback_fn(), and the OCI processing for the OCIStmtPrepare call is bypassed. Instead, pkg5_exit_callback_fn() is executed next.

OCI Callbacks From External Procedures

There are several OCI functions that can be used as callbacks from external procedures.

> **See Also:** These functions are listed in Chapter 19, "OCI Cartridge Functions". For information about writing C subroutines that can be called from PL/SQL code, including a list of which OCI calls can be used, and some example code, refer to the *Oracle9i Application Developer's Guide - Fundamentals.*

Application Failover Callbacks

Application failover callbacks can be used in the event of the failure of one database instance, and failover to another instance. Because of the delay which can occur during failover, the application developer may want to inform the user that failover is in progress, and request that the user stand by. Additionally, the session on the initial instance may have received some `ALTER SESSION` commands. These will not be automatically replayed on the second instance. Consequently, the developer may wish to replay these `ALTER SESSION` commands on the second instance.

> **See Also:** For more detailed information about application failover, refer to *Oracle9i Real Application Clusters Concepts* and *Net8 Reference*

Failover Callback Overview

To address the problems described above, the application developer can register a failover callback function. In the event of failover, the callback function is invoked several times during the course of reestablishing the user's session.

The first call to the callback function occurs when Oracle first detects an instance connection loss. This callback is intended to allow the application to inform the user of an upcoming delay. If failover is successful, a second call to the callback function occurs when the connection is reestablished and usable. At this time the client may wish to replay `ALTER SESSION` commands and inform the user that failover has happened. If failover is unsuccessful, then the callback is called to inform the application that failover will not take place. Additionally, the callback is called each time a user handle besides the primary handle is reauthenticated on the new connection. Since each user handle represents a server-side session, the client may wish to replay `ALTER SESSION` commands for that session.

An initial attempt at failover may not always successful. The OCI provides a mechanism for retrying failover after an unsuccessful attempt.

> **See Also:** See for more information about this scenario

Failover Callback Structure and Parameters

The basic structure of a user-defined application failover callback function is as follows:

```
sb4 appfocallback_fn ( dvoid      * svchp,
                       dvoid      * envhp,
```

```
dvoid      * fo_ctx,
ub4        fo_type,
ub4        fo_event );
```

An example is provided in the section "Failover Callback Example" on page 9-37 for the following parameters:

svchp
The first parameter, *svchp*, is the service context handle. It is of type **dvoid** *.

envhp
The second parameter, *envhp*, is the OCI environment handle. It is of type **dvoid** *.

fo_ctx
The third parameter, *fo_ctx*, is a client context. It is a pointer to memory specified by the client. In this area the client can keep any necessary state or context. It is passed as a **dvoid** *.

fo_type
The fourth parameter, *fo_type*, is the failover type. This lets the callback know what type of failover the client has requested. The usual values are:

- OCI_FO_SESSION, which indicates that the user has requested only session failover.

- OCI_FO_SELECT, which indicates that the user has requested select failover as well.

fo_event
The last parameter is the failover event. This indicates to the callback why it is being called. It has several possible values:

- OCI_FO_BEGIN indicates that failover has detected a lost connection and failover is starting.

- OCI_FO_END indicates successful completion of failover.

- OCI_FO_ABORT indicates that failover was unsuccessful, and there is no option of retrying.

OCI_FO_ERROR also indicates that failover was unsuccessful, but it gives the application the opportunity to handle the error and retry failover.

> **See Also:** See "Handling OCI_FO_ERROR" on page 9-39 for more information about this value

- OCI_FO_REAUTH indicates that a user handle has been reauthenticated. To find out which, the application should check the OCI_ATTR_SESSION attribute of the service context handle (which is the first parameter).

Failover Callback Registration

For the failover callback to be used, it must be registered on the server context handle. This registration is done by creating a callback definition structure and setting the OCI_ATTR_FOCBK attribute of the server handle to this structure.

The callback definition structure must be of type **OCIFocbkStruct**. It has two fields: *callback_function*, which contains the address of the function to call, and *fo_ctx* which contains the address of the client context.

An example of callback registration is included as part of the example in the next section.

Failover Callback Example

The following code shows an example of a simple user-defined callback function definition and registration.

Part 1, Failover Callback Definition

```
sb4  callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event )
dvoid * svchp;
dvoid * envhp;
dvoid *fo_ctx;
ub4 fo_type;
ub4 fo_event;
{
switch (fo_event)
   {
   case OCI_FO_BEGIN:
   {
     printf(" Failing Over ... Please stand by \n");
     printf(" Failover type was found to be %s \n",
                   ((fo_type==OCI_FO_SESSION) ? "SESSION"
                   :(fo_type==OCI_FO_SELECT) ? "SELECT"
                   : "UNKNOWN!"));
     printf(" Failover Context is :%s\n",
                   (fo_ctx?(char *)fo_ctx:"NULL POINTER!"));
     break;
   }
```

```
    case OCI_FO_ABORT:
    {
      printf(" Failover aborted. Failover will not take place.\n");
      break;
    }
    case    OCI_FO_END:
    {
        printf(" Failover ended ...resuming services\n");
      break;
    }
    case OCI_FO_REAUTH:
    {
        printf(" Failed over user. Resuming services\n");
      break;
    }
    default:
    {
      printf("Bad Failover Event: %d.\n",  fo_event);
      break;
    }
    }
    return 0;
}
```

Part 2, Failover Callback Registration

```
int register_callback(svrh, errh)
dvoid *svrh; /* the server handle */
OCIError *errh; /* the error handle */
{
  OCIFocbkStruct failover;                    /*  failover callback structure */
  /* allocate memory for context */
  if (!(failover.fo_ctx = (dvoid *)malloc(strlen("my context."))))
      return(1);
  /* initialize the context. */
  strcpy((char *)failover.context_function, "my context.");
  failover.callback_function = &callback_fn;
  /* do the registration */
  if (OCIAttrSet(srvh, (ub4) OCI_HTYPE_SERVER,
                (dvoid *) &failover, (ub4) 0,
                (ub4) OCI_ATTR_FOCBK, errh)  != OCI_SUCCESS)
      return(2);
  /* successful conclusion */
  return (0);
}
```

Handling OCI_FO_ERROR

A failover attempt is not always successful. If the attempt fails, the callback function receives a value of OCI_FO_ABORT or OCI_FO_ERROR in the `fo_event` parameter. A value of OCI_FO_ABORT indicates that failover was unsuccessful, and no further failover attempts are possible. OCI_FO_ERROR, on the other hand, provides the callback function with the opportunity to handle the error in some way. For example, the callback may choose to wait a specified period of time and then indicate to the OCI library that it should reattempt failover.

> **Note:** This functionality is only available to applications linked with the 8.0.5 or later OCI libraries running against any Oracle server.

Consider the following timeline of events:

Time	Event
T0	Database crashes (crash lasts until T5).
T1	Failover triggered by user activity.
T2	User attempts to reconnect; attempt fails.
T3	Failover callback invoked with OCI_FO_ERROR.
T4	Failover callback enters predetermined sleep period.
T5	Database comes back up again.
T6	Failover callback triggers new failover attempt; it is successful.
T7	User successfully reconnects

The callback function triggers the new failover attempt by returning a value of OCI_FO_RETRY from the function.

The following example code shows a callback function which might be used to implement the failover strategy similar to the scenario described above. In this case the failover callback enters a loop in which it sleeps and then reattempts failover until it is successful:

```
/*------------------------------------------------------------------*/
/* the user defined failover callback  */
/*------------------------------------------------------------------*/
```

```
sb4  callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event )
dvoid * svchp;
dvoid * envhp;
dvoid *fo_ctx;
ub4 fo_type;
ub4 fo_event;
{
   OCIError *errhp;
   OCIHandleAlloc(envhp, (dvoid **)&errhp, (ub4) OCI_HTYPE_ERROR,
               (size_t) 0, (dvoid **) 0);
   switch (fo_event)
   {
   case OCI_FO_BEGIN:
   {
     printf(" Failing Over ... Please stand by \n");
     printf(" Failover type was found to be %s \n",
             ((fo_type==OCI_FO_NONE) ? "NONE"
              :(fo_type==OCI_FO_SESSION) ? "SESSION"
              :(fo_type==OCI_FO_SELECT) ? "SELECT"
              :(fo_type==OCI_FO_TXNAL) ? "TRANSACTION"
              : "UNKNOWN!"));
     printf(" Failover Context is :%s\n",
             (fo_ctx?(char *)fo_ctx:"NULL POINTER!"));
     break;
   }
   case OCI_FO_ABORT:
   {
     printf(" Failover aborted. Failover will not take place.\n");
     break;
   }
   case    OCI_FO_END:
   {
       printf("\n Failover ended ...resuming services\n");
     break;
   }
   case OCI_FO_REAUTH:
   {
       printf(" Failed over user. Resuming services\n");
     break;
   }
   case OCI_FO_ERROR:
   {
     /* all invocations of this can only generate one line. The newline
      * will be put at fo_end time.
      */
```

```
      printf(" Failover error gotten. Sleeping...");
      sleep(3);
      printf("Retrying. ");
      return (OCI_FO_RETRY);
      break;
    }
    default:
    {
      printf("Bad Failover Event: %d.\n",  fo_event);
      break;
    }
    }
    return 0;
}
```

The following is sample output from a program containing this failover callback function:

```
executing select...
7369    SMITH    CLERK
7499    ALLEN    SALESMAN
 Failing Over ... Please stand by
 Failover type was found to be SELECT
 Failover Context is :My context.
 Failover error gotten. Sleeping...Retrying.  Failover error gotten.
Sleeping...Retrying.  Failover error gotten. Sleeping...Retrying.  Failover
error gotten. Sleeping...Retrying.  Failover error gotten. Sleeping...Retrying.
Failover error gotten. Sleeping...Retrying.  Failover error gotten.
Sleeping...Retrying.  Failover error gotten. Sleeping...Retrying.  Failover
error gotten. Sleeping...Retrying.  Failover error gotten. Sleeping...Retrying.
 Failover ended ...resuming services
7521    WARD     SALESMAN
7566    JONES    MANAGER
7654    MARTIN    SALESMAN
7698    BLAKE    MANAGER
7782    CLARK    MANAGER
7788    SCOTT    ANALYST
7839    KING     PRESIDENT
7844    TURNER    SALESMAN
7876    ADAMS    CLERK
7900    JAMES    CLERK
7902    FORD     ANALYST
```

OCI and Advanced Queuing

The OCI provides an interface to Oracle's Advanced Queuing feature. Oracle AQ provides message queuing as an integrated part of the Oracle server. Oracle AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database.* By providing an integrated solution Oracle AQ frees application developers to devote their efforts to their specific business logic rather than having to construct a messaging infrastructure.

> **Note:** In order to use Advanced Queuing, you must be using the Enterprise Edition

See Also:

- For detailed information about AQ, including concepts, features, and examples, refer to

- *Oracle9i Application Developer's Guide - Advanced Queuing*

- *Oracle9i Application Developer's Guide - XML*

- *Oracle Integration Server Overview*

- For example code demonstrating the use of the OCI with AQ, refer to the description of `OCIAQEnq()` on page 16-73

OCI Advanced Queuing Functions

The OCI library includes several functions related to Advanced Queuing:

- `OCIAQEnq()`

- `OCIAQDeq()`

- `OCIAQListen()`

> **See Also:** "Advanced Queuing and Publish-Subscribe Functions" on page 16-70, contains complete descriptions of these functions and their parameters

OCI Advanced Queuing Descriptors

The following descriptors are used by OCI AQ operations:

- **OCIAQEnqOptions**

- **OCIAQDeqOptions**

- **OCIAQMsgProperties**

- **OCIAQAgent**

You can allocate these descriptors with respect to the service handle using the standard `OCIDescriptorAlloc()` call. The following code shows examples of this:

```
OCIDescriptorAlloc(svch, &enqueue_options, OCI_DTYPE_AQENQ_OPTIONS, 0, 0 );
OCIDescriptorAlloc(svch, &dequeue_options, OCI_DTYPE_AQDEQ_OPTIONS, 0, 0 );
OCIDescriptorAlloc(svch, &message_properties, OCI_DTYPE_AQMSG_PROPERTIES, 0, 0);
OCIDescriptorAlloc(svch, &agent, OCI_DTYPE_AQAGENT, 0, 0 );
```

Each descriptor has a variety of attributes which can be set and/or read.

> **See Also:** These attributes are described in more detail in "Advanced Queuing Descriptor Attributes" on page A-42

Advanced Queuing in OCI vs. PL/SQL

The following tables compare functions, parameters, and options for OCI AQ functions and descriptors, and PL/SQL AQ functions in the dbms_aq package.

PL/SQL Function	OCI Function
DBMS_AQ.ENQUEUE	`OCIAQEnq()`
DBMS_AQ.DEQUEUE	`OCIAQDeq()`
DBMS_AQ.LISTEN	`OCIAQListen()`

DBMS_AQ.ENQUEUE Parameter	OCIAQEnq() Parameter
queue_name	queue_name
enqueue_options	enqueue_options
message_properties	message_properties
payload	payload
msgid	msgid

Note: `OCIAQEnq()` also requires the following additional parameters: *svch*, *errh, payload_tdo, payload_ind*, and *flags*

DBMS_AQ.DEQUEUE Parameter	OCIAQDeq() Parameter
queue_name	queue_name
dequeue_options	dequeue_options
message_properties	message_properties
payload	payload
msgid	msgid

Note: `OCIAQDeq()` also requires the following additional parameters: `svch`, `errh, queue_name, dequeue_options, message_properties, payload_tdo, payload, payload_ind`, and `flags`

DBMS_AQ.LISTEN Parameter	OCIAQListen() Parameter
agent_list	agent_list
wait	wait
agent	agent

Note: `OCIAQListen()` also requires the following additional parameters: `svchp, errhp, agent_list, num_agents, wait, agent,` and `flags`

PL/SQL Agent Parameter	OCIAQAgent Attribute
name	OCI_ATTR_AGENT_NAME
address	OCI_ATTR_AGENT_ADDRESS
protocol	OCI_ATTR_AGENT_PROTOCOL

PL/SQL Message Property	OCIAQMsgProperties Attribute
priority	OCI_ATTR_PRIORITY
delay	OCI_ATTR_DELAY
expiration	OCI_ATTR_EXPIRATION
correlation	OCI_ATTR_CORRELATION

PL/SQL Message Property	OCIAQMsgProperties Attribute
attempts	OCI_ATTR_ATTEMPTS
recipient_list	OCI_ATTR_RECIPIENT_LIST
exception_queue	OCI_ATTR_EXCEPTION_QUEUE
enqueue_time	OCI_ATTR_ENQ_TIME
state	OCI_ATTR_MSG_STATE
sender_id	OCI_ATTR_SENDER_ID
original_msgid	OCI_ATTR_ORIGINAL_MSGID

PL/SQL Enqueue Option	OCIAQEnqOptions Attribute
visibility	OCI_ATTR_VISIBILITY
relative_msgid	OCI_ATTR_RELATIVE_MSGID
sequence_deviation	OCI_ATTR_SEQUENCE_DEVIATION

PL/SQL Dequeue Option	OCIAQDeqOptions Attribute
consumer_name	OCI_ATTR_CONSUMER_NAME
dequeue_mode	OCI_ATTR_DEQ_MODE
navigation	OCI_ATTR_NAVIGATION
visibility	OCI_ATTR_VISIBILITY
wait	OCI_ATTR_WAIT
msgid	OCI_ATTR_DEQ_MSGID
correlation	OCI_ATTR_CORRELATION

Publish-Subscribe Notification

The publish-subscribe notification feature allows an OCI application to receive client notifications directly, register an e-mail address to which notifications can be sent, register a HTTP URL to which notifications can be posted, or register a PL/SQL procedure to be invoked on a notification. Figure 9–2, "Publish-Subscribe Model" illustrates the process.

Figure 9–2   Publish-Subscribe Model
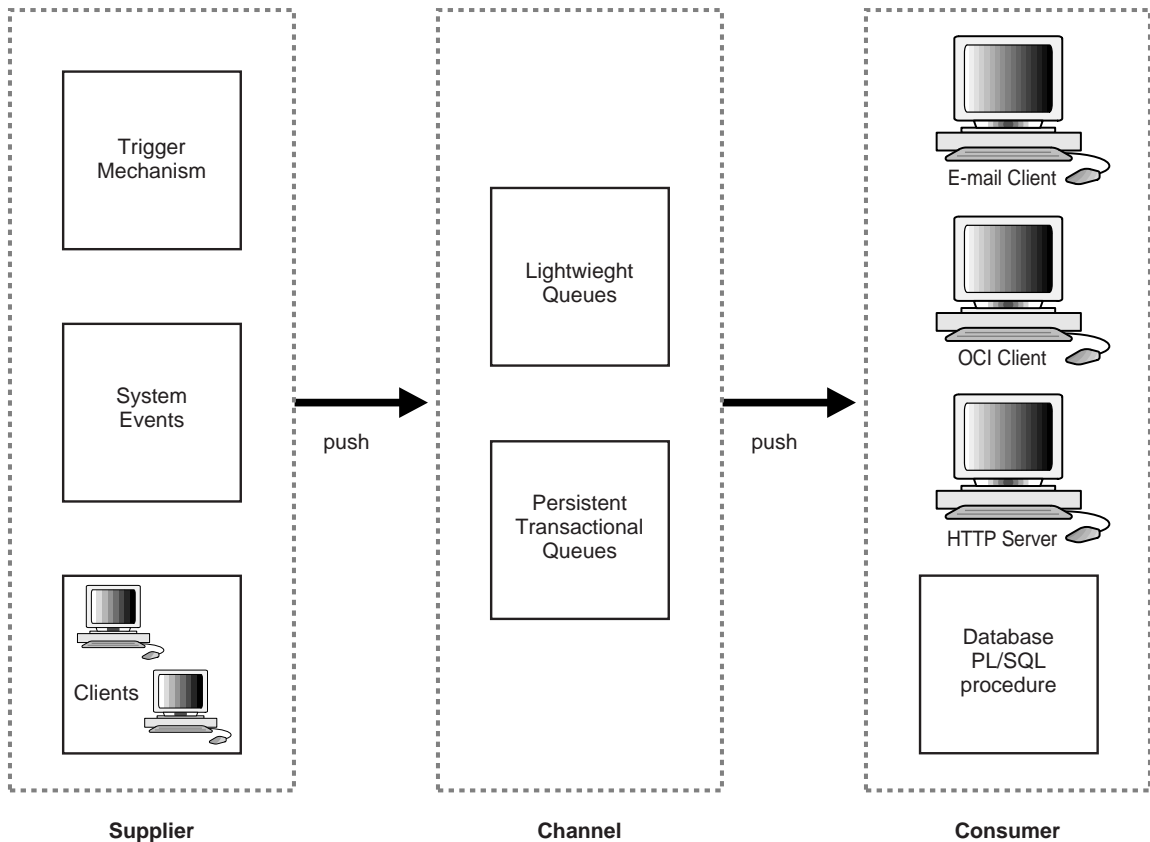
An OCI application can:

- register interest in notifications in the AQ namespace and be notified when an enqueue occurs.

- register interest in subscriptions to database events and receive notifications when the events are triggered.

- manage registrations, such as disabling registrations temporarily or dropping the registrations entirely.

- post, or send, notifications to registered clients.

In all the above scenarios the notification can be received directly by the OCI application, or the notification can be sent to a pre-specified e-mail address, or it can be sent to a pre-defined HTTP URL, or a pre-specified database PL/SQL procedure can be invoked as a result of a notification.

Registered clients are notified asynchronously when events are triggered or on an explicit AQ enqueue. Clients do not need to be connected to a database.

See Also:

- For information on Advanced Queuing, see "OCI and Advanced Queuing" on page 9-42

- For information on creating queues and about AQ, including concepts, features, and examples, refer to the chapter on Advanced Queuing in the *Oracle9i Application Developer's Guide - Advanced Queuing.*

- For information on creating triggers, refer to the chapter on Commands in the *Oracle9i SQL Reference.*

Publish-Subscribe Registration Functions

Registration can be done in two ways:

- You register directly to the database. This way is simple and the registration will take effect immediately.

- Open Registration. You register using LDAP, from which the database receives the registration request. This is useful when the client cannot have a database connection (the client wants to register for a database open event while the database is down), or if the client wants to register for the same event or events in multiple databases at one time.

Let us next consider these two alternative ways of registration.

Publish-Subscribe Register Directly to the Database

The following steps are required in an OCI application to register and receive notifications for events. It is assumed that the appropriate event trigger or AQ queue has been set up. Also, the initialization parameter COMPATIBLE must be set to 8.1 or higher.

> **See Also:**
>
> - Detailed descriptions of the functions noted can be found in"Advanced Queuing and Publish-Subscribe Functions" on page 16-70
>
> - For examples of the use of these functions in an application, see "Publish-Subscribe Direct Registration Example" on page 9-54

Note: The publish-subscribe feature is only available on multithreaded platforms.

---

1. Execute OCIInitialize() with OCI_EVENTS mode to specify that the application is interested in registering for and receiving notifications. This starts a dedicated listening thread for notifications on the client.

2. Execute OCIHandleAlloc() with handle type OCI_HTYPE_SUBSCRIPTION to allocate a subscription handle.

3. Execute OCIAttrSet() to set the subscription handle attributes for:

   - OCI_ATTR_SUBSCR_NAME - subscription name

   - OCI_ATTR_SUBSCR_NAMESPACE - subscription namespace

   - OCI_ATTR_SUBSCR_CALLBACK - notification callback

   - OCI_ATTR_SUBSCR_CTX - callback context

   - OCI_ATTR_SUBSCR_PAYLOAD - payload buffer for posting

   - OCI_ATTR_SUBSCR_RECPT - recipient name

   - OCI_ATTR_SUBSCR_RECPTPROTO - protocol to receive notification with

   - OCI_ATTR_SUBSCR_RECPTPRES - presentation to receive notification with

OCI_ATTR_SUBSCR_NAME, OCI_ATTR_SUBSCR_NAMESPACE and OCI_ATTR_SUBSCR_RECPTPROTO must be set before registering a subscription.

If OCI_ATTR_SUBSCR_RECPTPROTO is set to OCI_SUBSCR_PROTO_OCI, then OCI_ATTR_SUBSCR_CALLBACK and OCI_ATTR_SUBSCR_CTX also need to be set.

If OCI_ATTR_SUBSCR_RECPTPROTO is set to OCI_SUBSCR_PROTO_MAIL, OCI_SUBSCR_PROTO_SERVER, or OCI_SUBSCR_PROTO_HTTP, then OCI_ATTR_SUBSCR_RECPT also needs to be set.

Setting OCI_ATTR_SUBSCR_CALLBACK and OCI_ATTR_SUBSCR_RECPT at the same time will cause an application error.

OCI_ATTR_SUBSCR_PAYLOAD is required before posting to a subscription.

> **See Also:** For information on these attributes, see "Subscription Handle Attributes" on page A-53

4. If OCI_ATTR_SUBSCR_RECPTPROTO is set to OCI_SUBSCR_PROTO_OCI, then define the callback routine to be used with the subscription handle.

> **See Also:** "Notification Callback" on page 9-52

5. If OCI_ATTR_SUBSCR_RECPTPROTO is set to OCI_SUBSCR_PROTO_SERVER, then define the PL/SQL procedure, to be invoked on notification, in the database.

> **See Also:** "Notification Procedure" on page 9-53

6. Execute OCISubscriptionRegister() to register with the subscriptions. This call can register interest in several subscriptions at the same time.

Open Registration for Publish-Subscribe

Prerequisites for this method are:

- Registering using LDAP (open registration) requires the client to be an enterprise user.

> **See Also:** *Oracle Advanced Security Administrator's Guide,* sections on managing enterprise user security

- The compatibility of the database has to be 9.0 or higher.

- LDAP_REGISTRATION_ENABLED must be set to TRUE. This can be done this way:

```
ALTER SYSTEM SET LDAP_REGISTRATION_ENABLED=TRUE
```

The default is FALSE.

- LDAP_REG_SYNC_INTERVAL must be set to the time interval (in seconds) to refresh   registrations from LDAP:

```
ALTER SYSTEM SET LDAP_REG_SYNC_INTERVAL = time_interval
```

The default is 0, which means do not refresh.

- To force a database refresh of LDAP registration information immediately:

```
ALTER SYSTEM REFRESH LDAP_REGISTRATION
```

The steps in open registration using Oracle Enterprise Security Manager (OESM) are:

1. In each enterprise domain, create enterprise role, ENTERPRISE_AQ_USER_ROLE.

2. For each database in the enterprise domain, add global role GLOBAL_AQ_USER_ROLE to enterprise role ENTERPRISE_AQ_USER_ROLE.

3. For each enterprise domain, add enterprise role ENTERPRISE_AQ_USER_ROLE to privilege group cn=OracleDBAQUsers, under cn=oraclecontext, under the administrative context.

4. For each enterprise user that should be authorized to register for events in the database, grant enterprise role ENTERPRISE_AQ_USER_ROLE.

Using OCI toOpen Register with LDAP

1. Call `OCIInitialize()` with *mode* set to OCI_EVENTS | OCI_USE_LDAP.

2. Call `OCIAttrSet()` to set the following environment handle attributes for accessing LDAP:

- OCI_ATTR_LDAP_HOST: the host name on which the LDAP server resides

- OCI_ATTR_LDAP_PORT: the port on which the LDAP server is listening

- OCI_ATTR_BIND_DN: the distinguished name to login to the LDAP server, usually the DN of the enterprise user

- OCI_ATTR_LDAP_CRED: the credential used to authenticate the client, for example, the password for simple authentication (username/password)

- OCI_ATTR_WALL_LOC: for SSL authentication, the location of the client wallet

- OCI_ATTR_LDAP_AUTH: the authentication method code

    See Also: "OCI_ATTR_LDAP_AUTH" on page A-6 for a complete list of authentication modes

- OCI_ATTR_LDAP_CTX: the administrative context for Oracle in the LDAP server

3. Call `OCIHandleAlloc()` with handle type OCI_HTYPE_SUBSCRIPTION, to allocate a subscription handle.

4. Call `OCIDescriptorAlloc()` with descriptor type OCI_DTYPE_SRVDN, to

   allocate a server DN descriptor.

5. Call `OCIAttrSet()` to set the server DN descriptor attributes for OCI_ATTR_SERVER_DN, the distinguished name of the database in which the client wants to receive notifications. `OCIAttrSet()` can be called multiple times for this attribute so that more than one database server is included in the registration

6. Call `OCIAttrSet()` to set the subscription handle attributes for:

- OCI_ATTR_SUBSCR_NAME: subscription name

- OCI_ATTR_SUBSCR_NAMESPACE: subscription namespace

- OCI_ATTR_SUBSCR_CALLBACK: notification callback

- OCI_ATTR_SUBSCR_CTX: callback context

- OCI_ATTR_SUBSCR_PAYLOAD: payload buffer for posting

- OCI_ATTR_SUBSCR_RECPT: recipient name

- OCI_ATTR_SUBSCR_RECPTPROTO: protocol to receive notification

- OCI_ATTR_SUBSCR_RECPTRES: presentation to receive notification with

- OCI_ATTR_SUBSCR_SERVER_DN: the descriptor handles you populated in step 5.

7. Call `OCISubscriptionRegister()` to register the subscriptions. The registration will take effect when the database accesses LDAP to pick up new

registrations. The frequency of pick-ups is determined by the value of REG_SYNC_INTERVAL.

OCI Functions Used to Manage Publish-Subscribe Notification

The following functions are used to manage publish-subscribe notification.

Table 9–1   Publish-Subscribe Functions

Function	Purpose
OCISubscriptionDisable()	Disables a subscription.
OCISubscriptionEnable()	Enables a subscription.
OCISubscriptionPost()	Posts a subscription.
OCISubscriptionRegister()	Registers a subscription.
OCISubscriptionUnRegister()	Unregisters a subscription.

Notification Callback

The client needs to register a notification callback that gets invoked when there is some activity on the subscription for which interest has been registered. In the AQ namespace, for instance, this occurs when a message of interest is enqueued.

This callback is typically set through the OCI_ATTR_SUBSCR_CALLBACK attribute of the subscription handle.

> **See Also:**   For information, see "Subscription Handle Attributes" on page A-53

The callback must return a value of OCI_CONTINUE and adhere to the following specification:

```
typedef ub4 (*OCISubscriptionNotify) ( dvoid *pCtx,
                                        OCISubscription *pSubscrHp,
                                        dvoid *pPayload,
                                        ub4 *iPayloadLen,
                                        dvoid *pDescriptor,
                                        ub4 iMode);
```

The parameters are described as follows:

pCtx (IN) A user-defined context specified when the callback was registered.

pSubscrHp (IN) The subscription handle specified when the callback was registered.

pPayload (IN) The payload for this notification. For this release, only ub1 * (a sequence of bytes) for the payload is supported.

iPayloadLen (IN) The length of the payload for this notification.

pDescriptor (IN) The namespace-specific descriptor. Namespace-specific parameters can be extracted from this descriptor. The structure of this descriptor is opaque to the user and its type is dependent on the namespace.

The attributes of the descriptor are namespace-specific. For Advanced Queuing, the descriptor is OCI_DTYPE_AQNFY. The attributes of this descriptor are:

- Queue Name - OCI_ATTR_QUEUE_NAME

- Consumer Name - OCI_ATTR_CONSUMER_NAME

- Message Id - OCI_ATTR_NFY_MSGID

- Message Properties - OCI_ATTR_MSG_PROP

> **See Also:** For more information about OCI and Advanced Queuing, refer to "OCI and Advanced Queuing" on page 9-42

iMode (IN) Call-specific mode. Valid value:

- OCI_DEFAULT - executes the default call

Notification Procedure

The PL/SQL procedure that will be invoked when there is some activity on the subscription for which interest has been registered, has to be created in the database.

This procedure is typically set through the OCI_ATTR_SUBSCR_RECPT attribute of the subscription handle. For information, see "Subscription Handle Attributes".

> **See Also:** ■
>
> - See "Subscription Handle Attributes" on page A-53.
>
> - For the PL/SQL procedure specification please see "AQ PL/SQL Callback" in *Oracle9i Supplied PL/SQL Packages and Types Reference.*

Publish-Subscribe Direct Registration Example

This example shows how system events, client notification, and Advanced Queuing work together to implement publish subscription notification.

The following PL/SQL code creates all objects necessary to support a publish-subscribe mechanism under the user schema, *pubsub*. In this code, the Agent *snoop* subscribes to messages that are published at logon events. Note that the user *pubsub* needs AQ_ADMINISTRATOR_ROLE and AQ_USER_ROLE privileges to use Advance Queuing functionality. Also, the initialization parameter _SYSTEM_TRIG_ENABLED must be set to TRUE (the default) to enable triggers for system events.

```
Rem -------------------------------------------------------
REM create queue table for persistent multiple consumers
Rem -------------------------------------------------------
connect pubsub/pubsub;
Rem  Create or replace a queue table
begin
  DBMS_AQADM.CREATE_QUEUE_TABLE(
  QUEUE_TABLE=>'pubsub.raw_msg_table',
  MULTIPLE_CONSUMERS => TRUE,
  QUEUE_PAYLOAD_TYPE =>'RAW',
  COMPATIBLE => '8.1.5');
end;
/
Rem -------------------------------------------------------
Rem  Create a persistent queue for publishing messages
Rem -------------------------------------------------------
Rem  Create a queue for logon events
begin
  DBMS_AQADM.CREATE_QUEUE(QUEUE_NAME=>'pubsub.logon',
  QUEUE_TABLE=>'pubsub.raw_msg_table',
  COMMENT=>'Q for error triggers');
end;
/
Rem -------------------------------------------------------
Rem  Start the queue
Rem -------------------------------------------------------
begin
  DBMS_AQADM.START_QUEUE('pubsub.logon');
end;
/
Rem -------------------------------------------------------
```

```
Rem  define new_enqueue for convenience
Rem ------------------------------------------------------
create or replace procedure new_enqueue(queue_name  in varchar2,
                                         payload  in raw ,
correlation in varchar2 := NULL,
exception_queue in varchar2 := NULL)
as
  enq_ct    dbms_aq.enqueue_options_t;
  msg_prop  dbms_aq.message_properties_t;
  enq_msgid raw(16);
  userdata  raw(1000);
begin
  msg_prop.exception_queue := exception_queue;
  msg_prop.correlation := correlation;
  userdata := payload;
  DBMS_AQ.ENQUEUE(queue_name,enq_ct, msg_prop,userdata,enq_msgid);
end;
/
Rem ------------------------------------------------------
Rem  add subscriber with rule based on current user name,
Rem  using correlation_id
Rem ------------------------------------------------------
declare
subscriber sys.aq$_agent;
begin
  subscriber := sys.aq$_agent('SNOOP', null, null);
  dbms_aqadm.add_subscriber(queue_name => 'pubsub.logon',
                            subscriber => subscriber,
                            rule => 'CORRID = ''SCOTT'' ');
end;
/
Rem ------------------------------------------------------
Rem  create a trigger on logon on database
Rem ------------------------------------------------------
Rem  create trigger on after logon
create or replace trigger systrig2
   AFTER LOGON
   ON DATABASE
   begin
     new_enqueue('pubsub.logon', hextoraw('9999'), dbms_standard.login_user);
   end;
/

Rem ------------------------------------------------------
Rem  create a PL/SQL callback for notification of logon
```

```
Rem  of user 'scott' on database
Rem -------------------------------------------------------
Rem
create or replace procedure plsqlnotifySnoop(
  context raw, reginfo sys.aq$_reg_info, descr sys.aq$_descriptor,
  payload raw, payloadl number)
as
begin
 dbms_output.putline('Notification : User Scott Logged on\n');
end;
/
```

After the subscriptions are created, the client needs to register for notification using callback functions. The following sample code performs the necessary steps for registration. The initial steps of allocating and initializing session handles are omitted here for sake of clarity.

```
...
ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;
/* callback function for notification of logon of user 'scott' on database */
ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
    dvoid *ctx;
    OCISubscription *subscrhp;
    dvoid *pay;
    ub4 payl;
    dvoid *desc;
    ub4 mode;
{
    printf("Notification : User Scott Logged on\n");
}
int main()
{
    OCISession *authp = (OCISession *) 0;
    OCISubscription *subscrhpSnoop = (OCISubscription *)0;
    OCISubscription *subscrhpSnoopMail = (OCISubscription *)0;
    OCISubscription *subscrhpSnoopServer = (OCISubscription *)0;

/***************************************************
Initialize OCI Process/Environment
Initialize Server Contexts
Connect to Server
Set Service Context
***************************************************/
    /* Registration Code Begins */
/* Each call to initSubscriptionHn allocates
```

```
                    and Initialises a Registration Handle */

    /* Register for OCI notification */
        initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle */
            "PUBSUB.SNOOP:ADMIN", /* subscription name */
    /* <queue_name>:<agent_name> */
            (dvoid*)notifySnoop,  /* callback function */
            OCI_SUBSCR_PROTO_OCI, /* receive with protocol */
             (char *)0, /* recipient address */
             OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

    /* Register for email notification */
        initSubscriptionHn(    &subscrhpSnoopMail,    /* subscription handle */
            "PUBSUB.SNOOP:ADMIN", /* subscription name */
    /* <queue_name>:<agent_name> */
            (dvoid*)0, /* callback function */
            OCI_SUBSCR_PROTO_MAIL, /* receive with protocol */
             "xyz@company.com", /* recipient address */
            OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

    /* Register for server to server notification */
        initSubscriptionHn(    &subscrhpSnoopServer,    /* subscription handle */
            "PUBSUB.SNOOP:ADMIN", /* subscription name */
    /* <queue_name>:<agent_name> */
            (dvoid*)0, /* callback function */
            OCI_SUBSCR_PROTO_SERVER, /* receive with protocol */
             "pubsub.plsqlnotifySnoop", /* recipient address */
            OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */
/****************************************************
The Client Process does not need a live Session for Callbacks
End Session and Detach from Server
****************************************************/
    OCISessionEnd ( svchp, errhp, authp, (ub4) OCI_DEFAULT);
    /* detach from server */
    OCIServerDetach( srvhp, errhp, OCI_DEFAULT);
    while (1)     /* wait for callback */
        sleep(1);
}
void initSubscriptionHn (subscrhp,
                         subscriptionName,
                         func,
                         recpproto,
                         recpaddr,
                         recppres)
OCISubscription **subscrhp;
```

```
                char * subscriptionName;
                dvoid * func;
                ub4 recpproto;
                char * recpaddr;
                ub4 recppres;
                {
                    /* allocate subscription handle */
                    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)subscrhp,
                        (ub4) OCI_HTYPE_SUBSCRIPTION,
                        (size_t) 0, (dvoid **) 0);

                    /* set subscription name in handle */
                    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                        (dvoid *) subscriptionName,
                        (ub4) strlen((char *)subscriptionName),
                        (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

                    /* set callback function in handle */
                    if (func)
                      (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                          (dvoid *) func, (ub4) 0,
                          (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

                    /* set context in handle */
                    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                        (dvoid *) 0, (ub4) 0,
                        (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

                    /* set namespace in handle */
                    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                        (dvoid *) &namespace, (ub4) 0,
                        (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);
                    checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,
                        OCI_DEFAULT));

                    /* set receive with protocol in handle */
                    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                        (dvoid *) &recpproto, (ub4) 0,
                        (ub4) OCI_ATTR_SUBSCR_RECPTPROTO, errhp);

                    /* set recipient address in handle */
                    if (recpaddr)
                      (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                          (dvoid *) &recpaddr, (ub4) strlen(recpaddr),
                          (ub4) OCI_ATTR_SUBSCR_RECPT, errhp);
```

```
      /* set receive with presentation in handle */
      (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
          (dvoid *) &recppres, (ub4) 0,
          (ub4) OCI_ATTR_SUBSCR_RECPTPRES, errhp);
}
...
```

If user SCOTT logs on to the database, the client is notified by e-mail, and the
callback function notifySnoop is called. An e-mail notification will be sent to the
address xyz@company.com and the PL/SQL procedure plsqlnotifySnoop will
also be called in the database.

Publish-Subscribe LDAP Registration Example

The following code fragment illustrates how to do LDAP registration. Please read
all the program comments:

```
...

/* TO use LDAP registration feature, OCI_EVENTS | OCI_USE_LDAP must be set
   in OCIInitialize: */

(void) OCIInitialize((ub4) OCI_EVENTS|OCI_OBJECT|OCI_USE_LDAP, (dvoid *)0,
                      (dvoid * (*)(dvoid *, size_t)) 0,
                      (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                      (void (*)(dvoid *, dvoid *)) 0 );

...

/* set LDAP attributes in the environment handle */

/* LDAP host name */
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"yow", 3
                  OCI_ATTR_LDAP_HOST, (OCIError *)errhp);

/* LDAP server port */
ldap_port = 389;
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&ldap_port,
                  (ub4)0, OCI_ATTR_LDAP_PORT, (OCIError *)errhp);

/* bind DN of the client, normally the enterprise user name */
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"cn=orcladmin",
```

```
                       12, OCI_ATTR_BIND_DN, (OCIError *)errhp);

     /* password of the client */
     (void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"welcome",
                       7, OCI_ATTR_LDAP_CRED, (OCIError *)errhp);

     /* authentication method is "simple", username/password authentication */
     ldap_auth = 0x01;
     (void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&ldap_auth,
                       (ub4)0, OCI_ATTR_LDAP_AUTH, (OCIError *)errhp);

     /* adminstrative context: this is the DN above cn=oraclecontext */
     (void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"cn=acme,cn=com",
                       14, OCI_ATTR_LDAP_CTX, (OCIError *)errhp);

     ...

     /* retrieve the LDAP attributes from the environment handle */

     /* LDAP host */
     (void) OCIAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                       &szp,  OCI_ATTR_LDAP_HOST,  (OCIError *)errhp);

     /* LDAP server port */
     (void) OCIAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&intval,
                       0,  OCI_ATTR_LDAP_PORT,  (OCIError *)errhp);

     /* client binding DN */
     (void) OCIAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                       &szp,  OCI_ATTR_BIND_DN,  (OCIError *)errhp);

     /* client password */
     (void) OCIAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                       &szp,  OCI_ATTR_LDAP_CRED,  (OCIError *)errhp);

     /* adminstrative context */
     (void) OCIAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                       &szp,  OCI_ATTR_LDAP_CTX,  (OCIError *)errhp);

     /* client authentication method */
     (void) OCIAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&intval,
                       0,  OCI_ATTR_LDAP_AUTH,  (OCIError *)errhp);

     ...
```

```
/* to set up the server DN descriptor in the subscription handle */

/* allocate a server DN descriptor, dn is of type "OCIServerDNs **",
   subhp is of type "OCISubscription **" */
(void) OCIDescriptorAlloc((dvoid *)envhp, (dvoid **)dn,
                     (ub4) OCI_DTYPE_SRVDN, (size_t)0, (dvoid **)0);

/* now *dn is the server DN descriptor, add the DN of the first database
   that we want to register */
(void) OCIAttrSet((dvoid *)*dn, (ub4) OCI_DTYPE_SRVDN,
                 (dvoid *)"cn=server1,cn=oraclecontext,cn=acme,cn=com",
                 42, (ub4)OCI_ATTR_SERVER_DN, errhp);
/* add the DN of another database in the descriptor */
(void) OCIAttrSet((dvoid *)*dn, (ub4) OCI_DTYPE_SRVDN,
                 (dvoid *)"cn=server2,cn=oraclecontext,cn=acme,cn=com",
                 42, (ub4)OCI_ATTR_SERVER_DN, errhp);

/* set the server DN descriptor into the subscription handle */
(void) OCIAttrSet((dvoid *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
             (dvoid *) *dn, (ub4)0, (ub4) OCI_ATTR_SERVER_DNS, errhp);

...

/* now we will try to get the server DN information from the subscription
   handle */

/* first, get the server DN descriptor out */
(void) OCIAttrGet((dvoid *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                 (dvoid *)dn, &szp, OCI_ATTR_SERVER_DNS, errhp);

/* then, get the number of server DNs in the descriptor */
(void) OCIAttrGet((dvoid *) *dn, (ub4)OCI_DTYPE_SRVDN, (dvoid *)&intval,
                 &szp, (ub4)OCI_ATTR_DN_COUNT, errhp);

/* allocate an array of char * to hold server DN pointers returned by
   oracle */
  if (intval)
  {
    arr = (char **)malloc(intval*sizeof(char *));
    (void) OCIAttrGet((dvoid *)*dn, (ub4)OCI_DTYPE_SRVDN, (dvoid *)arr,
                     &intval, (ub4)OCI_ATTR_SERVER_DN, errhp);
  }

/* OCISubscriptionRegister() calls have two modes: OCI_DEFAULT and
   OCI_REG_LDAPONLY. If OCI_DEFAULT is used, there should be only one
```

server DN in the server DN descriptor. The registration request will
be sent to the database. If a database connection is not available,
the registration request will be detoured to the LDAP server. On the
other hand, if mode OCI_REG_LDAPONLY is used the registration request
will be directly sent to LDAP. This mode should be used when there are
more than one server DNs in the server DN descriptor, or we are sure
that a database connection is not available.

```
    In this example, two DNs are entered; so we should use mode
    OCI_REG_LDAPONLY in register. */
OCISubscriptionRegister(svchp, subhp, 1, errhp, OCI_REG_LDAPONLY);

 ...

 /* as OCISubscriptionRegister(), OCISubscriptionUnregister() also has
    mode OCI_DEFAULT and OCI_REG_LDAPONLY. The usage is the same. */

 OCISubscriptionUnRegister(svchp, *subhp, errhp, OCI_REG_LDAPONLY);
}

...
```

Part II

OCI Object Concepts

This part of the book contains chapters that describe the use of user-defined objects with OCI:

- Chapter 10, "OCI Object-Relational Programming", provides an introduction to object concepts and object-relational programming with OCI.

- Chapter 11, "Object-Relational Datatypes", discusses object datatypes and how you can represent database objects as C structures. This chapter also describes OCI functions that map and manipulate datatypes. It then covers binding and defining object-relational datatypes.

- Chapter 12, "Direct Path Loading" A discussion of the direct path loading of data is presented.

- Chapter 13, "Object Cache and Object Navigation", describes the object cache and how to navigate between objects.

- Chapter 14, "The Object Type Translator (OTT)", discusses how the OTT is used to convert database type definitions into host language representations.

10

OCI Object-Relational Programming

This chapter introduces the OCI's facility for working with objects in an Oracle database server. It also discusses the OCI's object navigational function calls. The following sections are included in this chapter:

- Overview of Objects
- OCI Object Overview
- Working with Objects in OCI
- Developing an OCI Object Application
- Type Inheritance

Overview of Objects

This chapter is divided into several sections that cover the basic concepts involved in writing OCI applications to manipulate Oracle objects and the OCI navigational function calls.

> **See Also:** The following chapters contain additional information about using the OCI to work with objects:

- Chapter 11, "Object-Relational Datatypes", discusses the datatypes used by OCI object-relational applications. This information supplements that found in Chapter 3, "Datatypes". This chapter also includes a discussion of the OCI datatype mapping and manipulation functions.

- Chapter 12, "Direct Path Loading", discusses information about bind and define operations specific to object-relational datatypes. This information supplements that in Chapter 2, "OCI Programming Basics", and Chapter 5, "Binding and Defining".

- Chapter 13, "Object Cache and Object Navigation", discusses the object cache and object navigation. This chapter includes a discussion of the OCI navigational functions.

- Chapter 14, "The Object Type Translator (OTT)" discusses the Object Type Translator.

- Complete descriptions of the OCI object-relational functions are contained in Chapter 17, "OCI Navigational and Type Functions", and Chapter 18, "OCI Datatype Mapping and Manipulation Functions".

- Additionally, some object functionality is included in those functions described in the other reference chapters.

OCI Object Overview

OCI provides functions for managing database access and processing SQL statements. These functions are described in detail in Part I of this guide.

OCI allows applications to access any of the datatypes found in the Oracle database server, including scalar values, collections, and instances of any object type. This includes all of the following:

- objects

- variable-length arrays (VARRAYs)

- nested tables (multisets)

- references (REFs)

- LOBs

To take full advantage of Oracle server object capabilities, most applications need to do more than just access objects. After an object has been retrieved, the application must navigate through references from that object to other objects. OCI provides the capability to do this. Through OCI's object *navigational calls*, an application can perform any of the following functions on objects:

- creating, accessing, locking, deleting, copying, and flushing objects

- getting references to the objects and their meta-objects

- dynamically getting and setting values of objects' attributes

The OCI navigational calls are discussed in more detail later in this chapter.

OCI also provides the ability to access type information stored in an Oracle database. The OCIDescribeAny() function enables an application to access most information relating to types stored in the database, including information about methods, attributes, and type meta-data.

> **See Also:** OCIDescribeAny() is discussed in Chapter 6, "Describing Schema Metadata"

Applications interacting with Oracle objects need a way to represent those objects in a host language format. Oracle provides a utility called the Object Type Translator (OTT), which can convert type definitions in the database to C struct declarations. The declarations are stored in a header file that can be included in an OCI application.

When type definitions are represented in C, the types of attributes are mapped to special C variable types. The OCI includes a set of *datatype mapping and manipulation functions* that enable an application to manipulate these datatypes, and thus manipulate the attributes of objects.

> **See Also:** These functions are discussed in more detail in Chapter 11, "Object-Relational Datatypes"

The terminology for objects can occasionally become confusing. In the remainder of this chapter, the terms *object* and *instance* both refer to an object that is either stored in the database or is present in the object cache.

Working with Objects in OCI

Many of the programming principles that govern a relational OCI application are the same for an object-relational application. An object-relational application uses the standard OCI calls to establish database connections and process SQL statements. The difference is that the SQL statements issued retrieve object references, which can then be manipulated with OCI's object functions. An object can also be directly manipulated as a value instance (without using its object reference).

Basic Object Program Structure

The basic structure of an OCI application that uses objects is essentially the same as that for a relational OCI application, as described in the section "OCI Program Structure" on page 2-2. That paradigm is reproduced here, with extra information covering basic object functionality.

1. Initialize the OCI programming environment. You *must* initialize the environment in object mode.

    Your application will most likely also need to include C struct representations of database objects in a header file.

    > **See Also:** These structs can be created by the programmer, or, more easily, they can be generated by the Object Type Translator (OTT), as described in Chapter 14, "The Object Type Translator (OTT)"

2. Allocate necessary handles, and establish a connection to a server.

3. Prepare a SQL statement for execution. This is a local (client-side) step, which may include binding placeholders and defining output variables. In an object-relational application, this SQL statement should return a reference (REF) to an object.

> **Note:** It is also possible to fetch an entire object, rather than just a reference (REF). If you SELECT a referenceable object, rather than pinning it, you get that object *by value.* Alternately, you can select a non-referenceable object, as described in "Fetching Embedded Objects" on page 10-16.

4. Associate the prepared statement with a database server, and execute the statement.

5. Fetch returned results.

   In an object-relational application, this step entails retrieving the REF, and then pinning the object to which it refers. Once the object is pinned, your application will do some or all of the following:

   - Manipulate the attributes of the object and mark it as *dirty*

   - Follow a REF to another object or series of objects

   - Access type and attribute information

   - Navigate a complex object retrieval graph

   - Flush modified objects to the server

6. Commit the transaction. This step implicitly flushes all modified objects to the server and commits the changes.

7. Free statements and handles not to be reused or re-execute prepared statements again.

All of these steps are discussed in more detail in the remainder of this chapter.

> **See Also:**
>
> - For information about using the OCI to connect to a server, process SQL statements, and allocate handles, see Chapter 2, "OCI Programming Basics" and the description of the OCI relational functions in Chapter 15, "OCI Relational Functions"
>
> - For information about OTT, refer to the section "Representing Objects in C Applications" on page 10-8, and Chapter 14, "The Object Type Translator (OTT)"

Persistent Objects, Transient Objects, and Values

Instances of an Oracle type are categorized into *persistent objects* and *transient objects* based on their lifetime. Instances of persistent objects can be further divided into *standalone objects* and *embedded objects* depending on whether or not they are referenceable by way of an object identifier.

> **Note:** The terms *object* and *instance* are used interchangeably in this manual.

> **See Also:** For more information about objects, refer to the *Oracle9i Application Developer's Guide - Object-Relational Features.*

Persistent Object

A persistent object is an object which is stored in an Oracle database. It may be fetched into the object cache and modified by an OCI application. The lifetime of a persistent object can exceed that of the application which is accessing it. Once it is created, it remains in the database until it is explicitly deleted. There are two types of persistent objects:

- Standalone instances are stored in rows of a object table, and each one has a unique object identifier. An OCI application can retrieve a REF to a standalone instance, pin the object and navigate from the pinned object to other related objects. Standalone object may also be referred to as *referenceable objects.*

  It is also possible to SELECT a referenceable object, in which case you fetch the object *by value* instead of fetching its REF.

- Embedded instances are not stored as rows in a object table. They are embedded within other structures. Examples of embedded objects are objects which are attributes of another object, or instances which exist in an object column of a database table. Embedded instances do not have object identifiers, and OCI applications cannot get REFs to embedded instances.

  Embedded objects may also be referred to as *non-referenceable objects* or *value instances.* You may sometimes see them referred to as *values,* which is not to be confused with scalar data values. The context should make the meaning clear.

The following SQL examples demonstrate the difference between these two types of persistent objects.

Example 1, Standalone Objects

```
CREATE TYPE person_t AS OBJECT
```

```
    (name       varchar2(30),
     age        number(3));
CREATE TABLE person_tab OF person_t;
```

Objects which are stored in the object table `person_tab` are standalone instances. They have object identifiers and are referenceable. They can be pinned in an OCI application.

Example 2, Embedded Objects
```
CREATE TABLE department
   (deptno      number,
    deptname    varchar2(30),
    manager     person_t);
```

Objects which are stored in the `manager` column of the `department` table are embedded objects. They do not have object identifiers, and they are not referenceable. This means they cannot be pinned in an OCI application, and they also never need to be unpinned. They are always retrieved into the object cache *by value*.

Transient Objects

A transient object is an instance of an object type. It may have an object identifier. Its lifetime cannot exceed that of the application. The application can also delete a transient object at any time.

Transient objects are often created by the application using the `OCIObjectNew()` function to store temporary values for computation. Transient objects cannot be converted to persistent objects. Their role is fixed at the time they are instantiated.

> **See Also:** See the section "Creating Objects" on page 10-33 for more information about using `OCIObjectNew()`.

Values

In the context of this manual, a *value* refers to either:

- a scalar value which is stored in a non-object column of a database table. An OCI application can fetch values from a database by issuing SQL statements.
- an embedded or non-referenceable object.

The context should make it clear which meaning is intended.

> **Note:** It is possible to SELECT a referenceable object into the
> object cache, rather than pinning it, in which case you fetch the
> object *by value* instead of fetching its REF.

Developing an OCI Object Application

This section discusses the steps involved in developing a basic OCI object application. Each step discussed in the section "Basic Object Program Structure" on page 10-4 is described here in more detail.

The following figure shows a simple program logic flow for how an application might work with objects. For simplicity, some required steps are omitted. Each step in this diagram is discussed in the following sections.

Figure 10–1   Basic Object Operational Flow
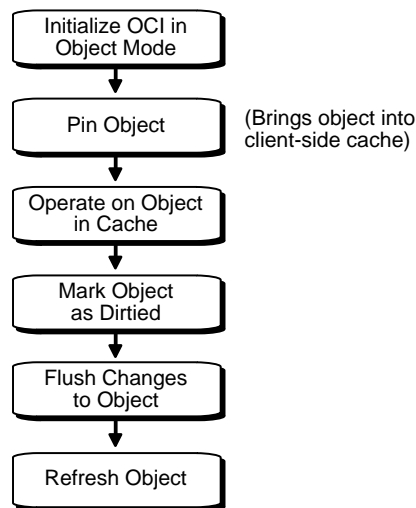
Representing Objects in C Applications

Before an OCI application can work with object types, those types must exist in the database. Typically, you create types with SQL DDL statements, such as CREATE TYPE.

When the Oracle server processes the type definition DDL commands, it stores the type definitions in the data dictionary as *type descriptor objects* (TDOs).

When your application retrieves instances of object types from the database, it needs to have a client-side representation of the objects. In a C program, the representation of an object type is a `struct`. In an OCI object application, you may also include a NULL indicator structure corresponding to each object type structure.

> **See Also:** Application programmers who wish to utilize object representations other than the default structs generated by the object cache should refer to "The Object Cache and Memory Management" on page 13-2.

Oracle provides a utility called the Object Type Translator (OTT), which generates C struct representations of database object types for you. For example, if you have a type in your database declared as

```
CREATE TYPE emp_t AS OBJECT
( name        VARCHAR2(30),
  empno       NUMBER,
  deptno      NUMBER,
  hiredate    DATE,
  salary      NUMBER);
```

OTT produces the following C struct and corresponding NULL indicator struct:

```
struct emp_t
{
  OCIString   * name;
  OCINumber   empno;
  OCINumber   deptno;
  OCIDate     hiredate;
  OCINumber   salary;
};
typedef struct emp_t emp_t

struct emp_t_ind
{
  OCIInd      _atomic;
  OCIInd      name;
  OCIInd      empno;
  OCIInd      deptno;
  OCIInd      hiredate;
  OCIInd      salary;
};
typedef struct emp_t_ind emp_t_ind;
```

The variable types used in the struct declarations are special types employed by the OCI object calls. A subset of OCI functions manipulate data of these types.

> **See Also:** These functions are mentioned later in this chapter, and are discussed in more detail in Chapter 11, "Object-Relational Datatypes".

These struct declarations are automatically written to a .h file whose name is determined by the OTT input parameters. You can include this header file in the code files for an application to provide access to objects.

> **See Also:**
>
> - For more information about OTT, see Chapter 14, "The Object Type Translator (OTT)".
>
> - For more information on the use of the NULL indicator struct, see the section "Nullity" on page 10-30.

Initializing Environment and Object Cache

If your OCI application will be accessing and manipulating objects, it is essential that you specify a value of OCI_OBJECT for the *mode* parameter of the OCIEnvCreate() call, which is the first OCI call in any OCI application. Specifying this value for *mode* indicates to the OCI libraries that your application will be working with objects. This notification has the following important effects:

- it establishes the *object run-time environment*

- it sets up the *object cache*

Memory for the object cache is allocated on demand when objects are loaded into the cache.

If the *mode* parameter of OCIInitialize() is not set to OCI_OBJECT, any attempt to use an object-related function will result in an error.

The client-side object cache is allocated in the program's process space. This cache is the memory for objects that have been retrieved from the server and are available to your application.

> **Note:** If you initialize the OCI environment in object mode, your application allocates memory for the object cache, whether or not the application actually uses object calls.

> **See Also:** The object cache is mentioned throughout this chapter. For a detailed explanation of the object cache, see Chapter 13, "Object Cache and Object Navigation".

Making Database Connections

Once the OCI environment has been properly initialized, the application can connect to a server. This is accomplished through the standard OCI connect calls described in "OCI Programming Steps" on page 2-20. When using these calls, no additional considerations need to be made because this application will be accessing objects.

There is only one object cache allocated per OCI environment. All objects retrieved or created through different connections within the environment use the same physical object cache. Each connection has its own logical object cache.

Retrieving an Object Reference from the Server

In order to work with objects, your application must first retrieve one or more objects from the server. You accomplish this by issuing a SQL statement that returns REFs to one or more objects.

> **Note:** It is also possible for a SQL statement to fetch embedded objects, rather than REFs, from a database. See the section "Fetching Embedded Objects" on page 10-16 for more information.

In the following example, the application declares a text block that stores a SQL statement designed to retrieve a REF to a single employee object from a object table of employees (emp_tab) in the database, given a particular employee number which is passed as an input variable (:emp_num) at runtime:

```
text *selemp = (text *) "SELECT REF(e)
                        FROM emp_tab e
                        WHERE empno = :emp_num";
```

Your application should prepare and process this statement in the same way that it would handle any relational SQL statement, as described in Chapter 2, "OCI Programming Basics":

- Prepare an application request, using OCIStmtPrepare().
- Bind the host input variable using the appropriate bind call(s).

- Declare and prepare an output variable to receive the employee object reference. Here you would use an employee object reference, like the one declared in "Representing Objects in C Applications" on page 10-8:

```
OCIRef  *emp1_ref = (OCIRef *) 0; /* reference to an employee object */
```

  When defining the output variable, set the *dty* datatype parameter for the define call to SQLT_REF, the datatype constant for REF.

- Execute the statement with OCIStmtExecute().

- Fetch the resulting REF into emp1_ref, using OCIStmtFetch().

At this point, you could use the object reference to access and manipulate an object or objects from the database.

> **See Also:**
>
> - For general information about preparing and executing SQL statements, see the section "OCI Programming Steps" on page 2-20. For specific information about binding and defining REF variables, refer to the sections "Advanced Bind Operations" on page 5-10 and "Advanced Define Operations" on page 5-22.
>
> - For a code example showing REF retrieval and pinning, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".

Pinning an Object

Upon completion of the fetch step, your application has a REF, or pointer, to an object. The actual object is not currently available to work with. Before you can manipulate an object, it must be *pinned*. Pinning an object loads the object instance into the object cache, and enables you to access and modify the instance's attributes and follow references from that object to other objects, if necessary. Your application also controls when modified objects are written back to the server.

> **Note:** This section deals with a simple pin operation involving a single object at a time. For information about retrieving multiple objects through complex object retrieval, see the section "Complex Object Retrieval" on page 10-22.

An application pins an object by calling the function `OCIObjectPin()`. The parameters for this function allow you to specify the *pin option, pin duration,* and *lock option* for the object.

The following sample code illustrates a pin operation for the employee reference we retrieved in the previous section:

```
if (OCIObjectPin(env, err, &emp1_ref, (OCIComplexObject *) 0,
    OCI_PIN_ANY,
    OCI_DURATION_TRANS,
    OCI_LOCK_X,  &emp1) != OCI_SUCCESS)
    process_error(err);
```

In this example, `process_error()` represents an error-handling function. If the call to `OCIObjectPin()` returns anything but OCI_SUCCESS, the error-handling function is called. The parameters of the `OCIObjectPin()` function are as follows:

- `env` is the OCI environment handle.

- `err` is the OCI error handle.

- `emp1_ref` is the reference that was retrieved through SQL.

- `(OCIComplexObject *) 0` indicates that this pin operation is not utilizing complex object retrieval.

- `OCI_PIN_ANY` is the pin option. See "Pinning an Object Copy" on page 13-7 for more information.

- `OCI_DURATION_TRANS` is the pin duration. See "Object Duration" on page 13-15 for more information.

- `OCI_LOCK_X` is the lock option. See "Locking Objects For Update" on page 13-13 for more information.

- `emp1` is an out parameter, which returns a pointer to the pinned object.

Now that the object has been pinned, the OCI application can modify that object. In this simple example, the object contains no references to other objects.

> **See Also:** For an example of navigation from one instance to another, see the section "Simple Object Navigation" on page 13-18.

Array Pin

Given an array of references, an OCI application can pin an array of objects by calling `OCIObjectArrayPin()`. The references may point to objects of different

types. This function provides the ability for fetching objects of different types from different tables in one network round trip.

Manipulating Object Attributes

Once an object has been pinned, an OCI application can modify its attributes. The OCI provides a set of function for working with datatypes of object type structs, known as the OCI datatype mapping and manipulation functions.

> **Note:** Changes made to objects pinned in the object cache affect only those object copies (instances), and *not* the original object in the database. In order for changes made by the application to reach the database, those changes must be flushed/committed to the server. See "Marking Objects and Flushing Changes" on page 10-15 for more information.

For example, assume that the employee object in the previous section was pinned so that the employee's salary could be increased. Assume also that at this company, yearly salary increases are prorated for employees who have been at the company for less than 180 days.

For this example we will need to access the employee's hire date and check whether it is more or less than 180 days prior to the current date. Based on that calculation, the employee's salary is increased by either $5000 (for more than 180 days) or $3000 (for less than 180 days). The sample code on the following page demonstrates this process.

Note that the datatype mapping and manipulation functions work with a specific set of datatypes; you must convert other types, like **int**, to the appropriate OCI types before using them in calculations.

```
/* assume that sysdate has been fetched into sys_date, a string. */
/* emp1 and emp1_ref are the same as in previous sections. */
/* err is the OCI error handle. */
/* NOTE: error handling code is not included in this example. */

sb4 num_days;        /* the number of days between today and hiredate */
OCIDate curr_date;          /* holds the current date for calculations */
int raise;   /* holds the employee's raise amount before calculations */
OCINumber raise_num;       /* holds employee's raise for calculations */
OCINumber new_sal;                 /* holds the employee's new salary */
```

```
/* convert date string to an OCIDate */
OCIDateFromText(err, (text *) sys_date, (ub4) strlen(sys_date), (text *)
         NULL, (ub1) 0, (text *) NULL, (ub4) 0, &curr_date);

  /* get number of days between hire date and today */
OCIDateDaysBetween(err, &curr_date, &emp1->hiredate, &num_days);

/* calculate raise based on number of days since hiredate */
if num_days > 180
    raise = 5000
else
    raise = 3000;

/* convert raise value to an OCINumber */
OCINumberFromInt(err, (dvoid *)&raise, (uword)sizeof(raise),
                OCI_NUMBER_SIGNED, &raise_num);

/* add raise amount to salary */
OCINumberAdd(err, &raise_num, &emp1->salary, &new_sal);
OCINumberAssign(err, &new_sal, &emp1->salary);
```

This example points out how values must be converted to OCI datatypes (e.g., **OCIDate**, **OCINumber**) before being passed as parameters to the OCI datatype mapping and manipulation functions.

> **See Also:** For more information about the OCI datatypes and the datatype mapping and manipulation functions, refer to Chapter 11, "Object-Relational Datatypes".

Marking Objects and Flushing Changes

In the example in the previous section, an attribute of an object instance was changed. At this point, however, that change exists only in the client-side object cache. The application must take specific steps to insure that the change is written in the database.

The first step is to indicate that the object has been modified. This is done with the OCIObjectMarkUpdate() function. This function marks the object as *dirty* (modified).

Objects that have had their dirty flag set must be flushed to the server for the changes to be recorded in the database. You can do this in three ways:

- Flush a single dirty object by calling OCIObjectFlush().

- Flush the entire cache using `OCICacheFlush()`. In this case the OCI traverses the dirty list maintained by the cache and flushes the dirty objects to the server.

- Call `OCITransCommit()` to commit a transaction. Doing so also traverses the dirty list and flushes objects to the server.

The flush operations work only on persistent objects in the cache. Transient objects are never flushed to the server.

Flushing an object to the server can activate triggers in the database. In fact, on some occasions an application may want to explicitly flush objects just to fire triggers on the server side.

> **See Also:**
>
> - For more information about `OCITransCommit()` see the section "OCI Support for Transactions" on page 8-2
>
> - For information about transient and persistent objects, see the section "Creating Objects" on page 10-33
>
> - For information about seeing and checking object meta-attributes, such as *dirty,* see the section "Object Meta-Attributes" on page 10-18

Fetching Embedded Objects

If your application needs to fetch an embedded object instance—an object stored in a column of a regular table, rather than an object table—you cannot use the REF retrieval mechanism described in the section "Retrieving an Object Reference from the Server" on page 10-11. Embedded instances do not have object identifiers, so it is not possible to get a REF to them. This means that they cannot serve as the basis for object navigation. There are still many situations, however, in which an application will want to fetch embedded instances.

For example, assume that an `address` type has been created.

```
CREATE TYPE address AS OBJECT
( street1          varchar2(50),
  street2          varchar2(50),
  city             varchar2(30),
  state            char(2),
  zip              number(5))
```

You could then use that type as the datatype of a column in another table:

```
CREATE TABLE clients
```

```
( name            varchar2(40),
  addr            address)
```

Your OCI application could then issue the following SQL statement:

```
SELECT addr FROM clients
WHERE name='BEAR BYTE DATA MANAGEMENT'
```

This statement would return an embedded `address` object from the `clients` table. The application could then use the values in the attributes of this object for other processing.

Your application should prepare and process this statement in the same way that it would handle any relational SQL statement, as described in Chapter 2, "OCI Programming Basics":

- Prepare an application request, using `OCIStmtPrepare()`.

- Bind the input variable using the appropriate bind call(s).

- Define an output variable to receive the `address` instance. You use a C struct representation of the object type that was generated by OTT, as described in the section "Representing Objects in C Applications" on page 10-8:

  ```
  addr1      *address; /* variable of the address struct type */
  ```

  When defining the output variable, set the *dty* datatype parameter for the define call to SQLT_NTY, the datatype constant for named data types.

- Execute the statement with `OCIStmtExecute()`

- Fetch the resulting instance into addr1, using `OCIStmtFetch()`.

Following this, you can access the attributes of the instance, as described in the section "Manipulating Object Attributes" on page 10-14, or pass the instance as an input parameter for another SQL statement.

> **Note:** Changes made to an embedded instance can be made persistent only by executing a SQL UPDATE statement.

> **See Also:** For more information about preparing and executing SQL statements, see the section "OCI Programming Steps" on page 2-20.

Object Meta-Attributes

An object's *meta-attributes* serve as flags which can provide information to an application, or to the object cache, about the status of an object. For example, one of the meta-attributes of an object indicates whether or not it has been flushed to the server. These can help an application control the behavior of instances.

Persistent and transient object instances have different sets of meta-attributes. The meta-attributes for persistent objects are further broken down into *persistent meta-attributes* and *transient meta-attributes*. Transient meta-attributes exist only when an instance is in memory. Persistent meta-attributes also apply to objects stored in the server.

Persistent Object Meta-Attributes

The following table shows the meta-attributes for *standalone* persistent objects.

Persistent Meta-Attributes	Meaning
existent	does the object exist?
nullity	null information of the instance
locked	has the object been locked?
dirty	has the object been marked as *dirtied*?
Transient Meta-Attributes	
pinned	is the object pinned?
allocation duration	see "Object Duration" on page 13-15
pin duration	see "Object Duration" on page 13-15

> **Note:** Embedded persistent objects only have the *nullity* and *allocation duration* attributes, which are transient.

The OCI provides the OCIObjectGetProperty() function, which allows an application to check the status of a variety of attributes of an object. The syntax of the function is:

```
sword OCIObjectGetProperty ( OCIEnv            *envh,
                             OCIError          *errh,
```

```
CONST dvoid          *obj,
OCIObjectPropId      propertyId,
dvoid                *property,
ub4                  *size );
```

The *propertyId* and *property* parameters are used to retrieve information about any of a variety of properties or attributes

The different property ids and the corresponding type of *property* argument are given below.

> **See Also:** For more information, see
> OCIObjectGetProperty() on page 17-26.

OCI_OBJECTPROP_LIFETIME
This identifies whether the given object is a persistent object or a transient object or a value instance. The *property* argument must be a pointer to a variable of type **OCIObjectLifetime**. Possible values include:

- OCI_OBJECT_PERSISTENT

- OCI_OBJECT_TRANSIENT

- OCI_OBJECT_VALUE

OCI_OBJECTPROP_SCHEMA
This returns the schema name of the table in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the schema name an error is returned, the error message will communicate the required size. Upon success, the size of the returned schema name in bytes is returned by *size*. The *property* argument must be an array of type **text** and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_TABLE
This returns the table name in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the table name an error is returned, the error message will communicate the required size. Upon success, the size of the returned table name in bytes is returned by *size*. The *property* argument must be an array of type text and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_PIN_DURATION

This returns the pin duration of the object. An error is returned if the given object points to a value instance. The `property` argument must be a pointer to a variable of type **OCIDuration**. Valid values include:

- OCI_DURATION_SESSION

- OCI_DURATION_TRANS

  See Also: For more information about durations, see "Object Duration" on page 13-15.

OCI_OBJECTPROP_ALLOC_DURATION

This returns the allocation duration of the object. The `property` argument must be a pointer to a variable of type **OCIDuration**. Valid values include:

- OCI_DURATION_SESSION

- OCI_DURATION_TRANS

  See Also: For more information about durations, see "Object Duration" on page 13-15.

OCI_OBJECTPROP_LOCK

This returns the lock status of the object. The possible lock status is enumerated by **OCILockOpt**. An error is returned if the given object points to a transient or value instance. The `property` argument must be a pointer to a variable of type **OCILockOpt**. Note, the lock status of an object can also be retrieved by calling `OCIObjectIsLocked()`.

OCI_OBJECTPROP_MARKSTATUS

This returns the dirty status and indicates whether the object is a new object, updated object or deleted object. An error is returned if the given object points to a transient or value instance. The `property` argument must be of type **OCIObjectMarkStatus**. Valid values include:

- OCI_OBJECT_NEW

- OCI_OBJECT_DELETED

- OCI_OBJECT_UPDATED

The following macros are available to test the object mark status:

- OCI_OBJECT_IS_UPDATED (flag)

- OCI_OBJECT_IS_DELETED (flag)
- OCI_OBJECT_IS_NEW (flag)
- OCI_OBJECT_IS_DIRTY (flag)

OCI_OBJECTPROP_VIEW
This identifies whether the specified object is a view object or not. If the property value returned is TRUE, it indicates the object is a view otherwise it is not. An error is returned if the given object points to a transient or value instance. The *property* argument must be of type boolean.

Additional Attribute Functions
The OCI also provides routines which allow an application to set or check some of these attributes directly or indirectly, as shown in the following table:

Meta-Attribute	Set With	Check With
nullity	\<none\>	OCIObjectGetInd()
existence	\<none\>	OCIObjectExists()
locked	OCIObjectLock()	OCIObjectIsLocked()
dirty	OCIObjectMark()	OCIObjectIsDirty()

Transient Object Meta-Attributes
Transient objects have no persistent attributes, and the following transient attributes:

Transient Meta-Attributes	Meaning
existent	does the object exist?
pinned	is the object being accessed by the application?
dirty	has the object been marked as *dirtied*?
nullity	null information of the instance
allocation duration	see "Object Duration" on page 13-15
pin duration	see "Object Duration" on page 13-15

Complex Object Retrieval

In the examples earlier in this chapter, only a single instance at a time was fetched or pinned. In these cases, each pin operation involved a separate server round trip to retrieve the object.

Object-oriented applications often model their problems as a set of interrelated objects that form graphs of objects. The applications process objects by starting at some initial set of objects, and then using the references in these initial objects to traverse the remaining objects. In a client-server setting, each of these traversals could result in costly network roundtrips to fetch objects.

Application performance when dealing with objects may be increased through the use of *complex object retrieval (COR)*. This is a prefetching mechanism in which an application specifies a criteria for retrieving a set of linked objects in a single operation.

> **Note:** As described below, this does not mean that these prefetched objects are all pinned. They are fetched into the object cache, so that subsequent pin calls are local operations.

A *complex object* is a set of logically related objects consisting of a root object, and a set of objects each of which is prefetched based on a given *depth level*. The *root object* is explicitly fetched or pinned. The depth level is the shortest number of references that need to be traversed from the root object to a given prefetched object in a complex object.

An application specifies a complex object by describing its content and boundary. The fetching of complex objects is constrained by an environment's *prefetch limit*, the amount of memory in the object cache that is available for prefetching objects.

> **Note:** The use of COR does not add functionality; it only improves performance so its use is optional.

As an example for this discussion, consider the following type declaration:

```
CREATE TYPE customer(...);
CREATE TYPE line_item(...);
CREATE TYPE line_item_varray as VARRAY(100) of REF line_item;
CREATE TYPE purchase_order AS OBJECT
( po_number          NUMBER,
  cust               REF customer,
```

```
related_orders    REF purchase_order,
line_items        line_item_varray)
```

The `purchase_order` type contains a scalar value for `po_number`, a VARRAY of line items, and two references. The first is to a `customer` type, and the second is to a `purchase_order` type, indicating that this type may be implemented as a linked list.

When fetching a complex object, an application must specify the following:

1. a REF to the desired root object.

2. one or more pairs of type and depth information to specify the boundaries of the complex object. The type information indicates which REF attributes should be followed for COR, and the depth level indicates how many levels deep those links should be followed.

In the case of the purchase order object above, the application must specify the following:

1. the REF to the root purchase order object

2. one or more pairs of type and depth information for `cust`, `related_orders`, or `line_items`

An application fetching a purchase order will very likely need access to the customer information for that order. Using simple navigation, this would require two server accesses to retrieve the two objects. Through complex object retrieval, the customer can be prefetched when the application pins the purchase order. In this case, the complex object would consist of the purchase order object and the customer object it references.

In the previous example, the application would specify the `purchase_order` REF, and would indicate that the `cust` REF attribute should be followed to a depth level of 1:

1. `REF(PO object)`

2. `{(customer, 1)}`

If the application wanted to prefetch the `purchase_order` object and all objects in the object graph it contains, the application would specify that both the `cust` and `related_orders` should be followed to the maximum depth level possible.

1. `REF(PO object)`

2. `{(customer, UB4MAXVAL), (purchase_order, UB4MAXVAL)}`

where `UB4MAXVAL` specifies that all objects of the specified type reachable through references from the root object should be prefetched.

If an application wanted to fetch a PO and all the associated line items, it would specify:

1. `REF(PO object)`

2. `{(line_item, 1)}`

The application can also choose to fetch all objects reachable from the root object by way of `REF`s (transitive closure) to a certain depth. To do so, set the level parameter to the depth desired. For the above two examples, the application could also specify `(PO object REF, UB4MAXVAL)` and `(PO object REF, 1)` respectively to prefetch required objects. Doing so results in many extraneous fetches but is quite simple to specify, and requires only one server round trip.

Prefetching Objects

After specifying and fetching a complex object, subsequent fetches of objects contained in the complex object do not incur the cost of a network round trip, because these objects have already been prefetched and are in the object cache. Keep in mind that excessive prefetching of objects can lead to a flooding of the object cache. This flooding, in turn, may force out other objects that the application had already pinned leading to a performance degradation instead of performance improvement.

> **Note:** If there is insufficient memory in the cache to hold all prefetched objects, some objects may not be prefetched. The application will then incur a network round trip when those objects are accessed later.

The `SELECT` privilege is needed for all prefetched objects. Objects in the complex object for which the application does not have `SELECT` privilege will not be prefetched.

Implementing Complex Object Retrieval in the OCI

Complex Object Retrieval (COR) allows an application to prefetch a complex object while fetching the root object. The complex object specifications are passed to the same `OCIObjectPin()` function used for simple objects.

An application specifies the parameters for complex object retrieval using a *complex object retrieval handle.* This handle is of type **OCIComplexObject** and is allocated in the same way as other OCI handles.

The complex object retrieval handle contains a list of *complex object retrieval descriptors.* The descriptors are of type **OCIComplexObjectComp**, and are allocated in the same way as other OCI descriptors.

Each COR descriptor contains a type REF and a depth level. The type REF specifies a type of reference to be followed while constructing the complex object. The depth level indicates how far a particular type of reference should be followed. Specify an integer value, or the constant UB4MAXVAL for the maximum possible depth level.

The application can also specify the depth level in the COR handle without creating COR descriptors for type and depth parameters. In this case, all REFs are followed to the depth specified in the COR handle. The COR handle can also be used to specify whether a collection attribute should be fetched separately on demand (out-of-line) as opposed to the default case of fetching it along with the containing object (inline).

The application uses OCIAttrSet() to set the attributes of a COR handle. The attributes are:

> OCI_ATTR_COMPLEXOBJECT_LEVEL - the depth level
>
> OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE - fetch collection attribute in an object type out-of-line

The application allocates the COR descriptor using OCIDescriptorAlloc() and then can set the following attributes:

> OCI_ATTR_COMPLEXOBJECTCOMP_TYPE - the type REF
>
> OCI_ATTR_COMPLEXOBJECTCOMP_LEVEL - the depth level for references of the above type

Once these attributes are set, the application calls OCIParamSet() to put the descriptor into a complex object retrieval handle. The handle has an OCI_ATTR_PARAM_COUNT attribute which specifies the number of descriptors on the handle. This attribute can be read with OCIAttrGet().

Once the handle has been populated, it can be passed to the OCIObjectPin() call to pin the root object and prefetch the remainder of the complex object.

The complex object retrieval handles and descriptors must be freed explicitly when they are no longer needed.

> **See Also:** For more information about handles and descriptors, see "Handles" on page 2-5 and "Descriptors" on page 2-15.

COR Prefetching

The application specifies a complex object while fetching the root object. The prefetched objects are obtained by doing a breadth-first traversal of the graph(s) of objects rooted at a given root object(s). The traversal stops when all required objects have been prefetched, or when the total size of all the prefetched objects exceeds the *prefetch limit.*

COR interface

The interface for fetching complex objects is the OCI pin interface. The application can pass an initialized COR handle to `OCIObjectPin()` (or an array of handles to `OCIObjectArrayPin()`) to fetch the root object and the prefetched objects specified in the COR handle.

```
sword OCIObjectPin ( OCIEnv            *env,
                     OCIError          *err,
                     OCIRef            *object_ref,
                     OCIComplexObject  *corhdl,
                     OCIPinOpt         pin_option,
                     OCIDuration       pin_duration,
                     OCILockOpt        lock_option,
                     dvoid             **object );

sword OCIObjectArrayPin ( OCIEnv            *env,
                          OCIError          *err,
                          OCIRef            **ref_array,
                          ub4               array_size,
                          OCIComplexObject  **cor_array,
                          ub4               cor_array_size,
                          OCIPinOpt         pin_option,
                          OCIDuration       pin_duration,
                          OCILockOpt        lock,
                          dvoid             **obj_array,
                          ub4               *pos );
```

Note the following points when using COR:

1. A null COR handle argument defaults to pinning just the root object.

2. A COR handle with type of the root object and a depth level of 0 fetches only the root object and is thus equivalent to a null COR handle.

3. The lock options apply only to the root object.

> **Note:** In order to specify lock options for prefetched objects, the
> application can visit all the objects in a complex object, create an
> array of REFs, and lock the entire complex object in another round
> trip using the array interface (OCIObjectArrayPin()).

Example of COR

The following example illustrates how an application program can be modified to
use complex object retrieval.

Consider an application that displays a purchase order and the line items associated
with it. The code in boldface accomplishes this. The rest of the code uses complex
object retrieval for prefetching and thus enhances the application's performance.

```
OCIEnv *envhp;
OCIError *errhp;
OCIRef *liref;
OCIRef *poref;
OCIIter *itr;
boolean  eoc;
purchase_order *po = (purchase_order *)0;
line_item *li = (line_item *)0;
OCISvcCtx *svchp;
OCIComplexObject *corhp;
OCIComplexObjectComp *cordp;
OCIType *litdo;
ub4 level = 0;

/* get COR Handle */
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &corhp, (ub4)
               OCI_HTYPE_COMPLEXOBJECT, 0, (dvoid **)0);

/* get COR descriptor for type line_item */
OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &cordp, (ub4)
               OCI_DTYPE_COMPLEXOBJECTCOMP, 0, (dvoid **) 0);

/* get type of line_item to set in COR descriptor */
OCITypeByName(envhp, errhp, svchp, (const text *) 0, (ub4) 0,
               const text *) "LINE_ITEM", (ub4) strlen((const char *)
               "LINE_ITEM"), OCI_DURATION_SESSION, &litdo);

/* set line_item type in COR descriptor */
```

```
                    OCIAttrSet( (dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
                                dvoid *) litdo, (ub4) sizeof(dvoid *), (ub4)
                                OCI_ATTR_COMPLEXOBJECTCOMP_TYPE, (OCIError *) errhp);
                    level = 1;

                    /* set depth level for line_item_varray in COR descriptor */
                    OCIAttrSet( (dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
                                (dvoid *) &level, (ub4) sizeof(ub4), (ub4)
                                OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL, (OCIError *) errhp);

                    /* put COR descriptor in COR handle */
                    OCIParamSet(corhp, OCI_HTYPE_COMPLEXOBJECT, &errhp, cordp,
                                OCI_DTYPE_COMPLEXOBJECTCOMP, 1);
```

```
                    /* pin the purchase order */
                    OCIObjectPin(envhp, errhp, poref, corhp, OCI_PIN_LATEST,
                                 OCI_REFRESH_LOADED, OCI_DURATION_SESSION,
                                 OCI_LOCK_NONE, (ub2) 1, (dvoid **)&po)
```

```
                    /* free COR descriptor and COR handle */
                    OCIDescriptorFree((dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP);
                    OCIHandleFree((dvoid *) corhp, (ub4) OCI_HTYPE_COMPLEXOBJECT);
```

```
                    /* iterate and print line items for this purchase order */
                    OCIIterCreate(envhp, errhp, po.line_items, &itr);
```

```
                    /* get first line item */
                    OCIIterNext(envhp, errhp, itr, &liref, (dvoid **)0, &eoc);
                    while (!eoc)           /* not end of collection */
                    {
                    /* pin line item */
                     OCIObjectPin(envhp, errhp, liref, (dvoid *)0, OCI_PIN_RECENT,
                                  OCI_REFRESH_LOADED, OCI_DURATION_SESSION,
                                  OCI_LOCK_NONE, (ub2) 1, (dvoid **)&li);
                      display_line_item(li);
```

```
                    /* get next line item */
                    OCIIterNext(envhp, errhp, itr, &liref, (dvoid **)0, &eoc);
                    }
```

OCI Versus SQL Access to Objects

If an application needs to manipulate a graph of objects (inter-related by object references) then it is more effective to use the OCI interface rather than the SQL

interface for accessing objects. Retrieving a graph of objects using the SQL interface may require executing multiple SELECT statements which would mean multiple network roundtrips. Using the complex object retrieval capability provided by the OCI, the application can retrieve the graph of objects in one OCIObjectPin() call.

Consider the update case where the application retrieves a graph of objects and modifies it based upon user interaction and then wishes to make the modifications persistent in the database. Using the SQL interface, the application would have to execute multiple UPDATE statements to update the graph of objects. If the modifications involved creation of new objects and deletion of existing objects then corresponding INSERT and DELETE statements would also need to be executed. In addition, the application would have to do more bookkeeping, such as keeping track of table names, because this information is required for executing the INSERT/UPDATE/DELETE statements.

Using the OCI's OCICacheFlush() function, the application can flush all modifications (insertion, deletion and update of objects) in a single operation. The OCI does all the bookkeeping, thereby requiring less coding on the part of the application. So for manipulating graph of objects the OCI is not only efficient but also provides an easy to use interface.

Consider a different case in which the application needs to fetch an object given its REF. In the OCI this is achieved by pinning the object using the OCIObjectPin() call. In the SQL interface this can be achieved by dereferencing the REF in a SELECT statement (e.g. SELECT DEREF(ref) from tbl;). Consider situations where the same REF (i.e. reference to the same object) is being dereferenced multiple times in a transaction. By calling OCIObjectPin() with the OCI_PIN_RECENT option, the object will be fetched from the server only once for the transaction and repeated pins on the same REF result in returning a pointer to the already-pinned object in the cache. In the case of the SQL interface, each execution of the SELECT DEREF... statement would result in fetching the object from the server and hence would result in multiple roundtrips to the server and multiple copies of the same object.

Finally, consider the case in which the application needs to fetch a non-referenceable object. For example,

```
CREATE TABLE department
(
deptno number,
deptname varchar2(30),
manager employee_t
);
```

employee_t instances stored in the manager column are non-referenceable. Only the SQL interface can be used to fetch manager column instances. But if employee_t has any REF attributes, OCI calls can then be used to navigate the REF.

Pin Count and Unpinning

Each object in the object cache has a *pin count* associated with it. The pin count essentially indicates the number of code modules that are concurrently accessing the object. The pin count is set to 1 when an object is pinned into the cache for the first time. Objects prefetched with complex object retrieval enter the object cache with a pin count of zero.

It is possible to pin an already-pinned object. Doing so increases the pin count by one. When a process finishes using an object, it should *unpin* it, using OCIObjectUnpin(). This call decrements the pin count by one.

When the pin count of an object reaches zero, that object is eligible to be aged out of the cache if necessary, freeing up the memory space occupied by the object.

The pin count of an object can be set to zero explicitly by calling OCIObjectPinCountReset().

An application can unpin all objects in the cache related to a specific connection, by calling OCICacheUnpin().

See Also:

- See the section "Freeing an Object Copy" on page 13-9 for more information about the conditions under which objects with zero pin count are removed from the cache.

- For information about explicitly flushing an object or the entire cache, see the section "Marking Objects and Flushing Changes" on page 10-15.

- See the section "Freeing an Object Copy" on page 13-9 for more information about objects being aged out of the cache.

Nullity

If a column in a row of a database table has no value, then that column is said to be NULL, or to contain a NULL. Two different types of NULLs can apply to objects:

- Any attribute of an object can have a NULL value. This indicates that the value of that attribute of the object is not known.

- An object instance may be *atomically NULL.* This means that the value of the entire object is unknown.

Atomic nullity is not the same thing as nonexistence. An atomically NULL instance still exists, its value is just not known. It may be thought of as an existing object with no data.

When working with objects in the OCI, an application can define a *NULL indicator structure* for each object type used by the application. In most cases, doing so simply requires including the NULL indicator structure generated by OTT along with the struct declaration. When the OTT output header file is included, the NULL indicator struct becomes available to your application.

For each type, the NULL indicator structure includes an atomic NULL indicator (whose type is **OCIInd**), and a NULL indicator for each attribute of the instance. If the type has an object attribute, the NULL indicator structure includes that attribute's NULL indicator structure. The following example shows the C representations of types with their corresponding NULL indicator structures.

```
struct address
{
   OCINumber    no;
   OCIString    *street;
   OCIString    *state;
   OCIString    *zip;
};
typedef struct address address;

struct address_ind
{
  OCIInd     _atomic;
  OCIInd     no;
  OCIInd     street;
  OCIInd     state;
  OCIInd     zip;
};
typedef struct address_ind address_ind;


struct person
{
   OCIString      *fname;
   OCIString      *lname;
   OCINumber      age;
   OCIDate        birthday;
```

```
        OCIArray        *dependentsAge;
        OCITable        *prevAddr;
        OCIRaw          *comment1;
        OCILobLocator   *comment2;
        address         addr;
        OCIRef          *spouse;
};
typedef struct person person;

struct person_ind
{
  OCIInd          _atomic;
  OCIInd          fname;
  OCIInd          lname;
  OCIInd          age;
  OCIInd          birthday;
  OCIInd          dependentsAge;
  OCIInd          prevAddr;
  OCIInd          comment1;
  OCIInd          comment2;
  address_ind     addr;
  OCIInd          spouse;
};
typedef struct person_ind person_ind;
```

> **Note:** The dependentsAge field of person_ind indicates
> whether the entire varray (dependentsAge field of person) is
> atomically NULL or not. Null information of individual elements of
> dependentsAge can be retrieved through the *elemind* parameter
> of a call to OCICollGetElem(). Similarly, the prevAddr field of
> person_ind indicates whether the entire nested table (prevAddr
> field of person) is atomically NULL or not. Null information of
> individual elements of prevAddr can be retrieved through the
> *elemind* parameter of a call to OCICollGetElem().

For an object type instance, the first field of the NULL-indicator structure is the
atomic NULL indicator, and the remaining fields are the attribute NULL indicators
whose layout resembles the layout of the object type instance's attributes.

Checking the value of the atomic NULL indicator allows an application to test
whether an instance is atomically NULL. Checking any of the others allows an
application to test the NULL status of that attribute, as in the following code sample:

```
person_ind *my_person_ind
if ( my_person_ind -> _atomic = OCI_IND_NULL)
{
    /* instance is atomically NULL */
}
if ( my_person_ind -> fname = OCI_IND_NULL)
{
    /* fname attribute is NULL */
}
```

In the above example, the value of the atomic NULL indicator, or one of the attribute NULL indicators, is compared to the predefined value OCI_IND_NULL to test its nullity. The following predefined values are available for such a comparison:

- OCI_IND_NOTNULL, indicating that the value is not NULL

- OCI_IND_NULL, indicating that the value is NULL

- OCI_IND_BADNULL, indicates that an enclosing object (or parent object) is NULL. This is used by PL/SQL, and may also be referred to as an INVALID_NULL. For example if a type instance is NULL, then its attributes are INVALID_NULLs.

Use the function OCIObjectGetInd() on page 17-37 function to allocate storage for and retrieve the NULL indicator structure of an object.

> **See Also:** For more information about OTT-generated NULL indicator structs, refer to Chapter 14, "The Object Type Translator (OTT)".

Creating Objects

An OCI application can create any object using OCIObjectNew(). To create a persistent object, the application must specify the object table where the new object will reside. This value can be retrieved by calling OCIObjectPinTable(), and it is passed in the *table* parameter. To create a transient object, the application needs to pass only the type descriptor object (retrieved by calling OCIDescribeAny()) for the type of object being created.

OCIObjectNew() can also be used to create instances of scalars (e.g., REF, LOB, string, raw, number, and date) and collections (e.g., varray and nested table) by passing the appropriate value for the *typecode* parameter.

Attribute Values of New Objects

By default, all attributes of a newly created objects have NULL values. After initializing attribute data, the user must change the corresponding NULL status of each attribute to non-NULL.

It is possible to have attributes set to non-NULL values when an object is created. This is accomplished by setting the OCI_OBJECT_NEWNOTNULL attribute of the environment handle to TRUE using OCIAttrSet(). This mode can later be turned off by setting the attribute to FALSE.

If OCI_OBJECT_NEWNOTNULL is set to TRUE, then OCIObjectNew() creates a non-NULL object. The attributes of the object have the default values described in the following table, and the corresponding NULL indicators are set to NOT NULL.

Table 10–1   Attribute Values for New Objects

Attribute Type	Default Value
REF	If an object has a REF attribute, the user must set it to a valid REF before flushing the object or an error is returned.
DATE	The earliest possible date Oracle allows, which is midnight, 01-JAN-4712 BCE (equivalent to Julian day 1).
ANSI DATE	The earliest possible date Oracle allows, 01-JAN-4712 BCE (equivalent to Julian day 1).
TIMESTAMP	The earliest possible date and time Oracle allows, which is midnight, 01-JAN-4712 BCE (equivalent to Julian day 1).
TIMESTAMP WITH TIME ZONE	The earliest possible date and time Oracle allows, which is midnight, 01-JAN-4712 BCE (equivalent to Julian day 1) at UTC (0:0) time zone.
TIMESTAMP WITH LOCAL TIME ZONE	The earliest possible date and time Oracle allows, which is midnight, 01-JAN-4712 BCE (equivalent to Julian day 1) at UTC (0:0) time zone.
INTERVAL YEAR TO MONTH	INTERVAL '0-0' YEAR TO MONTH.
INTERVAL DAY TO SECOND	INTERVAL '0 0:0:0' DAY TO SECOND.
FLOAT	0.
NUMBER	0
DECIMAL	0.

Table 10–1   Attribute Values for New Objects (Cont.)

Attribute Type	Default Value
RAW	Raw data with length set to 0. Note: the default value for a RAW attribute is the same as that for a NULL RAW attribute.
VARCHAR2	OCIString with 0 length and first char set to NULL. The default value is the same as that of a null string attribute.
CHAR	OCIString with 0 length and first char set to NULL. The default value is the same as that of a null string attribute.
VARCHAR	OCIString with 0 length and first char set to NULL. The default value is the same as that of a null string attribute.
VARRAY	collection with 0 elements.
NESTED TABLE	table with 0 elements.
CLOB	empty CLOB.
BLOB	empty BLOB.
BFILE	The user must initialize the BFILE to a valid value by setting the directory alias and filename.

Freeing and Copying Objects

Use `OCIObjectFree()` to free memory allocated through `OCIObjectNew()`. Freeing an object deallocates all the memory allocated for the object, including the associated NULL indicator structure. This procedure deletes a transient, but not a persistent, object before its lifetime expires. An application should use `OCIObjectMarkDelete()` to delete a persistent object.

An application can copy one instance to another instance of the same type using `OCIObjectCopy()`.

> **See Also:**   See the descriptions of these functions in Chapter 17, "OCI Navigational and Type Functions" for more information.

Object Reference and Type Reference

The object extensions to the OCI provide the application with the flexibility to access the contents of objects using their pointers or their references. The OCI provides the function `OCIObjectGetObjectRef()` to return a reference to an object given the object's pointer.

For applications that also want to access the type information of objects, the OCI provides the function `OCIObjectGetProperty()` to return a reference to an object's type descriptor object (TDO), given a pointer to the object.

Creating Objects Based on Object Views or User-defined OIDs

Applications can use the `OCIObjectNew()` call to create objects which are based on object views, or on tables with user-defined OIDs. If `OCIObjectNew()` receives a handle to an object view or a table with a user-defined OID, then the reference it returns is a *pseudo-reference*. This pseudo-reference cannot be saved into any other object, but it can be used to fill in the object's attributes so that a primary-key-based reference can be obtained with `OCIObjectGetObjectRef()`.

This process involves the following steps:

1. Pin the object view or object table on which the new object will be based.

2. Create a new object using `OCIObjectNew()`, passing in the handle to the table/view obtained by the pin operation in step 1.

3. Fill in the necessary values for the object. These include those attributes which make up the user-defined OID for the object table or object view.

4. Use `OCIObjectGetObjectRef()` to obtain the primary-key-based reference to the object, if necessary. If desired, return to step 2 to create more objects.

5. Flush the newly created object(s) to the server.

The following sample code shows how this process might be implemented to create a new object for the `emp_view` object view in the `scott` schema:

```
void object_view_new ()
{
dvoid    *table;
OCIRef   *pkref;
dvoid    *object;
....
/* Set up the service context, error handle etc.. */
...
/* Pin the object view */
OCIObjectPinTable(envp,errorp,svctx, "scott", strlen("scott"), "emp_view",
    strlen("emp_view"),(dvoid *) 0, OCI_DURATION_SESSION, (dvoid **) &table);

/* Create a new object instance  */
OCIObjectNew(envp, errorp, svctx, OCI_TYPECODE_OBJECT,(OCIType *)0, table,
OCI_DURATION_SESSION,FALSE,&object);
```

```
                /* Populate the attributes of "object" */
                OCIObjectSetAttr(...);
                ...
                /* Allocate an object reference */
                OCIObjectNew(envp, errorp, svctx, OCI_TYPECODE_REF, (OCIType *)0, (dvoid *)0,
                   OCI_DURATION_SESSION,TRUE,&pkref);

                /* Get the reference using OCIObjectGetObjectRef */
                OCIObjectGetObjectRef(envp,errorp,object,pkref);
                ...
                /* Flush new object(s) to server */
                ...
                } /* end function */
```

Error Handling in Object Applications

Error handling in OCI applications is the same, whether or not the application uses objects.

> **See Also:**   For more information about function return codes and error messages, see the section "Error Handling" on page 2-31.

Type Inheritance

Type inheritance of objects has many similarities to inheritance in C++ and Java. You can create an object type as a *subtype* of an existing object type. The subtype is said to inherit all the attributes and methods (member functions and procedures) of the *supertype*, which is the original type. Only single inheritance is supported; an object cannot have more than one supertype. The subtype can add new attributes and methods to the ones it inherits. It can also override (redefine the implementation) of any of its inherited methods. A subtype is said to *extend* (that is, inherit from) its supertype.

> **See Also:**   *Oracle9i Application Developer's Guide - Object-Relational Features* for a more complete discussion of this topic

As an example, a type Person_t can have a subtype Student_t and a subtype Employee_t. In turn, Student_t can have its own subtype, PartTimeStudent_t. A type declaration must have the flag NOT FINAL so that it can have subtypes. The default is FINAL, which means that the type can have no subtypes.

All types discussed so far in this chapter are FINAL. All types in applications developed before release 9.0 are FINAL. A type that is FINAL can be altered to be NOT FINAL. A NOT FINAL type with no subtypes can be altered to be FINAL. Person_t is declared as NOT FINAL for our example:

```
CREATE TYPE Person_t AS OBJECT
( ssn NUMBER,
  name VARCAHR2(30),
  address VARCHAR2(100)) NOT FINAL;
```

A subtype inherits all the attributes and methods declared in its supertype. It can also declare new attributes and methods, which must have different names than those of the supertype.The keyword UNDER identifies the supertype, like this:

```
CREATE TYPE Student_t UNDER Person_t
( deptid NUMBER,
  major  VARCHAR2(30)) NOT FINAL;
```

The newly declared attributes deptid and major belong to the subtype Student_t. The subtype Employee_t is declared as, for example:

```
CREATE TYPE Employee_t UNDER Person_t
( empid NUMBER,
  mgr   VARCHAR2(30));
```

The resulting structs generated by OTT for this example are shown at:

> **See Also:** "OTT Support for Type Inheritance" *on page 14-17*

This subtype Student_t, can have its own subtype, such as PartTimeStudent_t:

```
CREATE TYPE PartTimeStuden_t UNDER Student_t
( numhours NUMBER) ;
```

Substitutability

The benefits of polymorphism derive partially from the property *substitutability*. Substitutability allows a value of some subtype to be used by code originally written for the supertype, without any specific knowledge of the subtype being needed in advance. The subtype value behaves to the surrounding code just like a value of the supertype would, even if it perhaps uses different mechanisms within its specializations of methods.

Instance substitutability refers to the ability to use an object value of a subtype in a context declared in terms of a supertype. REF substitutability refers to the ability to use a REF to a subtype in a context declared in terms of a REF to a supertype.

REF type attributes are substitutable, that is, an attribute defined as REF T can hold a REF to an instance of T or any of its subtypes.

Object type attributes are substitutable: an attribute defined to be of (an object) type T can hold an instance of T or any of its subtypes.

Collection element types are substitutable: if we define a collection of elements of type T, it can hold instances of type T and any of its subtypes. Here is an example of object attribute substitutability:

```
CREATE TYPE Book_t AS OBJECT
( title VARCHAR2(30),
  author Person_t      /* substitutable */);
```

Thus, a Book_t instance can be created by specifying a title string and a Person_t (or any subtype of Person_t) instance:

```
Book_t('My Oracle Experience',
        Employee_t(12345, 'Joe', 'SF', 1111, NULL))
```

NOT INSTANTIABLE Types and Methods

A type can be declared to be NOT INSTANTIABLE, which means that there is no constructor (default or user defined) for the type. Thus, it will not be possible to construct instances of this type. The typical usage would be define instantiable subtypes for such a type. Here is how this property is used:

```
CREATE TYPE Address_t AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_t UNDER Address_t(...);
CREATE TYPE IntlAddress_t UNDER Address_t(...);
```

A method of a type can be declared to be NOT INSTANTIABLE. Declaring a method as NOT INSTANTIABLE means that the type is not providing an implementation for that method. Further, a type that contains any NOT INSTANTIABLE methods must necessarily be declared as NOT INSTANTIABLE. For example:

```
CREATE TYPE T AS OBJECT
(
  x NUMBER,
  NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE;
```

A subtype of a NOT INSTANTIABLE type can override any of the NOT INSTANTIABLE methods of the supertype and provide concrete implementations. If there are any NOT INSTANTIABLE methods remaining, the subtype must also necessarily be declared as NOT INSTANTIABLE.

A NOT INSTANTIABLE subtype can be defined under an instantiable supertype. Declaring a NOT INSTANTIABLE type to be FINAL is not useful and is not allowed.

OCI Support for Type Inheritance

The following calls support type inheritance.

OCIDescribeAny()

This function has been enhanced to provide information specific to inherited types. Additional attributes have been added for the properties of inherited types. For example, you can get the supertype of a type.

> **See Also:** Table 6–7, "Attributes Belonging to Types" and Table 6–9, "Attributes Belonging to Type Methods" for attributes that OCIDescribeAny() can be used to read

Bind and Define Functions

OCI bind functions support REF, instance and collection element substitutability (subtype instances can be passed in where supertype is expected). There are no changes to the OCI bind interface, since all type checking and conversions are done at the server side.

OCI define functions also support substitutability (subtype instances can be fetched into define variables declared to hold the supertype). However, note that this might require the system to resize the memory to hold the subtype instance.

> **Note:** The client program must use objects that are allocated out of the object cache (and are thus re-sizeable) in such scenarios.

The client should not use a struct (allocated on the stack) as the define variable if the value is potentially polymorphic.

> **See Also:** Chapter 11, "Object-Relational Datatypes" for details of the bind and define processes

OCIObjectGetTypeRef()

This function will return the TDO of the most specific type of the input object. This operation will return an error if the user does not have privileges on the most specific type.

OCIObjectCopy()

The source can be a instance of any subtype of the target type. For example, a source object holding an `Employee_t` instance can be assigned to a target `Person_t` object. The object copy will copy all the attributes from the source to the target, including the subtype attributes. The copy changes the most specific type of the target object to `Employee_t`.

The TDO argument refers to the most specific type of the source object.

OCICollAssignElem()

The input element can be a instance of the subtype of the declared type. If the collection is of type `Person_t`, this function can be used to assign an `Employee_t` instance as an element of the collection.

OCICollAppend()

The input element can be a instance of the subtype of the declared type; if the collection is of type `Person_t`, this function can be used to append an `Employee_t` instance to the collection.

OCICollGetElem()

The collection element returned could be an instance of the subtype of the declared type.

OTT Support for Type Inheritance

The Object Type Translator (OTT) supports type inheritance of objects by declaring first the inherited attributes in an encapsulated struct called '_super', followed by the new declared attributes. This is done because C does not support type inheritance.

> **See Also:** "OTT Reference" *on page 14-26* for an example and discussion

Type Evolution

Adding, dropping and modifying type attributes are supported. This concept is known as *type evolution.* It is discussed in:

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features*

OCIDescribeAny() will return information about the latest version of the requested type if the type of the input object is OCI_OTYPE_NAME, and the type of the described object is OCI_PTYPE_TYPE, that is, the name input to OCIDescribeAny() is a type name.

To access type information use these functions, as well as OCIDescribeAny():

> **See Also:** OCITypeArrayByName() and OCITypeByName()

For a discussion of the impact of type evolution on the object cache:

> **See Also:** Type Evolution and the Object Cache *on page 13-22*

11

Object-Relational Datatypes

The concepts of binding and defining were introduced and discussed in Chapter 2, "OCI Programming Basics" and in Chapter 5, "Binding and Defining". This chapter provides additional information necessary for users who are developing object applications. The following topics are covered in this chapter:

- Overview of OCI Functions for Objects

- Mapping Oracle Datatypes to C

- Manipulating C Datatypes With OCI

- Date (OCIDate)

- Datetime and Interval (OCIDateTime, OCIInterval)

- Number (OCINumber)

- Fixed or Variable-Length String (OCIString)

- Raw (OCIRaw)

- Collections (OCITable, OCIArray, OCIColl, OCIIter)

- Multilevel Collection Types

- REF (OCIRef)

- Object Type Information Storage and Access

- Any Type, AnyData and AnyDataSet Interfaces

- Binding Named Datatypes

- Defining Named Datatypes

- Binding And Defining Oracle C Datatypes

- SQLT_NTY Bind/Define Example

Overview of OCI Functions for Objects

The OCI datatype mapping and manipulation functions provide the ability to manipulate instances of predefined Oracle C datatypes. These datatypes are used to represent the attributes of user-defined datatypes, including object types in Oracle.

Each group of functions within the OCI is distinguished by a particular naming convention. The datatype mapping and manipulation functions, for example, can be easily recognized because the function names start with the prefix *OCI*, followed by the name of a datatype, as in `OCIDateFromText()` and `OCIRawSize()`. As will be explained later, the names can be further broken down into function groups that operate on a particular type of data.

Additionally, the predefined Oracle C types on which these functions operate are also distinguished by names which begin with the prefix *OCI*, as in **OCIDate** or **OCIString**.

The datatype mapping and manipulation functions are used when an application needs to manipulate, bind, or define attributes of objects that are stored in an Oracle database, or which have been retrieved by a SQL query. Retrieved objects are stored in the client-side object cache, as was described in Chapter 13, "Object Cache and Object Navigation".

This chapter describes the purpose and structure of each of the datatypes that can be manipulated by the OCI datatype mapping and manipulation functions. It also summarizes the different function groups, and gives lists of available functions and their purposes.

This chapter also provides information about how to use these datatypes in bind and define operations within an OCI application.

These functions are valid only when an OCI application is running in object mode. For information about initializing the OCI in object mode, and creating an OCI application that accesses and manipulates objects, refer to the section "Initializing Environment and Object Cache" on page 10-10.

> **See Also:** For detailed information about object types, attributes, and collection datatypes, refer to *Oracle9i Database Concepts*.

Mapping Oracle Datatypes to C

Oracle provides a rich set of predefined datatypes with which you can create tables and specify user-defined datatypes (including object types). Object types extend the functionality of Oracle by allowing you to create datatypes that precisely model the

types of data with which they work. This can provide increased efficiency and ease-of-use for programmers who are accessing the data.

Database tables and object types are based upon the datatypes supplied by Oracle. These tables and types are created with SQL statements and stored using a specific set of Oracle internal datatypes, like VARCHAR2 or NUMBER. For example, the following SQL statements create a user-defined address datatype and an object table to store instances of that type:

```
CREATE TYPE address AS OBJECT
(street1     varchar2(50),
street2     varchar2(50),
city        varchar2(30),
state       char(2),
zip         number(5));
CREATE TABLE address_table OF address;
```

The new address type could also be used to create a regular table with an object column:

```
CREATE TABLE employees
(name         varchar2(30),
birthday      date,
home_addr     address);
```

An OCI application can manipulate information in the name and birthday columns of the employees table using straightforward bind and define operations in association with SQL statements. Accessing information stored as attributes of objects requires some extra steps.

The OCI application first needs a way to represent the objects in a C-language format. This is accomplished by using the Object Type Translator (OTT) to generate C struct representations of user-defined types. The elements of these structs have datatypes that represent C language mappings of Oracle datatypes.

> **See Also:** Table 14–1, "Object Datatype Mappings for Object Type Attributes" for the available Oracle types you can use as object attribute types and their C mappings

An additional C type, **OCIInd**, is used to represent null indicator information corresponding to attributes of object types.

> **See Also:** For more information and examples regarding the use of the OTT, refer to Chapter 14, "The Object Type Translator (OTT)".

OCI Type Mapping Methodology

Oracle followed a distinct design philosophy when specifying the mappings of Oracle predefined types. The current system has the following benefits and advantages:

- The actual representation of datatypes like **OCINumber** is opaque to client applications, and the datatypes are manipulated with a set of predefined functions. This allows for the internal representation to change to accommodate future enhancements without breaking user code.

- The implementation is consistent with object-oriented paradigms in which class implementation is hidden and only the required operations are exposed.

- This implementation can have advantages for programmers. Consider a C program that wants to manipulate Oracle number variables without losing the accuracy provided by Oracle numbers. To do this in Oracle release 7, you would have had to issue a "SELECT ... FROM DUAL" statement. In later releases, this is accomplished by invoking the OCINumber*() functions.

Manipulating C Datatypes With OCI

In an OCI application, the manipulation of data may be as simple as adding together two integer variables and storing the result in a third variable:

```
int     int_1, int_2, sum;
...
/* some initialization occurs */
...
sum = int_1 + int_2;
```

The C language provides a set of predefined operations on simple types like **integer**. However, the C datatypes listed in Table 14–1, "Object Datatype Mappings for Object Type Attributes" are not simple C primitives. Types like **OCIString** and **OCINumber** are actually structs with a specific Oracle-defined internal structure. It is not possible to simply add together two **OCINumbers** and store the value in the third.

The following is not valid:

```
OCINumber     num_1, num_2, sum;
...
/* some initialization occurs */
...
sum = num_1 + num_2;          /* NOT A VALID OPERATION */
```

The OCI datatype mapping and manipulation functions are provided to enable you to perform operations on these new datatypes. For example, the above addition of **OCINumbers** could be accomplished as follows, using the `OCINumberAdd()` function:

```
OCINumber    num_1, num_2, sum;
...
/* some initialization occurs */
...
OCINumberAdd(errhp, &num_1, &num_2, &sum): /* errhp is error handle */
```

The OCI provides functions to operate on each of the new datatypes. The names of the functions provide information about the datatype on which they operate. The first three letters, *OCI*, indicate that the function is part of the OCI. The next part of the name indicates the datatype on which the function operates. The following table shows the various function prefixes, along with example function names and the datatype on which those functions operate:

Function Prefix	Example	Operates On
OCIColl	`OCICollGetElem()`	OCIColl, OCIIter, OCITable, OCIArray
OCIDate	`OCIDateDaysBetween()`	OCIDate
OCIDateTime	`OCIDateTimeSubtract()`	OCIDate, OCIDateTime
OCIInter	`OCIInterToText()`	OCIInterval
OCIIter	`OCIIterInit()`	OCIIter
OCINumber	`OCINumberAdd()`	OCINumber
OCIRaw	`OCIRawResize()`	OCIRaw *
OCIRef	`OCIRefAssign()`	OCIRef *
OCIString	`OCIStringSize()`	OCIString *
OCITable	`OCITableLast()`	OCITable *

The structure of each of the datatypes is described later in this chapter, along with a list of the functions that manipulate that type.

Precision of Oracle Number Operations

Oracle numbers have a precision of 38 decimal digits. All Oracle number operations are accurate to the full precision, with the following exceptions:

- Inverse trigonometric functions are accurate to 28 decimal digits.

- Other transcendental functions, including trigonometric functions, are accurate to approximately 37 decimal digits.

- Conversions to and from native floating-point types have the precision of the relevant floating-point type, not to exceed 38 decimal digits.

Date (OCIDate)

The Oracle date format is mapped in C by the **OCIDate** type, which is an opaque C struct. Elements of the struct represent the year, month, day, hour, minute, and second of the date. The specific elements can be set and retrieved using the appropriate OCI functions.

The **OCIDate** datatype can be bound or defined directly using the external typecode SQLT_ODT in the bind or define call.

The OCI date manipulation functions are listed in the following tables, which are organized according to functionality. Unless otherwise specified, the term *date* in these tables refers to a value of type **OCIDate**.

> **See Also:** The prototypes and descriptions for all the functions are provided in Chapter 18, "OCI Datatype Mapping and Manipulation Functions".

Date Conversion Functions

The following functions perform date conversion.

Function	Purpose
OCIDateToText()	convert date to string
OCIDateFromText()	convert text string to date
OCIDateZoneToZone()	convert date from one time zone to another

Date Assignment and Retrieval Functions

The following functions retrieve and assign date elements.

Function	Purpose
OCIDateAssign()	OCIDate assignment
OCIDateGetDate()	get the date portion of an OCIDate
OCIDateSetDate()	set the date portion of an OCIDate
OCIDateGetTime()	get the time portion of an OCIDate
OCIDateSetTime()	set the time portion of an OCIDate

Date Arithmetic and Comparison Functions

The following functions perform date arithmetic and comparison.

Function	Purpose
OCIDateAddDays()	add days
OCIDateAddMonths()	add months
OCIDateCompare()	compare dates
OCIDateDaysBetween()	calculate the number of days between two dates

Date Information Accessor Functions

The following functions access date information.

Function	Purpose
OCIDateLastDay()	the last day of the month
OCIDateNextDay()	the first named day after a given date
OCIDateSysDate()	the system date

Date Validity Checking Functions

The following function checks date validity.

Function	Purpose
OCIDateCheck()	check whether a given date is valid

Date Example

The following code provides examples of how to manipulate an attribute of type **OCIDate** using OCI calls.

```
#define FMT "DAY, MONTH DD, YYYY"
#define LANG "American"
struct person
{
OCIDate start_date;
};
typedef struct person person;

OCIError *err;
person *tim;
sword status;                        /* error status */
uword invalid;
OCIDate last_day, next_day;
text buf[100], last_day_buf[100], next_day_buf[100];
ub4 buflen = sizeof(buf);

/* For this example, assume the OCIEnv and OCIError have been
* initialized as described in Chapter 2. */
/* Pin tim person object in the object cache. See Chapter 13 for
* information about pinning. For this example, assume that
* tim is pointing to the pinned object. */
/* set the start date of tim */
OCIDateSetTime(&tim->start_date,8,0,0);
OCIDateSetDate(&tim->start_date,1990,10,5)

/* check if the date is valid */
if (OCIDateCheck(err, &tim->start_date, &invalid) != OCI_SUCCESS)
/* error handling code */

if (invalid)
/* error handling code */

/* get the last day of start_date's month */
if (OCIDateLastDay(err, &tim->start_date, &last_day) != OCI_SUCCESS)
/* error handling code */
```

```
/* get date of next named day */
if (OCIDateNextDay(err, &tim->start_date, "Wednesday",    strlen("Wednesday"),
&next_day) != OCI_SUCCESS)
/* error handling code */
/* convert dates to strings and print the information out */
/* first convert the date itself*/
buflen = sizeof(buf);
if (OCIDateToText(err, &tim->start_date, FMT, sizeof(FMT)-1, LANG,
    sizeof(LANG)-1, &buflen, buf) != OCI_SUCCESS)
/* error handling code */

/* now the last day of the month */
buflen = sizeof(last_day_buf);
if (OCIDateToText(err, &last_day, FMT, sizeof(FMT)-1, LANG,    sizeof(LANG)-1,
&buflen, last_day_buf) != OCI_SUCCESS)
/* error handling code */

/* now the first Wednesday after this date */
buflen = sizeof(next_day_out);
if (OCIDateToText(err, &next_day, FMT, sizeof(FMT)-1, LANG,
    sizeof(LANG)-1, &buflen, next_day_buf) != OCI_SUCCESS)
/* error handling code */

/* print out the info */
printf("For: %s\n", buf);
printf("The last day of the month is: %s\n", last_day_buf);
printf("The next Wednesday is: %s\n", next_day_buf);
```

The output will be:

```
For: Monday, May 13, 1996
The last day of the month is: Friday, May 31
The next Wednesday is: Wednesday, May 15
```

Datetime and Interval (OCIDateTime, OCIInterval)

The **OCIDateTime** datatype is an opaque structure used to represent Oracle time
and timestamp datatypes (TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP
WITH TIME ZONE) and the ANSI DATE datatype. You can set or retrieve the data in
these types (that is, year, day, fractional second) using the appropriate OCI
functions.

The **OCIInterval** datatype is also an opaque structure and is used to represent Oracle interval datatypes (INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND).

You can bind and define **OCIDateTime** and **OCIInterval** data using the following external typecodes in the bind or define call:

OCI Datatype	Type of Data	External Typecode for Binding/Defining
OCIDateTime	ANSI DATE	SQLT_DATE
OCIDateTime	TIMESTAMP	SQLT_TIMESTAMP
OCIDateTime	TIMESTAMP WITH TIME ZONE	SQLT_TIMESTAMP_TZ
OCIDateTime	TIMESTAMP WITH LOCAL TIME ZONE	SQLT_TIMESTAMP_LTZ
OCIInterval	INTERVAL YEAR TO MONTH	SQLT_INTERVAL_YM
OCIInterval	INTERVAL DAY TO SECOND	SQLT_INTERVAL_DS

The OCI functions which operate on datetime and interval data are listed in the following tables. More detailed information about these functions can be found in OCI Date, Datetime, and Interval Functions on page 18-28.

In general, functions which operate on **OCIDateTime** data are also valid for **OCIDate** data.

Datetime Functions

The following functions operate on **OCIDateTime** values. Some of these functions also perform arithmetic operations on datetime and interval values. Some functions may only work for certain datetime types. The possible types are:

- SQLT_DATE - DATE
- SQLT_TIMESTAMP - TIMESTAMP
- SQLT_TIMESTAMP_TZ - TIMESTAMP WITH TIME ZONE
- SQLT_TIMESTAMP_LTZ -TIMESTAMP WITH LOCAL TIME ZONE

See the individual function descriptions for more information about input types which are valid for a particular function.

Function	Purpose
OCIDateTimeAssign() on page 18-49	Performs datetime assignment
OCIDateTimeCheck() on page 18-50	Checks if the given date is valid
OCIDateTimeCompare() on page 18-52	Compares two datetime values
OCIDateTimeConstruct() on page 18-53	Constructs a datetime descriptor
OCIDateTimeConvert() on page 18-55	Converts one datetime type to another
OCIDateTimeFromArray() on page 18-56	Converts an array of size OCI_DT_ARRAYLEN to an **OCIDateTime** descriptor
OCIDateTimeFromText() on page 18-58	Converts the given string to Oracle datetime type in the **OCIDateTime** descriptor, according to the specified format
OCIDateTimeGetDate() on page 18-60	Gets the date (year, month, day) portion of a datetime value
OCIDateTimeGetTime() on page 18-61	Gets the time (hour, minute, second, fractional second) out of a datetime value
OCIDateTimeGetTimeZoneName() on page 18-63	Gets the time zone name portion of a datetime value
OCIDateTimeGetTimeZoneOffset() on page 18-64	Gets the time zone (hour, minute) portion of a datetime value
OCIDateTimeIntervalAdd() on page 18-65	Adds an interval to a datetime to produce a resulting datetime
OCIDateTimeIntervalSub() on page 18-66	Subtracts an interval from a datetime and stores the result in a datetime
OCIDateTimeSubtract() on page 18-67	Takes two datetimes as input and stores their difference in an interval
OCIDateTimeSysTimeStamp() on page 18-68	Gets the system current date and time as a timestamp with time zone
OCIDateTimeToArray() on page 18-69	Converts a **OCIDateTime** descriptor to an array
OCIDateTimeToText() on page 18-71	Converts the given date to a string according to the specified format

Function	Purpose
OCIDateZoneToZone() on page 18-73	Converts date from one time zone to another zone

Datetime Example

The following snippet of code shows how to use an **OCIDateTime** datatype to select data from a TIMESTAMP WITH LOCAL TIME ZONE column:

```
...

/* allocate the program variable for storing the data */
OCIDateTime *tstmpltz = (OCIDateTime *)NULL;

/* Col1 is a timestamp with local time zone column */
OraText *sqlstmt = (OraText *)"SELECT col1 FROM foo";

/* Allocate the descriptor (storage) for the datatype */
status = OCIDescriptorAlloc(envhp,(dvoid **)&tstmpltz, OCI_DTYPE_TIMESTAMP_LTZ,
        0, (dvoid **)0);
....

status = OCIStmtPrepare (stmthp, errhp, sqlstmt, (ub4)strlen ((char *)sqlstmt),
        (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);

/* specify the define buffer for col1 */
status = OCIDefineByPos(stmthp, &defnp, errhp, 1, &tstmpltz, sizeof(tstmpltz),
        SQLT_TIMESTAMP_LTZ, 0, 0, 0, OCI_DEFAULT));

/* Execute and Fetch */
OCIStmtExecute(svchp, stmthp, errhp, 1, 0,(OCISnapshot *) NULL,
        (OCISnapshot *)NULL, OCI_DEFAULT)

At this point tstmpltz contains a valid timestamp with local time zone data. You
can get the time zone name of the datetime data using:

status = OCIDateTimeGetTimeZoneName(envhp, errhp, tstmpltz, (ub1 *)buf,
        (ub4 *)buflen);
...
```

Interval Functions

The following functions operate exclusively on interval data. In some cases it is necessary to specify the type of interval involved. Possible types include:

- SQLT_INTERVAL_YM - interval year to month

- SQLT_INTERVAL_DS - interval day to second

See the individual function descriptions for more detailed information.

> **See Also:** Complete lists of the names and purposes as well as more detailed information about these functions can be found in OCI Date, Datetime, and Interval Functions on page 18-28.

Function	Purpose
`OCIIntervalAdd()` on page 18-75	Adds two intervals to produce a resulting interval
`OCIIntervalAssign()` on page 18-76	Copies one interval to another
`OCIIntervalCheck()` on page 18-77	Checks the validity of an interval
`OCIIntervalCompare()` on page 18-79	Compares two intervals
`OCIIntervalDivide()` on page 18-81	Divides an interval by an Oracle Number to produce an interval
`OCIIntervalFromNumber()` on page 18-82	Converts an Oracle Number to an interval
`OCIIntervalFromText()` on page 18-83	Given an interval string, produces the interval represented by the string
`OCIIntervalGetDaySecond()` on page 18-86	Gets values of day and second from an interval
`OCIIntervalGetYearMonth()` on page 18-88	Gets year and month from an interval
`OCIIntervalMultiply()` on page 18-89	Multiplies an interval by an Oracle Number to produce an interval
`OCIIntervalSetDaySecond()` on page 18-90	Sets day and second in an interval

Function	Purpose
OCIIntervalSetYearMonth() on page 18-92	Sets year and month in an interval
OCIIntervalSubtract() on page 18-93	Subtracts two intervals and stores the result in an interval
OCIIntervalToNumber() on page 18-94	Converts an interval to an Oracle Number
OCIIntervalToText() on page 18-95	Given an interval, produces a string representing the interval

Number (OCINumber)

The **OCINumber** datatype is an opaque structure used to represent Oracle numeric datatypes (NUMBER, FLOAT, DECIMAL, and so forth). You can bind or define this type using the external typecode SQLT_VNU in the bind or define call.

The **OCINumber** manipulation functions are listed in the following tables, which are organized according to functionality. Unless otherwise specified, the term *number* in these tables refers to a value of type **OCINumber**.

> **See Also:** The prototypes and descriptions for all the functions are provided in Chapter 18, "OCI Datatype Mapping and Manipulation Functions".

Number Arithmetic Functions

The following functions perform arithmetic operations.

Function	Purpose
OCINumberAbs()	get the absolute value of a number
OCINumberAdd()	add two numbers together
OCINumberCeil()	get the ceiling value of a number
OCINumberDec()	decrement a number
OCINumberDiv()	divide one number by another
OCINumberFloor()	get the floor value of a number
OCINumberInc()	increment a number

Function	Purpose
OCINumberMod()	get the modulus from the division of two numbers
OCINumberMul()	multiply two numbers together
OCINumberNeg()	negate a number
OCINumberRound()	round a number to a specified decimal place
OCINumberShift()	shifts a number a certain number of decimal places
OCINumberSign()	get the sign of a number
OCINumberSqrt()	get the square root of a number
OCINumberSub()	subtract one number from another
OCINumberTrunc()	truncate a number to a specified decimal place
OCINumberSIgn()	returns the sign of a given number

Number Conversion Functions

The following functions perform conversions between numbers and reals, integers, and strings.

Function	Purpose
OCINumberToInt()	convert number to integer
OCINumberFromInt()	convert integer to number
OCINumberToReal()	convert number to real
OCINumberFromReal()	convert real to number
OCINumberToText()	convert number to string
OCINumberFromText()	convert string to number

Exponential and Logarithmic Functions

The following functions perform exponential and logarithmic operations.

Function	Purpose
OCINumberPower()	take a number base to a given number exponent
OCINumberExp()	take the exponent with base e

Function	Purpose
OCINumberLog()	take the logarithm of a given base
OCINumberLn()	take the natural logarithm (base *e*)
OCINumberIntPower()	take a number base to a given integer power

Trigonometric Functions

The following functions perform trigonometric operations on numbers.

Function	Purpose
OCINumberArcCos()	calculate arc cosine
OCINumberArcSin()	calculate arc sine
OCINumberArcTan() / OCINumberArcTan2()	calculate arc tangent / of two numbers
OCINumberCos()	calculate cosine
OCINumberHypCos()	calculate cosine hyperbolic
OCINumberSin()	calculate sine
OCINumberHypSin()	calculate sine hyperbolic
OCINumberTan()	calculate tangent
OCINumberHypTan()	calculate tangent hyperbolic

Number Assignment, Comparison, and Evaluation Functions

The following functions perform assign and compare operations on numbers.

Function	Purpose
OCINumberAssign()	assign one number to another
OCINumberCmp()	compare two numbers
OCINumberIsInt()	test if an integer
OCINumberIsZero()	test if equal to zero
OCINumberPrec()	sets the precision
OCINumberSetPi()	set a number to pi

Function	Purpose
OCINumberSetZero()	initialize number to zero

Number Example

The following example shows how to manipulate an attribute of type **OCINumber**.

```
struct person
{
OCINumber sal;
};
typedef struct person person;
OCIError *err;
person* steve;
person* scott;
person* jason;
OCINumber  *stevesal;
OCINumber  *scottsal;
OCINumber *debsal;
sword   status;
int     inum;
double  dnum;
OCINumber ornum;
char    buffer[21];
ub4     buflen;
sword   result;

/* For this example, assume OCIEnv and OCIError are initialized. */
/* For this example, assume that steve, scott and jason are pointing to
  person objects which have been pinned in the object cache. */
stevesal = &steve->sal;
scottsal = &scott->sal;
debsal = &jason->sal;

/* initialize steve's salary to be $12,000 */
OCINumberInit(err, stevesal);
inum = 12000;
status = OCINumberFromInt(err, &inum, sizeof(inum), OCI_NUMBER_SIGNED,
    stevesal);
if (status != OCI_SUCCESS)  /* handle error from OCINumberFromInt */;

/* initialize scott's salary to be same as steve */
OCINumberAssign(err, stevesal, scottsal);
```

```
/* initialize jason's salary to be 20% more than steve's */
dnum = 1.2;
status = OCINumberFromReal(err, &dnum, DBL_DIG, &ornum);
if (status != OCI_SUCCESS)  /* handle error from OCINumberFromReal */;
status = OCINumberMul(err, stevesal, &ornum, debsal);
if (status != OCI_SUCCESS)  /* handle error from OCINumberMul */;

/* give scott a 50% raise */
dnum = 1.5;
status = OCINumberFromReal(err, &dnum, DBL_DIG, &ornum);
if (status != OCI_SUCCESS)  /* handle error from OCINumberFromReal */;
status = OCINumberMul(err, scottsal, &ornum, scottsal);
if (status != OCI_SUCCESS)  /* handle error from OCINumberMul */;

/* double steve's salary */
status = OCINumberAdd(err, stevesal, stevesal, stevesal);
if (status != OCI_SUCCESS)  /* handle error from OCINumberAdd */;

/* get steve's salary in integer */
status = OCINumberToInt(err, stevesal, sizeof(inum), OCI_NUMBER_SIGNED,
    &inum);
if (status != OCI_SUCCESS)  /* handle error from OCINumberToInt */;

/* inum is set to 24000 */
/* get jason's salary in double */
status = OCINumberToReal(err, debsal, sizeof(dnum), &dnum);
if (status != OCI_SUCCESS)  /* handle error from OCINumberToReal */;

/* dnum is set to 14400 */
/* print scott's salary as DEM0001`8000.00 */
buflen = sizeof(buffer);
status = OCINumberToText(err, scottsal, "C0999G9999D99", 13,
    "NLS_NUMERIC_CHARACTERS='.`' NLS_ISO_CURRENCY='Germany'",
    54, &buflen, buffer);
if (status != OCI_SUCCESS)  /* handle error from OCINumberToText */;
printf("scott's salary = %s\n", buffer);

/* compare steve and scott's salaries */
status = OCINumberCmp(err, stevesal, scottsal, &result);
if (status != OCI_SUCCESS)  /* handle error from OCINumberCmp */;

/* result is positive */
/* read jason's new salary from string */
status = OCINumberFromText(err, "48`000.00", 9, "99G999D99", 9,
    "NLS_NUMERIC_CHARACTERS='.`'", 27, debsal);
```

```
if (status != OCI_SUCCESS)  /* handle error from OCINumberFromText */;
/* jason's salary is now 48000.00 */
```

Fixed or Variable-Length String (OCIString)

Fixed or variable-length string data is represented to C programs as an **OCIString** *.

The length of the string does not include the null character.

For binding and defining variables of type **OCIString** * use the external typecode
SQLT_VST.

> **See Also:** The prototypes and descriptions for all the functions
> are provided in Chapter 18, "OCI Datatype Mapping and
> Manipulation Functions".

String Functions

The following functions allow the C programmer to manipulate an instance of a
string.

Function	Purpose
OCIStringAllocSize()	get allocated size of string memory in bytes
OCIStringAssign()	assign one string to another
OCIStringAssignText()	assign text string to string
OCIStringPtr()	get pointer to string part of string
OCIStringResize()	resize string memory
OCIStringSize()	get string size

String Example

This example assigns a text string to a string, then gets a pointer to the string part of
the string, as well as the string size, and prints it out.

Note the double indirection used in passing the *vstring1* parameter in
OCIStringAssignText().

```
OCIEnv       *envhp;
OCIError     *errhp;
OCIString    *vstring1 = (OCIString *)0;
OCIString    *vstring2 = (OCIString *)0;
```

```
text            c_string[20];
text           *text_ptr;
sword          status;

strcpy(c_string, "hello world");
/* Assign a text string to an OCIString */
status = OCIStringAssignText(envhp, errhp, c_string,
     (ub4)strlen(c_string),&vstring1);
/* Memory for vstring1 is allocated as part of string assignment */

status = OCIStringAssignText(envhp, errhp, "hello again",
     (ub4)strlen("This is a longer string."),&vstring1);
/* vstring1 is automatically resized to store the longer string */

/* Get a pointer to the string part of vstring1 */
text_ptr = OCIStringPtr(envhp, vstring1);
/* text_ptr now points to "hello world" */
printf("%s\n", text_ptr);
```

Raw (OCIRaw)

Variable-length raw data is represented in C using the **OCIRaw** * datatype.

For binding and defining variables of type **OCIRaw** *, use the external typecode SQLT_LVB.

> **See Also:**   The prototypes and descriptions for all the functions
> are provided in Chapter 18, "OCI Datatype Mapping and
> Manipulation Functions"

Raw Functions

The following functions perform OCIRaw operations.

Function	Purpose
OCIRawAllocSize()	get the allocated size of raw memory in bytes
OCIRawAssignBytes()	assign raw data (ub1 *) to OCIRaw *
OCIRawAssignRaw()	assign one OCIRaw * to another
OCIRawPtr()	get pointer to raw data
OCIRawResize()	resize memory of variable-length raw data

Function	Purpose
OCIRawSize()	get size of raw data

Raw Example

In this example, a raw data block is set up and a pointer to its data is obtained.

Note the double indirection in the call to OCIRawAssignBytes().

```
OCIEnv       *envhp;
OCIError     *errhp;
sword        status;
ub1          data_block[10000];
ub4          data_block_len = 10000;
OCIRaw       *raw1;
ub1 *raw1_pointer;

/* Set up the RAW */
/* assume 'data_block' has been initialized */
status = OCIRawAssignBytes(envhp, errhp, data_block, data_block_len, &raw);

/* Get a pointer to the data part of the RAW */
raw1_pointer = OCIRawPtr(envhp, raw1);
```

Collections (OCITable, OCIArray, OCIColl, OCIIter)

Oracle provides two types of collections: variable-length arrays (VARRAYs) and nested tables. In C applications, varrays are represented as **OCIArray** *, and nested tables are represented as **OCITable** *. Both of these datatypes (along with OCIColl and **OCIIter**, described later) are opaque structures.

A variety of generic collection functions enable you to manipulate collection data. You can use these functions on both varrays and nested tables. In addition, there is a set of functions specific to nested tables.

> **See Also:** "Nested Table Manipulation Functions" on page 11-24

You can allocate an instance of a varray or nested table using OCIObjectNew() and free it using OCIObjectFree().

> **See Also:** The prototypes and descriptions for all the functions are provided in "OCI Collection and Iterator Functions" on page 18-5

Generic Collection Functions

Oracle provides two types of collections: variable-length arrays (varrays) and nested tables. Both varrays and nested tables can be viewed as sub-types of a generic collection type.

In C, a generic collection is represented as **OCIColl** *, a varray is represented as **OCIArray** *, and a nested table as **OCITable** *. Oracle provides a set of functions to operate on generic collections (such as **OCIColl** *). These functions start with the prefix OCIColl, as in OCICollGetElem(). The OCIColl*() functions can also be called to operate on varrays and nested tables.

The generic collection functions are grouped into two main categories:

- manipulating varray or nested table data
- scanning through a collection with a collection iterator

The generic collection functions represent a complete set of functions for manipulating varrays. Additional functions are provided to operate specifically on nested tables. They are identified by the prefix OCITable, as in OCITableExists().

See Also:

Note: Indexes passed to collection functions are zero-based

Collection Data Manipulation Functions

The following generic functions manipulate collection data:

Function	Purpose
OCICollAppend()	append an element
OCICollAssignElem()	assign element at given index
OCICollAssign()	assign one collection to another
OCICollGetElem()	get pointer to an element given its index
OCICollMax()	get upper bound of collection
OCICollSize()	get current size of collection
OCICollTrim()	trim n elements from the end of the collection

Collection Scanning Functions

The following generic functions enable you to scan collections with a collection iterator. The iterator is of type **OCIIter**, and is created by first calling `OCIIterCreate()`.

Function	Purpose
OCIIterCreate()	create an iterator for scanning collection
OCIIterDelete()	delete iterator
OCIIterGetCurrent()	get pointer to current element pointed by iterator
OCIIterInit()	initialize iterator to scan the given collection
OCIIterNext()	get pointer to next element
OCIIterPrev()	get pointer to previous element

Varray/Collection Iterator Example

This example creates and uses a collection iterator to scan through a varray.

```
OCIEnv      *envhp;
OCIError    *errhp;
text        *text_ptr;
sword       status;
OCIArray    *clients;
OCIString   *client_elem;
OCIIter     *iterator;
boolean     eoc;
dvoid       *elem;
OCIInd      *elemind;

/* Assume envhp, errhp have been initialized */
/* Assume clients points to a varray */

/* Print the elements of clients */
/* To do this, create an iterator to scan the varray */
status = OCIIterCreate(envhp, errhp, clients, &iterator);

/* Get the first element of the clients varray */
printf("Clients' list:\n");
status = OCIIterNext(envhp, errhp, iterator, &elem,
                     (dvoid **) &elemind, &eoc);
```

```
                  while (!eoc && (status == OCI_SUCCESS))
                  {
                    client_elem = *(OCIString)**elem;
                                        /* client_elem points to the string */

                   /*
                      the element pointer type returned by OCIIterNext() through 'elem' is
                      the same as that of OCICollGetElem(). Refer to OCICollGetElem() for
                      details.  */

                    /*
                      client_elem points to an OCIString descriptor, so to print it out,
                      get a pointer to where the text begins
                    */
                    text_ptr = OCIStringPtr(envhp, client_elem);

                    /*
                      text_ptr now points to the text part of the client OCIString, which is a
                  NULL-terminated string
                    */
                    printf("  %s\n", text_ptr);
                    status = OCIIterNext(envhp, errhp, iterator, &elem,
                                      (dvoid **)&elemind, &eoc);
                  }

                  if (status != OCI_SUCCESS)
                  {
                    /* handle error */
                  }

                  /* destroy the iterator */
                  status = OCIIterDelete(envhp, errhp, &iterator);
```

Nested Table Manipulation Functions

As its name implies, one table may be *nested,* or contained within another, as a variable, attribute, parameter or column. Nested tables may have elements deleted, by means of the OCITableDelete() function.

For example, suppose a table is created with 10 elements, and OCITableDelete() is used to delete elements at index 0 through 4 and 9. The first existing element is now element 5, and the last existing element is element 8.

As noted above, the generic collection functions may be used to map to and manipulate nested tables. In addition, the following functions are specific to nested tables. They should not be used on varrays.

Function	Purpose
OCITableDelete()	delete an element at a given index
OCITableExists()	test whether an element exists at a given index
OCITableFirst()	return index for first existing element of table
OCITableLast()	return index for last existing element of table
OCITableNext()	return index for next existing element of table
OCITablePrev()	return index for previous existing element of table
OCITableSize()	return table size, not including deleted elements

Nested Table Element Ordering

When a nested table is fetched into the object cache, its elements are given a transient ordering, numbered from zero to the number of elements, minus 1. For example, a table with 40 elements would be numbered from 0 to 39.

You can use these position ordinals to fetch and assign the values of elements (for example, fetch to element *i*, or assign to element *j*, where *i* and *j* are valid position ordinals for the given table).

When the table is copied back to the database, its transient ordering is lost. Delete operations may be performed against elements of the table. Delete operations create transient *holes*; that is, they do not change the position ordinals of the remaining table elements.

Nested Table Locators

You can retrieve a locator to a nested table. A locator is like a handle to a collection value, and it contains information about the database snapshot which exists at the time of retrieval. This snapshot information helps the database retrieve the correct instantiation of a collection value at a later time when collection elements are fetched using the locator.

Unlike a LOB locator, a collection locator cannot be used to modify a collection instance, they merely locate the correct data. Using the locator enables an application to return a handle to a nested table without having to retrieve the entire collection, which may be quite large.

A user specifies when a table is created if a locator should be returned when a collection column or attribute is fetched, using the RETURN AS LOCATOR specification.

> **See Also:** *Oracle9i SQL Reference* for more information

You can use the OCICollIsLocator() function to determine whether a collection is a locator.

Multilevel Collection Types

The collection element itself can be directly or indirectly another collection type. Multilevel collection type is the name given to such a top-level collection type.

Multilevel collections have the following characteristics:

- They can be collections of other collection types.
- They can be collections of objects with collection attributes.
- They have no limit to the number of nesting levels.
- They can contain any combination of varrays and nested tables.
- They can be used as columns in tables.

OCI routines work with multilevel collections. The following routines can return in parameter *elem* a **OCIColl***, which can be used in any of the collection routines:

- OCICollgetElem()
- OCIIterGetCurrent()
- OCIIterNext()
- OCIIterPrev()

The following functions take a collection element and add it to an existing collection. Parameter *elem* could be an **OCIColl*** if the element type is another collection:

- OCICollAssignElem()
- OCICollAppend()

Multilevel Collection Type Example

Assume that the following types and tables are used for the example:

```
type_1 (a NUMBER, b NUMBER)
NT1 TABLE OF type_1
NT2 TABLE OF NT1
```

The following snippet of code iterates over the multilevel collection:

```
...
OCIColl *outer_coll;
OCIColl *inner_coll;
OCIIter *itr1, *itr2;
Type_1 *type_1_instance;
....
/* assume outer_coll points to a valid coll of type NT2 */
checkerr(OCIIterCreate(envhp, errhp, outer_coll, &itr1));
for(eoc = FALSE;!OCIIterNext(envhp, errhp, itr, (dvoid **) &elem,
                               (dvoid **) &elem_null, &eoc) && !eoc;)
{
   inner_coll = (OCIColl *)elem;
   /* iterate over inner collection.. */
   checkerr(errhp, OCIIterCreate(envhp, errhp, inner_coll, &itr2));
   for(eoc2 = FALSE;!OCIIterNext(envhp, errhp, itr2, (dvoid **)&elem2,
            (dvoid **) &elem2_null, &eoc2) && !eoc2;)
      {
       type_1_instance = (Type_1 *)elem2;
       /* use the fields of type_1_instance */
      }
   /* close iterator over inner collection */
   checkerr(errhp, OCIIterDelete(envhp, errhp, &itr2));
}
/* close iterator over outer collection */
checkerr(errhp, OCIIterDelete(envhp, errhp, &itr));
...
```

REF (OCIRef)

A REF (reference) is an identifier to an object. It is an opaque structure that uniquely locates the object. An object may point to another object by way of a REF.

In C applications, the REF is represented by **OCIRef ***.

> **See Also:**   The prototypes and descriptions for all the functions
> are provided in Chapter 18, "OCI Datatype Mapping and
> Manipulation Functions".

REF Manipulation Functions

The following functions perform REF operations.

Function	Purpose
OCIRefAssign()	assign one REF to another
OCIRefClear()	clear or nullify a REF
OCIRefFromHex()	convert hexadecimal string to a REF
OCIRefHexSize()	return size of hex string representation of REF
OCIRefIsEqual()	compare two REFs for equality
OCIRefIsNull()	test whether a REF is NULL
OCIRefToHex()	convert REF to a hexadecimal string

REF Example

This example tests two REFs for NULL, compares them for equality, and assigns one REF to another. Note the double indirection in the call to OCIRefAssign().

```
OCIEnv        *envhp;
OCIError      *errhp;
sword         status;
boolean       refs_equal;
OCIRef        *ref1, *ref2;

/* assume refs have been initialized to point to valid objects */
/*Compare two REFs for equality */
refs_equal = OCIRefIsEqual(envhp, ref1, ref2);
printf("After first OCIRefIsEqual:\n");
if(refs_equal)
   printf("REFs equal\n");
else
   printf("REFs not equal\n");

/*Assign ref1 to ref2 */
status = OCIRefAssign (envhp, errhp, ref1, &ref2);
if(status != OCI_SUCCESS)
/*error handling*/

/*Compare the two REFs again for equality */
refs_equal = OCIRefIsEqual(envhp, ref1, ref2);
```

```
printf("After second OCIRefIsEqual:\n");
if(refs_equal)
   printf("REFs equal\n");
else
   printf("REFs not equal\n");
```

Object Type Information Storage and Access

Descriptor Objects

When a given type is created with the CREATE TYPE statement, it is stored in the server and associated with a type descriptor object (TDO). In addition, the database stores descriptor objects for each data attribute of the type, each method of the type, each parameter of each method, and the results returned by methods. The following table lists the OCI datatypes associated with each type of descriptor object.

Information Type		OCI Datatype
Type		OCIType
Type Attributes	Collection Elements	OCITypeElem
Method Parameters	Method Results	
Method		OCITypeMethod

Several OCI functions (including OCIBindObject() and OCIObjectNew()) require a TDO as an input parameter. An application can obtain the TDO by calling OCITypeByName(), which gets the type's TDO in an **OCIType** variable. Once you obtain the TDO, you can pass it, as necessary to other calls.

Any Type, AnyData and AnyDataSet Interfaces

These capabilities allow you to model self-descriptive data. You can store heterogeneous data types in the same column and query the type of data in an application.

These definitions are used in the discussion in the following sections:

- *Persistent types.* These are created using the SQL statement CREATE TYPE. They are stored persistently in the database.

- *Transient types.* Anonymous type descriptions that are not stored persistently in the database.They are created by programs on the fly. They are useful for

exchanging type information, if necessary, between various components of an application in a dynamic fashion.

- *Self-descriptive data.* Data encapsulating type information with its actual contents. The **OCIAnyData** datatype models such data in OCI. A data value of most SQL types can be converted to an **OCIAnyData** which can then be converted back to the old data value. The type SYS.ANYDATA models such data in SQL or PL/SQL.

- *Self-descriptive dataset.* Encapsulation of a set of data instances (all of the same type) along with their type description. They should all have the same type description.The **OCIDataAnySet** datatype models this data in OCI. The type SYS.ANYDATASET models such data in SQL or PL/SQL.

Interfaces are available in both OCI (C language) as well as in SQL and PL/SQL for constructing and manipulating these type descriptions as well as self-descriptive data. The following sections describe the relevant OCI interfaces.

> **See Also:**   For more information see "Persistent Objects, Transient Objects, and Values" on page 10-6, and *Oracle9i SQL Reference,* section *"Oracle Supplied Types"* for an overview

Type Interfaces

The type interfaces can be used to construct named as well as anonymous transient object types (structured with attributes) and collection types. The OCITypeBeginCreate() call is used to begin type construction of transient object types as well as collection types (the typecode parameter determines which one is being constructed).

You need to allocate a parameter handle using OCIDescriptorAlloc(). Subsequently, type information (for attributes of an object type as well as for the collection element's type) needs to be set using OCIAttrSet(). For object types, use OCITypeAddAttr() to add the attribute information to the type. After information on the last attribute has been added, you must call OCITypeEndCreate().

For example:

```
OCITypeBeginCreate( ...)        /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeAddAttr(...)             /* Add attribute 1 */
OCIAttrSet(...)
OCITypeAddAttr(...)             /* Add attribute 2 */
...
```

```
OCITypeEndCreate(...)            /* End Type Creation */
```

For collection types, the information on the collection element type needs to be set with `OCITypeSetCollection()`. Subsequently, `OCITypeEndCreate()` is called to finish construction.

For example:

```
OCITypeBeginCreate( ...)         /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeSetCollection(...)        /* Set information on collection element etc.*/
OCITypeEndCreate(...)            /* End Type Creation */
```

The `OCIDescribeAny()` call can be used to obtain the **OCIType** corresponding to a persistent type.

Creating a Parameter Descriptor for OCIType Calls

The `OCIDescriptorAlloc()` call can be used to allocate an **OCIParam** (with the parent handle being the environment handle). Subsequently, `OCIAttrSet()` can be called with the following allowed attribute types to set relevant type information:

- OCI_ATTR_PRECISION

To set numeric precision. Pass a (`ub1 *`) attribute value to the buffer holding precision value.

- OCI_ATTR_SCALE

To set numeric scale. Pass a (`sb1 *`) attribute value to the buffer holding scale value.

- OCI_ATTR_CHARSET_ID

To set the character set id for character types. Pass a (`ub2 *`) attribute value to the buffer holding char set id.

- OCI_ATTR_CHARSET_FORM

To set the character set form for character types. Pass a (`ub1 *`) attribute value to the buffer holding character set form value.

- OCI_ATTR_DATA_SIZE

Length of VARCHAR2, RAW, etc. Pass a (`ub4 *`) attribute value to the buffer holding length.

- OCI_ATTR_TYPECODE

To set typecode. Pass a (ub2 *) attribute value to the buffer holding typecode. This attribute needs to be set first.

- OCI_ATTR_TDO

To set **OCIType** of an object or collection attribute. Pass a (OCIType *) attribute value to the **OCIType** corresponding to the attribute. It is your responsibility to make sure that the **OCIType** is pinned when this **OCIParam** is used during **AnyType** construction. If it is a transient type attribute, its allocation duration should be at least as much as the top level **OCIType** being created. There will be an exception returned otherwise.

- For builtin types, here are the acceptable typecodes (the permissible values for OCI_ATTR_TYPECODE) for SQL type attributes:

  OCI_TYPECODE_DATE, OCI_TYPECODE_NUMBER, OCI_TYPECODE_VARCHAR, OCI_TYPECODE_ RAW, OCI_TYPECODE_CHAR, OCI_TYPECODE_VARCHAR2, OCI_TYPECODE_VARCHAR, OCI_TYPECODE_BLOB, OCI_TYPECODE_BFILE, OCI_TYPECODE_CLOB

- If the attribute/collection element type is itself another transient type, set OCI_ATTR_TYPECODE to:

  OCI_TYPECODE_OBJECT or OCI_TYPECODE_REF (for REFS) or OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE and set the OCI_ATTR_TDO to the **OCIType** corresponding to the transient type.

- For user defined type attributes, the permissible values for OCI_ATTR_TYPECODE are:

  - OCI_TYPECODE_OBJECT (for an Object Type),

  - OCI_TYPECODE_REF (for a REF type)

  - and OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE (for collections).

    The OCI_ATTR_TDO should be set in these cases to the appropriate user defined type's **OCIType**.

Obtaining the OCIType for Persistent Types

The OCIDescribeAny() call can be used to obtain the **OCIType** corresponding to a persistent type. For example:

```
OCIDescribeAny(svchp, errhp. (dvoid *)"SCOTT.EMP", (ub4)strelen("SCOTT.EMP"),
               (ub1)OCI_OTYPE_NAME, (ub1)OCI_DEFAULT, OCI_PTYPE_TYPE, dschp);
```

From the describe handle (*dschp*), the **OCIType** can be obtained using
OCIAttrGet() calls.

Type Access Calls

OCIDescribeAny() can be called with these transient type descriptions for a
dynamic description of the type. The **OCIType** pointer can be passed directly to
OCIDescribeAny() (with *objtype* set to OCI_OTYPE_PTR). This provides a
way to obtain attribute information by name as well as position.

Extensions to OCIDescribeAny()

For transient types that represent builtin types (created with a builtin typecode), the
parameter handle that describes these types (which will be of type
OCI_PTYPE_TYPE) will support the following extra attributes.

OCI_ATTR_DATA_SIZE,

OCI_ATTR_TYPECODE,

OCI_ATTR_DATA_TYPE,

OCI_ATTR_PRECISION,

OCI_ATTR_SCALE,

OCI_ATTR_CHARSET_ID

OCI_ATTR_CHARSET_FORM,

OCI_ATTR_LFPRECISION,

OCI_ATTR_FSPRECISION

These attributes will have the usual meanings they have while describing a type
attribute.

> **Note:**   These attributes are supported only for transient builtin
> types. The attributes OCI_ATTR_IS_TRANSIENT_TYPE and
> OCI_ATTR_IS_PREDEFINED_TYPE are true for these types. For
> persistent types, these attributes are supported only from the
> parameter handle of the type's attributes (which will be of type
> OCI_PTYPE_TYPE_ATTR).

OCIAnyData Interfaces

An **OCIAnyData** encapsulates type information as well as a data instance of that type (that is, self descriptive data). An **OCIAnyData** can be created from any builtin or user-defined type instance using the `OCIAnyDataConvert()` call. This call does a conversion (cast) to an **OCIAnyData**.

Alternatively, object types and collection types can be constructed piece by piece (an attribute at a time for object types or a collection element at a time). To construct in this fashion, `OCIAnyDataBeginCreate()` should be called with the type information (**OCIType**). Subsequently `OCIAnyDataAttrSet()` can be used for object types and `OCIAnyDataCollAddElem()` can be used for collection types. `OCIAnyDataEndCreate()` must then be called to finish the construction process.

Subsequently, the access routines can be invoked. To convert (cast) an **OCIAnyData** to the corresponding type instance, the `OCIAnyDataAccess()` call can be used.

An **OCIAnyData** that is based on an object or collection type can also be accessed piece by piece.

Special collection construction and access calls are provided for performance improvement. These calls can be used to avoid unnecessary creation and copying of the entire collection in memory. For example:

```
OCIAnyDataConvert(...)        /* Cast a builtin or user-defined type instance
                                 to an OCIAnyData in 1 call. */

OCIAnyDataBeginCreate(...)    /* Begin AnyData Creation */

OCIAnyDataAttrSet(...)        /* Attribute-wise construction for object types */

or,

OCIAnyDataCollAddElem(...)    /* Element-wise construction for collections */

OCIAnyDataEndCreate(...)      /* End OCIAnyData Creation */
```

OCIAnyDataSet Interfaces

An **OCIAnyDataSet** encapsulates type information as well as *a set of instances* of that type. `OCIAnyDataSetBeginCreate()` is called to begin the construction process. `OCIAnyDataSetAddInstance()` is called to add a new instance and this call returns the **OCIAnyData** corresponding to that instance.

Then, the OCIAnyData functions can be invoked to construct this instance. `OCIAnyDataSetEndCreate()` is called once all instances have been added.

For access, call `OCIAnyDataSetGetInstance()` to get the **OCIAnyData** corresponding to the instance. Only sequential access is supported. Subsequently, the OCIAnyData access functions can be invoked.For example:

```
OCIAnyDataSetBeginCreate(...)    /* Begin AnyDataSet Creation */
OCIAnyDataSetAddInstance(...)    /* Add a new instance to the AnyDataSet */
                                 /* Use the OCIAnyData*() functions to create
                                    the instance */
OCIAnyDataSetEndCreate(...)      /* End OCIAnyDataSet Creation */
```

> **Note:** For complete descriptions of all the calls in these interfaces, see Chapter 20, "OCI Any Type and Data Functions".

Binding Named Datatypes

This section provides information on binding named datatypes, such as objects and collections, and REFs.

Named Datatype Binds

For a named datatype (object type or collection) bind, a second bind call is necessary following `OCIBindByName()`, or `OCIBindByPos()`. The OCI Bind Object Type call, `OCIBindObject()`, sets up additional attributes specific to the object type bind. An OCI application uses this call when fetching data from a table which has a column with an object datatype.

The `OCIBindObject()` call takes, among other parameters, a Type Descriptor Object (TDO) for the named data type. The TDO, of datatype **OCIType** is created and stored in the database when a named data type is created. It contains information about the type and its attributes. An application can obtain a TDO by calling `OCITypeByName()`.

The `OCIBindObject()` call also sets up the indicator variable or structure for the named data type bind.

When binding a named data type, use the SQLT_NTY datatype constant to indicate the datatype of program variable being bound. SQLT_NTY indicates that a C struct representing the named data type is being bound. A pointer to this structure is passed to the bind call.

With inheritance and instance substitutability, you can bind a subtype instance where the supertype is expected.

It is possible that working with named data types may require the use of three bind calls in some circumstances. For example, to bind a static array of named data types to a PL/SQL table, three calls must be invoked: `OCIBindByName()`, `OCIBindArrayOfStruct()`, and `OCIBindObject()`.

See Also:

- For information about using these data types to fetch an embedded object from the database, refer to the section "Fetching Embedded Objects" on page 10-16.

- For additional important information, see the section "Information for Named Datatype and REF Binds" on page 11-37

- For more information about descriptor objects, see "Descriptor Objects" on page 11-29.

Binding REFs

As with named data types, binding REFs is a two-step process. First, call OCIBindByName() or OCIBindByPos(), and then call OCIBindObject().

REFs are bound using the SQLT_REF datatype. When SQLT_REF is used, then the program variable being bound must be of type OCIRef *.

With inheritance and REF substitutability, you can bind a REF value to a subtype instance where a REF to the supertype is expected.

See Also:

- For information about binding and pinning REFs to objects, see "Retrieving an Object Reference from the Server" on page 10-11.

- For additional important information, see the section "Information for Named Datatype and REF Binds" on page 11-37.

Information for Named Datatype and REF Binds

This section presents some additional important information to keep in mind when working with named data type and REF binds. It includes pointers about memory allocation and indicator variable usage.

- If the datatype being bound is SQLT_NTY, the indicator struct parameter of the OCIBindObject() call (*dvoid ** indpp*) is used, and the scalar indicator is completely ignored.

- If the datatype is SQLT_REF, the scalar indicator is used, and the indicator struct parameter of OCIBindObject() is completely ignored.

- The use of indicator structures is optional. The user can pass a NULL pointer in the *indpp* parameter for the OCIBindObject() call. During the bind, this means that the object is not atomically NULL and none of its attributes are NULL.

- The indicator struct size pointer, *indsp*, and program variable size pointer, *pgvsp*, in the OCIBindObject() call is optional. Users can pass NULL if these parameters are not needed.

Information Regarding Array Binds

For doing array binds of named data types or REFs, for array inserts or fetches, the user needs to pass in an array of pointers to buffers (pre-allocated or otherwise) of the appropriate type. Similarly, an array of scalar indicators (for SQLT_REF types) or an array of pointers to indicator structs (for SQLT_NTY types) needs to be passed.

> **See Also:** For more information about SQLT_NTY, see the section "New External Datatypes" on page 3-19.

Defining Named Datatypes

This section provides information on defining named data types (for example, objects, collections) and REFs.

Defining Named Datatype Output Variables

For a named datatype (object type, nested table, varray) define, two define calls are necessary. The application should first call OCIDefineByPos(), specifying SQLT_NTY in the dty parameter. Following OCIDefineByPos(), the application must call OCIDefineObject(). In this case, the data buffer pointer in OCIDefineByPos() is ignored and additional attributes pertaining to a named data type define are set up using the OCI Define Object attributes call, OCIDefineObject().

There SQLT_NTY datatype constant is specified for a named datatype define. In this case, the application fetches the result data into a host-language representation of the named data type. In most cases, this will be a C struct generated by the Object Type Translator.

When making an OCIDefineObject() call, a pointer to the address of the C struct (preallocated or otherwise) must be provided. The object may have been

created with `OCIObjectNew()`, allocated in the cache, or with user-allocated memory.

However, in the presence of inheritance, we strongly recommend using objects in the object cache and *not* passing objects allocated out of user memory from the stack. The reason is that due to instance substitutability, the server may send back a *subtype* instance when the client is expecting a supertype instance. This requires the server to dynamically re-size the object -- which is possible only for objects in the cache.

> **Note:** Please refer to the section"Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-39 for more important information about defining named data types.

Defining REF Output Variables

As with named data types, defining for a REF output variable is a two-step process. The first step is a call to `OCIDefineByPos()`, and the second is a call to `OCIDefineObject()`. Also as with named data types, the SQLT_REF datatype constant is passed to the *dty* parameter of `OCIDefineByPos()`.

SQLT_REF indicates that the application will be fetching the result data into a variable of type **OCIRef ***. This REF can then be used as part of object pinning and navigation, as described in Chapter 6.

> **Note:** Please refer to the section"Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-39 for more important information about defining REFs.

Information for Named Datatype and REF Defines, and PL/SQL OUT Binds

This section presents some additional important information to keep in mind when working with named data type and REF defines. It includes pointers about memory allocation and indicator variable usage.

A PL/SQL OUT bind refers to binding a placeholder to an output variable in a PL/SQL block. Unlike a SQL statement, where output buffers are set up with define calls, in a PL/SQL block, output buffers are set up with bind calls. Refer to the section "Binding Placeholders in PL/SQL" on page 5-5 for more information.

- If the datatype being defined is SQLT_NTY, the indicator struct parameter of the OCIDefineObject() call (*dvoid ** indpp*) is used, and the scalar indicator is completely ignored.

- If the datatype is SQLT_REF, the scalar indicator is used, and the indicator struct parameter of OCIDefineObject() is completely ignored.

- The use of indicator structures is optional. The user can pass a NULL pointer in the *indpp* parameter for the OCIDefineObject() call. During a fetch or PL/SQL OUT bind, this means that the user is not interested in any information about nullity.

- In a SQL define or PL/SQL OUT bind, you can pass in preallocated memory for either the output variable or the indicator. Then that preallocated memory is used to store result data, and all secondary memory (out-of-line memory), if any, will be deallocated. The pre-allocated memory can either come from the cache (the result of an OCIObjectNew() call), or from the client's private memory space.

    > **Note:** If a client application wants to allocate memory from its own private memory space, instead of the cache, it must insure that there is no secondary out-of-line memory in the object.

For an object define with type SQLT_NTY, client applications wanting to pre-allocate object memory must use the OCIObjectNew() function. Client applications should not allocate the object in its own private memory space, such as with malloc() or on the stack. The OCIObjectNew() function allocates the object in the object cache. The allocated object can be freed using OCIObjectFree(). Refer to Chapter 17, "OCI Navigational and Type Functions" for details on OCIObjectNew() and OCIObjectFree().

    > **Note:** There is no change to the behavior of OCIDefineObject() when the user does not pre-allocate the object memory and instead initializes the output variable to null pointer value. In this case, the object will be implicitly allocated in the object cache by the OCI library.

- In a SQL define or PL/SQL OUT bind, if the user passes in a NULL address for the output variable or the indicator, memory for the variable or the indicator will be implicitly allocated by OCI.

- If an output object of type SQLT_NTY is atomically NULL (in a SQL define or PL/SQL OUT bind), only the NULL indicator struct will get allocated (implicitly if necessary) and populated accordingly to indicate the atomic nullity of the object. The top-level object, itself, will not get implicitly allocated.

- An application can free indicators by calling OCIObjectFree(). If there is a top-level object (as in the case of a non-atomically NULL object), then the indicator is freed when the top-level object is freed with OCIObjectFree(). If the object is atomically null, then there is no top-level object, so the indicator must be freed separately.

- The indicator struct size pointer, *indsp*, and program variable size pointer, *pgvsp*, in the OCIDefineObject() call is optional. Users can pass NULL if these parameters are not needed.

Information About Array Defines

For doing array defines of named data types or REFs, the user needs to pass in an array of pointers to buffers (pre-allocated or otherwise) of the appropriate type. Similarly, an array of scalar indicators (for SQLT_REF types) or an array of pointers to indicator structs (for SQLT_NTY types) needs to be passed.

Binding And Defining Oracle C Datatypes

Previous chapters of this book have discussed OCI bind and define operations. "What is Binding?" on page 4-6 discussed the basics of OCI bind operations, while "What is Defining?" on page 4-15 discusses the basics of OCI define operations. Information specific to binding and defining named data types and REFs is found in Chapter 5, "Binding and Defining".

The sections covering basic bind and define functionality showed how an application could use a scalar variable or array of scalars as an input (bind) value in a SQL statement, or as an output (define) buffer for a query.

The sections covering named data types and REFs showed how to bind or define an object or reference. Chapter 10, "OCI Object-Relational Programming" expanded on this to talk about pinning object references, object navigation, and fetching embedded instances.

The purpose of this section is to cover binding and defining of individual attribute values, using the datatype mappings explained in this chapter.

Variables of one of the types defined in this chapter, such as **OCINumber** or **OCIString**, can typically be declared in an application and used directly in an OCI

bind or define operation as long as the appropriate datatype code is specified. The following table lists the datatypes that can be used for binds and defines, along with their C mapping, and the OCI external datatype which must be specified in the *dty* (datatype code) parameter of the bind or define call.

Table 11–1   Datatype Mappings for Binds and Defines

Datatype	C Mapping	OCI External Datatype and Code
Oracle number	OCINumber	VARNUM (SQLT_VNU)
Oracle date	OCIDate	SQLT_ODT
VARCHAR2	OCIString *	SQLT_VST (see Note 1 below)
RAW	OCIRaw *	SQLT_LVB (see Note 1 below)
CHAR	OCIString *	SQLT_VST
OBJECT	struct *	Named Data Type (SQLT_NTY)
REF	OCIRef *	REF (SQLT_REF)
VARRAY	OCIArray *	Named Data Type (SQLT_NTY)
Nested Table	OCITable *	Named Data Type (SQLT_NTY)
DATETIME	OCIDateTime *	See "Datetime and Interval (OCIDateTime, OCIInterval)" on page 11-9.
INTERVAL	OCIInterval *	See "Datetime and Interval (OCIDateTime, OCIInterval)" on page 11-9.

Note: Before fetching data into a define variable of type **OCIString ***, the size of the string must first be set using the `OCIStringResize()` routine. This may require a describe operation to obtain the length of the select-list data. Similarly, an **OCIRaw *** must be first sized with `OCIRawResize()`.

The following section presents examples of how to use C-mapped datatypes in an OCI application.

See Also: For a discussion of OCI external datatypes, and a list of datatype codes, refer to Chapter 3, "Datatypes".

Bind and Define Examples

The examples in this section demonstrate how variables of type **OCINumber** can be used in OCI bind and define operations.

> **Note:** The examples in this section are intended to demonstrate the flow of calls used to perform certain OCI tasks. An expanded pseudocode is used for the examples in this section. Actual function names are used, but for the sake of simplicity not all parameters and typecasts are filled in. Additionally, other necessary OCI calls, like handle allocations, have been omitted.

Assume, for this example, that the following `person` object type was created:

```
CREATE TYPE person AS OBJECT
(name     varchar2(30),
salary    number);
```

This type is then used to create an `employees` table which has a column of type `person`.

```
CREATE TABLE employees
(emp_id    number,
job_title  varchar2(30),
emp        person);
```

The Object Type Translator (OTT) generates the following C struct and null indicator struct for `person`:

```
struct person
{   OCIString * name;
    OCINumber salary;};
typedef struct person person;

struct person_ind
{   OCIInd  _atomic;
    OCIInd  name;
    OCIInd  salary;}
typedef struct person_ind person_ind;
```

> **See Also:** For a complete discussion of OTT, see Chapter 14, "The Object Type Translator (OTT)"

Assume that the `employees` table has been populated with values, and an OCI application has declared a `person` variable:

```
person *my_person;
```

and fetched an object into that variable through a `SELECT` statement, like

```
text *mystmt = (text *) "SELECT person FROM employees
                        WHERE emp.name='Andrea'";
```

This would require defining `my_person` to be the output variable for this statement, using appropriate OCI define calls for named datatypes, as described in the section "Advanced Define Operations" on page 5-22. Executing the statement would retrieve the `person` object named `Andrea` into the `my_person` variable.

Once the object is retrieved into `my_person`, the OCI application now has access to the attributes of `my_person`, including the name and the salary.

The application could go on to update another employee's salary to be the same as Andrea's, as in

```
text *updstmt = (text *) "UPDATE employees SET emp.salary = :newsal
                          WHERE emp.name = 'MONGO'"
```

Andrea's salary (stored in `my_person->salary`) would be bound to the placeholder `:newsal`, specifying an external datatype of `VARNUM` (datatype code=6) in the bind operation:

```
OCIBindByName(...,":newsal",...,&my_person->salary,...,6,...);
OCIStmtExecute(...,updstmt,...)
```

Executing the statement updates Mongo's salary in the database to be equal to Andrea's, as stored in `my_person`.

Conversely, the application could update Andrea's salary to be the same as Mongo's, by querying the database for Mongo's salary, and then making the necessary salary assignment:

```
text *selstmt = (text *) "SELECT emp.salary FROM employees
                          WHERE emp.name = 'MONGO'"
OCINumber mongo_sal;
...
OCIDefineByPos(...,1,...,&mongo_sal,...,6,...);
OCIStmtExecute(...,selstmt,...);
OCINumberAssign(...,&mongo_sal, &my_person->salary);
```

In this case, the application declares an output variable of type **OCINumber** and uses it in the define step. In this case we define an output variable for position 1, and use the appropriate datatype code (6 for VARNUM).

The salary value is fetched into the mongo_sal **OCINumber**, and the appropriate OCI function, OCINumberAssign(), is used to assign the new salary to the copy of the Andrea object currently in the cache. To modify the data in the database, the change must be flushed to the server.

Salary Update Examples

The examples in the previous section should give some idea of the flexibility which the Oracle datatypes provide for bind and define operations. The goal of this section is to show how the same operation can be performed in several different ways. The goal is to give you some idea of the variety of ways in which these datatypes can be used in OCI applications.

The examples in this section are intended to demonstrate the flow of calls used to perform certain OCI tasks. An expanded pseudocode is used for the examples in this section. Actual function names are used, but for the sake of simplicity not all parameters and typecasts are filled in. Additionally, other necessary OCI calls, like handle allocations, have been omitted.

The Scenario
The scenario for these examples is as follows:

1. An employee named *BRUCE* exists in the employees table for a hospital. See person type and employees table creation statements in the previous section.

2. Bruce's current job title is *RADIOLOGIST*.

3. Bruce is being promoted to *RADIOLOGY_CHIEF*, and along with the promotion comes a salary increase.

4. Hospital salaries are in whole dollar values, are set according to job title, and stored in a table called salaries, defined as follows:

```
CREATE TABLE salaries
(job_title   varchar2(20),
salary       integer));
```

5. Bruce's salary needs to be updated to reflect his promotion.

Accomplishing the above task requires that the application retrieve the salary corresponding to *RADIOLOGY_CHIEF* from the salaries table, and update

Bruce's salary. A separate step would write his new title and the modified object back to the database.

Assuming that a variable of type `person` has been declared

```
person * my_person;
```

and the object corresponding to Bruce has been fetched into it, the following sections present three different ways in which the salary update could be performed.

Method 1 - fetch, convert, assign

This example uses the following method:

1. Do a traditional OCI define using an integer variable to retrieve the new salary from the database.

2. Convert the integer to an **OCINumber.**

3. Assign the new salary to Bruce.

```
#define INT_TYPE 3        /* datatype code for sword integer define */

text *getsal = (text *) "SELECT salary FROM salaries
                    WHERE job_title='RADIOLOGY_CHIEF'
sword     new_sal;
OCINumber    orl_new_sal;
...
OCIDefineByPos(...,1,...,new_sal,...,INT_TYPE,...);
                    /* define int output */
OCIStmtExecute(...,getsal,...);
                    /* get new salary as int */
OCINumberFromInt(...,new_sal,...,&orl_new_sal);
                    /* convert salary to OCINumber */
OCINumberAssign(...,&orl_new_sal, &my_person->salary);
                    /* assign new salary */
```

Method 2 - fetch, assign

This method eliminates one of the steps in Method 1:

1. Define an output variable of type **OCINumber**, so that no conversion is necessary after the value is retrieved.

2. Assign the new salary to Bruce

```
#define VARNUM_TYPE 6        /* datatype code for defining VARNUM */
```

```
text *getsal = (text *) "SELECT salary FROM salaries
                            WHERE job_title='RADIOLOGY_CHIEF'
OCINumber   orl_new_sal;
...
OCIDefineByPos(...,1,...,orl_new_sal,...,VARNUM_TYPE,...);
                                    /* define OCINumber output */
OCIStmtExecute(...,getsal,...);       /* get new salary as OCINumber */
OCINumberAssign(...,&orl_new_sal, &my_person->salary);
                                    /* assign new salary */
```

Method 3 - direct fetch

This method accomplishes the entire operation with a single define and fetch. No intervening output variable is used, and the value retrieved from the database is fetched directly into the salary attribute of the object stored in the cache.

1. Since Bruce is pinned in the object cache, use the location of his salary attribute as the define variable, and execute/fetch directly into it.

```
#define VARNUM_TYPE 6          /* datatype code for defining VARNUM */

text *getsal = (text *) "SELECT salary FROM salaries
                            WHERE job_title='RADIOLOGY_CHIEF'
...
OCIDefineByPos(...,1,...,&my_person->salary,...,VARNUM_TYPE,...);
          /* define bruce's salary in cache as output variable */
OCIStmtExecute(...,getsal,...);
              /* execute and fetch directly */
```

Summary and Notes

As the previous three examples show, the C datatypes provide flexibility for binding and defining. In these examples an integer can be fetched, and then converted to an **OCINumber** for manipulation. An **OCINumber** can be used as an intermediate variable to store the results of a query. Or, data can be fetched directly into a desired **OCINumber** attribute of an object.

> **Note:**   In all of these examples it is important to keep in mind that in OCI, if an output variable is defined before the execution of a query, the resulting data will be prefetched directly into the output buffer.

In the above examples, extra steps would be necessary to insure that changes are written to the database permanently. This may involve SQL UPDATE calls and OCI transaction commit calls.

These examples all dealt with define operations, but a similar situation applies for binding.

Similarly, although these examples dealt exclusively with the **OCINumber** type, a similar variety of operations are possible for the other Oracle C types described in the remainder of this chapter.

SQLT_NTY Bind/Define Example

The following code fragments demonstrate the use of SQLT_NTY bind and define calls, including OCIBindObject() and OCIDefineObject(). In each example, a previously defined SQL statement is being processed.

Bind Example

```
/*
** This example performs a SQL insert statement
*/
void insert(envhp, svchp, stmthp, errhp, insstmt, nrows)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
text *insstmt;
ub2    nrows;
{
  orttdo *addr_tdo = NULLP(orttdo);
  address  addrs;
  null_address naddrs;
  address *addr = &addrs;
  null_address *naddr = &naddrs;
  sword custno =300;
  OCIBind *bnd1p, *bnd2p;
  ub2 i;

  /* define the application request  */
  checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) insstmt,
          (ub4) strlen((char *)insstmt),
          (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));
```

```
                    /* bind the input variable */
           checkerr(errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":custno",
                      (sb4) -1, (dvoid *) &custno,
                      (sb4) sizeof(sword), SQLT_INT,
                      (dvoid *) 0, (ub2 *)0, (ub2 *)0, (ub4) 0, (ub4 *) 0,
                      (ub4) OCI_DEFAULT));

           checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":addr",
                      (sb4) -1, (dvoid *) 0,
                      (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                      (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT));

           checkerr(errhp, OCITypeByName(envhpx, errhp, svchpx, (const text *)
                  SCHEMA, (ub4) strlen((char *)SCHEMA), (const text *)
                   "ADDRESS_VALUE", (ub4) strlen((char *)"ADDRESS_VALUE"),
                   OCI_DURATION_SESSION, &addr_tdo));

           if(!addr_tdo)
           {
             printf("Null tdo returned\n");
             goto done_insert;
           }

           checkerr(errhp, OCIBindObject(bnd2p, errhp, addr_tdo, (dvoid **) &addr,
                 (ub4 *) 0, (dvoid **) &naddr, (ub4 *) 0));
```

Define Example

```
/*
** This example executes a SELECT statement from a table which includes
** an object.
*/

void selectval(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
{
  orttdo *addr_tdo = NULLP(orttdo);
  OCIDefine *defn1p, *defn2p;
  address *addr = (address *)NULL;
  sword custno =0;
  sb4 status;
```

```
      /* define the application request  */
      checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) selvalstmt,
                            (ub4) strlen((char *)selvalstmt),
                            (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    /* define the output variable */
  checkerr(errhp, OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *)
          &custno, (sb4) sizeof(sword), SQLT_INT, (dvoid *) 0, (ub2 *)0,
          (ub2 *)0, (ub4) OCI_DEFAULT));

  checkerr(errhp, OCIDefineByPos(stmthp, &defn2p, errhp, (ub4) 2, (dvoid *)
           0, (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
           (ub2 *)0, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCITypeByName(envhpx, errhp, svchpx, (const text *)
          SCHEMA, (ub4) strlen((char *)SCHEMA), (const text *)
          "ADDRESS_VALUE", (ub4) strlen((char *)"ADDRESS_VALUE"),OROODTSES,
          &addr_tdo));

    if(!addr_tdo)
    {
      printf("NULL tdo returned\n");
      goto done_selectval;
    }


    checkerr(errhp, OCIDefineObject(defn2p, errhp, addr_tdo, (dvoid **)
          &addr, (ub4 *) 0, (dvoid **) 0, (ub4 *) 0));

    checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
          (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));
```

12

Direct Path Loading

The direct path loading functions are used to load data from external files into tables and partitions.

This chapter includes the following sections:

Direct Path Loading Overview

The direct path load interface enables an OCI application to access the direct path
load engine of the Oracle database server to perform the functions of the Oracle
SQL*Loader utility. This functionality provides the ability to load data from
external files into either a table or a partition of a partitioned table.

Figure 12–1   Direct Path Loading
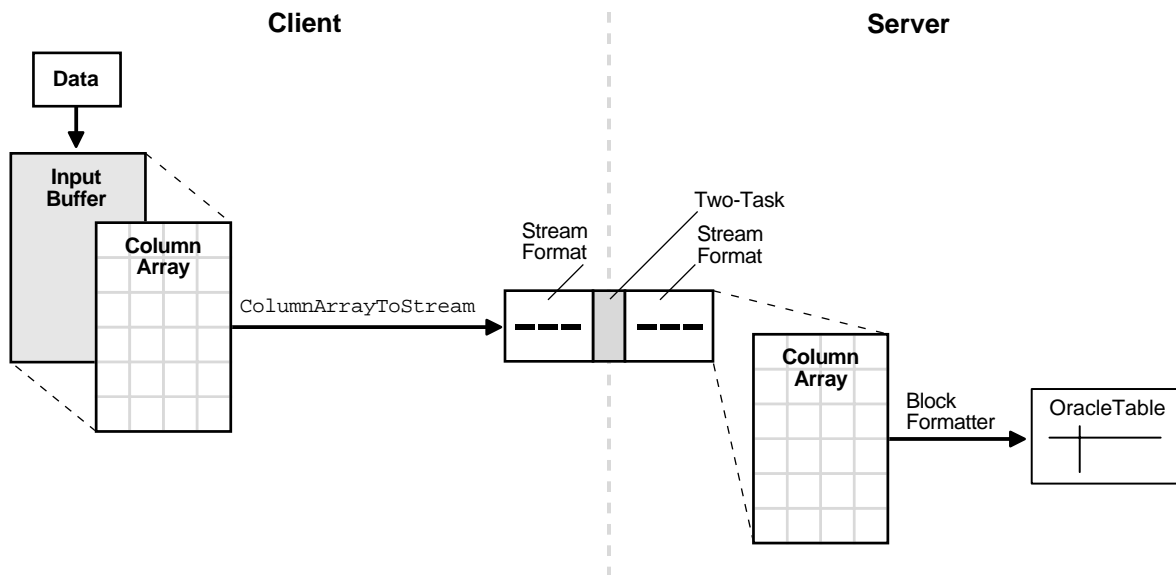
The OCI direct path load interface has the ability to load multiple rows by loading a
direct path stream that contains data for multiple rows.

To use the direct path API, the client application performs the following steps:

1. Perform the OCI initialization.

2. Allocate a direct path context handle and set the attributes.

3. Supply the name of the object (table, partition, or sub-partition) to be loaded.

4. Describe the external data types of the columns of the object(s).

5. Prepare the direct path interface.

6. Allocate one or more column arrays.

7. Allocate one or more direct path streams.

8. Set entries in the column array to point to the input data value for each column.

9. Convert a column array to a direct path stream format.

10. Load the direct path stream.

11. Retrieve any errors that may have occurred.

12. Invoke the direct path finishing function.

13. Free handles and data structures.

14. Disconnect from the server.

Steps 8 through 11 can be repeated many times, depending on the data to be loaded.

A direct load operation requires that the object being loaded is locked to prevent DML on the object. Note that queries are lock-free and are allowed while the object is being loaded. The mode of the DML lock, and which DML locks are obtained depend upon the specification of the OCI_DIRPATH_PARALLEL_LOAD option, and if a partition or sub-partition load is being done as opposed to an entire table load.

> **See Also:** For more information on
> OCI_DIRPATH_PARALLEL_LOAD, see `OCIDirPathPrepare()`
> on page 16-112

- For a table load, if the OCI_DIRPATH_PARALLEL_LOAD option is set to:
  - FALSE, then the table DML X-Lock is acquired.
  - TRUE, then the table DML S-Lock is acquired.
- For a partition load, if the OCI_DIRPATH_PARALLEL_LOAD option is set to:
  - FALSE, then the table DML SX-Lock and partition DML X-Lock is acquired.
  - TRUE, then the table DML SS-Lock and partition DML S-Lock is acquired.

Datatypes Supported for Direct Path Loading

The following external datatypes are valid for scalar columns in a direct path load operation:

- SQLT_CHR
- SQLT_DAT

- SQLT_INT
- SQLT_UIN
- SQLT_FLT
- SQLT_BIN
- SQLT_NUM
- SQLT_PDN

The following external object datatypes are supported:

- SQLT_NTY - column objects (FINAL and NOT FINAL) and SQL string columns
- SQLT_REF - REF columns (FINAL and NOT FINAL)

The following table types are supported:

- Nested tables
- Object tables (FINAL and NOT FINAL)

> **See Also:** For information on setting or retrieving the datatype of a column, see OCI_ATTR_DATA_TYPE on page A-69. For information on datatypes, see Table 3–2, "External Datatypes and Codes".

Direct Path Handles

A direct path load corresponds to a direct path array insert operation. The direct path load interface uses the following handles to keep track of the objects loaded and the specification of the data operated on:

- direct path context
- direct path function context
- direct path column array
- direct path function context column array
- direct path stream

> **See Also:** "Direct Path Loading Handle Attributes" on page A-58 and all the descriptions of direct path attributes following

Direct Path Context

This handle must be allocated for each object, either a table or a partition of a partitioned table, being loaded. Because a **OCIDirPathCtx** handle is the parent handle of the **OCIDirPathFuncCtx**, **OCIDirPathColArray,** and **OCIDirPathStream** handles, freeing a **OCIDirPathCtx** handle frees its child handles also (although for good coding practices, free child handles individually before you free the parent handle).

A direct path context is allocated with `OCIHandleAlloc()`.

```
OCIEnv *envp;
OCIDirPathCtx *dpctx;
sword error;
    error = OCIHandleAlloc((dvoid *)envp, (dvoid **)&dpctx,
                           OCI_HTYPE_DIRPATH_CTX, 0,(dvoid **)0);
```

Note that the parent handle of a direct path context is always the environment handle. A direct path context is freed with `OCIHandleFree()`.

```
error = OCIHandleFree(dpctx, OCI_HTYPE_DIRPATH_CTX);
```

OCI Direct Path Function Context

> **See Also:** For more about the datatypes supported, see *Oracle9i Application Developer's Guide - Object-Relational Features*

This handle, of type **OCIDirPathFuncCtx**, is used to describe the following named type and REF columns:

- Column objects. The function context here describes the object type, which will be used as the default constructor to construct the object, and the object attributes of the constructor.

- REF columns. The function context here describes a single object table (optional) to reference row objects from, and the REF arguments that identify the row object.

- SQL string columns. The function context here describes a SQL string and its arguments to compute the value to be loaded into the column.

The handle type OCI_HTYPE_DIRPATH_FN_CTX is passed to `OCIHandleAlloc()` to indicate that a function context is to be allocated, as in the following example.

```
OCIDirPathCtx *dpctx;        /* direct path context */
```

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */

OCIHandleAlloc((dvoid *)dpctx,
               (dvoid **)&dpfnctx,
               (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
               (size_t)0, (dvoid **)0));
```

Note that the parent handle of a direct path function context is always the direct path context handle. A direct path function context handle is freed with:

```
error = OCIHandleFree(dpfnctx, OCI_HTYPE_DIRPATH_FN_CTX);
```

Direct Path Column Array and Direct Path Function Column Array

These handles are used to present an array of rows to the direct path interface. A row is represented by three arrays: column values, column lengths, and column flags. Methods used on a column array include: allocate the array handle and set or get values corresponding to an array entry.

Both handles share the same data structure, **OCIDirPathColArray**. But these column array handles differ in parent handles and handle types.

A direct path column array handle is allocated with OCIHandleAlloc(). The following code fragment shows explicit allocation of the direct path column array handle:

```
OCIDirPathCtx *dpctx;       /* direct path context */
OCIDirPathColArray *dpca;   /* direct path column array */
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpca,
                       OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                       (size_t)0, (dvoid **)0);
```

A direct path column array handle is freed with OCIHandleFree().

```
error = OCIHandleFree(dpca, OCI_HTYPE_DIRPATH_COLUMN_ARRAY);
```

A direct path function column array handle is allocated in almost the same way:

```
OCIDirPathFuncCtx *dpfnctx;   /* direct path fuction context */
OCIDirPathColArray *dpfnca;   /* direct path function column array */
sword error;
error = OCIHandleAlloc((dvoid *)dpfnctx, (dvoid **)&dpfnca,
                       (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                       (size_t)0, (dvoid **)0);
```

A direct path column array is freed with `OCIHandleFree()`:

```
error = OCIHandleFree(dpfnca, OCI_HTYPE_DIRPATH_FN_COL_ARRAY);
```

Freeing a **OCIDirPathColArray** handle also frees the column array associated with the handle.

Direct Path Stream

This handle is used by the conversion operation, `OCIDirPathColArrayToStream()`, and by the load operation, `OCIDirPathLoadStream()`.

Direct path stream handles is allocated by the client with `OCIHandleAlloc()`. The structure of a **OCIDirPathStream** handle can be thought of as a pair in the form (buffer, buffer length).

A direct path stream is a linear representation of Oracle table data. The conversion operations always append to the end of the stream. Load operations always start from the beginning of the stream. After a stream is completely loaded, the stream must be reset by calling `OCIDirPathStreamReset()`.

The following example shows a direct path stream handle allocated with `OCIHandleAlloc()`. The parent handle is always an **OCIDirPathCtx** handle:

```
OCIDirPathCtx *dpctx;    /* direct path context */
OCIDirPathStream *dpstr; /* direct path stream */
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpstr,
                       OCI_HTYPE_DIRPATH_STREAM, (size_t)0,(dvoid **)0);
```

A direct path stream handle is freed using `OCIHandleFree()`.

```
error = OCIHandleFree(dpstr, OCI_HTYPE_DIRPATH_STREAM);
```

Freeing an **OCIDirPathStream** handle also frees the stream buffer associated with the handle.

Direct Path Interface Functions

The functions listed in this section are used with the direct path load interface.

> **See Also:** Detailed descriptions of each function can be found in

Operations on the direct path context are performed by the functions in Table 12–1, "Direct Path Context Functions".

Table 12–1   Direct Path Context Functions

Function	Purpose
OCIDirPathAbort()	Aborts a direct path operation
OCIDirPathDataSave()	Executes a data savepoint
OCIDirPathFinish()	Commits the loaded data
OCIDirPathFlushRow()	Flushes a partial row from the database server
OCIDirPathLoadStream()	Loads data that has been converted to direct path stream format
OCIDirPathPrepare()	Prepares direct path interface to convert or load rows

Operations on the direct path column array are performed by the functions in Table 12–2, "Direct Path Column Array Functions".

Table 12–2   Direct Path Column Array Functions

Function	Purpose
OCIDirPathColArrayEntryGet()	Gets a specified entry in a column array
OCIDirPathColArrayEntrySet()	Sets a specified entry in a column array to a specific value
OCIDirPathColArrayRowGet()	Gets the base row pointers for a specified row number
OCIDirPathColArrayReset()	Resets the row array state
OCIDirPathColArrayToStream()	Converts from a column array format to a direct path stream format

Operations on the direct path stream are performed by the functions in Table 12–3, "Direct Path Stream Functions".

Table 12–3   Direct Path Stream Functions

Function	Purpose
OCIDirPathStreamReset()	Resets the direct stream state

Limitations and Restrictions of the Direct Path Load Interface

The direct path load interface has the following limitations that are the same as SQL*Loader:

- Triggers are not supported.

- Referential integrity constraints are not supported.

- Clustered tables are not supported.

- Loading of remote objects is not supported.

- LONGs must be specified last.

- SQL strings that return LOBs, objects, or collections are not supported.

- Loading of varray columns is not supported.

- All partitioning columns must come before any LOBs. This is because we need to determine what partition the LOB will go into before we start writing to it.

Direct Path Load Example for Scalar Columns

Data Structures Used in Direct Path Loading Example

The following data structure is used in the example.

```
/* load control structure */
struct loadctl
{
  ub4                nrow_ctl;          /* number of rows in column array */
  ub2                ncol_ctl;        /* number of columns in column array */
  OCIEnv             *envhp_ctl;                    /* environment handle */
  OCIServer          *srvhp_ctl;                         /* server handle */
  OCIError           *errhp_ctl;                          /* error handle */
  OCIError           *errhp2_ctl;                 /* another error handle */
  OCISvcCtx          *svchp_ctl;                       /* service context */
  OCISession         *authp_ctl;                /* authentication context */
  OCIParam           *colLstDesc_ctl;      /* column list parameter handle */
  OCIDirPathCtx      *dpctx_ctl;                    /* direct path context */
  OCIDirPathColArray *dpca_ctl;          /* direct path column array handle */
  OCIDirPathStream   *dpstr_ctl;               /* direct path stream handle */
  ub1                *buf_ctl;    /* pre-alloc'd buffer for out-of-line data */
  ub4                bufsz_ctl;                   /* size of buf_ctl in bytes */
  ub4                bufoff_ctl; /* offset into buf_ctl which is not in use */
  ub4                *otor_ctl;                   /* Offset to Recnum mapping */
```

```
ub1                 *inbuf_ctl;                      /* buffer for input records */
struct pctx          pctx_ctl;                       /* partial field context */
};
```

The header file cdemodp.h from the demo directory defines several structs:

```
#ifndef cdemodp_ORACLE
# define cdemodp_ORACLE

# include <oratypes.h>

# ifndef externdef
#   define externdef
# endif

/* External column attributes */
struct col
{
  text *name_col;                                        /* column name */
  ub2   id_col;                                          /* column load id */
  ub2   exttyp_col;                                      /* external type */
  text *datemask_col;                             /* datemask, if applicable */
  ub1   prec_col;                                 /* precision, if applicable */
  sb1   scale_col;                                   /* scale, if applicable */
  ub2   csid_col;                                      /* character set id */
  ub1   date_col;             /* is column a chrdate or date? 1=TRUE. 0=FALSE */
};

/* Input field descriptor
 * For this example (and simplicity),
 * fields are strictly positional.
 */
struct fld
{
  ub4  begpos_fld;                               /* 1-based beginning position */
  ub4  endpos_fld;                               /* 1-based ending    position */
  ub4  maxlen_fld;                        /* max length for out of line field */
  ub4    flag_fld;
#define FLD_INLINE          0x1
#define FLD_OUTOFLINE       0x2
#define FLD_STRIP_LEAD_BLANK  0x4
#define FLD_STRIP_TRAIL_BLANK 0x8
};

struct tbl
```

```
{
  text        *owner_tbl;                                        /* table owner */
  text        *name_tbl;                                          /* table name */
  text        *subname_tbl;                             /* subname, if applicable */
  ub2          ncol_tbl;                        /* number of columns in col_tbl */
  text        *dfltdatemask_tbl;           /* table level default date mask */
  struct col  *col_tbl;                                    /* column attributes */
  struct fld  *fld_tbl;                                     /* field descriptor */
  ub1          parallel_tbl;                         /* parallel: 1 for true */
  ub1          nolog_tbl;                           /* no logging: 1 for true */
  ub4          xfrsz_tbl;                      /* transfer buffer size in bytes */
};

struct sess                             /* options for a direct path load session */
{
  text        *username_sess;                                        /* user */
  text        *password_sess;                                    /* password */
  text        *inst_sess;                              /* remote instance name */
  text        *outfn_sess;                                 /* output filename */
  ub4          maxreclen_sess;         /* max size of input record in bytes */
};
#endif                                              /* cdemodp_ORACLE */
```

Outline of an Example of a Direct Path Load for Scalar Columns

The following sample code illustrates the use of several of the OCI direct path interfaces. It is not a complete code example.

The *init_load* function performs a direct path load using the direct path API on the table described by *tblp*. The *loadctl* structure given by *ctlp* has an appropriately initialized environment and service context. A connection has been made to the server.

```
STATICF void
init_load(ctlp, tblp)
struct loadctl *ctlp;
struct tbl     *tblp;
{
  struct  col   *colp;
  struct  fld   *fldp;
  sword          ociret;                          /* return code from OCI calls */
  OCIDirPathCtx *dpctx;                                /* direct path context */
  OCIParam      *colDesc;                      /* column parameter descriptor */
  ub1            parmtyp;
```

```
ub1             *timestamp = (ub1 *)0;
ub4              size;
ub4              i;
ub4              pos;

/* allocate and initialize a direct path context */
OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
        OCIHandleAlloc((dvoid *)ctlp->envhp_ctl,
                       (dvoid **)&ctlp->dpctx_ctl,
                       (ub4)OCI_HTYPE_DIRPATH_CTX,
                       (size_t)0, (dvoid **)0));

dpctx = ctlp->dpctx_ctl;                                   /* shorthand */

OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
        OCIAttrSet((dvoid *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX,
                   (dvoid *)tblp->name_tbl,
                   (ub4)strlen((const char *)tblp->name_tbl),
                   (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl));
...
```

Additional attributes, such as OCI_ATTR_SUB_NAME and
OCI_ATTR_SCHEMA_NAME, are also set here. After the attributes have been set,
prepare the load.

```
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
        OCIDirPathPrepare(dpctx, ctlp->svchp_ctl, ctlp->errhp_ctl));
```

Allocate the Column Array and Stream Handles.

Note that the direct path context handle is the parent handle for the column array
and stream handles. Also note that errors are returned with the environment handle
associated with the direct path context.

```
OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
        OCIHandleAlloc((dvoid *)ctlp->dpctx_ctl, (dvoid **)&ctlp->dpca_ctl,
                       (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                       (size_t)0, (dvoid **)0));

OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
        OCIHandleAlloc((dvoid *)ctlp->dpctx_ctl,(dvoid **)&ctlp->dpstr_ctl,
                       (ub4)OCI_HTYPE_DIRPATH_STREAM,
                       (size_t)0, (dvoid **)0));
```

Get Number of Rows and Columns

Get number of rows and columns in the column array just allocated.

```
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
         OCIAttrGet(ctlp->dpca_ctl, (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                    &ctlp->nrow_ctl, 0, OCI_ATTR_NUM_ROWS,
                    ctlp->errhp_ctl));

 OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
         OCIAttrGet(ctlp->dpca_ctl, (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                    &ctlp->ncol_ctl, 0, OCI_ATTR_NUM_COLS,
                    ctlp->errhp_ctl));
```

Set Input Data Fields

Set the input data fields to their corresponding data columns.

```
OCIDirPathColArrayEntrySet(ctlp->dpca_ctl, ctlp->errhp_ctl,
                                    rowoff, colp->id_col,
                                    cval, clen, cflg));
```

Reset Column Array State

Reset column array state in case a previous conversion needed to be continued or a
row is expecting more data.

```
(void) OCIDirPathColArrayReset(ctlp->dpca_ctl, ctlp->errhp_ctl);
```

Reset the Stream State

Reset the stream state to start a new stream. Otherwise, data in the stream is
appended to existing data.

```
(void) OCIDirPathStreamReset(ctlp->dpstr_ctl, ctlp->errhp_ctl);
```

Convert Data to Stream Format

After inputting the data, convert the data in the column array to stream format and
filter out any bad records.

```
ocierr = OCIDirPathColArrayToStream(ctlp->dpca_ctl, ctlp->dpctx_ctl,
                                    ctlp->dpstr_ctl, ctlp->errhp_ctl,
                                    rowcnt, startoff);
```

Load the Stream.

Note that the position in the stream is maintained internally to the stream handle, along with offset information for the column array which produced the stream. When the conversion to stream format is done, the data is appended to the stream. It is the responsibility of the caller to reset the stream when appropriate. On errors, the position is moved to the next row, or the end of the stream if the error occurs on the last row. The next `OCIDirPathLoadStream()` call starts on the next row, if any. If a `OCIDirPathLoadStream()` call is made, and the end of a stream has been reached, OCI_NO_DATA is returned.

```
ocierr = OCIDirPathLoadStream(ctlp->dpctx_ctl, ctlp->dpstr_ctl,
            ctlp->errhp_ctl);
```

Finish the Direct Path Load

```
OCIDirPathFinish(ctlp->dpctx_ctl, ctlp->errhp_ctl);
```

Free the Direct Path Handles

Free all the direct path handles allocated. Note that direct path column array and stream handles are freed before the parent direct path context handle is freed.

```
ociret = OCIHandleFree((dvoid *)ctlp->dpca_ctl,
                          OCI_HTYPE_DIRPATH_COLUMN_ARRAY);
ociret = OCIHandleFree((dvoid *)ctlp->dpstr_ctl,
                          OCI_HTYPE_DIRPATH_STREAM);
ociret = OCIHandleFree((dvoid *)ctlp->dpctx_ctl,
                          OCI_HTYPE_DIRPATH_CTX);
```

Direct Path Loading of Object Types

The use of the direct path function contexts to load various non-scalar types is discussed in this section.

The non-scalar types are:

- nested tables
- object tables (`FINAL` and `NOT FINAL`)
- column objects (`FINAL` and `NOT FINAL`)
- REF columns (`FINAL` and `NOT FINAL`)
- SQL string columns

See Also: Table B–1, "OCI Demonstration Programs" for a listing of the programs demonstrating direct path loading that are available with your Oracle installation.

Direct Path Loading of Nested Tables

Nested tables are stored in a separate table. Using the direct path loading API, a nested table is loaded separately from its parent table with a foreign key, called a SETID, to link the two tables together.

Note:

- Currently, the SETIDs must be user-supplied, and are not system-generated.

- When loading the parent and child tables separately, it is possible that orphaned children can be created when the rows are inserted in the child table, but the corresponding parent row is not inserted in the parent table. It is also possible to insert a parent row in the parent table, but that the child rows are not inserted in the child table and therefore it will have missing children.

Describing a Nested Table Column and Its Nested Table

Note: Steps that are different from loading scalar data are in italics.

Loading the parent table with a nested table column is a separate action from loading the child nested table.

- *To load the parent table with a nested-table column:*

  Describe the parent table and its columns as usual, except:

  When describing the nested-table column, this is the column that stores the SETIDs. Its external data type is SQLT_CHR if the SETIDs in the data file are in characters, SQLT_BIN if binary.

- *To load the nested table (child):*

  1. Describe the nested table and its columns as usual.

2. *The SETID column is required.*

a. *Set its OCI_ATTR_NAME using a dummy name (for example "setid") because the API does not expect you to know its system name.*

b. *Set the column attribute with OCI_ATTR_DIRPATH_SID to indicate that this is a SETID column:*

```
ub1 flg = 1;
OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM, (dvoid *)&flg,
          (ub4)0, (ub4)OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
```

Direct Path Loading of Column Objects

A column object is a table column that is defined as an object. Currently only the default constructor, which consists of all of the constituent attributes, is supported.

Describing a Column Object

To describe a column object and its object attributes, use a direct path function context. Describing a column object requires setting its object constructor. Describing object attributes is similar to describing a list of scalar columns.

To describe a column object:

Note:

- Nested column objects are supported.

- The steps here are similar to that of describing a list of scalar columns to be loaded for a table. Steps that are new are in italics.

---

1. Allocate a parameter handle on the column object with OCI_DTYPE_PARAM. This parameter handle is used to set the column's external attributes.

2. Set the column name and its other external column attributes (for example, maximum data size, precision, scale).

3. *Set the external type as SQLT_NTY (named type) with OCI_ATTR_DATA_TYPE.*

4. *Allocate a direct path function context handle. This context will be used to describe the column's object type and attributes:*

```
OCIDirPathCtx *dpctx;        /* direct path context */
OCIDirPathFuncCtx *dpfnctx  /* direct path function context */;
sword error;
```

```
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpfnctx,
                        OCI_HTYPE_DIRPATH_FN_CTX, (size_t)0, (dvoid **)0);
```

5. Set the column's object type name (for example, "Employee") with OCI_ATTR_NAME in the function context:

```
text *obj_type;   /* column object's object type */
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)obj_type, (ub4)strlen((const char *)obj_type),
           (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl);
```

6. Set the expression type, OCI_ATTR_DIRPATH_EXPR_TYPE, to be OCI_DIRPATH_EXPR_OBJ_CONSTR. This indicates that the expression set with OCI_ATTR_NAME will be used as the default object constructor:

```
ub1 expr_type = OCI_DIRPATH_EXPR_OBJ_CONSTR;
OCIAttrSet((dvoid *)dpfnctx,(ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)&expr_type,(ub4)0, (ub4)OCI_ATTR_DIRPATH_EXPR_TYPE,
           ctlp->errhp_ctl);
```

7. Set the number of columns or object attributes that will be loaded for this column object using OCI_ATTR_NUM_COLS.

8. Get the column/attribute parameter list for the function context OCIDirPathFuncCtx.

9. For each object attribute:

- Get the column descriptor for the object attribute with OCI_DTYPE_PARAM.

- Set the attribute's column name with OCI_ATTR_NAME.

- Set the external column type (the type of the data that will be passed to the direct path API) with OCI_ATTR_DATA_TYPE.

- Set any other external column attributes (maximum data size, precision, scale, etc.)

- *If this attribute column is a column object, then do steps 3-10 for its object attributes.*

- Free the handle to the column descriptor.

10. Set the function context OCIDirPathFuncCtx that was created in step 4 into the parent column object's parameter handle with OCI_ATTR_DIRPATH_FN_CTX.

Allocating the Array Column for the Column Object

When loading a column object, the data for its object attributes will be loaded into a separate column array created just for that object. A child column array is allocated for each column object, whether it is nested or not. Each row of object attributes in the child column array maps back to the corresponding non-NULL row of its parent column object in the parent column array.

Use the column object's direct path function context handle and column array type OCI_HTYPE_DIRPATH_FN_COL_ARRAY.

To allocate a child column array for a column object:

```
OCIDirPathFuncCtx *dpfnctx;    /* direct path function context */
OCIDirPathColArray *dpfnca;    /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
               (dvoid **)&dpfnca, (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
               (size_t)0, (dvoid **)0);
```

Loading Column Object Data into the Column Array

If a column is scalar, its value is set in the column array by passing the address of its value to OCIDirPathColArrayEntrySet(). And if a column is an object, the address of its child column array handle is passed instead. The child column array will contain the data of the object attributes.

To load data into an column object:

Note: Steps that are different from loading scalar data are in italics.

(Start.) For each column object:

1. *If the column is non-NULL:*

   a. *For each of its object attribute columns:*

   ■ *If an object attribute is a nested column object, then go to (Start.) and do this entire procedure recursively.*

   ■ *Set the data in the child column array using* OCIDirPathColArrayEntrySet().

   b. *Set the column object's data in the column array by passing the address of its child column array handle to* OCIDirPathColArrayEntrySet().

2. Else if the column is NULL:

    ■   Set the column object's data in the column array by passing a NULL address for the data, a length of 0, and an OCI_DIRPATH_COL_NULL flag to `OCIDirPathColArrayEntrySet()`.

Direct Path Loading of SQL String Columns

A column value can be computed by a SQL string. SQL strings can be used for scalar column types. SQL strings cannot be used for object types, but can be used for object attributes of scalar column types. They cannot be used for nested tables and LONGs.

A SQL expression is represented to the direct path API using the **OCIDirPathFuncCtx**. Its OCI_ATTR_NAME value will be the SQL string with the parameter list of the named bind variables for the expression.

A SQL string example is:

```
substr(substr(:string, :offset, :length), :offset, :length)
```

Things to note about this example are:

■   SQL expressions can be nested.

■   Bind variable names can be repeated within the expression.

Describing a SQL String Column

> **Note:**   Steps that are different from loading scalar data are in italics.

1. Allocate a parameter handle on the SQL string column with OCI_DTYPE_PARAM. This parameter handle is used to set the column's external attributes.

2. Set the column name and its other external column attributes (for example, maximum data size, precision, scale).

3. *Set the SQL string column's external type as SQLT_NTY with OCI_ATTR_DATA_TYPE.*

4. *Allocate a direct path function context handle. This context will be used to describe the arguments of the SQL string.*

```
OCIDirPathCtx *dpctx;        /* direct path context */
OCIDirPathFuncCtx *dpfnctx  /* direct path function context */;
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpfnctx,
                       OCI_HTYPE_DIRPATH_FN_CTX, (size_t)0, (dvoid **)0);
```

5. *Set the column's SQL string in OCI_ATTR_NAME in the function context.*

```
text *sql_str;   /* column's SQL  string expression */
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)sql_str, (ub4)strlen((const char *)sql_str),
           (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl);
```

6. *Set the expression type, OCI_ATTR_DIRPATH_EXPR_TYPE, to be*
   OCI_DIRPATH_EXPR_SQL. This indicates that the expression set with
   OCI_ATTR_NAME will be used as the SQL string to derive the value from.

```
ub1 expr_type = OCI_DIRPATH_EXPR_SQL;
OCIAttrSet((dvoid *)dpfnctx,(ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)&expr_type,(ub4)0, (ub4)OCI_ATTR_DIRPATH_EXPR_TYPE,
           ctlp->errhp_ctl);
```

7. *Set the number of arguments that will be passed to the SQL string with*
   OCI_ATTR_NUM_COLS.

8. *Get the column/attribute parameter list for the function context.*

9. *For each SQL string argument:*

   ■ Get the column descriptor for the object attribute with
     OCI_DTYPE_PARAM.

   ■ Set the attribute's column name with OCI_ATTR_NAME.

     The order in which the SQL string arguments are defined does not matter. The
     order does not have to match the order used in the SQL string.

     There is a naming convention for SQL string arguments.

   ■ *The argument names must match the bind variable names used in the SQL string*
     in content but not in case. For example, if the SQL string is
     "substr(:INPUT_STRING, 3, 5)", then it is acceptable if you give the argument
     name as "input_string".

   ■ *If an argument is used multiple times in an SQL string, declaring it once and*
     counting it as one argument only is correct.

- Set the external column type (the type of the data that will be passed to the direct path API) with OCI_ATTR_DATA_TYPE.

- Set any other external column attributes (maximum data size, precision, scale, etc.).

- Free the handle to the column descriptor.

10. *Set the function context OCIDirPathFuncCtx that was created in step 4 into the parent column object's parameter handle with OCI_ATTR_DIRPATH_FN_CTX.*

Allocating the Column Array for SQL String Columns

When loading a SQL string column, the data for its arguments will be loaded into a separate column array created just for that SQL string column. A child column array is allocated for each SQL string column. Each row of arguments in the child column array maps back to the corresponding non-NULL row of its parent SQL string column in the parent column array.

To allocate a child column array for a SQL string column:

```
OCIDirPathFuncCtx *dpfnctx;    /* direct path function context */
OCIDirPathColArray *dpfnca;    /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
               (dvoid **)&dpfnca, (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
               (size_t)0, (dvoid **)0);
```

Loading the SQL String Data into the Column Array

If a column is scalar, its value would be set in the column array by passing the address of its value to `OCIDirPathColArrayEntrySet()`. If a column is of a SQL string type, the address of its child column array handle would be passed instead. The child column array would contain the SQL string's argument data.

To load data into a SQL string column:

> **Note:** Steps that are different from loading scalar data are in italics.

For each SQL string column:

1. *If the column is non-NULL:*

   a. *For each of its function argument columns:*

Set the data in the child column array using `OCIDirPathColArrayEntrySet().`

b. *Set the SQL string column's data into the column array by passing the address of its child column array handle to* `OCIDirPathColArrayEntrySet().`

2. Else if the column is NULL:

Set the SQL string column data into the column array by passing a NULL address for the data, a length of 0, and an OCI_DIRPATH_COL_NULL flag to `OCIDirPathColArrayEntrySet().`

This process is similar to that for column objects.

Direct Path Loading of REF Columns

The REF type is a pointer, or reference, to a row object in an object table.

Describing the REF Column

Describing the arguments to a REF column is similar to describing the list of columns to be loaded for a table.

> **Note:** A REF column can be a top-table-level column or nested as an object attribute to a column object.
>
> Steps that are different from loading scalar data are in italics.

1. Get a parameter handle on the REF column with OCI_DTYPE_PARAM. This parameter handle is used to set the column's external attributes.

2. Set the column name and its other external column attributes (for example, maximum data size, precision, scale).

3. *Set the REF column's external type as SQLT_REF with OCI_ATTR_DATA_TYPE.*

4. *Allocate a direct path function context handle. This context is used to describe the REF column's arguments.*

```
OCIDirPathCtx *dpctx;        /* direct path context */
OCIDirPathFuncCtx *dpfnctx  /* direct path function context */;
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpfnctx,
                        OCI_HTYPE_DIRPATH_FN_CTX, (size_t)0, (dvoid **)0);
```

5. *OPTIONAL: Set the REF column's table name in OCI_ATTR_NAME in the function context. See the next step for more details.*

```
text *ref_tbl;      /* column's reference table */
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)ref_tbl, (ub4)strlen((const char *)ref_tbl),
           (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl);
```

6. *OPTIONAL: Set the expression type, OCI_ATTR_DIRPATH_EXPR_TYPE, to be OCI_DIRPATH_EXPR_REF_TBLNAME. Set this only if step 5 was done. This indicates that the expression set with OCI_ATTR_NAME will be used as the object table to reference row objects from.*

    This parameter is optional. The behavior for this parameter varies for the REF type.

    a. *Unscoped REF columns (unscoped, system-OID-based):*

        If not set, then by the definition of an "unscoped" REF column, this REF column is required to have a reference table name as its argument for every data row.

        If set, this REF column can only refer to row objects from this specified object table for the duration of the load. And the REF column is not allowed to have a reference table name as its argument. (The direct path API is providing this parameter as a short cut to users who will be loading to an unscoped REF column that refers to the same reference object table during the entire load.)

    b. *Scoped REF columns (scoped, system-OID-based and primary-key-based):*

        If not set, the direct path API will use the reference table specified in the schema.

        If set, the reference table name must match the object table specified in the schema for this scoped REF column. An error occurs if the table names do not match.

        Whether this parameter is set or not, it does not matter to the API whether this reference table name is in the data row or not. If the name is in the data row, it has to match the table name specified in the schema. If it is not in the data row, the API will use the reference table specified in the schema.

7. *Set the number of REF arguments that will be used to reference a row object. with OCI_ATTR_NUM_COLS.*

    The number of arguments required varies for the REF column type. This number is derived from step 6 above.

    a. *Unscoped REF columns (unscoped, system-OID-based REF columns):*

        One if OCI_DIRPATH_EXPR_REF_TBLNAME is used. None for the reference table name, and one for the OID value.

        Two if OCI_DIRPATH_EXPR_REF_TBLNAME is not used. One for the reference table name, and one for the OID value.

    b. *Scoped REF columns (scoped, system-OID-based and primary-key-based):*

        N or N+1 are acceptable, where N is the number of columns making up the object id, regardless if OCI_DIRPATH_EXPR_REF_TBLNAME is used or not. Minimum is N if the reference table name is not in the data row. It's N+1 if the reference table name is in the data row. Note: If the REF is system-OID-based, then N is one. If the REF is primary-key-based, then N is the number of component columns that make up the primary key. If the reference table name is in the data row, then add one to N.

Note: To simplify the error message if you were to pass in a number of REF arguments other than N or N+1, the error message will say that it found so-and-so number of arguments when it expects N. Although N+1 is not stated in the message, N+1 is acceptable (even though the reference table name is not needed) and will not invoke an error message.

---

8. *Get the column/attribute parameter list for the function context.*

9. *For each REF argument or attribute:*

    a. Get the column descriptor for the REF argument using OCI_DTYPE_PARAM.

    b. Set the attribute's column name using OCI_ATTR_NAME.

    The order of the REF arguments given matter.

        The reference table name comes first, if given.

        The object id, whether it is system-generated or primary-key-based, comes next.

    There is a naming convention for the REF arguments.

        Since the reference table name is not a table column, you can use any dummy names for its column name, such as "ref-tbl".

For a system-generated OID column, you can use any dummy names for its column name, such as. "sys-OID".

For a primary-key-based object id, list all the primary-key columns to load into. There is no need to create a dummy name for OID. The component column names, if given (see short cut note below), can be given in any order.

Do not set the attribute column name(s) for the object id if you want to use the short cut.

Short cut. If loading a system-OID-based REF column, do not set the column name with a name. The API will figure it out. But you will still have to set other column attributes, such as external data type.

If loading a primary-key REF column and its primary key consists of multiple columns, the short cut is not to set their column names. But you will still have to set other column attributes, such as external data type.

> **Note:** If the component column names are NULL, then the API code determines the column names in the position or order in which they were defined for the primary key. So, when you set column attributes other than the name, make sure the attributes are set for the component columns in the correct order.

c. Set the external column type (the type of the data that will be passed to the direct path API) using OCI_ATTR_DATA_TYPE.

d. Set any other external column attributes (max data size, precision, scale, etc.).

e. Free the handle to the column descriptor.

f. *Set the function context OCIDirPathFuncCtx that was created in step 4 in the parent column object's parameter handle using OCI_ATTR_DIRPATH_FN_CTX.*

Allocating the Column Array for a REF Column

To allocate a child column array for a REF column:

```
OCIDirPathFuncCtx *dpfnctx;   /* direct path function context */
OCIDirPathColArray *dpfnca;   /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
               (dvoid **)&dpfnca, (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
               (size_t)0, (dvoid **)0);
```

Loading the REF Data Into The Column Array

If a column is scalar, its value would be set in the column array by passing the address of its value to `OCIDirPathColArrayEntrySet()`. If a column is a REF, the address of its child column array handle would be passed instead. The child column array will contain the REF arguments' data.

To load data into a REF column:

> **Note:** Steps that are different from loading scalar data are in italics.

For each REF column:

1. *If the column is non-NULL:*

    a. *For each of its REF argument columns:*

    > *Set its data in the child column array using `OCIDirPathColArrayEntrySet()`.*

    b. *Set the REF column's data into the column array by passing the address of its child column array handle to `OCIDirPathColArrayEntrySet()`.*

2. Else if the column is NULL:

    > Set the REF column's data into the column array by passing a NULL address for the data, length of 0, and a OCI_DIRPATH_COL_NULL flag to OCIDirPathColArrayEntrySet().

NOT FINAL Object and REF Columns

An example of an inheritance hierarchy is shown below. In this example, `Person` is at the top of the hierarchy. It has two sub-types, `Employee` and `Student`. `ParttimeEmployee` is a sub-type of `Employee`. Therefore, the types which can be stored in a `Person` column are shown in this diagram:

Inheritance Hierarchy Diagram

```
                    Person (Name, Address)
                       |           |
                       |           |
                       |           |
        Student(Units, GPA)     Employee (Manager, Deptid)
                                   |
                                   |
                                   |
                                ParttimeEmployee (Hours)
```
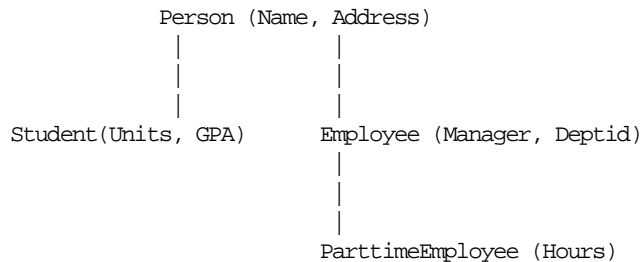
When loading a table which contains a column of type `Person`, the actual set of types could include any of these four: the `NOT FINAL` type `Person`, and its three sub-types: `Student`, `Employee`, and `ParttimeEmployee`. The direct path API only supports the loading of one fixed, derived type to this `NOT FINAL` column for the duration of this load. Thus, the API needs to know which one of these types will be loaded, the attributes to load for this type, and the function used to create this type.

Note:

- A `NOT FINAL` column in a table can only store one fixed, derived type for the duration of the load.

- When describing and loading a derived type, all of the attributes for that type that are to be loaded must be specified. Think of a subtype as a flattened representation of all the object attributes that are unique to this type plus all the attributes of its ancestors. Therefore, any of these attribute columns that are to be loaded into will have to be described and counted.

- For example, if loading to all columns in `ParttimeEmployee`, there are 5 object attributes to load into: `Name`, `Address`, `Manager`, `Deptid`, and `Hours`.

Describing a Fixed, Derived Type to be Loaded

To describe `NOT FINAL` or substitutable object and REF columns of a fixed, derived type:

> **Note:** The steps describing a `NOT FINAL` column of a fixed, derived type is similar to describing its `FINAL` counterpart.

To describe a `NOT FINAL` column of type X (where X is object or REF), refer to previous sections to describe a `FINAL` column of this type. Because the derived type (could be a supertype or a subtype) is fixed for the duration of the load, the client interface for describing a `NOT FINAL` column is the same as for a `FINAL` column.

A subtype can be thought of as a flattened representation of all the object attributes that are unique to this type plus all the attributes of its ancestors. Therefore, any of these attribute columns that are to be loaded into will have to be described and counted.

Allocating the Column Array

This is the same as for a `FINAL` column of the same type.

Loading the Data into the Column Array

This is the same as for a `FINAL` column of the same type.

Direct Path Loading of Object Tables

An object table is a table in which each row is an object (or row object). Each column in the table is an object attribute.

Describing an Object Table

Describing an object table is very similar to describing a non-object table. Each object attribute is a column in the table. The only difference is that you may need to describe the OID, which could be system-generated, user-generated, or primary-key based.

To describe an object table:

> **Note:** Steps that are different from loading a non-object table are in italics.

For each object attribute column:

Describe each object attribute column as it needs to be described, depending on its type (for example, NUMBER, REF):

For the object table OID:

1. *If the object id is system-generated:*

   Nothing extra to do. The system will generate OIDs for each row object.

2. *If the object id is user-generated:*

   a. *Use a dummy name to represent the column name for the OID (for example, "cust_oid").*

   b. *Set the OID column attribute with OCI_ATTR_DIRPATH_OID.*

3. *If the object id is primary-key-based:*

   a. *All of the primary-key columns making up the OID must be loaded.*

   b. *Do not set OCI_ATTR_DIRPATH_OID, because no OID column with a dummy name was created.*

Allocating the Column Array for the Object Table

This is the same as allocating a column array for a non-object table.

```
OCIDirPathCtx *dpctx;        /* direct path context */
OCIDirPathColArray *dpca;    /* direct path column array */
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpca,
                       OCI_HTYPE_DIRPATH_COLUMN_ARRAY, 0, (dvoid **)0);
```

Loading Data into the Column Array

This is the same as loading data into a non-object table.

Direct Path Loading a NOT FINAL Object Table

A NOT FINAL object table supports inheritance and a FINAL object table cannot.

Describing a NOT FINAL Object Table

Describing a NOT FINAL object table of a fixed derived type is very similar to describing a FINAL object table.

To describe a NOT FINAL object table of a fixed type:

> **Note:** Steps that are different from loading a FINAL object table are in italics.

1. *Set the object table's object type in the direct path context with OCI_ATTR_DIRPATH_OBJ_CONSTR. This indicates that the object type, whether it is a supertype or a derived type, will be used as the default object constructor when loading to this table for the duration of the load.*

```
text *obj_type;            /* the object type to load into this NOT FINAL */
                           /* object table */
OCIAttrSet((dvoid *)dpctx,
           (ub4)OCI_HTYPE_DIRPATH_CTX,
           (dvoid *) obj_type,
           (ub4)strlen((const char *) obj_type),
           (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

2. For each of the object attribute columns to be loaded, describe them according to their datatypes. Describe the object id, if needed. This is the same as describing a FINAL object table.

Allocating the Column Array for the NOT FINAL Object Table

This is the same as for a FINAL object table.

Direct Path Loading in Pieces

To support loading data that will not all fit in memory at one time, use loading in pieces.

The direct path API already supports loading LONGs and LOBs incrementally. This is accomplished through the following sequence of steps:

1. Set the first piece into the column array using `OCIDirPathColArrayEntrySet()` and passing in the `OCI_DIRPATH_COL_PARTIAL` flag to indicate that all the data for this column has not been loaded yet.

2. Convert the column array to a stream.

3. Load the stream.

4. Set the next piece of that data into the column array. If it is not complete, set the partial flag and go back to step 2. If it is complete, then set the OCI_DIRPATH_COL_COMPLETE flag and continue on to the next column.

This approach is essentially the same for dealing with large attributes for column objects and large arguments for SQL string types.

> **Note:** Collections are not loaded in pieces, as such. Nested tables are loaded separately and are loaded like a top-level table. Nested tables can be loaded incrementally and can have columns which are loaded in pieces. Therefore, do not set the OCI_DIRPATH_COL_PARTIAL flag for the column containing the collection.

Loading Object Types in Pieces

Objects are loaded into a separate column array from the parent table which contains them. Therefore, when they need to be loaded in pieces you must set the elements in the child column array up to and including the pieced element.

The general steps are:

1. For the pieced element, set the OCI_DIRPATH_COL_PARTIAL flag.

2. Set the child column array handle into the parent column array and mark that entry with the OCI_DIRPATH_COL_PARTIAL flag as well.

3. At this point, convert the parent column array to a stream. This will convert the child column array as well.

4. Then load the stream.

5. Go back to step one and continue loading the remaining data for that element until it is complete.

Here are some rules about loading in pieces:

- There can only be one partial element at a time at any level. Once one partial element is marked complete then another one at that level could be partial.

- If an element is partial and it is not top-level, then all of its ancestors up the containment hierarchy must be marked partial as well.

- If there are multiple levels of nesting, it is necessary to go up to a level where the data can be converted into a stream. This will be a top-level table.

Direct Path Context Handles and Attributes for Object Types

The following discussion gives the supplemental details of the handles and attributes that are listed in the appendix A.

Direct Path Context Attributes

OCI_ATTR_DIRPATH_OBJ_CONSTR

Indicates the object type to load into a NOT FINAL object table.

```
text *obj_type;              /* the object type to load into this NOT FINAL */
                             /* object table */
OCIAttrSet((dvoid *)dpctx,
           (ub4)OCI_HTYPE_DIRPATH_CTX,
           (dvoid *) obj_type,
           (ub4)strlen((const char *) obj_type),
           (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

Direct Path Function Context and Attributes

Here is a summary of the attributes for function context handles.

> **See Also:** "Direct Path Context Handle (OCIDirPathCtx) Attributes" on page A-58

OCI_ATTR_DIRPATH_OBJ_CONSTR

Indicates the object type to load into a substitutable object table.

```
text *obj_type; /* stores an object type name */
OCIAttrSet((dvoid *)dpctx,
                       (ub4)OCI_HTYPE_DIRPATH_CTX,
                       (dvoid *) obj_type,
                       (ub4)strlen((const char *) obj_type),
                       (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

OCI_ATTR_NAME

When a function context is created, set OCI_ATTR_NAME equal to the expression that describes the non-scalar column. Then set an OCI attribute to indicate the type of the expression. The expression type varies as follows:

1. Column objects:

    a. This required expression is the object type name. The object type will be used as the default object constructor.

    b. Set the expression type OCI_ATTR_DIRPATH_EXPR_TYPE to OCI_DIRPATH_EXPR_OBJ_CONSTR to indicate this expression is an object type name.

2. REF columns:

    a. This optional expression is the reference table name. This table is the object table from which the REF column will be referencing row objects.

    b. Set the expression type OCI_ATTR_DIRPATH_EXPR_TYPE to OCI_DIRPATH_EXPR_REF_TBLNAME to indicate this expression is a reference object table.

    c. The behavior for this parameter, set or not set, varies for each REF type.

    ■ Unscoped REF columns (unscoped, system-OID-based):

        If not set, then by the definition of an "unscoped" REF column, this REF column is required to have a reference table name as its argument for every data row.

        If set, this REF column can only refer to row objects from this specified object table for the duration of the load. And the REF column is not allowed to have a reference table name as its argument. (Direct path API is providing this parameter as a short cut to the users who will be loading to an unscoped REF column that refers to the same reference object table during the entire load.)

    ■ Scoped REF columns (scoped, system-OID-based and primary-key-based):

        If not set, the direct path API will use the reference table specified in the schema.

        If set, the reference table name must match the object table specified in the schema for this scoped REF column. An error occurs if the table names do not match.

        Whether this parameter is set or not, it will not matter to the API whether this reference table name is in the data row or not. If the name is in the data row, it has to match the table name specified in the schema. If it is not in the data row, the API will use the reference table defined in the schema.

3. SQL string columns:

This mandatory expression contains a SQL string to derive the value that will be stored in the column.

Set the expression type OCI_ATTR_DIRPATH_EXPR_TYPE to OCI_DIRPATH_EXPR_SQL to indicate that this expression is a SQL string.

OCI_ATTR_DIRPATH_EXPR_TYPE

This attribute is used to indicate the type of the expression specified in OCI_ATTR_NAME for the non-scalar column's function context.

If OCI_ATTR_NAME is set, then OCI_ATTR_DIRPATH_EXPR_TYPE is required.

The possible values for OCI_ATTR_DIRPATH_EXPR_TYPE are:

1. OCI_DIRPATH_EXPR_OBJ_CONSTR:

   - Indicates that the expression is an object type name and will be used as the default object constructor for a column object.

   - Required for column objects.

2. OCI_DIRPATH_EXPR_REF_TBLNAME:

   - Indicates that the expression is a reference object table name. This table is the object table from which the REF column will be referencing row objects.

   - Optional for REF columns.

3. OCI_DIRPATH_EXPR_SQL:

   - Indicates that the expression is a SQL string, which is executed to derive a value to be stored in the column.

   - Required for SQL string columns.

The following pseudocode example illustrates the above rules:

```
OCIDirPathFuncCtx  *dpfnctx; /* function context for this non-scalar column */

ub1 exprtype; /* expression type */

if (column type is an object) then exprtype = OCI_DIRPATH_EXPR_OBJ_CONSTR;

if (column_type is a REF && function context name exists) then exprtype =
OCI_DIRPATH_EXPR_REF_TBLNAME;

if (column_type is a SQL string) then exprtype = OCI_DIRPATH_EXPR_SQL;
```

```
OCIAttrSet((dvoid *)(dpfnctx),(ub4)OCI_HTYPE_DIRPATH_FN_CTX,(dvoid *)
&exprtype,(ub4) 0, (ub4)OCI_ATTR_DIRPATH_EXPR_TYPE, ctlp->errhp_ctl);
```

OCI_ATTR_NUM_COLS

This attribute describes the number of attributes or arguments that will be loaded or processed for a non-scalar column. This parameter must be set before the column list can be retrieved.

1. Column objects:

   The number of object attribute columns to be loaded for this column object.

2. SQL string columns:

   a. The number of arguments to be passed to the SQL string.

   b. If an argument is used multiple times in the function, counting it as one is correct.

3. REF columns:

   a. The number of REF arguments to identify the row object the REF column should point to.

   b. The number of arguments required varies for the REF column type:

   ▪ Unscoped REF columns (unscoped, system-OID-based REF columns):

      If OCI_DIRPATH_EXPR_REF_TBLNAME is used. None for the reference table name, and one for the OID value. (Only the OID values will be in the data rows.)

      If OCI_DIRPATH_EXPR_REF_TBLNAME is not used. One for the reference table name, and one for the OID value. (Both the reference table names and the OID values will be in the data rows.)

   ▪ Scoped REF columns (scoped, system-OID-based and primary-key-based):

      N or N+1 are acceptable, where N is the number of columns making up the object id, regardless if OCI_DIRPATH_EXPR_REF_TBLNAME is used or not. The minimum is N if the reference table name is not in the data row. Use N+1 if the reference table name is in the data row.

      If the REF is system-OID-based, then N is one. If the REF is primary-key-based, then N is the number of component columns that make up the primary key. If the reference table name is in the data row, then add one to N.

> **Note:** To simplify the error message if you pass in a number of
> REF arguments other than N or N+1, the error message will say
> that it found so-and-so number of arguments when it expects N.
> Although N+1 is not stated in the message, N+1 is acceptable (even
> though the reference table name is not needed) and will not invoke
> an error message.

OCI_ATTR_NUM_ROWS

This attribute, when used for a OCI_HTYPE_DIRPATH_FN_CTX (function
context), is retrievable only, and cannot be set by the user. You can only use this
attribute in `OCIAttrGet()` and not `OCIAttrSet()`. When called with
`OCIAttrGet()`, the number of rows loaded so far is returned.

However, the attribute OCI_ATTR_NUM_ROWS, when used for a
OCI_HTYPE_DIRPATH_CTX (table-level context), can be set and can be retrieved
by the user.

Calling `OCIAttrSet()` with OCI_ATTR_NUM_ROWS and
OCI_HTYPE_DIRPATH_CTX sets the number of rows to be allocated for the
table-level column array. If not set, the direct path API code will derive a
"reasonable" number based on the maximum record size and the transfer buffer
size. To see how many rows were allocated, call `OCIAttrGet()` with
OCI_ATTR_NUM_ROWS on OCI_HTYPE_DIRPATH_COLUMN_ARRAY for a
table-level column array, and with OCI_HTYPE_DIRPATH_FN_COL_ARRAY for a
function column array.

Calling `OCIAttrGet()` with OCI_ATTR_NUM_ROWS and
OCI_HTYPE_DIRPATH_CTX returns the number of rows loaded so far.

This attribute cannot be set by the user for a function context. You are not allowed
to specify the number of rows desired in a function column array through
OCI_ATTR_NUM_ROWS with OCIAttrSet() because then all function column
arrays will have the same number of rows as the table-level column array. Thus this
attribute can only be set for a table-level context and not for a function context.

Direct Path Column Parameter Attributes

When describing an object, SQL string, or REF column, one of its column attributes
is a function context.

If a column is an object, then its function context describes its object type and object attributes. If a SQL string, the expression to be called. If REF, its reference table name and row object identifiers.

When setting a function context as a column attribute, OCI_ATTR_DIRPATH_FN_CTX is used in `OCIAttrSet()`:

```
OCIParam *colDesc;          /* column parameter descriptor */
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
OCIError *errhp;            /* error handle */

OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
           (dvoid *)(dpfnctx), (ub4)0,
           (ub4)OCI_ATTR_DIRPATH_FN_CTX, errhp);
```

Attributes for column parameter context handles follow.

> **See Also:** "Direct Path Column Parameter Attributes" on page A-66

OCI_ATTR_NAME

The naming conventions when loading nested tables, object tables, SQL string columns, and REF columns are described in the following paragraphs.

In general, a dummy column name is used if are loading data into a column that is a system column with a system name that you are not aware of (for example, an object table's system-generated object id (OID) column or a nested table's SETID (SID) column) or if a column is an argument that doesn't have a database table column (for example, SQL string and REF arguments).

If the column is a database table column, but a dummy name was used, then a column attribute has to be set so that the function can identify the column even though it's not under the name known to the database.

The naming rules are:

1. Child nested tables's SETID (SID) column:

   The SETID column is required. Set its OCI_ATTR_NAME using a dummy name, because the API doesn't expect the user to know its system name. Then set the column attribute with OCI_ATTR_DIRPATH_SID to indicate that this is a SID column.

2. Object table's object id (OID) column:

   An object id is required if:

    a. If the object id is system-generated:

    Use a dummy name as its column name (for example, "cust_oid").

    Set its column attribute with OCI_ATTR_DIRPATH_OID. So if you have multiple columns with dummy names, you know which one represents the system-generated OID.

    b. If the object id is primary-key-based:

    You cannot use a dummy name as its column name. Therefore, you do not need to set its column attribute with OCI_ATTR_DIRPATH_OID.

3. SQL string argument:

Set the attribute's column name with OCI_ATTR_NAME.

The order of the SQL string arguments given does not matter. The order does not have to match the order used in the SQL string.

There is a naming convention for SQL string arguments.

    a. The argument names must match the bind variable names used in the SQL string in content but not in case. For example, if the SQL string is `substr(:INPUT_STRING, 3, 5)`, then you can give the argument name as "input_string".

    b. If an argument is used multiple times in an SQL string, then you can declare it once and count it as only one argument.

4. REF argument:

    a. Set the attribute's column name using OCI_ATTR_NAME.

The order of the REF arguments does matter.

- The reference table name comes first, if given.

- The object id, whether it is system-generated or primary-key-based, comes next.

    b. There is a naming convention for the REF arguments.

- For the reference table name argument, use any dummy names for its column name, for example, "ref-tbl".

- For the system-generated OID argument, use any dummy names for its column name, such as "sys-OID". Note: Since this column is used as an argument and not as a column to load into, do not set this column with OCI_ATTR_DIRPATH_OID.

- For a primary-key-based object id, list all the primary-key columns to load into. There is no need to create a dummy name for OID. The component column names, if given (see step for short cut below), can be given in any order.

c. Do not set the attribute column name(s) for the object id if you want to use the short cut.

- **Short cut.** If loading a system-OID-based REF column, do not set the column name with a name. The API will figure it out. But you still have to set other column attributes, such as external data type.

- If loading a primary-key REF column and its primary key consists of multiple columns, the short cut is not to set their column names. But user will still have to set other column attributes, such as external data type.

Note: If the component column names are NULL, then the API code determines the column names in the position or order in which they were defined for the primary key. So, when you set column attributes other than the name, make sure the attributes are set for the component columns in the correct order.

OCI_ATTR_DIRPATH_SID

Indicates that a column is a nested table's SETID column. Required if loading to a nested table.

```
ub1 flg = 1;
OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM, (dvoid *)&flg, (ub4)0,
          (ub4)OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
```

OCI_ATTR_DIRPATH_OID

Indicates that a column is an object table's object id column.

```
ub1 flg = 1;
OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM, (dvoid *)&flg, (ub4)0,
          (ub4)OCI_ATTR_DIRPATH_OID, ctlp->errhp_ctl);
```

Direct Path Function Column Array Handle for Non-scalar Columns

See Also: "Direct Path and Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes" on page A-63

The handle type OCI_HTYPE_DIRPATH_FN_COL_ARRAY is used if the column is an object, SQL string, or REF. The structure **OCIDirPathColArray** is the same for both scalar and non-scalar columns.

To allocate a child column array for a function context:

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
OCIDirPathColArray *dpfnca; /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
               (dvoid **)&dpfnca,
               (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
               (size_t)0, (dvoid **)0);
```

OCI_ATTR_NUM_ROWS Attribute

This attribute, when used for a OCI_HTYPE_DIRPATH_FN_COL_ARRAY (function column array), is retrievable only, and cannot be set by the user. When called with the function `OCIAttrGet()`, the number of rows allocated for the function column array is returned.

13

Object Cache and Object Navigation

This chapter introduces OCI's facility for working with objects in an Oracle database server. It also discusses OCI's object navigational function calls. This chapter includes the following sections:

- Overview of Using Objects in OCI Programs
- The Object Cache and Memory Management
- Object Navigation
- OCI Navigational Functions
- Type Evolution and the Object Cache

Overview of Using Objects in OCI Programs

This chapter is broken down into several main sections that discuss the basic concepts involved in writing OCI applications to manipulate Oracle objects. The chapter also covers the OCI navigational function calls.

The following specific sections are included:

- **The Object Cache and Memory Management** - This section discusses OCI object programming in more detail, including more sophisticated options.

- **Object Navigation** - This section discusses basic object navigation using OCI.

- **OCI Navigational Functions** - This section introduces the OCI functions that enable an application to navigate through a graph of objects.

> **See Also:** Complete descriptions of the OCI navigational functions can be found in Chapter 17, "OCI Navigational and Type Functions"

The Object Cache and Memory Management

The object cache is a client-side memory buffer that provides lookup and memory management support for objects. It stores and tracks object instances that have been fetched by an OCI application. The object cache provides memory management.

When objects are fetched by the application through a SQL SELECT statement, or through an OCI pin operation, a copy of the object is stored in the object cache. Objects that are fetched directly through a SELECT statement are fetched *by value*, and they are non-referenceable objects which cannot be pinned. Only referenceable objects may be pinned.

If an object is being pinned, and an appropriate version already exists in the cache, it does not need to be fetched from the server.

Every client program that uses OCI to dereference REFs to retrieve objects utilizes the object cache. A client-side object cache is allocated for every OCI environment handle initialized in object mode. Multiple threads of a process can share the same client-side cache by sharing the same OCI environment handle.

Exactly one copy of each referenceable object exists in the cache for each connection. Dereferencing a REF many times or dereferencing several equivalent REFs returns the same copy of the object.

If you modify a copy of an object in the cache, you must flush the changes to the server before they are visible to other processes. Objects that are no longer needed

can be unpinned or freed; they can then be swapped out of the cache, freeing the memory space they occupied.

When database objects are loaded into the cache, they are transparently mapped into the C language structures. The object cache maintains the association between all object copies in the cache and their corresponding objects in the database. When the transaction is committed, changes made to the object copy in the cache are automatically propagated to the database.

The cache does not manage the contents of object copies; it does not automatically refresh object copies. The application must ensure the correctness and consistency of the contents of object copies. For example, if the application marks an object copy for insert, update, or delete, then aborts the transaction, the cache simply unmarks the object copy but does not purge or invalidate the copy. The application must pin *recent* or *latest*, or refresh the object copy in the next transaction. If it pins *any*, it may get the same object copy with its uncommitted changes from the previous aborted transaction.

> **See Also:**   For more information about pin options, see "Pinning an Object Copy" on page 13-7

The object cache is created when the OCI environment is initialized using `OCIEnvCreate()` with *mode* set to OCI_OBJECT.

The object cache maintains a fast look-up table for mapping `REF`s to objects. When an application de-references a `REF` and the corresponding object is not yet cached in the object cache, the object cache automatically sends a request to the server to fetch the object from the database and load it into the object cache.

Subsequent de-references of the same `REF` will be faster since they become local cache access and do not incur network round trips. To notify the object cache that an application is accessing an object in the cache, the application pins the object; when it is done with the object, it should unpin it. The object cache maintains a pin count for each object in the cache, the count is incremented upon a pin call and unpin call decrements it. When the pin count goes to zero, that means the object is no longer needed by the application.

The object cache uses an least-recently used (LRU) algorithm to manage the size of the cache. The LRU algorithm frees candidate objects when the cache reaches the maximum size. The candidate objects are objects with a pin count of zero.

Each application processes running against the same server has its own object cache, as shown in Figure 13–1, "The Object Cache".

Figure 13–1   The Object Cache
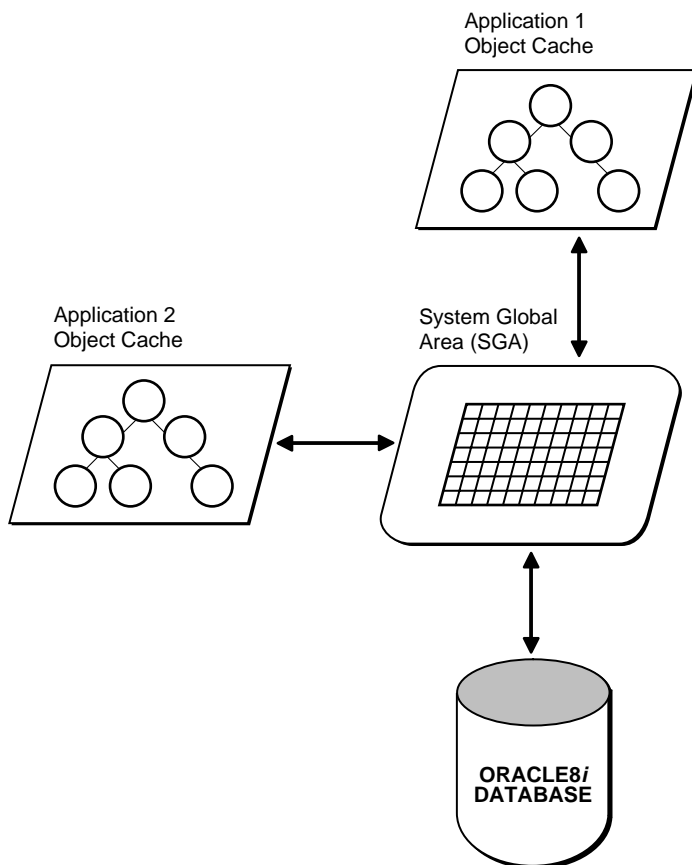
The object cache tracks the objects that are currently in memory, maintains references to the objects, manages automatic object swapping, and tracks object meta-attributes.

Cache Consistency and Coherency

The object cache does not automatically maintain value coherency or consistency between object copies and their corresponding objects in the database. In other words, if an application makes changes to an object copy, the changes are not automatically applied to the corresponding object in the database, and vice versa. The cache provides operations such as flushing a modified object copy to the

database and refreshing a stale object copy with the latest value from the database
to enable the program to maintain some coherency.

> **Note:**   Oracle does not support automatic cache coherency with
> the server's buffer cache or database. Automatic cache coherency
> refers to the mechanism by which the object cache refreshes local
> object copies when the corresponding objects have been modified
> in the server's buffer cache. This mechanism happens when the
> object cache flushes the changes made to local object copies to the
> buffer cache before any direct access of corresponding objects in the
> server. Direct access includes using SQL, triggers, or stored
> procedures to read or modify objects in the server.

Object Cache Parameters

The object cache has two important parameters associated with it, which are
attributes of the environment handle:

- OCI_ATTR_CACHE_MAX_SIZE, the maximum cache size
- OCI_ATTR_CACHE_OPT_SIZE, the optimal cache size

These parameters refer to levels of cache memory usage, and they help to determine
when the cache automatically ages out eligible objects to free up memory.

If the memory occupied by the objects currently in the cache reaches or exceeds the
maximum cache size, the cache automatically begins to free (or ages out) unmarked
objects which have a pin count of zero. The cache continues freeing such objects
until memory usage in the cache reaches the optimal size, or until it runs out of
objects eligible for freeing. Note that the cache can grow beyond the specified
maximum cache size.

OCI_ATTR_CACHE_MAX_SIZE is specified as a percentage of
OCI_ATTR_CACHE_OPT_SIZE. The maximum object cache size (in bytes) is
computed by incrementing OCI_ATTR_CACHE_OPT_SIZE by
OCI_ATTR_CACHE_MAX_SIZE percentage, as follows:

```
maximum_cache_size = optimal_size + optimal_size * max_size_percentage / 100
```

or

```
maximum_cache_size = OCI_ATTR_CACHE_OPT_SIZE + OCI_ATTR_CACHE_OPT_SIZE *
                                   OCI_ATTR_CACHE_MAX_SIZE / 100
```

The default value for OCI_ATTR_CACHE_MAX_SIZE is 10%. The default value for OCI_ATTR_CACHE_OPT_SIZE is 8M bytes.

The cache size attributes of the environment handle can be set with the `OCIAttrSet()` call and retrieved with the `OCIAttrGet()` function.

> **See Also:** See the section "Environment Handle Attributes" on page A-4 for more information.

Object Cache Operations

This section describes the most important functions the object cache provides to operate on object copies.

> **See Also:** All of the OCI's navigational and cache/object management functions are listed in the section "OCI Navigational Functions" on page 13-20.

Pinning and Unpinning

Pinning an object copy enables the application to access it in the cache by dereferencing the REF to it.

Unpinning an object indicates to the cache that the object currently is not being used. Objects should be unpinned when they are no longer needed to make them eligible for implicit freeing by the cache, thus freeing up memory.

Freeing

Freeing an object copy removes it from the cache and frees its memory.

Marking and Unmarking

Marking an object notifies the cache that an object copy has been updated in the cache and the corresponding object must be updated in the server when the object copy is flushed.

Unmarking an object removes the indication that the object has been updated.

Flushing

Flushing an object writes local changes made to marked object copies in the cache to the corresponding objects in the server. When this happens, the copies in the object cache are unmarked.

Refreshing

Refreshing an object copy in the cache replaces it with the latest value of the corresponding object in the server.

> **Note:** Pointers to top-level object memory are valid after a refresh. Pointers to secondary-level memory (e.g., string text pointers, collections, etc.) may become invalid after a refresh.

Loading and Removing Object Copies

Pin, unpin, and free functions are discussed in this section.

Pinning an Object Copy

When an application needs to dereference a REF in the object cache, it calls `OCIObjectPin()`. This call dereferences the REF and pins the object copy in the cache. As long as the object copy is pinned, it is guaranteed to be accessible by the application. Another variation of `OCIObjectPin()` is `OCIObjectArrayPin()` which takes an array of REFs, dereferences the REFs, and pins the object copies. Both `OCIObjectPin()` and `OCIObjectArrayPin()` take a pin option, *any*, *recent*, or *latest*. The datatype of the pin option is **OCIPinOpt**.

- If the *any* (OCI_PIN_ANY) option is specified, the object cache immediately returns the object copy that is already in the cache, if there is one. If no copy is in the cache, the object cache loads the latest object copy from the database and then returns the object copy. The *any* option is appropriate for read-only, informational, fact, or meta objects, such as products, salesmen, vendors, regions, parts, or offices. These objects usually do not change often, and even if they change, the change does not affect the application.

- If the *latest* (OCI_PIN_LATEST) option is specified, the object cache loads into the cache the latest object copy from the database. It returns that copy unless the object copy is locked in the cache, in which case the marked object copy is returned immediately. If the object is already in the cache and not locked, the latest object copy is loaded and overwrites the existing one. The *latest* option is appropriate for operational objects, such as purchase orders, bugs, line items, bank accounts, or stock quotes. These objects usually change often, and the program cares to access these objects at their latest possible state.

- If the *recent* (OCI_PIN_RECENT) option is specified, there are two possibilities:

- If in the same transaction the object copy has been previously pinned using the *latest* or *recent* option, the *recent* option becomes equivalent to the *any* option.

- If the previous condition does not apply, the *recent* option becomes equivalent to the *latest* option.

When the program pins an object, the program also specifies one of two possible values for the pin duration: *session* or *transaction*. The datatype of the duration is **OCIDuration**.

- If the pin duration is *session* (OCI_DURATION_SESSION), the object copy remains pinned until the end of session (i.e., end of connection) or until it is unpinned explicitly by the program (by calling OCIObjectUnpin()).

- If the pin duration is *transaction* (OCI_DURATION_TRANS), the object copy remains pinned until the end of transaction or until it is unpinned explicitly.

When loading an object copy into the cache from the database, the cache effectively executes

```
SELECT VALUE(t) FROM t WHERE REF(t) = :r
```

where t is the object table storing the object, and r is the REF, and the fetched value becomes the value of the object copy in the cache.

Since the object cache effectively executes a separate SELECT statement to load each object copy into the cache, in a read-committed transaction, object copies are not guaranteed to be read-consistent with each other.

In a serializable transaction, object copies pinned *recent* or *latest* are read-consistent with each other because the SELECT statements to load these object copies are executed based on the same database snapshot.

The object cache model is orthogonal to or independent of the Oracle transaction model. The behavior of the object cache does not change based on the transaction model, even though the objects that are retrieved from the server through the object cache can be different when running the same program under different transaction models (e.g., read committed versus serializable).

Unpinning an Object Copy

An object copy can be unpinned when it is no longer used by the program. It then becomes available to be freed. An object copy must be both completely unpinned and unmarked in order to become eligible to be implicitly freed by the cache when

the cache begins to run out of memory. To be completely unpinned, an object copy that has been pinned N times must be unpinned N times.

An unpinned but marked object copy is not eligible for implicit freeing until the object copy is flushed or explicitly unmarked by the user. However, the object cache implicitly frees object copies only when it begins to run out of memory, so an unpinned object copy need not necessarily be freed. If it has not been implicitly freed and is pinned again (with the any or recent options), the program gets the same object copy.

An application calls `OCIObjectUnpin()` or `OCIObjectPinCountReset()` to unpin an object copy. In addition, a program can call `OCICacheUnpin()` to completely unpin all object copies in the cache for a specific connection.

Freeing an Object Copy

Freeing an object copy removes it from the object cache and frees up its memory. The cache supports two methods for freeing up memory:

1. Explicit freeing - A program explicitly frees or removes an object copy from the cache by calling `OCIObjectFree()` which takes an option to (forcefully) free either a marked or pinned object copy. The program can also call `OCICacheFree()` to free all object copies in the cache.

2. Implicit freeing - Should the cache begin to run out of memory, it implicitly frees object copies that are both unpinned and unmarked. Unpinned objects that are marked are eligible for implicitly freeing only when the object copy is flushed or unmarked.

   See Also: For more information, see the section "Object Cache Parameters" on page 13-5.

For memory management reasons, it is important that applications unpin objects when they are no longer needed. This makes these objects available for aging out of the cache, and makes it easier for the cache to free memory when necessary.

OCI does not provide a function to free unreferenced objects in the client-side cache.

Making Changes to Object Copies

Functions for marking and unmarking object copies are discussed in this section.

Marking an Object Copy

An object copy can be created, updated, and deleted locally in the cache. If the object copy is created in the cache (by calling `OCIObjectNew()`), the object copy is marked for insert by the object cache, so that the object will be inserted in the server when the object copy is flushed.

If the object copy is updated in the cache, the user has to notify the object cache by marking the object copy for update (by calling `OCIObjectMarkUpdate()`). When the object copy is flushed, the corresponding object in the server is updated with the value in the object copy.

If the object copy is deleted, the object copy is marked for delete in the object cache (by calling `OCIObjectMarkDelete()`). When the object copy is flushed, the corresponding object in the server is deleted. The memory of the marked object copy is not freed until it is flushed and unpinned. When pinning an object marked for delete, the program receives an error, as if the program is dereferencing a dangling reference.

When a user makes multiple changes to an object copy, it is the final results of these changes which are applied to the object in the server when the copy is flushed. For example, if the user updates and deletes an object copy, the object in the server is simply deleted when the object copy is flushed. Similarly, if an attribute of an object copy is updated multiple times, it is the final value of this attribute which is updated in the server when the object copy is flushed.

The program can mark an object copy as updated or deleted only if the object copy has been loaded into the object cache.

Unmarking an Object Copy

A marked object copy can be unmarked in the object cache. By unmarking a marked object copy, the changes that are made to the object copy are not flushed to the server. The object cache does not undo the local changes that are already made to the object copy.

A program calls `OCIObjectUnmark()` to unmark an object. In addition, a program can call `OCICacheUnmark()` to unmark all object copies in the cache for a specific connection.

Synchronizing Object Copies with Server

Cache/server synchronization operations (flushing, refreshing) are discussed in this section.

Flushing Changes to Server

The local changes made to a marked object copy in the cache are written to the server when the object copy is flushed. The program can call `OCIObjectFlush()` to flush a single object copy or `OCICacheFlush()` to flush all marked object copies in the cache or a list of selected marked object copies. `OCICacheFlush()` flushes objects associated with a specific service context. See `OCICacheFlush()` on page 17-9.

After flushing an object copy, the object copy is unmarked. (Note that the object is locked in the server after it is flushed; the object copy is therefore marked as locked in the cache.)

> **Note:** The `OCICacheFlush()` operation incurs only a single server round-trip even if multiple objects are being flushed.

If an application wishes to flush only dirty objects of a certain type, this functionality is available through the callback function which is an optional argument to the `OCICacheFlush()` call. The application can define a callback which returns only the desired objects. In this case the operation still incurs only a single server round-trip for the flush.

In the default mode during `OCICacheFlush()`, the objects are flushed in the order that they are marked dirty. The performance of this flush operation can be considerably improved by setting the OCI_ATTR_CACHE_ARRAYFLUSH attribute in the environment handle.

> **See Also:** See OCI_ATTR_CACHE_ARRAYFLUSH on page A-4

However, OCI_ATTR_CACHE_ARRAYFLUSH mode should be used only if the order in which the objects are flushed is not important. During this mode, the dirty objects are grouped together and sent to the server in a manner that enables the server to efficiently update its tables. When this mode is enabled, it is not guaranteed that the order in which the objects are marked dirty is preserved.

Refreshing an Object Copy

When refreshed, an object copy is reloaded with the latest value of the corresponding object in the server. The latest value may contain changes made by other committed transactions and changes made directly (not through the object cache) in the server by the transaction. The program can change objects directly in the server using SQL DML, triggers, or stored procedures.

To refresh a marked object copy, the program must first unmark the object copy. An unpinned object copy is simply freed when it is refreshed (i.e., when the whole cache is refreshed).

The program can call `OCIObjectRefresh()` to refresh a single object copy or `OCICacheRefresh()` to refresh all object copies in the cache, all object copies that are loaded in a transaction (i.e., object copies that are pinned recent or pinned latest), or a list of selected object copies.

When an object is flushed to the server, triggers can be fired to modify more objects in the server. The same objects (modified by the triggers) in the object cache become out-of-date, and must be refreshed before they can be locked or flushed.

The various meta-attribute flags and durations of an object are modified as described in Table 13–1 after being refreshed:

Table 13–1   Object Attributes After Refresh

Object Attribute	Status After Refresh
existent	set to appropriate value
pinned	unchanged
flushed	reset
allocation duration	unchanged
pin duration	unchanged

During refresh, the object cache loads the new data into the top-level memory of an object copy, thus reusing the top level memory. The top-level memory of an object copy contains the in-line attributes of the object. On the other hand, the memory for the out-of-line attributes of an object copy may be freed and relocated, since the out-of-line attributes can vary in size.

> **See Also:**   See the section "Memory Layout of an Instance" on page 13-17 for more information about object memory

Object Locking

OCI functions related to object locking are discussed in this section.

Lock Options
When pinning an object you can specify whether the object should be locked or not via lock options. When an object is locked a server side lock is acquired and this

prevents any other user from modifying the object. The lock is released when the transaction commits or rollbacks. The different lock options are:

- The lock option OCI_LOCK_NONE instructs the cache to pin the object without locking.

- The lock option OCI_LOCK_X instructs the cache to pin the object only after acquiring a lock. If the object is currently locked by another user, the pin call with this option will wait until it can acquire the lock before returning to the caller. This is equivalent to executing a SELECT FOR UPDATE statement.

- The lock option OCI_LOCK_X_NOWAIT instructs the cache to pin the object only after acquiring a lock. Unlike the OCI_LOCK_X option, the pin call with OCI_LOCK_X_NOWAIT option will not wait if the object is currently locked by another user. This is equivalent to executing a SELECT FOR UPDATE WITH NOWAIT statement.

Locking Objects For Update

The program can optionally call OCIObjectLock() to lock an object for update. This call instructs the object cache to get a row lock on the object in the database. This is similar to executing

```
SELECT NULL FROM t WHERE REF(t) = :r FOR UPDATE
```

where t is the object table storing the object to be locked and r is the REF identifying the object. The object copy is marked locked in the object cache after OCIObjectLock() is called.

To lock a graph or set of objects, several OCIObjectLock() calls are required, one for each object, or the array pin OCIObjectArrayPin() call can be used for better performance.

By locking an object, the application is guaranteed that the object in the cache is up-to-date. No other transaction can modify the object while the application has it locked.

At the end of a transaction, all locks are released automatically by the server. The locked indicator in the object copy is reset.

Locking With the NOWAIT Option

In some cases, an application may attempt to lock an object which is currently locked by another user. In this case the application is blocked.

In order to avoid blocking when trying to lock an object, an application can use the `OCIObjectLockNoWait()` call instead of `OCIObjectLock()`. This function returns an error if it is unable to lock an object immediately because it is locked by another user.

The `NOWAIT` option is also available to pin calls by passing a value of OCI_LOCK_X_NOWAIT as the lock option parameter.

Implementing Optimistic Locking

There are two options available for implementing optimistic locking in an OCI application.

Optimistic Locking Option 1

The first optimistic locking option is for OCI applications that run transactions at the serializable level.

OCI supports calls that allow you to dereference and pin objects in the object cache without locking them, modify them in the cache (again without locking them), and then flush them (the dirtied objects) to the database.

During the flush, if a dirty object has been modified by another committed transaction since the beginning of your transaction, a non-serializable transaction error is returned. If none of the dirty objects has been modified by any other any other transaction since the beginning of your transaction, then the changes are written to the database successfully.

> **Note:** `OCITransCommit()` first flushes dirty objects into the database before committing a transaction.

The above mechanism effectively implements an optimistic locking model.

Optimistic Locking Option2

Alternately, an application can enable object change detection mode. To do this, set the OCI_ATTR_OBJECT_DETECTCHANGE attribute of the environment handle to a value of TRUE.

When this mode has been activated, the application receives an ORA-08179 error ("concurrency check failed") when attempting to flush an object that has been changed in the server by another committed transaction. The application can then handle this error in an appropriate manner.

Commit and Rollback in Object Applications

When a transaction is committed (OCITransCommit()), all marked objects are flushed to the server. If an object copy is pinned with a transaction duration, the object copy is unpinned.

When a transaction is rolled back, all marked objects are unmarked. If an object copy is pinned with a transaction duration, the object copy is unpinned.

Object Duration

In order to maintain free space in memory, the object cache attempts to reuse objects' memory whenever possible. The object cache reuses an object's memory when the object's lifetime (*allocation duration)* expires or when the object's *pin duration* expires. The allocation duration is set when an object is created with OCIObjectNew(), and the pin duration is set when an object is pinned with OCIObjectPin(). The datatype of the duration value is **OCIDuration**.

> **Note:** The pin duration for an object cannot be longer than the object's allocation duration.

When an object reaches the end of its allocation duration, it is automatically deleted and its memory can be reused. The pin duration indicates when an object's memory can be reused, and memory is reused when the cache is full.

OCI supports two predefined durations:

1. transaction (OCI_DURATION_TRANS)
2. session (OCI_DURATION_SESSION)

The *transaction duration* expires when the containing transaction ends (commits or aborts). The *session duration* expires when the containing session/connection ends.

The application can explicitly unpin an object using OCIObjectUnpin(). To minimize explicit unpinning of individual objects, the application can unpin all objects currently pinned in the object cache using the function OCICacheUnpin(). By default, all objects are unpinned at the end of the pin duration.

Durations Example

Table 13–2 illustrates the use of the different durations in an application. Four objects are created or pinned in this application over the course of one connection and three transactions. The first column indicates the action performed by the

database, and the second column indicates the function which performs the action. The remaining columns indicate the states of the various objects at each point in the application.

For example, Object 1 comes into existence at T2 when it is created with a connection duration, and it exists until T19 when the connection is terminated. Object 2 is pinned at T7 with a transaction duration, after being fetched at T6, and it remains pinned until T9 when the transaction is committed.

Table 13–2   Example of Allocation and Pin Durations

Time	Application Action	Function	Object 1	Object 2	Object 3	Object 4
T_1	Establish connection					
T_2	Create object 1 - allocation duration = connection	OCIObjectNew()	exists			
T_5	Start Transaction1	OCITransStart()	exists			
T_6	SQL - fetch REF to object 2		exists			
T_7	Pin object 2 - pin duration = transaction	OCIObjectPin()	exists	pinned		
T_8	Process application data		exists	pinned		
T_9	Commit Transaction1	OCITransCommit()	exists	unpinned		
T_{10}	Start Transaction2	OCITransStart()	exists			
T_{11}	Create object 3 - allocation duration = transaction	OCIObjectNew()	exists		exists	
T_{12}	SQL - fetch REF to object 4		exists		exists	
T_{13}	Pin object 4 - pin duration = connection	OCIObjectPin()	exists		exists	pinned
T_{14}	Commit Transaction2	OCITransCommit()	exists		deleted	pinned
T_{15}	Terminate session1	OCIDurationEnd()	exists			pinned
T_{16}	Start Transaction3	OCITransStart()	exists			pinned
T_{17}	Process application data		exists			pinned
T_{18}	Commit Transaction3	OCITransCommit()	exists			pinned
T_{19}	Terminate connection		deleted			unpinned

- See the descriptions of `OCIObjectNew()` and `OCIObjectPin()` in Chapter 17, "OCI Navigational and Type Functions" for specific information about parameter values which can be passed to these functions

- See the section "Creating Objects" on page 10-33 for information about freeing up an object's memory before its allocation duration has expired

Memory Layout of an Instance

An instance in memory is composed of a top-level memory chunk of the instance, a top-level memory of the null indicator structure and optionally, a number of secondary memory chunks. Consider a *DEPARTMENT* row type,

```
CREATE TYPE department AS OBJECT
( dep_name      varchar2(20),
  budget        number,
  manager       person,             /* person is an object type */
  employees     person_array );     /* varray of person objects */
```

and its C representation

```
struct department
{
OCIString * dep_name;
OCINumber budget;
struct person manager;
OCIArray * employees;
);
typedef struct department department;
```

Each instance of *DEPARTMENT* has a top-level memory chunk which contains the top-level attributes such as dep_name, budget, manager and employees. The attributes dep_name and employees are themselves actually pointers to the additional memory (the secondary memory chunks). The secondary memory is used to contain the actual data for the embedded instances (e.g. employees varray and dep_name string).

The top-level memory of the null indicator structure contains the null statuses of the attributes in the top level memory chunk of the instance. From the above example, the top level memory of the null structure contains the null statuses of the attributes dep_name, budget, manager and the atomic nullity of employees.

Object Navigation

This section discusses how OCI applications can navigate through graphs of objects in the object cache.

Simple Object Navigation

In the example in the previous sections, the object retrieved by the application was a simple object, whose attributes were all scalar values. If an application retrieves an object with an attribute which is a REF to another object, the application can use OCI calls to traverse the *object graph* and access the referenced instance.

As an example, consider the following declaration for a new type in the database:

```
CREATE TYPE person_t AS OBJECT
( name            VARCHAR2(30),
  mother          REF person_t,
  father          REF person_t);
```

An object table of person_t objects is created with the following statement:

```
CREATE TABLE person_table OF person_t;
```

Instances of the person_t type can now be stored in the typed table. Each instance of person_t includes references to two other objects, which would also be stored in the table. A NULL reference could represent a parent about whom information is not available.

An object graph is a graphical representation of the REF links between object instances. For example, Figure 13–2, "Object Graph of person_t Instances" on the following page depicts an object graph of person_t instances, showing the links from one object to another. The circles represent objects, and the arrows represent references to other objects.

Figure 13–2   Object Graph of `person_t` Instances
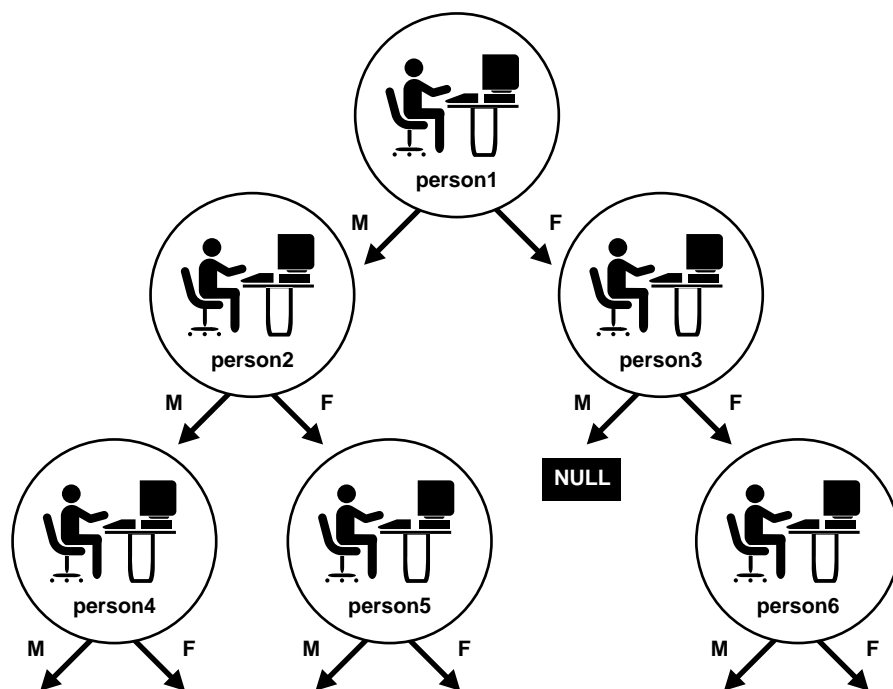
In this case, each object has links to two other instances of the same object. This need not always be the case. Objects may have links to other object types. Other types of graphs are also possible. For example, if a set of objects is implemented as a linked list, the object graph could be viewed as a simple chain, where each object references the previous and/or next objects in the linked list.

You can use the methods described earlier in this chapter to retrieve a reference to a person_t instance and then pin that instance. OCI provides functionality which enables you to traverse the object graph by following a reference from one object to another.

As an example, assume that an application fetches the person1 instance in the above graph and pins it as pers_1. Once that has been done, the application can access the mother instance of person1 and pin it into pers_2 through a second pin operation:

```
OCIObjectPin(env, err, pers_1->mother, OCI_PIN_ANY, OCI_DURATION_TRANS,
             OCI_LOCK_X, (OCIComplexObject *) 0, &pers_2);
```

In this case, an OCI fetch operation is not required to retrieve the second instance.

The application could then pin the father instance of person1, or it could operate on the reference links of person2.

> **Note:** Attempting to pin a NULL or dangling REF results in an error on the OCIObjectPin() call.

OCI Navigational Functions

This section provides a brief summary of the available OCI navigational functions. The functions are grouped according to their general functionality.

> **See Also:** More detailed descriptions of each of these functions can be found in Chapter 17, "OCI Navigational and Type Functions"

The use of these functions is described in the earlier sections of this chapter.

The navigational functions follow a naming scheme which uses different prefixes for different types of functionality:

OCICache*() - these functions are Cache operations

OCIObject*() - these functions are individual Object operations

Pin/Unpin/Free Functions

The following functions are available to pin, unpin, or free objects:

Function	Purpose
OCICacheFree()	Free all instances in the cache
OCICacheUnpin()	Unpin persistent objects in cache or connection
OCIObjectArrayPin()	Pin an array of references
OCIObjectFree()	Free and unpin a standalone instance
OCIObjectPin()	Pin an object
OCIObjectPinCountReset()	Unpin an object to zero pin count
OCIObjectPinTable()	Pin a table object with a given duration

Function	Purpose
OCIObjectUnpin()	Unpin an object

Flush and Refresh Functions

The following functions are available to flush modified objects to the server:

Function	Purpose
OCICacheFlush()	Flush modified persistent objects in cache to server
OCIObjectFlush()	Flush a modified persistent object to the server
OCICacheRefresh()	Refresh pinned persistent objects in the cache
OCIObjectRefresh()	Refresh a single persistent object

Mark and Unmark Functions

The following functions allow an application to mark or unmark an object by modifying one of its meta-attributes:

Function	Purpose
OCIObjectMarkDelByRef()	Mark an object deleted given a REF
OCIObjectMarkUpd()	Mark an object as updated/dirty
OCIObjectMarkDel()	Mark an object deleted / delete a value instance
OCICacheUnmark()	Unmarks all objects in the cache
OCIObjectUnmark()	Marks a given object as updated
OCIObjectUnmarkByRef()	Marks an object as updated, given a REF

Object Meta-Attribute Accessor Functions

The following functions allow an application to access the meta-attributes of an object:

Function	Purpose
OCIObjectExists()	Get existence status of an instance
OCIObjectFlushStatus()	Get the flush status of an instance

Function	Purpose
OCIObjectGetInd()	Get null structure of an instance
OCIObjectIsDirtied()	Has an object been marked as updated?
OCIObjectIsLocked()	Is an object locked?

Other Functions

The following functions provide additional object functionality for OCI applications:

Function	Purpose
OCIObjectCopy()	Copy one instance to another
OCIObjectGetObjectRef()	Return reference to a given object
OCIObjectGetTypeRef()	Get a reference to a TDO of an instance
OCIObjectLock()	Lock a persistent object
OCIObjectLockNoWait()	Lock an object in NOWAIT mode
OCIObjectNew()	Create a new instance

Type Evolution and the Object Cache

When type information is requested based on the type name, OCI returns the type descriptor object (TDO) corresponding to the latest version of the type. Since there is no synchronization between the server and the object cache, the TDO in the object cache may not be current.

It is possible that when pinning an object, the version of the image differs from the TDO's version. Then, an error will be issued. It is up to you to abort the application or refresh the TDO and re-pin the object. Continuing with the application may crash the application because even if the image and the TDO are at the same version, there is no guarantee that the object structure (i.e., C struct) defined in the application is compatible with the new type version, especially for the case when an attribute has been dropped from the type in the server.

Thus, when the structure of a type is altered, you must regenerate the header files of the changed type, modify their application, re-compile and re-link before executing the program again.

> **See Also:** "Type Evolution" on page 10-42

14

The Object Type Translator (OTT)

This chapter discusses the Object Type Translator (OTT), which is used to map database object types and named collection types to C structs for use in OCI and Pro*C/C++ applications. The chapter includes the following sections:

- OTT Overview
- What is the Object Type Translator
- The OTT Command Line
- The Intype File
- OTT Datatype Mappings
- The Outtype File
- Using OTT with OCI Applications
- OTT Reference

> **See Also:** For information specific to Pro*C/C++, please refer to the *Pro*C/C++ Precompiler Programmer's Guide.*

OTT Overview

The OTT (Object Type Translator) assists in the development of C language applications that make use of user-defined types in an Oracle server.

Through the use of SQL CREATE TYPE statements, you can create object types. The definitions of these types are stored in the database, and can be used in the creation of database tables. Once these tables are populated, an OCI or Pro*C/C++ programmer can access objects stored in the tables.

An application that accesses object data needs to be able to represent the data in a host language format. This is accomplished by representing object types as C structs. It would be possible for a programmer to code struct declarations by hand to represent database object types, but this can be very time-consuming and error-prone if many types are involved. The OTT simplifies this step by automatically generating appropriate struct declarations. For Pro*C/C++, the application only needs to include the header file generated by OTT. In OCI, the application also needs to call an initialization function generated by OTT.

In addition to creating structs which represent stored datatypes, OTT also generates parallel indicator structs which indicate whether an object type or its fields are null.

What is the Object Type Translator

The Object Type Translator (OTT) converts database definitions of object types and named collection types into C struct declarations which can be included in an OCI or Pro*C/C++ application.

You must explicitly invoke OTT to translate database types to C representations.

On most operating systems, OTT is invoked on the command line. It takes as input an *intype file*, and it generates an *outtype file* and one or more C *header files* and an optional *implementation file*. The following is an example of a command which invokes the OTT:

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=c \
hfile=demo.h initfile=demov.c
```

This command causes OTT to connect to the database with username 'scott' and password 'tiger', and translate database types to C structs, based on instructions in the intype file (demoin.typ). The resulting structs are output to the header file (demo.h) for the host language (C) specified by the code parameter. The outtype file (demoout.typ) receives information about the translation.

The implementation file (demov.c) contains the function to initialize the type version table with information about the user-defined types translated.

Each of these parameters is described in more detail in later sections of this chapter.

Sample demoin.typ file:

```
CASE=LOWER
TYPE employee
```

Sample demoout.typ file:

```
CASE = LOWER
TYPE SCOTT.EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
```

In this example, the demoin.typ file contains the type to be translated, preceded by TYPE (for example, TYPE employee). The structure of the outtype file is similar to the intype file, with the addition of information obtained by the OTT.

Once the OTT has completed the translation, the header file contains a C struct representation of each type specified in the intype file, and a null indicator struct corresponding to each type. For example, if the employee type listed in the intype file was defined as

```
CREATE TYPE employee AS OBJECT
(
    name        VARCHAR2(30),
    empno       NUMBER,
    deptno      NUMBER,
    hiredate    DATE,
    salary      NUMBER
);
```

the header file generated by the OTT (demo.h) includes, among other items, the following declarations:

```
struct employee
{
    OCIString * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate   hiredate;
    OCINumber salary;
};
typedef struct emp_type emp_type;
```

```
struct employee_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd deptno;
    OCIInd hiredate;
    OCIInd salary;
};
typedef struct employee_ind employee_ind;
```

A sample implementation file, `demov.c` produced by this command contains:

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword demov(OCIEnv *env, OCIError *err)
{
   sword status = OCITypeVTInit(env, err);
   if (status == OCI_SUCCESS)
       status = OCITypeVTInsert(env, err,
           "SCOTT", 5,
           "EMPLOYEE", 8,
           "$8.0", 4);
   return status;
}
```

Parameters in the intype file control the way generated structs are named. In this example, the struct name `employee` matches the database type name `EMPLOYEE`. The struct name is in lower case because of the line `CASE=lower` in the intype file.

The datatypes which appear in the struct declarations (e.g., **OCIString**, **OCIInd**) are special datatypes.

> **See Also:** For more information about these types, see the section "OTT Datatype Mappings" on page 14-10

The following sections describe these aspects of using the OTT:

- Creating Types in the Database
- Invoking OTT
- The OTT Command Line

- The Intype File
- OTT Datatype Mappings
- Null Indicator Structs
- The Outtype File

The remaining sections of the chapter discuss the use of the OTT with OCI, followed by a reference section which describes command line syntax, parameters, intype file structure, nested #include file generation, schema names usage, default name mapping, and restrictions.

Creating Types in the Database

The first step in using OTT is to create object types or named collection types and store them in the database. This is accomplished through the use of the SQL CREATE TYPE statement.

> **See Also:** For information about the CREATE TYPE statement, refer to the *Oracle9i SQL Reference.*

Invoking OTT

The next step is to invoke OTT. OTT parameters can be specified on the command line, or in a file called a configuration file. Certain parameters can also be specified in the intype file.

If a parameter is specified in more than one place, its value on the command line will take precedence over its value in the intype file, which takes precedence over its value in a user-defined configuration file, which takes precedence over its value in the default configuration file.

For global options -- that is, options on the command line or options at the beginning of the intype file before any TYPE statements -- the value on the command line overrides the value in the intype file. (The options that can be specified globally in the intype file are CASE, CODE, INITFILE, and INITFUNC, but not HFILE.) Anything in the intype file in a TYPE specification applies to a particular type only, and overrides anything on the command line that would otherwise apply to the type. So if you enter TYPE person HFILE=p.h, it applies to person only and overrides the HFILE on the command line. The statement is not considered a command-line parameter.

Command Line

Parameters (also called options) set on the command line override any set elsewhere.

> **See Also:** See "The OTT Command Line" on page 14-6, for more information.

Configuration File

A configuration file is a text file that contains OTT parameters. Each non-blank line in the file contains one parameter, with its associated value or values. If more than one parameter is put on a line, only the first one will be used. No whitespace may occur on any non-blank line of a configuration file.

A configuration file may be named on the command line. In addition, a default configuration file is always read. This default configuration file must always exist, but can be empty. The name of the default configuration file is `ottcfg.cfg`, and the location of the file is system-specific. For example, on Solaris, the file specification is `$ORACLE_HOME/precomp/admin/ottcfg.cfg`. See your platform-specific documentation for further information.

INTYPE File

The `intype` file gives a list of user defined types for OTT to translate.

The parameters `CASE`, `HFILE`, `INITFUNC`, and `INITFILE` can appear in the `intype` file.

> **See Also:** See "The Intype File" on page 14-9 for more information

The OTT Command Line

On most platforms, OTT is invoked on the command line.   You can specify the input and output files, and the database connection information, among other things. Consult your platform-specific documentation to see how to invoke OTT on your platform.

OTT Command Line Invocation Example

The following is an example of an OTT invocation from the command line:

```
ott userid=bren/bigkitty intype=demoin.typ outtype=demoout.typ code=c \
    hfile=demo.h initfile=demov.c
```

> **Note:** No spaces are permitted around the equals sign (=).

The following sections describe the elements of the command line used in this example.

> **See Also:** For a detailed discussion of the various OTT command line options, please refer to the section "OTT Reference" on page 14-26

OTT

Causes OTT to be invoked. It must be the first item on the command line.

USERID

Specifies the database connection information which OTT will use.

In Example 1, OTT will attempt to connect with username 'bren' and password 'bigkitty'.

INTYPE

Specifies the name of the intype file which will be used.

In Example 1, the name of the intype file is specified as demoin.typ.

OUTTYPE

Specifies the name of the outtype file. When OTT generates the C header file, it also writes information about the translated types into the outtype file. This file contains an entry for each of the types which is translated, including its version string, and the header file to which its C representation was written.

In "OTT Command Line Invocation Example" on page 14-6, the name of the outtype file is specified as demoout.typ.

> **Note:** If the file specified by the outtype keyword already exists, it is overwritten when OTT runs. If the name of the outtype file is the same as the name of the intype file, the outtype information overwrites the intype file.

CODE

Specifies the target language for the translation. The following options are available:

- C (equivalent to ANSI_C)

- ANSI_C (for ANSI C)

- KR_C (for Kernighan & Ritchie C)

There is currently no default option, so this parameter is required.

Struct declarations are identical in both C dialects. The style in which the initialization function defined in the INITFILE file is defined depends on whether KR_C is used. If the INITFILE option is not used, all three options are equivalent.

HFILE

Specifies the name of the C header file to which the generated structs should be written.

In "OTT Command Line Invocation Example" on page 14-6, the generated structs will be stored in a file called demo.h.

> **Note:** If the file specified by the hfile keyword already exists, it will be overwritten when OTT runs, with one exception: if the contents of the file as generated by OTT are identical to the previous contents of the file, OTT will not actually write to the file. This preserves the modification time of the file so that UNIX make and similar facilities on other platforms do not perform unnecessary recompilations.

INITFILE

Specifies the name of the C source file into which the type initialization function is to be written.

> **Note:** If the file specified by the initfile keyword already exists, it will be overwritten when OTT runs, with one exception: if the contents of the file as generated by OTT are identical to the previous contents of the file, OTT will not actually write to the file. This preserves the modification time of the file so that UNIX make and similar facilities on other platforms do not perform unnecessary recompilations.

The Intype File

When running OTT, the `intype` file tells OTT which database types should be translated, and it can also control the naming of the generated structs. The `intype` file can be a user-created file, or it may be the `outtype` file of a previous invocation of OTT. If the `intype` parameter is not used, all types in the schema to which OTT connects are translated.

The following is a simple example of a user-created `intype` file:

```
CASE=LOWER
TYPE employee
  TRANSLATE SALARY$ AS salary
           DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

The first line, with the `CASE` keyword, indicates that generated C identifiers should be in lower case. However, this `CASE` option is only applied to those identifiers that are not explicitly mentioned in the intype file. Thus, `employee` and `ADDRESS` would always result in C structures `employee` and `ADDRESS`, respectively. The members of these structures would be named in lower case.

> **See Also:**   See the description of "CASE" on page 14-32 for further information regarding the `CASE` option

The lines which begin with the `TYPE` keyword specify which types in the database should be translated: in this case, the `EMPLOYEE`, `ADDRESS`, `ITEM`, `PERSON`, and `PURCHASE_ORDER` types.

The `TRANSLATE...AS` keywords specify that the name of an object attribute should be changed when the type is translated into a C struct. In this case, the `SALARY$` attribute of the employee type is translated to `salary`.

The `AS` keyword in the final line specifies that the name of an object type should be changed when it is translated into a struct. In this case, the `purchase_order` database type is translated into a struct called `p_o`.

If `AS` is not used to translate a type or attribute name, the database name of the type or attribute will be used as the C identifier name, except that the `CASE` option will be observed, and any characters that cannot be mapped to a legal C identifier character will be replaced by an underscore. Reasons for translating a type or attribute name include:

■ The name contains characters other than letters, digits, and underscores

■ The name conflicts with a C keyword

■ The type name conflicts with another identifier in the same scope. This may happen, for example, if the program uses two types with the same name from different schemas.

■ The programmer prefers a different name

The OTT may need to translate additional types which are not listed in the `intype` file. This is because the OTT analyzes the types in the `intype` file for type dependencies before performing the translation, and translates other types as necessary. For example, if the ADDRESS type were not listed in the `intype` file, but the "Person" type had an attribute of type ADDRESS, OTT would still translate ADDRESS because it is required to define the "Person" type.

If you specify FALSE as the value of the TRANSITIVE parameter, then OTT will not generate types that are not specified in the `intype` file.

A normal case-insensitive SQL identifier can be spelled in any combination of upper and lower case in the `intype` file, and is not quoted.

Use quotation marks, such as TYPE "Person", to reference SQL identifiers that have been created in a case-sensitive manner, e.g., CREATE TYPE "Person". A SQL identifier is case-sensitive if it was quoted when it was declared. Quotation marks can also be used to refer to a SQL identifier that is an OTT-reserved word, for example, TYPE "CASE". When a name is quoted for this reason, the quoted name must be in upper case if the SQL identifier was created in a case-insensitive manner, for example, CREATE TYPE Case. If an OTT-reserved word is used to refer to the name of a SQL identifier but is not quoted, the OTT will report a syntax error in the `intype` file.

> **See Also:** For a more detailed specification of the structure of the `intype` file and the available options, refer to the section "Structure of the Intype File" on page 14-34

OTT Datatype Mappings

When OTT generates a C struct from a database type, the struct contains one element corresponding to each attribute of the object type. The datatypes of the attributes are mapped to types which are used in Oracle's object data types. The datatypes found in Oracle include a set of predefined, primitive types, and provide for the creation of user-defined types, such as object types and collections.

Oracle also includes a set of predefined types which are used to represent object type attributes in C structs. As an example, consider the following object type definition, and its corresponding OTT-generated struct declarations:

```
CREATE TYPE employee AS OBJECT
(    name       VARCHAR2(30),
     empno      NUMBER,
     deptno     NUMBER,
     hiredate   DATE,
     salary$    NUMBER);
```

The OTT output, assuming CASE=LOWER and no explicit mappings of type or attribute names, is:

```
struct employee
{    OCIString * name;
     OCINumber empno;
     OCINumber department;
     OCIDate   hiredate;
     OCINumber salary_;
};
typedef struct emp_type emp_type;
struct employee_ind
{
     OCIInd _atomic;
     OCIInd name;
     OCIInd empno;
     OCIInd department;
     OCIInd hiredate;
     OCIInd salary_;
}
typedef struct employee_ind employee_ind;
```

> **See Also:**   The indicator struct (struct employee_ind) is explained in the section, "Null Indicator Structs" on page 14-16

The datatypes in the struct declarations—**OCIString**, **OCINumber**, **OCIDate**, **OCIInd**—are used here to map the datatypes of the object type attributes. The NUMBER datatype of the empno attribute, maps to the **OCINumber** datatype, for example. These datatypes can also be used as the types of bind and define variables.

Mapping Object Datatypes to C

This section describes the mappings of Oracle object attribute types to C types generated by OTT. The following section, "OTT Type Mapping Example" on page 14-13, includes examples of many of these different mappings. The following table lists the mappings from types which can be used as attributes to object datatypes which are generated by OTT.

Table 14–1   Object Datatype Mappings for Object Type Attributes

Object Attribute Types	C Mapping
BFILE	OCIBFileLocator*
BLOB	OCILobLocator * or OCIBlobLocator *
CHAR(N), CHARACTER(N)	OCIString *
CLOB	OCILobLocator * or OCIClobLocator *
DATE	OCIDate
ANSI DATE	OCIDateTime *
TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime *
INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND	OCIInterval *
DEC, DEC(N), DEC(N,N)	OCINumber
DECIMAL, DECIMAL(N), DECIMAL(N,N)	OCINumber
FLOAT, FLOAT(N), DOUBLE PRECISION	OCINumber
INT, INTEGER, SMALLINT	OCINumber
Nested Object Type	C name of the nested object type
Nested Table	OCITable *
NUMBER, NUMBER(N), NUMBER(N,N)	OCINumber
NUMERIC, NUMERIC(N), NUMERIC(N,N)	OCINumber
RAW(N)	OCIRaw *
REAL	OCINumber
REF	OCIRef *

Table 14–1   Object Datatype Mappings for Object Type Attributes (Cont.)

Object Attribute Types	C Mapping
VARCHAR(N)	OCIString *
VARCHAR2(N)	OCIString *
VARRAY	OCIArray *

> **Note:** For REF, varray, and nested table types, OTT
> generates a typedef. The type declared in the typedef is then used
> as the type of the data member in the struct declaration. For
> examples, see the next section, "OTT Type Mapping Example".

If an object type includes an attribute of a REF or collection type, a typedef for the
REF or collection type is first generated. Then the struct declaration corresponding
to the object type is generated. The struct includes an element whose type is a
pointer to the REF or collection type.

If an object type includes an attribute whose type is another object type, OTT first
generates the nested type (if TRANSITIVE=TRUE). It then maps the object type
attribute to a nested struct of the type of the nested object type.

The Oracle C datatypes to which the OTT maps non-object database attribute types
are structures, which, except for **OCIDate**, are opaque.

OTT Type Mapping Example

The following example is presented to demonstrate the various type mappings
created by OTT.

Given the following database types

```
CREATE TYPE my_varray AS VARRAY(5) of integer;

CREATE TYPE object_type AS OBJECT
(object_name    VARCHAR2(20));

CREATE TYPE my_table AS TABLE OF object_type;

CREATE TYPE other_type AS OBJECT (object_number NUMBER);

CREATE TYPE many_types AS OBJECT
```

```
( the_varchar    VARCHAR2(30),
  the_char       CHAR(3),
  the_blob       BLOB,
  the_clob       CLOB,
  the_object     object_type,
  another_ref    REF other_type,
  the_ref        REF many_types,
  the_varray     my_varray,
  the_table      my_table,
  the_date       DATE,
  the_num        NUMBER,
  the_raw        RAW(255));
```

and an `intype` file which includes

```
CASE = LOWER
TYPE many_types
```

OTT would generate the following C structs:

> **Note:** Comments are provided here to help explain the structs.
> These comments are not part of actual OTT output.

```
#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE

#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCIRef many_types_ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;            /* used in many_types */
typedef OCITable my_table;             /* used in many_types*/
typedef OCIRef other_type_ref;
struct object_type                     /* used in many_types */
{
   OCIString * object_name;
};
typedef struct object_type object_type;
```

```
struct object_type_ind                  /*indicator struct for*/
{                                                  /*object_types*/
   OCIInd _atomic;
   OCIInd object_name;
};
typedef struct object_type_ind object_type_ind;

struct many_types
{
   OCIString *        the_varchar;
   OCIString *        the_char;
   OCIBlobLocator *   the_blob;
   OCIClobLocator *   the_clob;
   struct object_type the_object;
   other_type_ref *   another_ref;
   many_types_ref *   the_ref;
   my_varray *        the_varray;
   my_table *         the_table;
   OCIDate            the_date;
   OCINumber          the_num;
   OCIRaw *           the_raw;
};
typedef struct many_types many_types;

struct many_types_ind                    /*indicator struct for*/
{                                                  /*many_types*/
   OCIInd _atomic;
   OCIInd the_varchar;
   OCIInd the_char;
   OCIInd the_blob;
   OCIInd the_clob;
   struct object_type_ind the_object;            /*nested*/
   OCIInd another_ref;
   OCIInd the_ref;
   OCIInd the_varray;
   OCIInd the_table;
   OCIInd the_date;
   OCIInd the_num;
   OCIInd the_raw;
};
typedef struct many_types_ind many_types_ind;

#endif
```

Notice that even though only one item was listed for translation in the intype file, two object types and two named collection types were translated. This is because the OTT parameter TRANSITIVE on page 14-33, has the default value of TRUE. As described in that section, when TRANSITIVE=TRUE, OTT automatically translates any types which are used as attributes of a type being translated, in order to complete the translation of the listed type.

This is not the case for types which are only accessed by a pointer or ref in an object type attribute. For example, although the many_types type contains the attribute another_ref REF other_type, a declaration of struct other_type was not generated.

This example also illustrates how typedefs are used to declare varray, nested table, and REF types.

The typedefs occur near the beginning:

```
typedef OCIRef many_types_ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;
typedef OCITable my_table;
typedef OCIRef other_type_ref;
```

In the struct many_types, the varray, nested table, and REF attributes are declared:

```
struct many_types
{  ...
   other_type_ref *   another_ref;
   many_types_ref *   the_ref;
   my_varray *        the_varray;
   my_table *         the_table;
   ...
}
```

Null Indicator Structs

Each time OTT generates a C struct to represent a database object type, it also generates a corresponding null indicator struct. When an object type is selected into a C struct, null indicator information may be selected into a parallel struct.

For example, the following null indicator struct was generated in the example in the previous section:

```
struct many_types_ind
{
```

```
OCIInd _atomic;
OCIInd the_varchar;
OCIInd the_char;
OCIInd the_blob;
OCIInd the_clob;
struct object_type_ind the_object;
OCIInd another_ref;
OCIInd the_ref;
OCIInd the_varray;
OCIInd the_table;
OCIInd the_date;
OCIInd the_num;
OCIInd the_raw;
};
typedef struct many_types_ind many_types_ind;
```

The layout of the null struct is important. The first element in the struct (_atomic) is the *atomic null indicator*. This value indicates the null status for the object type as a whole. The atomic null indicator is followed by an indicator element corresponding to each element in the OTT-generated struct representing the object type.

Notice that when an object type contains another object type as part of its definition (in the above example it is the object_type attribute), the indicator entry for that attribute is the null indicator struct (object_type_ind) corresponding to the nested object type (if TRANSITIVE=TRUE).

varrays and nested tables contain the null information for their elements.

The datatype for all other elements of a null indicator struct is **OCIInd**.

> **See Also:**   For more information about atomic nullity, refer to the
> section "Nullity" on page 10-30

OTT Support for Type Inheritance

To support type inheritance of objects, OTT generates a C struct to represent an object subtype by declaring the inherited attributes in an encapsulated struct with the special name '_super', before declaring the new attributes. Thus, for an object subtype that inherits from a supertype, the first element in the struct is named '_super', followed by elements corresponding to each attribute of the subtype. The type of the element named '_super' is the name of the supertype.

For example, given the type Person_t, with subtype Student_t and subtype Employee_t, which are created as follows:

```
CREATE TYPE Person_t AS OBJECT
( ssn     NUMBER,
  name    VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;

CREATE TYPE Student_t UNDER Person_t
( deptid NUMBER,
  major  VARCHAR2(30)) NOT FINAL;

CREATE TYPE Employee_t UNDER Person_t
( empid NUMBER,
  mgr   VARCHAR2(30));
```

and, given an `intype` file which includes:

```
CASE=SAME
TYPE EMPLOYEE_T
TYPE STUDENT_T
TYPE PERSON_T
```

OTT generates the following C structs for `Person_t`, `Student_t`, and `Employee_t` and their null indicator structs:

```
#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE

#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCIRef EMPLOYEE_T_ref;
typedef OCIRef STUDENT_T_ref;
typedef OCIRef PERSON_T_ref;

struct PERSON_T
{
   OCINumber SSN;
   OCIString * NAME;
   OCIString * ADDRESS;
};
typedef struct PERSON_T PERSON_T;

struct PERSON_T_ind
{
   OCIInd _atomic;
   OCIInd SSN;
```

```
   OCIInd NAME;
   OCIInd ADDRESS;
};
typedef struct PERSON_T_ind PERSON_T_ind;

struct EMPLOYEE_T
{
   PERSON_T _super;
   OCINumber EMPID;
   OCIString * MGR;
};
typedef struct EMPLOYEE_T EMPLOYEE_T;

struct EMPLOYEE_T_ind
{
   PERSON_T _super;
   OCIInd EMPID;
   OCIInd MGR;
};
typedef struct EMPLOYEE_T_ind EMPLOYEE_T_ind;

struct STUDENT_T
{
   PERSON_T _super;
   OCINumber DEPTID;
   OCIString * MAJOR;
};
typedef struct STUDENT_T STUDENT_T;

struct STUDENT_T_ind
{
   PERSON_T _super;
   OCIInd DEPTID;
   OCIInd MAJOR;
};
typedef struct STUDENT_T_ind STUDENT_T_ind;

#endif
```

The above C mapping convention allows simple up-casting from an instance of a subtype to an instance of a supertype in C to work properly. For example:

```
STUDENT_T *stu_ptr = some_ptr;                 /* some STUDENT_T instance  */
PERSON_T  *pers_ptr = (PERSON_T *)stu_ptr;  /* up-casting */
```

The null indicator structs are generated similarly. Note that for the supertype `Person_t` null indicator struct, the first element is `'_atomic'`, and that for the subtypes `Employee_t` and `Student_t` null indicator structs, the first element is `'_super'` (no atomic element is generated for subtypes).

Substitutable Object Attributes

For attributes of `NOT FINAL` types (and therefore potentially substitutable), the embedded attribute is represented as a pointer.

Consider a type `Book_t` created as:

```
CREATE TYPE Book_t AS OBJECT
( title   VARCHAR2(30),
  author  Person_t      /* substitutable */);
```

The corresponding C struct generated by OTT contains a pointer to `Person_t`:

```
struct Book_t
{
  OCIString  *title;
  Person_t   *author;    /* pointer to Person_t struct */
}
```

The null indicator struct corresponding to the above type is:

```
struct Book_t_ind
{
  OCIInd  _atomic;
  OCIInd  title;
  OCIInd  author;
}
```

Note that the null indicator struct corresponding to the `author` attribute can be obtained from the `author` object itself. See `OCIObjectGetInd()`.

If a type is defined to be `FINAL`, it cannot have any subtypes. An attribute of a `FINAL` type is therefore not substitutable. In such cases, the mapping is as before: the attribute struct is inline. Now, if the type is altered and defined to be `NOT FINAL`, the mapping will have to change. The new mapping is generated by running OTT again.

The Outtype File

The `outtype` file is named on the OTT command line. When OTT generates the C header file, it also writes the results of the translation into the `outtype` file. This file contains an entry for each of the types which is translated, including its version string, and the header file to which its C representation was written.

The `outtype` file from one OTT run can be used as the `intype` file for a subsequent OTT invocation.

For example, given the simple `intype` file used earlier in this chapter:

```
CASE=LOWER
TYPE employee
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

the user has chosen to specify the case for the OTT-generated C identifiers, and has provided a list of types which should be translated. In two of these types, naming conventions are specified.

The following is an example of what the `outtype` file might look like after running OTT:

```
CASE = LOWER
TYPE EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS AS ADDRESS
  VERSION = "$8.0"
  HFILE = demo.h
TYPE ITEM AS item
  VERSION = "$8.0"
  HFILE = demo.h
TYPE "Person" AS Person
  VERSION = "$8.0"
  HFILE = demo.h
TYPE PURCHASE_ORDER AS p_o
  VERSION = "$8.0"
  HFILE = demo.h
```

When examining the contents of the `outtype` file, you might discover types listed which were not included in the `intype` specification. For example, if the `intype` file only specified that the `person` type was to be translated

```
CASE = LOWER
TYPE PERSON
```

and the definition of the `person` type includes an attribute of type `address`, then the `outtype` file will include entries for both PERSON and ADDRESS. The `person` type cannot be translated completely without first translating `address`.

When the parameter TRANSITIVE has been set to TRUE (it is the default), OTT analyzes the types in the intype file for type dependencies before performing the translation, and translates other types as necessary.

Using OTT with OCI Applications

C header and implementation files that have been generated by OTT can be used by an OCI application that accesses objects in an Oracle server. The header file is incorporated into the OCI code with an `#include` statement.

Once the header file has been included, the OCI application can access and manipulate object data in the host language format.

Figure 14–1, "Using OTT with OCI" shows the steps involved in using OTT with the OCI for the simplest applications:

1. SQL is used to create type definitions in the database.

2. OTT generates a header file containing C representations of object types and named collection types. It also generates an implementation file, as named with the INITFILE option.

3. The application is written. User-written code in the OCI application declares and calls the INITFUNC function.

4. The header file is included in an OCI source code file.

5. The OCI application, including the implementation file generated by OTT, is compiled and linked with the OCI libraries.

6. The OCI executable is run against the Oracle server.

Figure 14–1   Using OTT with OCI
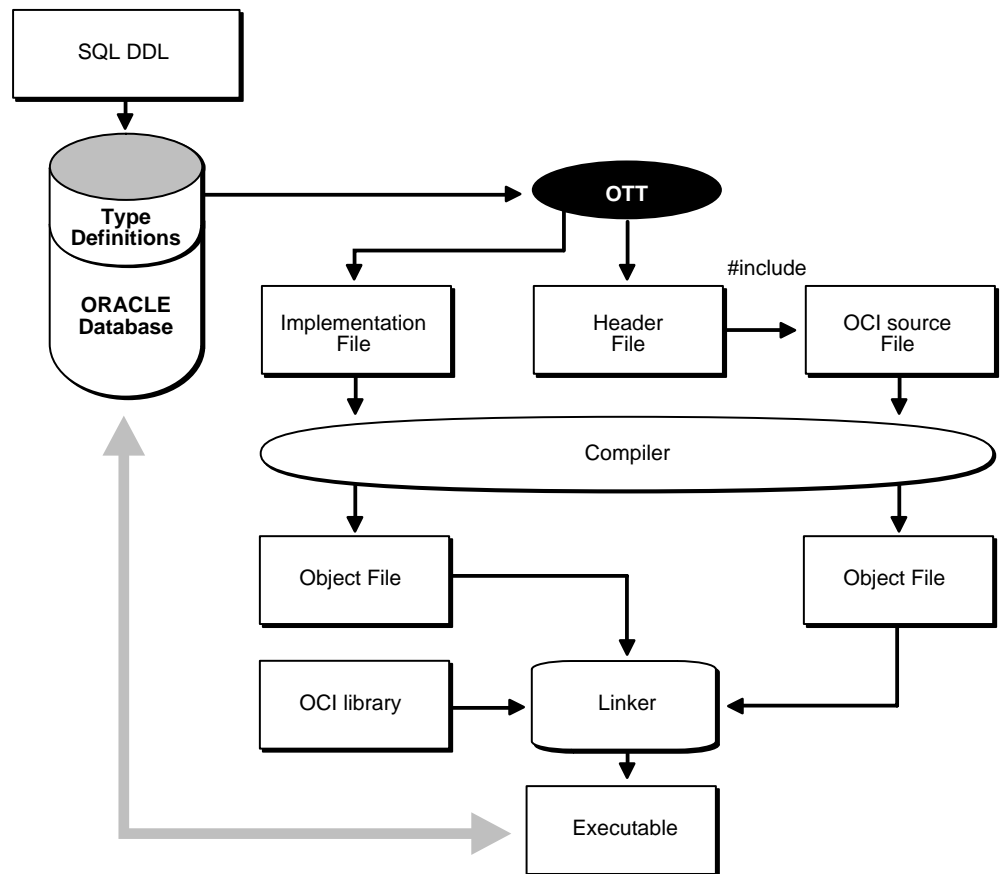
Accessing and Manipulating Objects with OCI

Within the application, the OCI program can perform bind and define operations using program variables declared to be of types which appear in OTT-generated header file.

For example, an application might fetch a REF to an object using a SQL SELECT statement and then pin that object using the appropriate OCI function. Once the object has been pinned, its attribute data can be accessed and manipulated with other OCI functions.

OCI includes a set of datatype mapping and manipulation functions which are specifically designed to work on attributes of object types and named collection types.

The following are examples of the available functions:

- `OCIStringSize()` gets the size of an **OCIString** string.

- `OCINumberAdd()` adds two **OCINumber** numbers together.

- `OCILobIsEqual()` compares two LOB locators for equality.

- `OCIRawPtr()` gets a pointer to an **OCIRaw** raw datatype.

- `OCICollAppend()` appends an element to a collection type (**OCIArray** or **OCITable**).

- `OCITableFirst()` returns the index for the first existing element of a nested table (**OCITable**).

- `OCIRefIsNull()` tests if a REF (**OCIRef**) is null

These functions are described in detail in other chapters of this guide.

Calling the Initialization Function

OTT generates a C initialization function if requested. The initialization function tells the environment, for each object type used in the program, which version of the type is used. You may specify a name for the initialization function when invoking OTT with the `INITFUNC` option, or may allow OTT to select a default name based on the name of the implementation file (`INITFILE`) containing the function.

The initialization function takes two arguments, an environment handle pointer and an error handle pointer. There is typically a single initialization function, but this is not required. If a program has several separately compiled pieces requiring different types, you may want to execute OTT separately for each piece requiring, for each piece, one initialization file, containing an initialization function.

After an environment handle is created by an explicit OCI object call, for example, by calling `OCIEnvCreate()`, you must also explicitly call the initialization functions. All the initialization functions must be called for each explicitly created environment handle. This gives each handle access to all the Oracle datatypes used in the entire program.

If an environment handle is implicitly created by embedded SQL statements, such as `EXEC SQL CONTEXT USE` and `EXEC SQL CONNECT`, the handle is initialized

implicitly, and the initialization functions need not be called. This is only relevant when Pro*C/C++ is being combined with OCI applications.

The following example shows an initialization function.

Given an `intype` file, `ex2c.typ`, containing

```
TYPE BREN.PERSON
TYPE BREN.ADDRESS
```

and the command line

```
ott userid=bren/bigkitty intype=ex2c outtype=ex2co hfile=ex2ch.h
initfile=ex2cv.c
```

OTT generates the following file, `ex2cv.c`:

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword ex2cv(OCIEnv *env, OCIError *err)
{
   sword status = OCITypeVTInit(env, err);
   if (status == OCI_SUCCESS)
      status = OCITypeVTInsert(env, err,
          "BREN", 5,
          "PERSON", 6,
          "$8.0", 4);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
           "BREN", 5,
           "ADDRESS", 7,
           "$8.0", 4);
    return status;
}
```

The function `ex2cv()` creates the type version table and inserts the types `BREN.PERSON` and `BREN.ADDRESS`.

If a program explicitly creates an environment handle, all the initialization functions must be generated, compiled, and linked, because they must be called for each explicitly created handle. If a program does not explicitly create any environment handles, initialization functions are not required.

A program that uses an OTT-generated header file must also use the initialization function generated at the same time. When a header file is generated by OTT and an

environment handle is explicitly created in the program, then the implementation file must also be compiled and linked into the executable.

Tasks of the Initialization Function

The C initialization function supplies version information about the types processed by OTT. It adds to the type-version table the name and version identifier of every OTT-processed object datatype.

The type-version table is used by Oracle's type manager to determine which version of a type a particular program uses. Different initialization functions generated by OTT at different times may add some of the same types to the type version table. When a type is added more than once, Oracle ensures the same version of the type is registered each time.

It is the OCI programmer's responsibility to declare a function prototype for the initialization function, and to call the function.

> **Note:** In the current release of Oracle, each type has only one version. Initialization of the type version table is required only for compatibility with future releases of Oracle.

OTT Reference

The behavior of the OTT is controlled by parameters which can appear on the OTT command line or in a CONFIG file. Certain parameters may also appear in the intype file.

This section provides detailed information about the following topics:

- OTT Command Line Syntax
- OTT Parameters
- Where OTT Parameters Can Appear
- Structure of the Intype File
- Nested Included File Generation
- SCHEMA_NAMES Usage
- Default Name Mapping
- OTT Restriction on File Name Comparison

The following conventions are used in this chapter to describe OTT syntax:

- Italic strings are variables or parameters to be supplied by the user.

- Strings in UPPERCASE are entered as shown, except that case is not significant.

- OTT keywords are listed in a lower-case monospaced font in examples and headings, but are printed in upper-case in text to make them more distinctive.

- Square brackets [...] enclose optional items.

- An ellipsis (...) immediately following an item (or items enclosed in brackets) means that the item can be repeated any number of times.

- Punctuation symbols other than those described above are entered as shown. These include '.', '@', etc.

OTT Command Line Syntax

The OTT command-line interface is used when explicitly invoking OTT to translate database types into C structs. This is always required when developing OCI applications that use objects.

An OTT command line statement consists of the keyword OTT, followed by a list of OTT parameters.

The parameters which can appear on an OTT command line statement are as follows:

```
[userid=username/password[@db_name]]

[intype=in_filename]

outtype=out_filename

code=C|ANSI_C|KR_C

[hfile=filename]

[errtype=filename]

[config=filename]

[initfile=filename]

[initfunc=filename]
```

```
[case=SAME|LOWER|UPPER|OPPOSITE]

[schema_name=ALWAYS|IF_NEEDED|FROM_INTYPE]

[transitive=TRUE|FALSE]
```

> **Note:** Generally, the order of the parameters following the OTT command does not matter, and only the OUTTYPE and CODE parameters are always required.

The HFILE parameter is almost always used. If omitted, HFILE must be specified individually for each type in the intype file. If OTT determines that a type not listed in the intype file must be translated, an error will be reported. Therefore, it is safe to omit the HFILE parameter only if the INTYPE file was previously generated as an OTT OUTTYPE file.

If the intype file is omitted, the entire schema will be translated. See the parameter descriptions in the following section for more information.

The following is an example of an OTT command line statement:

```
OTT userid=marc/cayman intype=in.typ outtype=out.typ code=c hfile=demo.h
errtype=demo.tls case=lower
```

Each of the OTT command line parameters is described in the following sections.

OTT Parameters

Enter parameters on the OTT command line using the following format:

```
parameter=value
```

where parameter is the literal parameter string and value is a valid parameter setting. The literal parameter string is not case sensitive.

Separate command-line parameters using either spaces or tabs.

Parameters can also appear within a configuration file, but, in that case, no whitespace is permitted within a line, and each parameter must appear on a separate line. Additionally, the parameters CASE, HFILE, INITFUNC, and INITFILE can appear in the intype file.

USERID

The USERID parameter specifies the Oracle username, password, and optional database name (Oracle Net Services database specification string). If the database name is omitted, the default database is assumed. The syntax of this parameter is:

userid=*username/password[@db_name]*

If this is the first parameter, "USERID=" may be omitted as shown here:

OTT *username/password...*

The USERID parameter is optional. If omitted, OTT automatically attempts to connect to the default database as user OPS$username, where username is the user's operating system user name.

INTYPE

The INTYPE parameter specifies the name of the file from which to read the list of object type specifications. OTT translates each type in the list.

The syntax for this parameter is

intype=*filename*

"INTYPE=" may be omitted if USERID and INTYPE are the first two parameters, in that order, and "USERID=" is omitted. If INTYPE is not specified, all types in the user's schema will be translated.

OTT *username/password filename...*

The intype file can be thought of as a makefile for type declarations. It lists the types for which C struct declarations are needed.

> **See Also:** The format of the intype file is described in section "Structure of the Intype File" on page 14-34

If the file name on the command line or in the intype file does not include an extension, a platform-specific extension such as "TYP" or ".typ" will be added.

OUTTYPE

The name of a file into which OTT will write type information for all the object datatypes it processes. This includes all types explicitly named in the intype file, and may include additional types that are translated because they are used in the

declarations of other types that need to be translated (if TRANSITIVE=TRUE). This file may be used as an `intype` file in a future invocation of OTT.

```
outtype=filename
```

If the INTYPE and OUTTYPE parameters refer to the same file, the new INTYPE information replaces the old information in the `intype` file. This provides a convenient way for the same `intype` file to be used repeatedly in the cycle of altering types, generating type declarations, editing source code, precompiling, compiling, and debugging.

OUTTYPE must be specified.

If the file name on the command line or in the `intype` file does not include an extension, a platform-specific extension such as "TYP" or ".typ" will be added.

CODE

This is the desired host language for OTT output, which may be specified as CODE=C, CODE=KR_C, or CODE=ANSI_C. "CODE=C" is equivalent to "CODE=ANSI_C".

```
CODE=C|KR_C|ANSI_C
```

There is no default value for this parameter; it must be supplied.

INITFILE

The INITFILE parameter specifies the name of the file where the OTT-generated initialization file is to be written. The initialization function will not be generated if this parameter is omitted.

For Pro*C/C++ programs, the INITFILE is not necessary, because the SQLLIB run-time library performs the necessary initializations. An OCI program user must compile and link the INITFILE file(s), and must call the initialization function(s) when an environment handle is created.

If the file name of an INITFILE on the command line or in the `intype` file does not include an extension, a platform-specific extension such as "C" or ".c" will be added.

```
initfile=filename
```

INITFUNC

The `INITFUNC` parameter is only used in OCI programs. It specifies the name of the initialization function generated by OTT. If this parameter is omitted, the name of the initialization function is derived from the name of the `INITFILE`.

`initfunc=`*`filename`*

HFILE

The name of the include (`.h`) file to be generated by OTT for the declarations of types that are mentioned in the `intype` file but whose include files are not specified there. This parameter is required unless the include file for each type is specified individually in the `intype` file. This parameter is also required if a type not mentioned in the `intype` file must be generated because other types require it, and these other types are declared in two or more different files, and `TRANSITIVE=TRUE`.

If the file name of an `HFILE` on the command line or in the `intype` file does not include an extension, a platform-specific extension such as "`H`" or "`.h`" will be added.

`hfile=`*`filename`*

CONFIG

The `CONFIG` parameter specifies the name of the OTT configuration file, which lists commonly used parameter specifications. Parameter specifications are also read from a system configuration file in a platform-dependent location. All remaining parameter specifications must appear on the command line, or in the `intype` file.

`config=`*`filename`*

> **Note:** `ACONFIG` parameter is not allowed in the `CONFIG` file.

ERRTYPE

If this parameter is supplied, a listing of the `intype` file is written to the `ERRTYPE` file, along with all informational and error messages. Informational and error messages are sent to the standard output whether or not `ERRTYPE` is specified.

Essentially, the `ERRTYPE` file is a copy of the `intype` file with error messages added. In most cases, an error message will include a pointer to the text which caused the error.

If the file name of an ERRTYPE on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as "TLS" or ".tls" will be added.

```
errtype=filename
```

CASE

This parameter affects the case of certain C identifiers generated by OTT. The possible values of CASE are SAME, LOWER, UPPER, and OPPOSITE. If CASE = SAME, the case of letters is not changed when converting database type and attribute names to C identifiers. If CASE=LOWER, all uppercase letters are converted to lowercase. If CASE=UPPER, all lowercase letters are converted to uppercase. If CASE=OPPOSITE, all uppercase letters are converted to lower-case, and vice-versa.

```
CASE=[SAME|LOWER|UPPER|OPPOSITE]
```

This option affects only those identifiers (attributes or types not explicitly listed) not mentioned in the intype file. Case conversion takes place after a legal identifier has been generated.

Note that the case of the C struct identifier for a type specifically mentioned in the INTYPE option is the same as its case in the intype file. For example, if the intype file includes the following line:

```
TYPE Worker
```

then the OTT generates

```
struct Worker {...};
```

On the other hand, if the intype file were written as

```
TYPE wOrKeR
```

the OTT generates

```
struct wOrKeR {...};
```

following the case of the intype file.

Case-insensitive SQL identifiers not mentioned in the intype file will appear in upper case if CASE=SAME, and in lower case if CASE=OPPOSITE. A SQL identifier is case-insensitive if it was not quoted when it was declared.

SCHEMA_NAMES

This option offers control in qualifying the database name of a type from the default schema with a schema name in the `outtype` file. The `outtype` file generated by OTT contains information about the types processed by OTT, including the type names.

> **See Also:** See "SCHEMA_NAMES Usage" on page 14-38 for further information

TRANSITIVE

Takes the values `TRUE` (the default) or `FALSE`. Indicates whether type dependencies not explicitly listed in the `intype` file are to be translated, or not.

If `TRANSITIVE=FALSE` is specified, then types not mentioned in the `intype` file are not generated, even if they were used as attribute types of other generated types.

Where OTT Parameters Can Appear

OTT parameters can appear on the command line, in a `CONFIG` file named on the command line, or both. Some parameters are also allowed in the `intype` file.

OTT is invoked as follows:

```
OTT username/password parameters
```

If one of the parameters on the command line is

```
config=filename
```

additional parameters are read from the configuration file `filename`.

In addition, parameters are also read from a default configuration file in a platform-dependent location. This file must exist, but can be empty. Parameters in a configuration file must appear one per line, with no whitespace on the line.

If OTT is executed without any arguments, an on-line parameter reference is displayed.

The types for OTT to translate are named in the file specified by the INTYPE parameter. The parameters CASE, INITFILE, INITFUNC, and HFILE may also appear in the `intype` file. `outtype` files generated by OTT include the CASE parameter, and include the INITFILE, and INITFUNC parameters if an

initialization file was generated. The `outtype` file specifies the `HFILE` individually for each type.

The case of the OTT command is platform-dependent.

Structure of the Intype File

The intype and outtype files list the types translated by OTT, and provide all the information needed to determine how a type or attribute name is translated to a legal C identifier. These files contain one or more type specifications. These files also may contain specifications of the following options:

- CASE
- HFILE
- INITFILE
- INITFUNC

If the `CASE`, `INITFILE`, or `INITFUNC` options are present, they must precede any type specifications. If these options appear both on the command line and in the intype file, the value on the command line is used.

> **See Also:** For an example of a simple user-defined `intype` file, and of the full `outtype` file that the OTT generates from it, see "The Outtype File" on page 14-21

Intype File Type Specifications

A type specification in the `INTYPE` names an object datatype that is to be translated. A type specification in the `outtype` file names an object datatype that has been translated,

```
TYPE employee
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS
TYPE PURCHASE_ORDER AS p_o
```

The structure of a type specification is as follows, where [] indicates optional inputs inside:

```
TYPE type_name [AS type_identifier]
[VERSION [=] version_string]
[HFILE [=] hfile_name]
[TRANSLATE{member_name [AS identifier]}...]
```

The syntax of `type_name` is:

`[schema_name.]type_name`

where *schema_name* is the name of the schema which owns the given object datatype, and `type_name` is the name of the type. The default schema is that of the user running OTT. The default database is the local database.

The components of a type specification are described below.

- *type_name* is the name of an Oracle object datatype.

- *type_identifier* is the C identifier used to represent the type. If omitted, the default name mapping algorithm will be used.

- *version_string* is the version string of the type which was used when the code was generated by a previous invocation of OTT. The version string is generated by OTT and written to the `outtype` file, which may later be used as the `intype` file when OTT is later executed. The version string does not affect OTT's operation, but will eventually be used to select which version of the object datatype should be used in the running program.

type_identifier is the C identifier used to represent the type. If omitted, the default type mapping algorithm will be used.

> **See Also:** "Default Name Mapping" on page 14-40

- *hfile_name* is the name of the header file in which the declarations of the corresponding struct or class appears or will appear. If *hfile_name* is omitted, the file named by the command-line HFILE parameter will be used if a declaration is generated.

- *member_name* is the name of an attribute (data member) which is to be translated to the following *identifier*.

- *identifier* is the C identifier used to represent the attribute in the user program. Identifiers may be specified in this way for any number of attributes. The default name mapping algorithm will be used for the attributes that are not mentioned.

An object datatype may need to be translated for one of two reasons:

- It appears in the `intype` file.

- It is required to declare another type that must be translated, and TRANSITIVE=TRUE.

If a type that is not mentioned explicitly is required by types declared in exactly one file, the translation of the required type is written to the same file(s) as the explicitly declared types that require it.

If a type that is not mentioned explicitly is required by types declared in two or more different files, the translation of the required type is written to the global HFILE file.

Nested Included File Generation

Every HFILE generated by OTT #includes other necessary files, and #defines a symbol constructed from the name of the file, which may be used to determine if the HFILE has already been included. Consider, for example, a database with the following types:

```
create type px1 AS OBJECT (col1 number, col2 integer);
create type px2 AS OBJECT (col1 px1);
create type px3 AS OBJECT (col1 px1);
```

where the intype file contains:

```
CASE=lower
type px1
  hfile tott95a.h
type px3
  hfile tott95b.h
```

If we invoke OTT with

```
ott scott/tiger tott95i.typ outtype=tott95o.typ code=c
```

then it will generate the two following header files.

File tott95b.h is:

```
#ifndef TOTT95B_ORACLE
#define TOTT95B_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#ifndef TOTT95A_ORACLE
#include "tott95a.h"
#endif
typedef OCIRef px3_ref;
struct px3
{
```

```
      struct px1 col1;
};
typedef struct px3 px3;
struct px3_ind
{
   OCIInd _atomic;
   struct px1_ind col1
};
typedef struct px3_ind px3_ind;
#endif
```

File tott95a.h is:

```
#ifndef TOTT95A_ORACLE
#define TOTT95A_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif
typedef OCIRef px1_ref;
struct px1
{
   OCINumber col1;
   OCINumber col2;
}
typedef struct px1 px1;
struct px1_ind
{
   OCIInd _atomic;
   OCIInd col1;
   OCIInd col2;
}
typedef struct px1_ind px1_ind;
#endif
```

In this file, the symbol TOTT95B_ORACLE is defined first so that the programmer may conditionally include tott95b.h without having to worry whether tott95b.h depends on the include file using the following construct:

```
#ifndef TOTT95B_ORACLE
#include "tott95b.h"
#endif
```

Using this technique, the programmer may include tott95b.h from some file, say foo.h, without having to know whether some other file included by foo.h also includes tott95b.h.

After the definition of the symbol `TOTT95B_ORACLE`, the file `oci.h` is `#included`. Every `HFILE` generated by OTT includes `oci.h`, which contains type and function declarations that the Pro*C/C++ or OCI programmer will find useful. This is the only case in which OTT uses angle brackets in a `#include`.

Next, the file `tott95a.h` is included. This file is included because it contains the declaration of "`struct px1`", which `tott95b.h` requires. When the user's `intype` file requests that type declarations be written to more than one file, OTT determines which other files each `HFILE` must include, and will generate the necessary `#includes`.

Note that OTT uses quotes in this `#include`. When a program including `tott95b.h` is compiled, the search for `tott95a.h` will begin where the source program was found, and will thereafter follow an implementation-defined search rule. If `tott95a.h` cannot be found in this way, a complete file name (e.g., a UNIX absolute pathname beginning with /) should be used in the `intype` file to specify the location of `tott95a.h`.

SCHEMA_NAMES Usage

This parameter affects whether the name of a type from the default schema to which OTT is connected is qualified with a schema name in the `outtype` file.

The name of a type from a schema other that the default schema is always qualified with a schema name in the `outtype` file.

The schema name, or its absence, determines in which schema the type is found during program execution.

There are three settings:

- `schema_names=ALWAYS` (default)

  All type names in the `outtype` file are qualified with a schema name.

- `schema_names=IF_NEEDED`

  The type names in the `OUTTYPE` file that belong to the default schema are not qualified with a schema name. As always, type names belonging to other schemas are qualified with the schema name.

- `schema_names=FROM_INTYPE`

  A type mentioned in the `intype` file is qualified with a schema name in the `OUTTYPE` file if, and only if, it was qualified with a schema name in the `intype` file. A type in the default schema that is not mentioned in the `intype` file but that has to be generated because of type dependencies will be written with a

schema name only if the first type encountered by OTT that depends on it was written with a schema name. However, a type that is not in the default schema to which OTT is connected will always be written with an explicit schema name.

The `outtype` file generated by OTT is an input parameter to Pro*C/C++. From the point of view of Pro*C/C++, it is the Pro*C/C++ `intype` file. This file matches database type names to C struct names. This information is used at run-time to make sure that the correct database type is selected into the struct. If a type appears with a schema name in the `outtype` file (Pro*C/C++ `intype` file), the type will be found in the named schema during program execution. If the type appears without a schema name, the type will be found in the default schema to which the program connects, which may be different from the default schema OTT used.

Example: Schema_Names Usage

If `SCHEMA_NAMES` is set to `FROM_INTYPE`, and the `intype` file reads:

```
TYPE Person
TYPE david.Dept
TYPE sam.Company
```

then the Pro*C/C++ application that uses the OTT-generated structs will use the types `sam.Company`, `david.Dept`, and `Person`. Using `Person` without a schema name refers to the `Person` type in the schema to which the application is connected.

If OTT and the application both connect to schema `david`, the application will use the same type (`david.Person`) that OTT used. If OTT connected to schema `david` but the application connects to schema `jana`, the application will use the type `jana.Person`. This behavior is appropriate only if the same "`CREATE TYPE Person`" statement has been executed in schema `david` and schema `jana`.

On the other hand, the application will use type `david.Dept` regardless of to which schema the application is connected. If this is the behavior you want, be sure to include schema names with your type names in the `intype` file.

In some cases, OTT translates a type that the user did not explicitly name. For example, consider the following SQL declarations:

```
CREATE TYPE Address AS OBJECT
( street    VARCHAR2(40),
  city      VARCHAR(30),
  state     CHAR(2),
  zip_code  CHAR(10) );
```

```
CREATE TYPE Person AS OBJECT
( name        CHAR(20),
  age         NUMBER,
  addr        ADDRESS );
```

Now suppose that OTT connects to schema david, SCHEMA_NAMES=FROM_INTYPE is specified, and the user's intype files include either

TYPE Person or TYPE david.Person

but do not mention the type david.Address, which is used as a nested object type in type david.Person. If "TYPE david.Person" appeared in the intype file, "TYPE david.Person" and "TYPE david.Address" will appear in the outtype file. If "Type Person" appeared in the intype file, "TYPE Person" and "TYPE Address" will appear in the outtype file.

If the david.Address type is embedded in several types translated by OTT, but is not explicitly mentioned in the intype file, the decision of whether to use a schema name is made the first time OTT encounters the embedded david.Address type. If, for some reason, the user wants type david.Address to have a schema name but does not want type Person to have one, the user should explicitly request

TYPE   david.Address

in the intype file.

The main point is that in the usual case in which each type is declared in a single schema, it is safest for the user to qualify all type names with schema names in the intype file.

Default Name Mapping

When OTT creates a C identifier name for an object type or attribute, it translates the name from the database character set to a legal C identifier. First, the name is translated from the database character set to the character set used by OTT. Next, if a translation of the resulting name is supplied in the intype file, that translation is used. Otherwise, OTT translates the name character-by-character to the compiler character set, applying the CASE option. The following describes this process in more detail.

When OTT reads the name of a database entity, the name is automatically translated from the database character set to the character set used by OTT. In order for OTT to read the name of the database entity successfully, all the characters of the name

must be found in the OTT character set, although a character may have different encodings in the two character sets.

The easiest way to guarantee that the character set used by OTT contains all the necessary characters is to make it the same as the database character set. Note, however, that the OTT character set must be a superset of the compiler character set. That is, if the compiler character set is 7-bit ASCII, the OTT character set must include 7-bit ASCII as a subset, and if the compiler character set is 7-bit EBCDIC, the OTT character set must include 7-bit EBCDIC as a subset. The user specifies the character set that OTT uses by setting the NLS_LANG environment variable, or by some other platform-specific mechanism.

Once OTT has read the name of a database entity, it translates the name from the character set used by OTT to the compiler's character set. If a translation of the name appears in the INTYPE file, OTT uses that translation.

Otherwise, OTT attempts to translate the name as follows:

1. First, if the OTT character set is a multibyte character set, all multibyte characters in the name that have single-byte equivalents are converted to those single-byte equivalents.

2. Next, the name is converted from the OTT character set to the compiler character set. The compiler character set is a single-byte character set such as US7ASCII.

3. Finally, the case of letters is set according to the CASE option in effect, and any character that is not legal in a C identifier, or that has no translation in the compiler character set, is replaced by an underscore. If at least one character is replaced by an underscore, OTT gives a warning message. If all the characters in a name are replaced by underscores, OTT gives an error message.

Character-by-character name translation does not alter underscores, digits, or single-byte letters that appear in the compiler character set, so legal C identifiers are not altered.

Name translation may, for example, translate accented single-byte characters such as "o" with an umlaut or "a" with an accent grave to "o" or "a", and may translate a multibyte letter to its single-byte equivalent. Name translation will typically fail if the name contains multibyte characters that lack single-byte equivalents. In this case, the user must specify name translations in the intype file.

OTT will not detect a naming clash caused by two or more database identifiers being mapped to the same C name, nor will it detect a naming problem where a database identifier is mapped to a C keyword.

OTT Restriction on File Name Comparison

Currently, the OTT determines if two files are the same by comparing the file names provided by the user on the command line or in the `intype` file. But one potential problem can occur when the OTT needs to know if two file names refer to the same file. For example, if the OTT-generated file `foo.h` requires a type declaration written to `foo1.h`, and another type declaration written to `/private/elias/foo1.h`, OTT should generate one `#include` if the two files are the same, and two `#includes` if the files are different. In practice, though, it would conclude that the two files are different, and would generate two **#include**s, as follows:

```
#ifndef FOO1_ORACLE
#include "foo1.h"
#endif
#ifndef FOO1_ORACLE
#include "/private/elias/foo1.h"
#endif
```

If `foo1.h` and `/private/elias/foo1.h` are different files, only the first one will be included. If `foo1.h` and `/private/elias/foo1.h` are the same file, a redundant `#include` will be written.

Therefore, if a file is mentioned several times on the command line or in the `intype` file, each mention of the file should use exactly the same file name.

Part III

OCI Reference

This part of the book contains the OCI function reference chapters:

See Also:

- For a a discussion of the OCI functions that apply to a Global Support environment, see the *Oracle9i Globalization and National Language Support Guide.*

- For a discussion of the OCI functions that apply to cartridge services, see the *Oracle9i Data Cartridge Developer's Guide.*

- For information about OCI calls pertaining to Advanced Queuing, see *Oracle9i Application Developer's Guide - Advanced Queuing.*

15

OCI Relational Functions

This chapter begins to describe the Oracle OCI relational functions for C. It includes information about calling OCI functions in your application, along with detailed descriptions of each function call. This chapter contains the following sections:

- Introduction to the Relational Functions
- Connect, Authorize, and Initialize Functions
- Handle and Descriptor Functions
- Bind, Define, and Describe Functions
- Statement Functions

Introduction to the Relational Functions

This chapter describes the OCI relational function calls. This chapter and the next, cover the functions in the basic OCI.

> **See Also:** For information about return codes and error handling, refer to the section "Error Handling" on page 2-31

Function Syntax

For each function, the following information is listed:

Purpose

A brief description of the action performed by the function.

Syntax

The function declaration.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below.

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

Comments

More detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Example

A complete or partial code example demonstrating the use of the function call being described. Not all function descriptions include an example.

Related Functions

A list of related function calls.

Calling OCI Functions

Unlike earlier versions of the OCI, in and after release 8, you cannot pass -1 for the string length parameter of a null-terminated string.

When you pass string lengths as parameters, do not include the null terminator byte in the length. The OCI does not expect strings to be null-terminated.

Buffer lengths that are OCI parameters are in bytes, except:

- the amount parameters in some LOB calls are in characters,
- when UTF-16 encoding of text is used in function parameters, the length is in character points.

Server Round-trips for LOB Functions

For a table showing the number of server round-trips required for individual OCI LOB functions, refer to Appendix C, "OCI Function Server Round-trips".

Connect, Authorize, and Initialize Functions

This section describes the OCI connect, authorize, and initialize functions.

Table 15–1   Connect, Authorize, and Initialize Functions

Function	Purpose
OCIConnectionPoolCreate() on page 15-5	Initializes the connection pool.
OCIConnectionPoolDestroy() on page 15-8	Destroys the connection pool.
OCIEnvCreate() on page 15-9	Creates and initializes an OCI environment
OCIEnvInit() on page 15-12	Initialize an environment handle
OCIInitialize() on page 15-14	Initialize OCI process environment
OCILogon() on page 15-18	Simplified single-session logon
OCILogon2() on page 15-20	Create a logon session in connection pooling mode.
OCIServerAttach() on page 15-22	Attach to a server; initialize server context handle
OCIServerDetach() on page 15-25	Detach from a server; uninitialize server context handle
OCISessionBegin() on page 15-26	Authenticate a user
OCISessionEnd() on page 15-30	Terminate a user session
OCITerminate() on page 15-31	Detaches from a shared memory subsystem.

OCIConnectionPoolCreate()

Purpose

Initializes the connection pool.

Syntax

```
sword OCIConnectionPoolCreate ( OCIEnv        *envhp,
                                OCIError      *errhp,
                                OCICPool      *poolhp,
                                OraText       **poolName,
                                sb4           *poolNameLen,
                                CONST OraText *dblink,
                                sb4           dblinkLen,
                                ub4           connMin,
                                ub4           connMax,
                                ub4           connIncr,
                                CONST OraText *poolUsername,
                                sb4           poolUserLen,
                                CONST OraText *poolPassword,
                                sb4           poolPassLen,
                                ub4           mode );
```

Parameters

envhp (IN)
A pointer to the environment where the connection pool is to be created

errhp (IN/OUT)
An error handle which can be passed to OCIErrorGet().

poolhp (IN)
An allocated pool handle.

poolName (OUT)
The name of the connection pool connected to.

poolNameLen (OUT)
The length of the string pointed to by poolName.

dblink (IN)
Specifies the database (server) to connect to.

dblinkLen (IN)
The length of the string pointed to by dblink.

connMin (IN)
Specifies the minimum number of connections in the connection pool. Valid values are 0 and above.

These number of connections are opened to the server by `OCIConnectionPoolCreate()`. After this, connections are opened only when necessary. Generally, it should be set to the number of concurrent statements the application is planning or expecting to run.

connMax (IN)
Specifies the maximum number of connections that can be opened to the database. Once this value is reached, no more connections are opened. Valid values are 1 and above.

connIncr (IN)
Allows the application to set the next increment for connections to be opened to the database if the current number of connections are less than `connMax`. Valid values are 0 and above.

poolUsername (IN)
Connection pooling requires an implicit primary session and this attribute provides a username for that session.

poolUserLen (IN)
The length of `poolUsername`.

poolPassword (IN)
The password for the username `poolUsername`.

poolPassLen (IN)
The length of `poolPassword`.

mode (IN)
The modes supported are

- OCI_DEFAULT

- OCI_CPOOL_REINITIALIZE.

Ordinarily, `OCIConnectionPoolCreate()` will be called with `mode` set to OCI_DEFAULT.

If you wish to change the pool attributes dynamically (for example: change the `connMin`, `connMax`, and `connIncr` parameters), call `OCIConnectionPoolCreate()` with `mode` set to OCI_CPOOL_REINITIALIZE. When this is done, the other parameters are ignored.

Comments

The OUT parameters *poolName* and *poolNameLen* will contain values to be used in subsequent `OCIServerAttach()` and `OCILogon2()` calls in place of the database name and the database name length arguments.

See Also: "Connection Pool Handle Attributes" on page A-21

Related Functions

`OCIConnectionPoolDestroy(), OCILogon2(), OCIServerAttach()`

OCIConnectionPoolDestroy()

Purpose

Destroys the connection pool.

Syntax

```
sword OCIConnectionPoolDestroy ( OCICPool     *poolhp,
                                 OCIError     *errhp,
                                 ub4          mode );
```

Parameters

poolhp (IN)
A pool handle for which a pool has been created.

errhp (IN/OUT)
An error handle which can be passed to OCIErrorGet().

mode (IN)
Currently, this function will support only the OCI_DEFAULT mode.

Related Functions

OCIConnectionPoolCreate()

OCIEnvCreate()

Purpose

Creates and initializes an environment for OCI functions to work under.

Syntax

```
sword OCIEnvCreate    ( OCIEnv        **envhpp,
                        ub4           mode,
                        CONST dvoid   *ctxp,
                        CONST dvoid   *(*malocfp)
                                        (dvoid *ctxp,
                                         size_t size),
                        CONST dvoid   *(*ralocfp)
                                        (dvoid *ctxp,
                                         dvoid *memptr,
                                         size_t newsize),
                        CONST void    (*mfreefp)
                                        (dvoid *ctxp,
                                         dvoid *memptr))
                        size_t    xtramemsz,
                        dvoid     **usrmempp );
```

Parameters

envhpp (OUT)

A pointer to an environment handle whose encoding setting is specified by *mode*. The setting will be inherited by statement handles derived from *envhpp*.

mode (IN)

Specifies initialization of the mode. Valid modes are:

- OCI_DEFAULT- the default value, which is non-UTF-16 encoding.

- OCI_THREADED - uses threaded environment. Internal data structures not exposed to the user are protected from concurrent accesses by multiple threads.

- OCI_OBJECT - uses object features.

- OCI_UTF16 - the environment handle and handles inherited from it assume UTF-16 encoding.

- OCI_SHARED - utilizes shared data structures.

- OCI_EVENTS - utilizes publish-subscribe notifications.

- OCI_NO_UCB - suppresses the calling of the dynamic callback routine *OCIEnvCallback*. The default behavior is to allow calling of *OCIEnvCallback* at the time that the environment is created.

    See Also: "Dynamic Callback Registrations" on page 9-28

- OCI_ENV_NO_MUTEX - no mutexing in this mode. All OCI calls done on the environment handle, or on handles derived from the environment handle, must be serialized.

ctxp (IN)
Specifies the user-defined context for the memory callback routines.

malocfp (IN)
Specifies the user-defined memory allocation function. If mode is OCI_THREADED, this memory allocation routine must be thread safe.

ctxp (IN)
Specifies the context pointer for the user-defined memory allocation function.

size (IN)
Specifies the size of memory to be allocated by the user-defined memory allocation function.

ralocfp (IN)
Specifies the user-defined memory re-allocation function. If the mode is OCI_THREADED, this memory allocation routine must be thread safe.

ctxp (IN)
Specifies the context pointer for the user-defined memory reallocation function.

memp (IN)
Pointer to memory block.

newsize (IN)
Specifies the new size of memory to be allocated

mfreefp (IN)
Specifies the user-defined memory free function. If mode is OCI_THREADED, this memory free routine must be thread-safe.

ctxp (IN)
Specifies the context pointer for the user-defined memory free function.

memptr (IN)
Pointer to memory to be freed

xtramemsz (IN)
Specifies the amount of user memory to be allocated for the duration of the environment.

usrmempp (OUT)
Returns a pointer to the user memory of size *xtramemsz* allocated by the call for the user.

Comments

This call creates an environment for all the OCI calls using the modes specified by the user.

> **Note:** This call should be invoked before any other OCI call and should be used instead of the `OCIInitialize()` and `OCIEnvInit()` calls. `OCIInitialize()` and `OCIEnvInit()` calls will be supported for backward compatibility.

This call is invoked before any other OCI calls, so it sets up any Unicode support in the environment handle, at the top-most level. The Unicode setting is made by the *mode* argument. Set *mode* to OCI_UTF16.

This call returns an environment handle which is then used by the remaining OCI functions. There can be multiple environments in OCI, each with its own environment modes. This function also performs any process level initialization if required by any mode. For example if the user wants to initialize an environment as OCI_THREADED, then all libraries that are used by OCI are also initialized in the threaded mode.

If you are writing a DLL or a shared library using OCI library then this call should definitely be used instead of `OCIInitialize()` and `OCIEnvInit()` call.

> **See Also:** For more information about the *xtramemsz* parameter and user memory allocation, refer to "User Memory Allocation" on page 2-14

Related Functions

`OCIHandleAlloc(),OCIHandleFree(),OCIEnvInit(),`
`OCIInitialize(),OCITerminate()`

OCIEnvInit()

Purpose

Allocates and initializes an OCI environment handle.

Syntax

```
sword OCIEnvInit ( OCIEnv    **envhpp,
                   ub4       mode,
                   size_t    xtramemsz,
                   dvoid     **usrmempp );
```

Parameters

envhpp (OUT)
A pointer to a handle to the environment.

mode (IN)
Specifies initialization of an environment mode. Valid modes are:

- OCI_DEFAULT

- OCI_NO_MUTEX

- OCI_ENV_NO_UCB

In OCI_DEFAULT mode, the OCI library always mutexes handles. In OCI_NO_MUTEX modes, there is no mutexing in this environment.

In OCI_NO_MUTEX mode, all OCI calls done on the environment handle, or on handles derived from the environment handle, must be serialized. This can be done by either doing your own mutexing or by having only one thread operating on the environment handle.

The OCI_ENV_NO_UCB mode is used to suppress the calling of the dynamic callback routine *OCIEnvCallback()* at environment initialization time. The default behavior is to allow such a call to be made.

> **See Also:** "Dynamic Callback Registrations" on page 9-28

xtramemsz (IN)
Specifies the amount of user memory to be allocated for the duration of the environment.

usrmempp (OUT)
Returns a pointer to the user memory of size *xtramemsz* allocated by the call for the user for the duration of the environment.

Comments

> **Note:** `OCIEnvCreate()` should be used instead of the `OCIInitialize()` and `OCIEnvInit()` calls. `OCIInitialize()` and `OCIEnvInit()` calls will be supported for backward compatibility.

This call allocates and initializes an OCI environment handle. No changes are done to an already initialized handle. If OCI_ERROR or OCI_SUCCESS_WITH_INFO is returned, the environment handle can be used to obtain ORACLE specific errors and diagnostics.

This call is processed locally, without a server round-trip.

The environment handle can be freed using `OCIHandleFree()`.

> **See Also:** For more information about the `xtramemsz` parameter and user memory allocation, refer to "User Memory Allocation" on page 2-14.

Related Functions

`OCIHandleAlloc()`,`OCIHandleFree()`,`OCIEnvCreate()`,`OCITerminate()`

OCIInitialize()

Purpose

Initializes the OCI process environment.

Syntax

```
sword OCIInitialize ( ub4          mode,
                      CONST dvoid  *ctxp,
                      CONST dvoid  *(*malocfp)
                                   (/* dvoid *ctxp,
                                       size_t size _*/),
                      CONST dvoid  *(*ralocfp)
                                   (/*_ dvoid *ctxp,
                                      dvoid *memptr,
                                      size_t newsize _*/),
                      CONST void   (*mfreefp)
                                   (/*_ dvoid *ctxp,
                                      dvoid *memptr _*/));
```

Parameters

mode (IN)
Specifies initialization of the mode. The valid modes are:

- OCI_DEFAULT - default mode.

- OCI_THREADED - threaded environment. In this mode, internal data structures not exposed to the user are protected from concurrent accesses by multiple threads.

- OCI_OBJECT - will use object features.

- OCI_SHARED - will utilize shared data structures.

- OCI_EVENTS - will utilize publish-subscribe notifications.

ctxp (IN)
User defined context for the memory call back routines.

malocfp (IN)
User-defined memory allocation function. If *mode* is OCI_THREADED, this memory allocation routine must be thread safe.

ctxp (IN/OUT)
Context pointer for the user-defined memory allocation function.

size (IN)

Size of memory to be allocated by the user-defined memory allocation function

ralocfp (IN)

User-defined memory re-allocation function. If *mode* is OCI_THREADED, this
memory allocation routine must be thread safe.

ctxp (IN/OUT)

Context pointer for the user-defined memory reallocation function.

memptr (IN/OUT)

Pointer to memory block

newsize (IN)

New size of memory to be allocated

mfreefp (IN)

User-defined memory free function. If *mode* is OCI_THREADED, this memory free
routine must be thread safe.

ctxp (IN/OUT)

Context pointer for the user-defined memory free function.

memptr (IN/OUT)

Pointer to memory to be freed

Comments

> **Note:** OCIEnvCreate() should be used instead of the
> OCIInitialize() and OCIEnvInit() calls.
> OCIInitialize() and OCIEnvInit() calls will be supported
> for backward compatibility.

This call initializes the OCI process environment. OCIInitialize() must be
invoked before any other OCI call.

This function provides the ability for the application to define its own memory
management functions through callbacks. If the application has defined such
functions (i.e., memory allocation, memory re-allocation, memory free), they should
be registered using the callback parameters in this function.

These memory callbacks are optional. If the application passes null values for the
memory callbacks in this function, the default process memory allocation
mechanism is used.

Shared Data Structures Mode

When a SQL statement is processed, certain underlying data is associated with the statement. This data includes information about statement text and bind data, as well as define and describe information for queries. This data remains the same from one execution of a statement to another, even if the statement is executed by different users.

When an OCI application is initialized in OCI_SHARED mode, common statement data is shared between multiple statement handles, thus providing memory savings for the application. This savings may be particularly valuable for applications which create multiple statement handles which execute the same SQL statement on different users' sessions, either on the same or multiple connections.

> **See Also:**
>
> - For more information, refer to "Shared Data Mode" on page 2-22.
>
> - For information about using the OCI to write multi-threaded applications, refer to "Thread Safety" on page 9-2.
>
> - For information about OCI programming with objects, refer to Chapter 10, "OCI Object-Relational Programming".

Example

The following statement shows an example of how to call `OCIInitialize()` in both threaded and object mode, with no user-defined memory functions:

```
OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT, (dvoid *)0,
     (dvoid * (*)()) 0, (dvoid * (*)()) 0,  (void (*)()) 0 );
```

Related Functions

OCIHandleAlloc(),OCIHandleFree(),OCIEnvCreate(),OCIEnvInit(),
OCITerminate()

OCILogoff()

Purpose

This function is used to terminate a connection and session created with `OCILogon()`.

Syntax

```
sword OCILogoff ( OCISvcCtx      *svchp
                  OCIError       *errhp );
```

Parameters

svchp (IN)
The service context handle which was used in the call to `OCILogon()`.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

Comments

This call is used to terminate a session and connection which were created with `OCILogon()`. This call implicitly deallocates the server, user session, and service context handles.

> **See Also:** For more information on logging on and off in an application, refer to the section "Application Initialization, Connection, and Session Creation" on page 2-26.

Related Functions

`OCILogon()`

OCILogon()

Purpose

This function is used to create a simple logon session.

Syntax

```
sword OCILogon ( OCIEnv          *envhp,
                 OCIError        *errhp,
                 OCISvcCtx       **svchp,
                 CONST OraText   *username,
                 ub4             uname_len,
                 CONST OraText   *password,
                 ub4             passwd_len,
                 CONST OraText   *dbname,
                 ub4             dbname_len );
```

Parameters

envhp (IN)
The OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

svchp (IN/OUT)
The service context pointer.

username (IN)
The username. Must be in UTF-16 encoding in OCI_UTF16 environment mode.

uname_len (IN)
The length of *username*, in number of bytes, regardless of the encoding.

password (IN)
The user's password. Must be in UTF-16 encoding in OCI_UTF16 environment mode.

passwd_len (IN)
The length of *password*, in number of bytes, regardless of the encoding.

dbname (IN)
The name of the database to connect to. Must be in UTF-16 encoding in OCI_UTF16 environment mode.

dbname_len (IN)

The length of *dbname*, in number of bytes, regardless of the encoding.

Comments

This function is used to create a simple logon session for an application.

> **Note:** Users requiring more complex sessions, such as TP monitor applications, should refer to the section "Application Initialization, Connection, and Session Creation" on page 2-26.

This call allocates the service context handles that are passed to it. This call also implicitly allocates server and user session handles associated with the session. These handles can be retrieved by calling `OCIAttrGet()` on the service context handle.

Related Functions

`OCILogoff()`

OCILogon2()

Purpose

This function is used to create a logon session in connection pooling mode.

Syntax

```
sword OCILogon2 ( OCIEnv          *envhp,
                  OCIError        *errhp,
                  OCISvcCtx       **svchp,
                  CONST OraText   *username,
                  ub4             uname_len,
                  CONST OraText   *password,
                  ub4             passwd_len,
                  CONST OraText   *dbname,
                  ub4             dbname_len );
                  ub4             mode );
```

Parameters

envhp (IN)
The OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

svchp (IN/OUT)
The address of a service context pointer. This will be returned with the service context handle, attached to the pool.

username (IN)
The username. Must be in UTF-16 encoding in OCI_UTF16 environment mode.

uname_len (IN)
The length of *username*, in number of bytes, regardless of the encoding.

password (IN)
The user's password. If it is null and *mode* is OCI_SPOOL, OCI, OCILogon2() assumes that the logon is for a proxy user. It implicitly creates a proxy connection in such a case, using the pool user to authenticate the proxy user. Must be in UTF-16 encoding in OCI_UTF16 environment mode.

passwd_len (IN)
The length of *password*, in number of bytes, regardless of the encoding.

dbname (IN)

The name of the database to connect to when *mode*=OCI_DEFAULT. Must be in UTF-16 encoding in OCI_UTF16 environment mode.

The name of the connection pool to connect to when *mode*=OCI_CPOOL. This must be the same as the *poolName* parameter of the connection pool created by OCIConnectionPoolCreate(). Must be in UTF-16 encoding in OCI_UTF16 environment mode.

dbname_len (IN)

The length of *dbname* when *mode*=OCI_DEFAULT, in number of bytes, regardless of the encoding.

The length of *poolName*, in number of bytes, regardless of the encoding, when *mode*=OCI_CPOOL.

mode (IN)

The values accepted are:

- OCI_CPOOL - use connection pooling,

- OCI_DEFAULT - no connection pooling, equivalent to calling OCILogon().

Comments

This call allocates the error and service context handles which are passed to it. This call also implicitly allocates server and authentication handles associated with the session. These handles can be retrieved by calling OCIAttrGet() on the service context handle.

This call assumes that OCIConnectionPoolCreate() has already been called, giving *poolName*, when connection pooling is in effect.

Related Functions

OCILogon()

OCIServerAttach()

Purpose

Creates an access path to a data source for OCI operations.

Syntax

```
sword OCIServerAttach ( OCIServer      *srvhp,
                        OCIError       *errhp,
                        CONST text     *dblink,
                        sb4            dblink_len,
                        ub4            mode );
```

Parameters

srvhp (IN/OUT)
An uninitialized server handle, which gets initialized by this call. Passing in an initialized server handle causes an error.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

dblink (IN)
Specifies the database server to use. This parameter points to a character string which specifies a connect string or a service point. If the connect string is null, then this call attaches to the default host. The string itself could be in UTF-16 or not, depending on mode or the setting in application's environment handle. The length of *dblink* is specified in *dblink_len*. The *dblink* pointer may be freed by the caller on return.

The name of the connection pool to connect to when *mode*=OCI_CPOOL. This must be the same as the *poolName* parameter of the connection pool created by `OCIConnectionPoolCreate()`. Must be in UTF-16 encoding in OCI_UTF16 environment mode.

dblink_len (IN)
The length of the string pointed to by *dblink*. For a valid connect string name or alias, *dblink_len* must be nonzero. Its value is in number of bytes.

The length of *poolName*, in number of bytes, regardless of the encoding, when *mode*=OCI_CPOOL.

mode (IN)
Specifies the various modes of operation. The valid modes are:

- OCI_DEFAULT. For encoding, this value tells the server handle to use the setting in the environment handle.

- OCI_CPOOL - use connection pooling,

Since an attached server handle can be set for any connection session handle, the *mode* value here does not contribute to any session handle.

Comments

This call is used to create an association between an OCI application and a particular server.

This call assumes that `OCIConnectionPoolCreate()` has already been called, giving *poolName*, when connection pooling is in effect.

This call initializes a server context handle, which must have been previously allocated with a call to `OCIHandleAlloc()`. The server context handle initialized by this call can be associated with a service context through a call to `OCIAttrSet()`. Once that association has been made, OCI operations can be performed against the server.

If an application is operating against multiple servers, multiple server context handles can be maintained. OCI operations are performed against whichever server context is currently associated with the service context.

When `OCIServerAttach()` is successfully completed, an Oracle shadow process is started. `OCISessionEnd()` and `OCIServerDetach()` should be called to clean up the Oracle shadow process. Otherwise, the shadow processes accumulate and cause the Unix system to run out of processes. If the database is restarted and there are not enough processes, the database may not startup.

Examples of Use of OCIServerAttach()

The following example demonstrates the use of `OCIServerAttach()`. This code segment allocates the server handle, makes the attach call, allocates the service context handle, and then sets the server context into it.

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
    OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
    OCI_HTYPE_SVCCTX, 0, (dvoid **) &tmp);
/* set attribute server context in the service context */
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) srvhp,
    (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
```

Related Functions

OCIServerDetach()

OCIServerDetach()

Purpose

Deletes an access to a data source for OCI operations.

Syntax

```
sword OCIServerDetach ( OCIServer    *srvhp,
                        OCIError     *errhp,
                        ub4          mode );
```

Parameters

srvhp (IN)
A handle to an initialized server context, which gets reset to uninitialized state. The handle is not de-allocated.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

mode (IN)
Specifies the various modes of operation. The only valid mode is OCI_DEFAULT for the default mode.

Comments

This call deletes an access to data source for OCI operations, which was established by a call to OCIServerAttach().

Related Functions

OCIServerAttach()

OCISessionBegin()

Purpose

Creates a user session and begins a user session for a given server.

Syntax

```
sword OCISessionBegin ( OCISvcCtx      *svchp,
                        OCIError       *errhp,
                        OCISession     *usrhp,
                        ub4            credt,
                        ub4            mode );
```

Parameters

svchp (IN)
A handle to a service context. There must be a valid server handle set in svchp.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

usrhp (IN/OUT)
A handle to an user session context, which is initialized by this call.

credt (IN)
Specifies the type of credentials to use for establishing the user session. Valid values for credt are:

- OCI_CRED_RDBMS - authenticate using a database username and password pair as credentials. The attributes OCI_ATTR_USERNAME and OCI_ATTR_PASSWORD should be set on the user session context before this call.

- OCI_CRED_EXT - authenticate using external credentials. No username or password is provided.

mode (IN)
Specifies the various modes of operation. Valid modes are:

- OCI_DEFAULT - in this mode, the user session context returned may only ever be set with the same server context specified in *svchp*. For encoding, the server handle uses the setting in the environment handle.

- OCI_MIGRATE - in this mode, the new user session context may be set in a service handle with a different server handle. This mode establishes the user session context. To create a migratable session, the service handle must already

be set with a non-migratable user session, which becomes the "creator" session of the migratable session. That is, a migratable session must have a non-migratable parent session.

- OCI_SYSDBA - in this mode, the user is authenticated for SYSDBA access.

- OCI_SYSOPER - in this mode, the user is authenticated for SYSOPER access.

- OCI_PRELIM_AUTH - this mode may only be used with OCI_SYSDBA or OCI_SYSOPER to authenticate for certain administration tasks.

Comments

The OCISessionBegin() call is used to authenticate a user against the server set in the service context handle.

> **Note:** Check for any errors returned when trying to start a session. For example, if the password for the account has expired, an ORA-28001 error is returned.

For release 8.1 or later, OCISessionBegin() must be called for any given server handle before requests can be made against it. OCISessionBegin() only supports authenticating the user for access to the Oracle server specified by the server handle in the service context. In other words, after OCIServerAttach() is called to initialize a server handle, OCISessionBegin() must be called to authenticate the user for that given server.

When using Unicode, when the *mode* or the environment handle has the appropriate setting, the username and password that have been set in the session handle *usrhp* should already be in Unicode. Before calling this function to start a session with a username and password, you must have called OCIAttrSet() to set these two Unicode strings into the session handle with corresponding length in bytes, because OCIAttrSet() only takes dvoid pointers. The string buffers then will be interpreted by OCISessionBegin().

When OCISessionBegin() is called for the first time for a given server handle, the user session may not be created in migratable (OCI_MIGRATE) mode.

After OCISessionBegin() has been called for a server handle, the application may call OCISessionBegin() again to initialize another user session handle with different (or the same) credentials and different (or the same) operation modes. If an application wants to authenticate a user in OCI_MIGRATE mode, the service handle must already be associated with a non-migratable user handle. The user ID

of that user handle becomes the ownership ID of the migratable user session. Every migratable session must have a non-migratable parent session.

If the OCI_MIGRATE mode is not specified, then the user session context can only be used with the same server handle set in *svchp*. If OCI_MIGRATE mode is specified, then the user authentication may be set with different server handles. However, the user session context may only be used with server handles which resolve to the same database instance. Security checking is done during session switching. A session can migrate to another process only if there is a non-migratable session currently connected to that process whose userid is the same as that of the creator's userid or its own userid.

OCI_SYSDBA, OCI_SYSOPER, and OCI_PRELIM_AUTH may only be used with a primary user session context.

To provide credentials for a call to OCISessionBegin(), one of two methods are supported. The first is to provide a valid username and password pair for database authentication in the user session handle passed to OCISessionBegin(). This involves using OCIAttrSet() to set the OCI_ATTR_USERNAME and OCI_ATTR_PASSWORD attributes on the user session handle. Then OCISessionBegin() is called with OCI_CRED_RDBMS.

> **Note:** When the user session handle is terminated using OCISessionEnd(), the username and password attributes remain unchanged and thus can be re-used in a future call to OCISessionBegin(). Otherwise, they must be reset to new values before the next OCISessionBegin() call.

The second type of credentials supported are external credentials. No attributes need to be set on the user session handle before calling OCISessionBegin(). The credential type is OCI_CRED_EXT. This is equivalent to the Oracle7 'connect /' syntax. If values have been set for OCI_ATTR_USERNAME and OCI_ATTR_PASSWORD, then these are ignored if OCI_CRED_EXT is used.

Another way of setting credentials is to use the session Id of an already authenticated user with the OCI_MIGSESSION attribute. This Id can be extracted from the session handle of an authenticated user using the OCIAttrGet() call.

Example

The following example demonstrates the use of OCISessionBegin(). This code segment allocates the user session handle, sets the username and password

attributes, calls `OCISessionBegin()`, and then sets the user session into the service context.

```
/* allocate a user session handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4)
    OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"jessica",
    (ub4)strlen("jessica"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"doogie",
    (ub4)strlen("doogie"), OCI_ATTR_PASSWORD, errhp);
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
    OCI_DEFAULT));
OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp,
    (ub4)0, OCI_ATTR_SESSION, errhp);
```

Related Functions

OCISessionEnd()

OCISessionEnd()

Purpose

Terminates a user session context created by OCISessionBegin()

Syntax

```
sword OCISessionEnd ( OCISvcCtx      *svchp,
                      OCIError       *errhp,
                      OCISession     *usrhp,
                      ub4            mode );
```

Parameters

svchp (IN/OUT)
The service context handle. There must be a valid server handle and user session handle associated with *svchp*.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

usrhp (IN)
De-authenticate this user. If this parameter is passed as null, the user in the service context handle is de-authenticated.

mode (IN)
The only valid mode is OCI_DEFAULT.

Comments

The user security context associated with the service context is invalidated by this call. Storage for the user session context is not freed. The transaction specified by the service context is implicitly committed. The transaction handle, if explicitly allocated, may be freed if not being used. Resources allocated on the server for this user are freed. The user session handle may be reused in a new call to OCISessionBegin().

Related Functions

OCISessionBegin()

OCITerminate()

Purpose

Detaches the process from the shared memory subsystem and releases the shared memory.

Syntax

```
sword OCITerminate ( ub4    mode);
```

Parameters

mode (IN)
Call-specific mode. Valid value:

- OCI_DEFAULT - executes the default call

Comments

OCITerminate() should be called only once for each process and is the counterpart of OCIInitialize() call. The call will try to detach the process from the shared memory subsystem and shut it down. It also performs additional process cleanup operations. When two or more processes connecting to the same shared memory are calling OCITerminate() simultaneously, the fastest one will release the shared memory subsystem completely and the slower ones will have to abort.

Related Functions

OCIInitialize()

Handle and Descriptor Functions

This section describes the OCI handle and descriptor functions.

Table 15–2   Handle and Descriptor Functions

Function	Purpose
OCIAttrGet() on page 15-33	Get the attributes of a handle
OCIAttrSet() on page 15-35	Set an attribute of a handle or descriptor
OCIDescriptorAlloc() on page 15-37	Allocate and initialize a descriptor or LOB locator
OCIDescriptorFree() on page 15-39	Free a previously allocated descriptor
OCIHandleAlloc() on page 15-41	Allocate and initialize a handle
OCIHandleFree() on page 15-44	Free a previously allocated handle
OCIParamGet() on page 15-46	Get a parameter descriptor
OCIParamSet() on page 15-48	Set parameter descriptor in COR handle

OCIAttrGet()

Purpose

This call is used to get a particular attribute of a handle.

Syntax

```
sword OCIAttrGet ( CONST dvoid    *trgthndlp,
                   ub4            trghndltyp,
                   dvoid          *attributep,
                   ub4            *sizep,
                   ub4            attrtype,
                   OCIError       *errhp );
```

Parameters

trgthndlp (IN)
Pointer to a handle type. The actual handle can be a statement handle, a session handle, etc. When this call is used to get encoding, users are allowed to check against either an environment or statement handle.

trghndltyp (IN)
The handle type. Valid types are:

- OCI_DTYPE_PARAM, for a parameter descriptor

- OCI_HTYPE_STMT, for a statement handle

- Any handle type in Table 2–1, "OCI Handle Types".

attributep (OUT)
Pointer to the storage for an attribute value. In OCI_UTF16 environment mode, the value of a string attribute will be returned as a UTF-16 string.

sizep (OUT)
The size of the attribute value, always in bytes because *attributep* is a **dvoid** pointer. This can be passed as null for most attributes because the sizes of non-string attributes are already known by the OCI library. For **text*** parameters, a pointer to a **ub4** must be passed in to get the length of the string.

attrtype (IN)
The type of attribute being retrieved. The types are listed in this document at:

> **See Also:** See Appendix A, "Handle and Descriptor Attributes", for a list of handle types and their readable attributes

errhp (IN/OUT)

An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

Comments

This call is used to get a particular attribute of a handle. OCI_DTYPE_PARAM is used to do implicit and explicit describes. The parameter descriptor is also used in direct path loading. For implicit describes, the parameter descriptor has the column description for each select list. For explicit describes, the parameter descriptor has the describe information for each schema object we are trying to describe. If the top-level parameter descriptor has an attribute which is itself a descriptor, use OCI_ATTR_PARAM as the attribute type in the subsequent call to `OCIAttrGet()`.

> **See Also:** For illustrative code fragments, see "Examples Using OCIDescribeAny()" on page 6-23 and "Describing Select-List Items" on page 4-12

Use this call to get the Unicode information in an environment or statement handle.

A function closely related to `OCIAttrGet()` is `OCIDescribeAny()`, which is a generic describe call that describes existing schema objects: tables, views, synonyms, procedures, functions, packages, sequences, and types. As a result of this call, the describe handle is populated with the object-specific attributes which can be obtained through an `OCIAttrGet()` call.

Then an `OCIParamGet()` on the describe handle returns a parameter descriptor for a specified position. Parameter positions begin with 1. Calling `OCIAttrGet()` on the parameter descriptor returns the specific attributes of a stored procedure or function parameter or a table column descriptor as the case may be. These subsequent calls do not need an extra round-trip to the server because the entire schema object description is cached on the client side by `OCIDescribeAny()`. Calling `OCIAttrGet()` on the describe handle can also return the total number of positions.

In UTF-16 mode, particularly when executing a loop, try to reuse the same pointer variable corresponding to an attribute and copy the contents to local variables after `OCIAttrGet()` is called. If multiple pointers are used for the same attribute, a memory leak can occur.

Related Functions

OCIAttrSet()

OCIAttrSet()

Purpose

This call is used to set a particular attribute of a handle or a descriptor.

Syntax

```
sword OCIAttrSet ( dvoid       *trgthndlp,
                   ub4         trghndltyp,
                   dvoid       *attributep,
                   ub4         size,
                   ub4         attrtype,
                   OCIError    *errhp );
```

Parameters

trgthndlp (IN/OUT)
Pointer to a handle type whose attribute gets modified.

trghndltyp (IN/OUT)
The handle type.

attributep (IN)
Pointer to an attribute value. The attribute value is copied into the target handle. If the attribute value is a pointer, then only the pointer is copied, not the contents of the pointer. String attributes must be in UTF-16 in OCI_UTF16 environment.

size (IN)
The size of an attribute value. This can be passed in as 0 for most attributes as the size is already known by the OCI library. For **text*** attributes, a **ub4** must be passed in set to the length of the string in bytes, regardless of encoding.

attrtype (IN)
The type of attribute being set:

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

Comments

See Appendix A, "Handle and Descriptor Attributes", for a list of handle types and their writable attributes.

Example

The following code sample demonstrates OCIAttrSet() being used several times near the beginning of an application.

```
int main()
{
OCIEnv *envhp;
OCIServer *srvhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCISession *usrhp;

OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT, (dvoid *)0,
    (dvoid * (*)()) 0,(dvoid * (*)()) 0,  (void (*)()) 0 );
OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
    0, (dvoid **) &tmp);
OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 0, (dvoid **) &tmp  );
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4)
    OCI_HTYPE_ERROR, 0, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
    OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp,
    (ub4) OCI_HTYPE_SVCCTX, , (dvoid **) &tmp);
  /* set attribute server context in the service context */
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) srvhp,
    (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
/* allocate a user session handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp,
    (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"sherry",
    (ub4)strlen("sherry"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"penfield",
    (ub4)strlen("penfield"), OCI_ATTR_PASSWORD, errhp);
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
    OCI_DEFAULT));
OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp,
    (ub4)0, OCI_ATTR_SESSION, errhp);
```

Related Functions

OCIAttrGet()

OCIDescriptorAlloc()

Purpose

Allocates storage to hold descriptors or LOB locators.

Syntax

```
sword OCIDescriptorAlloc ( CONST dvoid    *parenth,
                           dvoid          **descpp,
                           ub4            type,
                           size_t         xtramem_sz,
                           dvoid          **usrmempp);
```

Parameters

parenth (IN)
An environment handle.

descpp (OUT)
Returns a descriptor or LOB locator of desired type.

type (IN)
Specifies the type of descriptor or LOB locator to be allocated:

- OCI_DTYPE_SNAP - specifies generation of snapshot descriptor of C type **OCISnapshot**

- OCI_DTYPE_LOB - specifies generation of a LOB value type locator (for a BLOB or CLOB) of C type **OCILobLocator**

- OCI_DTYPE_FILE - specifies generation of a FILE value type locator of C type **OCILobLocator**.

- OCI_DTYPE_ROWID - specifies generation of a ROWID descriptor of C type **OCIRowid**.

- OCI_DTYPE_DATE - specifies generation of an ANSI DATE descriptor of C type **OCIDateTime**

- OCI_DTYPE_TIMESTAMP - specifies generation of a TIMESTAMP descriptor of C type **OCIDateTime**

- OCI_DTYPE_TIMESTAMP_TZ - specifies generation of a TIMESTAMP WITH TIME ZONE descriptor of C type **OCIDateTime**

- OCI_DTYPE_TIMESTAMP_LTZ - specifies generation of a TIMESTAMP WITH LOCAL TIME ZONE descriptor of C type **OCIDateTime**

- OCI_DTYPE_INTERVAL_YM - specifies generation of an INTERVAL YEAR TO MONTH descriptor of C type **OCIInterval**

- OCI_DTYPE_INTERVAL_DS - specifies generation of an INTERVAL DAY TO SECOND descriptor of C type **OCIInterval**

- OCI_DTYPE_COMPLEXOBJECTCOMP - specifies generation of a complex object retrieval descriptor of C type **OCIComplexObjectComp**.

- OCI_DTYPE_AQENQ_OPTIONS - specifies generation of an Advanced Queuing enqueue options descriptor of C type **OCIAQEnqOptions**.

- OCI_DTYPE_AQDEQ_OPTIONS - specifies generation of an Advanced Queuing dequeue options descriptor of C type **OCIAQDeqOptions**.

- OCI_DTYPE_AQMSG_PROPERTIES - specifies generation of an Advanced Queuing message properties descriptor of C type **OCIAQMsgProperties**.

- OCI_DTYPE_AQAGENT - specifies generation of an Advanced Queuing agent descriptor of C type **OCIAQAgent**.

xtramem_sz (IN)
Specifies an amount of user memory to be allocated for use by the application for the lifetime of the descriptor.

usrmempp (OUT)
Returns a pointer to the user memory of size *xtramem_sz* allocated by the call for the user for the lifetime of the descriptor.

Comments

Returns a pointer to an allocated and initialized descriptor, corresponding to the type specified in *type*. A non-null descriptor or LOB locator is returned on success. No diagnostics are available on error.

This call returns OCI_SUCCESS if successful, or OCI_INVALID_HANDLE if an out-of-memory error occurs.

> **See Also:** For more information about the *xtramem_sz* parameter and user memory allocation, refer to "User Memory Allocation" on page 2-14

Related Functions

OCIDescriptorFree()

OCIDescriptorFree()

Purpose

Deallocates a previously allocated descriptor.

Syntax

```
sword OCIDescriptorFree ( dvoid     *descp,
                          ub4       type );
```

Parameters

descp (IN)
An allocated descriptor.

type (IN)
Specifies the type of storage to be freed. The specific types are:

- OCI_DTYPE_SNAP - snapshot descriptor

- OCI_DTYPE_LOB - a LOB value type descriptor

- OCI_DTYPE_FILE - a FILE value type descriptor

- OCI_DTYPE_ROWID - a ROWID descriptor

- OCI_DTYPE_DATE - an ANSI DATE descriptor

- OCI_DTYPE_PARAM - a parameter descriptor

- OCI_DTYPE_TIMESTAMP - a TIMESTAMP descriptor

- OCI_DTYPE_TIMESTAMP_TZ - a TIMESTAMP WITH TIME ZONE descriptor

- OCI_DTYPE_TIMESTAMP_LTZ - a TIMESTAMP WITH LOCAL TIME ZONE descriptor

- OCI_DTYPE_INTERVAL_YM - an INTERVAL YEAR TO MONTH descriptor

- CI_DTYPE_INTERVAL_DS - an INTERVAL DAY TO SECOND descriptor

- OCI_DTYPE_COMPLEXOBJECTCOMP - a complex object retrieval descriptor

- OCI_DTYPE_AQENQ_OPTIONS - an AQ enqueue options descriptor

- OCI_DTYPE_AQDEQ_OPTIONS - an AQ dequeue options descriptor

- OCI_DTYPE_AQMSG_PROPERTIES - an AQ message properties descriptor

- OCI_DTYPE_AQAGENT - an AQ agent descriptor

Comments

This call frees storage associated with a descriptor. Returns OCI_SUCCESS or OCI_INVALID_HANDLE. All descriptors may be explicitly deallocated, however the OCI will deallocate a descriptor if the environment handle is deallocated.

Related Functions

OCIDescriptorAlloc()

OCIHandleAlloc()

Purpose

This call returns a pointer to an allocated and initialized handle.

Syntax

```
sword OCIHandleAlloc ( CONST dvoid    *parenth,
                       dvoid          **hndlpp,
                       ub4            type,
                       size_t         xtramem_sz,
                       dvoid          **usrmempp );
```

Parameters

parenth (IN)
An environment handle.

hndlpp (OUT)
Returns a handle.

type (IN)
Specifies the type of handle to be allocated. The allowed types are:

- OCI_HTYPE_CPOOL - specifies generation of a connection pooling handle of C type **OCICPool**

- OCI_HTYPE_ENV - specifies generation of an environment handle of C type **OCIEnv**

- OCI_HTYPE_ERROR - specifies generation of an error report handle of C type **OCIError**

- OCI_HTYPE_SVCCTX - specifies generation of a service context handle of C type **OCISvcCtx**

- OCI_HTYPE_STMT - specifies generation of a statement (application request) handle of C type **OCIStmt**

- OCI_HTYPE_DESCRIBE - specifies generation of a select list description handle of C type **OCIDescribe**

- OCI_HTYPE_SERVER - specifies generation of a server context handle of C type **OCIServer**

- OCI_HTYPE_SESSION - specifies generation of a user session handle of C type **OCISession**

- OCI_HTYPE_TRANS - specifies generation of a transaction context handle of C type **OCITrans**

- OCI_HTYPE_COMPLEXOBJECT - specifies generation of a complex object retrieval handle of C type **OCIComplexObject**

- OCI_HTYPE_SECURITY - specifies generation of a security handle of C type **OCISecurity**

- OCI_HTYPE_SUBSCR - specifies a generation of a subscription handle of C type **OCISubscription**

- OCI_HTYPE_DIRPATH_CTX - specifies a generation of a direct path context handle of C type **OCIDirPathCtx**

- OCI_HTYPE_DIRPATH_COLUMN_ARRAY - specifies a generation of a direct path column array handle of C type **OCIDirPathColArray**

- OCI_HTYPE_DIRPATH_STREAM - specifies a generation of a direct path stream handle of C type **OCIDirPathStream**

- OCI_HTYPE_PROCESS - specifies a generation of a process handle of C type **OCIProcess**

xtramem_sz (IN)
Specifies an amount of user memory to be allocated.

usrmempp (OUT)
Returns a pointer to the user memory of size $xtramem\_sz$ allocated by the call for the user.

Comments

Returns a pointer to an allocated and initialized handle, corresponding to the type specified in $type$. A non-null handle is returned on success. All handles are allocated with respect to an environment handle which is passed in as a parent handle.

No diagnostics are available on error. This call returns OCI_SUCCESS if successful, or OCI_INVALID_HANDLE if an error occurs.

Handles must be allocated using OCIHandleAlloc() before they can be passed into an OCI call.

To allocate and initialize an environment handle, call OCIEnvInit().

> **See Also:** For more information about using the $xtramem\_sz$ parameter for user memory allocation, refer to "User Memory Allocation" on page 2-14

Example

The following sample code shows `OCIHandleAlloc()` being used to allocate a variety of handles at the beginning of an application:

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4)
      OCI_HTYPE_ERROR, 0, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
      OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
      OCI_HTYPE_SVCCTX, 0, (dvoid **) &tmp);
```

Related Functions

OCIHandleFree(), OCIEnvInit()

OCIHandleFree()

Purpose

This call explicitly deallocates a handle.

Syntax

```
sword OCIHandleFree ( dvoid     *hndlp,
                      ub4        type );
```

Parameters

hndlp (IN)
A handle allocated by OCIHandleAlloc().

type (IN)
Specifies the type of storage to be freed. The specific types are:

- OCI_HTYPE_CPOOL - a connection pool handle

- OCI_HTYPE_ENV - an environment handle

- OCI_HTYPE_ERROR - an error report handle

- OCI_HTYPE_SVCCTX - a service context handle

- OCI_HTYPE_STMT - a statement (application request) handle

- OCI_HTYPE_DESCRIBE - a select list description handle

- OCI_HTYPE_SERVER - a server handle

- OCI_HTYPE_SESSION - a user session handle

- OCI_HTYPE_TRANS - a transaction handle

- OCI_HTYPE_COMPLEXOBJECT - a complex object retrieval handle

- OCI_HTYPE_SECURITY - a security handle

- OCI_HTYPE_SUBSCR - a subscription handle

- OCI_HTYPE_DIRPATH_CTX - direct path context handle

- OCI_HTYPE_DIRPATH_COLUMN_ARRAY - direct path column array handle

- OCI_HTYPE_DIRPATH_STREAM - direct path stream handle

- OCI_HTYPE_PROCESS - process handle

Comments

This call frees up storage associated with a handle, corresponding to the type specified in the `type` parameter.

This call returns either OCI_SUCCESS or OCI_INVALID_HANDLE.

All handles may be explicitly deallocated. The OCI will deallocate a child handle if the parent is deallocated.

Related Functions

OCIHandleAlloc(), OCIEnvInit()

OCIParamGet()

Purpose

Returns a descriptor of a parameter specified by position in the describe handle or statement handle.

Syntax

```
sword OCIParamGet ( CONST dvoid        *hndlp,
                    ub4               htype,
                    OCIError          *errhp,
                    dvoid             **parmdpp,
                    ub4               pos );
```

Parameters

hndlp (IN)
A statement handle or describe handle. The OCIParamGet() function will return a parameter descriptor for this handle.

htype (IN)
The type of the handle passed in the *hndlp* parameter. Valid types are:

- OCI_DTYPE_PARAM, for a parameter descriptor

- OCI_HTYPE_COMPLEXOBJECT, for a complex object retrieval handle

- OCI_HTYPE_STMT, for a statement handle

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

parmdpp (OUT)
A descriptor of the parameter at the position given in the *pos* parameter, of handle type OCI_DTYPE_PARAM.

pos (IN)
Position number in the statement handle or describe handle. A parameter descriptor will be returned for this position.

> **Note:** OCI_ERROR is returned if there are no parameter descriptors for this position.

Comments

This call returns a descriptor of a parameter specified by position in the describe handle or statement handle. Parameter descriptors are always allocated internally by the OCI library. They can be freed using `OCIDescriptorFree()`. For example, if you fetch the same column metadata for every execution of a statement, then the program will leak memory unless you explicitly free the parameter descriptor between each call to `OCIParamGet()`.

> **See Also:**   See Appendix A, "Handle and Descriptor Attributes", for more detailed information about parameter descriptor attributes.

Related Functions

`OCIAttrGet()`, `OCIAttrSet()`, `OCIParamSet()`, `OCIDescriptorFree()`

OCIParamSet()

Purpose

Used to set a complex object retrieval (COR) descriptor into a COR handle.

Syntax

```
sword OCIParamSet ( dvoid         *hndlp,
                    ub4           htype,
                    OCIError      *errhp,
                    CONST dvoid   *dscp,
                    ub4           dtyp,
                    ub4           pos );
```

Parameters

hndlp (IN/OUT)
Handle pointer.

htype (IN)
Handle type.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

dscp (IN)
Complex object retrieval descriptor pointer.

dtyp (IN)
Descriptor type. The descriptor type for a COR descriptor is OCI_DTYPE_COMPLEXOBJECTCOMP.

pos (IN)
Position number.

Comments

The COR handle must have been previously allocated using OCIHandleAlloc(), and the descriptor must have been previously allocated using OCIDescriptorAlloc(). Attributes of the descriptor are set using OCIAttrSet().

> **See Also:** For more information about complex object retrieval, see "Complex Object Retrieval" on page 10-22.

Related Functions

OCIParamGet()

Bind, Define, and Describe Functions

This section describes the bind, define, and describe functions.

Table 15–3   Bind, Define, and Describe Functions

Function	Purpose
OCIBindArrayOfStruct() on page 15-51	Set skip parameters for static array bind
OCIBindByName() on page 15-52	Bind by name
OCIBindByPos() on page 15-57	Bind by position
OCIBindDynamic() on page 15-62	Sets additional attributes after bind with OCI_DATA_AT_EXEC mode
OCIBindObject() on page 15-66	Set additional attributes for bind of named data type
OCIDefineArrayOfStruct() on page 15-68	Set additional attributes for static array define
OCIDefineByPos() on page 15-69	Define an output variable association
OCIDefineDynamic() on page 15-73	Sets additional attributes for define in OCI_DYNAMIC_FETCH mode
OCIDefineObject on page 15-76	Set additional attributes for define of named data type
OCIDescribeAny() on page 15-78	Describe existing schema objects
OCIStmtGetBindInfo() on page 15-81	Get bind and indicator variable names and handle

OCIBindArrayOfStruct()

Purpose

This call sets up the skip parameters for a static array bind.

Syntax

```
sword OCIBindArrayOfStruct ( OCIBind     *bindp,
                             OCIError    *errhp,
                             ub4         pvskip,
                             ub4         indskip,
                             ub4         alskip,
                             ub4         rcskip );
```

Parameters

bindp (IN/OUT)
The handle to a bind structure.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

pvskip (IN)
Skip parameter for the next data value.

indskip (IN)
Skip parameter for the next indicator value or structure.

alskip (IN)
Skip parameter for the next actual length value.

rcskip (IN)
Skip parameter for the next column-level return code value.

Comments

This call sets up the skip parameters necessary for a static array bind. It follows a call to OCIBindByName() or OCIBindByPos(). The bind handle returned by that initial bind call is used as a parameter for the OCIBindArrayOfStruct() call.

> **See Also:** For information about skip parameters, see the section "Binding and Defining Arrays of Structures" on page 5-26.

Related Functions

OCIBindByName(), OCIBindByPos()

OCIBindByName()

Purpose

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

Syntax

```
sword OCIBindByName ( OCIStmt         *stmtp,
                      OCIBind         **bindpp,
                      OCIError        *errhp,
                      CONST text      *placeholder,
                      sb4             placeh_len,
                      dvoid           *valuep,
                      sb4             value_sz,
                      ub2             dty,
                      dvoid           *indp,
                      ub2             *alenp,
                      ub2             *rcodep,
                      ub4             maxarr_len,
                      ub4             *curelep,
                      ub4             mode );
```

Parameters

stmtp (IN/OUT)
The statement handle to the SQL or PL/SQL statement being processed.

bindpp (IN/OUT)
A pointer to save the pointer of a bind handle which is implicitly allocated by this call. The bind handle maintains all the bind information for this particular input value. The default encoding for the call depends on the UTF-16 setting in *stmtp* unless the *mode* parameter has a different value. The handle is freed implicitly when the statement handle is deallocated. On input, the value of the pointer must be null or a valid bind handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

placeholder (IN)
The placeholder, specified by its name, which maps to a variable in the statement associated with the statement handle. The encoding of *placeholder* should always be consistent with that of the environment. That is, if the statement is

prepared in UTF-16, so is the placeholder. As a string type parameter, it should be cast as (**text ***) and terminated with NULL.

placeh_len (IN)
The length of the name specified in *placeholder*, in number of bytes regardless of the encoding.

valuep (IN/OUT)
The pointer to a data value or an array of data values of the type specified in the *dty* parameter. This data could be a UTF-16 (formerly known as UCS-2) string, if an OCIAttrSet() function has been called to set OCI_ATTR_CHARSET_ID as OCI_UTF16ID or the deprecated OCI_UCS2ID. OCI_UTF16ID is the new designation for OCI_UCS2ID.

Furthermore, as pointed out for OCIStmtPrepare(), the default encoding for the string type *valuep* will be UTF-16 if the environment mode is set as OCI_UTF16, unless users call OCIAttrSet() to manually reset the character set for the bind handle.

> **See Also:** Refer to OCI_ATTR_CHARSET_ID on page A-32.

An array of data values can be specified for mapping into a PL/SQL table or for providing data for SQL multiple-row operations. When an array of bind values is provided, this is called an array bind in OCI terms.

For SQLT_NTY or SQLT_REF binds, the *valuep* parameter is ignored. The pointers to OUT buffers are set in the pgvpp parameter initialized by OCIBindObject().

value_sz(IN)
The size in bytes of the data value pointed to by dvoid pointer *valuep*. Although the bind buffer valuep could be of string type, the length is measured in number of bytes because the pointer passed down is of (**dvoid ***) type. In the case of an array bind, this is the maximum size of any element possible with the actual sizes being specified in the *alenp* parameter.

For descriptors, locators, or REFs, whose size is unknown to client applications use the size of the structure you are passing in; sizeof(OCILobLocator *).

dty (IN)
The data type of the value(s) being bound. Named data types (SQLT_NTY) and REFs (SQLT_REF) are valid only if the application has been initialized in object mode. For named data types, or REFs, additional calls must be made with the bind handle to set up the datatype-specific attributes.

indp (IN/OUT)
Pointer to an indicator variable or array. For all data types except SQLT_NTY, this is a pointer to sb2 or an array of sb2s.

For SQLT_NTY, this pointer is ignored and the actual pointer to the indicator structure or an array of indicator structures is initialized in a subsequent call to `OCIBindObject()`. This parameter is ignored for dynamic binds.

> **See Also:** "Indicator Variables" on page 2-36

alenp (IN/OUT)
Pointer to array of actual lengths of array elements. Each element in *alenp* is the length of the data in the corresponding element in the bind value array before and after the execute. The length should be in bytes for strings passed in as a text type.. This parameter is ignored for dynamic binds.

rcodep (OUT)
Pointer to array of column level return codes. This parameter is ignored for dynamic binds.

maxarr_len (IN)
The maximum possible number of elements of type *dty* in a PL/SQL binds. This parameter is not required for non-PL/SQL binds. If *maxarr_len* is nonzero, then either `OCIBindDynamic()` or `OCIBindArrayOfStruct()` can be invoked to set up additional bind attributes.

curelep (IN/OUT)
A pointer to the actual number of elements. This parameter is only required for PL/SQL binds.

mode (IN)
To maintain coding consistency, theoretically, this parameter can take all three possible values used by `OCIStmtPrepare()`. Since the encoding of bind variables should always be same as that of the statement containing this variable, an error will be raised if the user specify an encoding other than that of the statement. So the recommended setting for mode is OCI_DEFAULT, which will make the bind variable have the same encoding as its statement.

The valid modes are:

- OCI_DEFAULT - default mode. The statement handle *stmtp* uses whatever is specified by its parent environment handle.

- OCI_DATA_AT_EXEC - When this mode is selected, the *value_sz* parameter defines the maximum size of the data that can ever be provided at runtime. The

application must be ready to provide the OCI library runtime IN data buffers at any time and any number of times. Runtime data is provided in one of these two ways:

- Callbacks using a user-defined function which must be registered with a subsequent call to `OCIBindDynamic()`.

- A polling mechanism using calls supplied by the OCI. This mode is assumed if no callbacks are defined.

  See Also: For more information about using the OCI_DATA_AT_EXEC mode, see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44.

When mode is set to OCI_DATA_AT_EXEC, do not provide values for *valuep*, *indp*, *alenp*, and *rcodep* in the main call. Pass zeroes for *indp* and *alenp*. Provide the values through the callback function registered using `OCIBindDynamic()`.

When the allocated buffers are not required any more, they should be freed by the client.

Comments

This call is used to perform a basic bind operation. The bind creates an association between the address of a program variable and a placeholder in a SQL statement or PL/SQL block. The bind call also specifies the type of data which is being bound, and may also indicate the method by which data will be provided at runtime.

Encoding is determined by either the bind handle using the setting in the statement handle as default, or you can override the setting by specifying the *mode* parameter explicitly.

This function also implicitly allocates the bind handle indicated by the *bindpp* parameter. If a non-null pointer is passed in `**bindpp`, the OCI assumes that this points to a valid handle that has been previously allocated with a call to `OCIHandleAlloc()` or `OCIBindByName()`.

Data in an OCI application can be bound to placeholders statically or dynamically. Binding is *static* when all the IN bind data and the OUT bind buffers are well-defined just before the execute. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execute time to the client library. Dynamic binding is indicated by setting the *mode* parameter of this call to OCI_DATA_AT_EXEC.

> **See Also:** For more information about dynamic binding, see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44

Both `OCIBindByName()` and `OCIBindByPos()` take as a parameter a bind handle, which is implicitly allocated by the bind call A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being utilized, `OCIBindArrayOfStruct()` must be called to set up the necessary skip parameters.

- If data is being provided dynamically at runtime, and the application will be using user-defined callback functions, `OCIBindDynamic()` must be called to register the callbacks.

- If lengths in `alenp` greater than 64Kbytes are required, use `OCIBindDynamic()`.

- If a named data type is being bound, `OCIBindObject()` must be called to specify additional necessary information.

- If a statement with RETURNING clause is used, a call to `OCIBindDynamic()` must follow this call.

Related Functions

`OCIBindDynamic()`, `OCIBindObject()`, `OCIBindArrayOfStruct()`

OCIBindByPos()

Purpose

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

Syntax

```
sword OCIBindByPos ( OCIStmt      *stmtp,
                     OCIBind      **bindpp,
                     OCIError     *errhp,
                     ub4          position,
                     dvoid        *valuep,
                     sb4          value_sz,
                     ub2          dty,
                     dvoid        *indp,
                     ub2          *alenp,
                     ub2          *rcodep,
                     ub4          maxarr_len,
                     ub4          *curelep,
                     ub4          mode );
```

Parameters

stmtp (IN/OUT)
The statement handle to the SQL or PL/SQL statement being processed.

bindpp (IN/OUT)
An address of a bind handle which is implicitly allocated by this call. The bind handle maintains all the bind information for this particular input value. The handle is freed implicitly when the statement handle is deallocated. On input, the value of the pointer must be null or a valid bind handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

position (IN)
The placeholder attributes are specified by position if `OCIBindByPos()` is being called.

valuep (IN/OUT)
An address of a data value or an array of data values of the type specified in the `dty` parameter. An array of data values can be specified for mapping into a PL/SQL

table or for providing data for SQL multiple-row operations. When an array of bind values is provided, this is called an array bind in OCI terms.

For SQLT_NTY or SQLT_REF binds, the *valuep* parameter is ignored. The pointers to OUT buffers are set in the *pgvpp* parameter initialized by OCIBindObject().

If the OCI_ATTR_CHARSET_ID attribute is set to OCI_UTF16ID (replaces the deprecated OCI_UCS2ID, which is retained for backward compatibility), all data passed to and received with the corresponding bind call is assumed to be in UTF-16 encoding.

> **See Also:** OCI_ATTR_CHARSET_ID on page A-32.

value_sz (IN)
The size of a data value. In the case of an array bind, this is the maximum size of any element possible with the actual sizes being specified in the *alenp* parameter.

For descriptors, locators, or REFs, whose size is unknown to client applications use the size of the structure you are passing in; e.g., sizeof (OCILobLocator *).

dty (IN)
The data type of the value(s) being bound. Named data types (SQLT_NTY) and REFs (SQLT_REF) are valid only if the application has been initialized in object mode. For named data types, or REFs, additional calls must be made with the bind handle to set up the datatype-specific attributes.

indp (IN/OUT)
Pointer to an indicator variable or array. For all data types, this is a pointer to sb2 or an array of sb2s. The only exception is SQLT_NTY, when this pointer is ignored and the actual pointer to the indicator structure or an array of indicator structures is initialized by OCIBindObject(). Ignored for dynamic binds.

> **See Also:** See the section "Indicator Variables" on page 2-36

alenp (IN/OUT)
Pointer to array of actual lengths of array elements. Each element in *alenp* is the length (in bytes, unless the data in *valuep* is in Unicode, when it is in codepoints) of the data in the corresponding element in the bind value array before and after the execute. This parameter is ignored for dynamic binds.

Note: If *alenp* is less than *value_sz*, data will be skipped.

---

rcodep (OUT)
Pointer to array of column level return codes. This parameter is ignored for dynamic binds.

maxarr_len (IN)
The maximum possible number of elements of type *dty* in a PL/SQL binds. This parameter is not required for non-PL/SQL binds. If *maxarr_len* is nonzero, then either `OCIBindDynamic()` or `OCIBindArrayOfStruct()` can be invoked to set up additional bind attributes.

curelep (IN/OUT)
A pointer to the actual number of elements. This parameter is only required for PL/SQL binds.

mode (IN)
The valid modes for this parameter are:

OCI_DEFAULT - This is default mode.

OCI_DATA_AT_EXEC - When this mode is selected, the *value_sz* parameter defines the maximum size of the data that can ever be provided at runtime. The application must be ready to provide the OCI library runtime IN data buffers at any time and any number of times. Runtime data is provided in one of the two ways:

- Callbacks using a user-defined function which must be registered with a subsequent call to `OCIBindDynamic()`.

- A polling mechanism using calls supplied by the OCI. This mode is assumed if no callbacks are define.

    See Also: For more information about using the OCI_DATA_AT_EXEC mode, see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44.

When mode is set to OCI_DATA_AT_EXEC, do not provide values for *valuep*, *indp*, *alenp*, and *rcodep* in the main call. Pass zeroes for *indp* and *alenp*. Provide the values through the callback function registered using `OCIBindDynamic()`.

When the allocated buffers are not required any more, they should be freed by the client.

Comments

This call is used to perform a basic bind operation. The bind creates an association between the address of a program variable and a placeholder in a SQL statement or PL/SQL block. The bind call also specifies the type of data which is being bound, and may also indicate the method by which data will be provided at runtime.

This function also implicitly allocates the bind handle indicated by the *bindpp* parameter. If a non-null pointer is passed in ***bindpp*, the OCI assumes that this points to a valid handle that has been previously allocated with a call to `OCIHandleAlloc()` or `OCIBindByPos()`.

Data in an OCI application can be bound to placeholders statically or dynamically. Binding is *static* when all the IN bind data and the OUT bind buffers are well-defined just before the execute. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execute time to the client library. Dynamic binding is indicated by setting the *mode* parameter of this call to OCI_DATA_AT_EXEC.

> **See Also:** For more information about dynamic binding, see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44.

Both `OCIBindByName()` and `OCIBindByPos()` take as a parameter a bind handle, which is implicitly allocated by the bind call A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being utilized, `OCIBindArrayOfStruct()` must be called to set up the necessary skip parameters.

- If data is being provided dynamically at runtime, and the application will be using user-defined callback functions, `OCIBindDynamic()` must be called to register the callbacks.

- If lengths in *alenp* greater than 64Kbytes are required, use `OCIBindDynamic()`.

- If a named data type is being bound, `OCIBindObject()` must be called to specify additional necessary information.

- If a statement with RETURNING clause is used, a call to `OCIBindDynamic()` must follow this call.

Related Functions

`OCIBindDynamic()`, `OCIBindObject()`, `OCIBindArrayOfStruct()`

OCIBindDynamic()

Purpose

This call is used to register user callbacks for dynamic data allocation.

Syntax

```
sword OCIBindDynamic ( OCIBind      *bindp,
                       OCIError     *errhp,
                       dvoid        *ictxp,
                       OCICallbackInBind          (icbfp)(/*_
                               dvoid        *ictxp,
                               OCIBind      *bindp,
                               ub4          iter,
                               ub4          index,
                               dvoid        **bufpp,
                               ub4          *alenp,
                               ub1          *piecep,
                               dvoid        **indpp */),
                               dvoid        *octxp,
                       OCICallbackOutBind         (ocbfp)(/*_
                               dvoid        *octxp,
                               OCIBind      *bindp,
                               ub4          iter,
                               ub4          index,
                               dvoid        **bufpp,
                               ub4          **alenpp,
                               ub1          *piecep,
                               dvoid        **indpp,
                               ub2          **rcodepp _*/)   );
```

Parameters

bindp (IN/OUT)
A bind handle returned by a call to `OCIBindByName()` or `OCIBindByPos()`.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

ictxp (IN)
The context pointer required by the call back function *icbfp*.

icbfp (IN)
The callback function which returns a pointer to the IN bind value or piece at run time. The callback takes in the following parameters:

ictxp (IN/OUT)

The context pointer for this callback function.

bindp (IN)

The bind handle passed in to uniquely identify this bind variable.

iter (IN)

0-based execute iteration value.

index (IN)

Index of the current array, for an array bind in PL/SQL. For SQL it is the row index. The value is 0-based and not greater than *curelep* parameter of the bind call.

bufpp (OUT)

The pointer to the buffer or storage. For descriptors, **bufpp* contains a pointer to the descriptor. For example if you define

```
OCILOBLocator    *lobp;
```

then you would set **bufpp* to *lobp* not **lobp*.

For REFs, pass the address of the ref; i.e., pass &my_ref for **bufpp*.

If the OCI_ATTR_CHARSET_ID attribute is set to OCI_UTF16ID (replaces the deprecated OCI_UCS2ID, which is retained for backward compatibility), all data passed to and received with the corresponding bind call is assumed to be in UTF-16 encoding.

> **See Also:** OCI_ATTR_CHARSET_ID on page A-32.

alenp (OUT)

A pointer to a storage for OCI to fill in the size of the bind value/piece after it has been read. For descriptors, pass the size of the pointer to the descriptor; e.g., `sizeof(OCILobLocator *)`.

piecep (OUT)

Which piece of the bind value. This can be one of the following values OCI_ONE_PIECE, OCI_FIRST_PIECE, OCI_NEXT_PIECE and OCI_LAST_PIECE. For datatypes that do not support piecewise operations, you must pass OCI_ONE_PIECE or an error will be generated.

indp (OUT)

Contains the indicator value. This is a pointer to either an sb2 value or a pointer to an indicator structure for binding named data types.

octxp (IN)
The context pointer required by the callback function *ocbfp*.

ocbfp (IN)
The callback function which returns a pointer to the OUT bind value or piece at run time. The callback takes in the following parameters:

octxp (IN/OUT)
The context pointer for this call back function.

bindp (IN)
The bind handle passed in to uniquely identify this bind variable.

iter (IN)
0-based execute iteration value.

index (IN)
For PL/SQL index of the current array, for an array bind. For SQL, the index is the row number in the current iteration. It is 0-based, and must not be greater than *curelep* parameter of the bind call.

bufpp (OUT)
A pointer to a buffer to write the bind value/piece.

If the OCI_ATTR_CHARSET_ID attribute is set to OCI_UTF16ID (replaces the deprecated OCI_UCS2ID, which is retained for backward compatibility), all data passed to and received with the corresponding bind call is assumed to be in UTF-16 encoding. For more information, refer to OCI_ATTR_CHARSET_ID on page A-32.

alenpp (IN/OUT)
A pointer to a storage for OCI to fill in the size of the bind value/piece after it has been read. It is in bytes except for Unicode encoding (if the OCI_ATTR_CHARSET_ID attribute is set to OCI_UTF16ID), when it is in codepoints.

piecep (IN/OUT)
Returns a piece value from the callback (application) to Oracle, as follows:

- **IN** - The value can be OCI_ONE_PIECE or OCI_NEXT_PIECE.

- **OUT** - Depends on the IN value:

    If IN value is OCI_ONE_PIECE, then OUT value can be OCI_ONE_PIECE or OCI_FIRST_PIECE

    If IN value is OCI_NEXT_PIECE then OUT value can be OCI_NEXT_PIECE or OCI_LAST_PIECE

indpp (OUT)

> Returns a pointer to contain the indicator value which either an sb2 value or a pointer to an indicator structure for named data types.

rcodepp (OUT)

> Returns a pointer to contains the return code.

Comments

This call is used to register user-defined callback functions for providing or receiving data if OCI_DATA_AT_EXEC mode was specified in a previous call to `OCIBindByName()` or `OCIBindByPos()`.

The callback function pointers must return OCI_CONTINUE if it the call is successful. Any return code other than OCI_CONTINUE signals that the client wishes to abort processing immediately.

> **See Also:**   For more information about the OCI_DATA_AT_EXEC mode, see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44.

When passing the address of a storage area, make sure that the storage area will exist even after the application returns from the callback. This means that you should not allocate such storage on the stack.

Related Functions

`OCIBindByName()`, `OCIBindByPos()`

OCIBindObject()

Purpose

This function sets up additional attributes which are required for a named data type (object) bind.

Syntax

```
sword OCIBindObject ( OCIBind          *bindp,
                      OCIError         *errhp,
                      CONST OCIType    *type,
                      dvoid            **pgvpp,
                      ub4              *pvszsp,
                      dvoid            **indpp,
                      ub4              *indszp, );
```

Parameters

bindp (IN/OUT)
The bind handle returned by the call to `OCIBindByName()` or `OCIBindByPos()`.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

type (IN)
Points to the TDO which describes the type of the program variable being bound. Retrieved by calling `OCITypeByName()`. Optional for REFs in SQL, but required for REFs in PL/SQL.

pgvpp (IN/OUT)
Address of the program variable buffer. For an array, *pgvpp* points to an array of addresses. When the bind variable is also an OUT variable, the OUT Named Data Type value or REF is allocated in the Object Cache, and a REF is returned.

pgvpp is ignored if the OCI_DATA_AT_EXEC mode is set. Then the Named Data Type buffers are requested at runtime. For static array binds, skip factors may be specified using the `OCIBindArrayOfStruct()` call. The skip factors are used to compute the address of the next pointer to the value, the indicator structure and their sizes.

pvszsp (OUT) [optional]
Points to the size of the program variable. The size of the named data type is not required on input. For an array, *pvszsp* is an array of **ub4**s. On return, for OUT bind variables, this points to size(s) of the Named Data Types and REFs received.

pvszsp is ignored if the OCI_DATA_AT_EXEC mode is set. Then the size of the buffer is taken at runtime.

indpp (IN/OUT) [optional]
Address of the program variable buffer containing the parallel indicator structure. For an array, points to an array of pointers. When the bind variable is also an OUT bind variable, memory is allocated in the object cache, to store the OUT indicator values. At the end of the execute when all OUT values have been received, *indpp* points to the pointer(s) to these newly allocated indicator structure(s). Required only for SQLT_NTY binds. *indpp* is ignored if the OCI_DATA_AT_EXEC mode is set. Then the indicator is requested at runtime.

indszp (IN/OUT)
Points to the size of the IN indicator structure program variable. For an array, it is an array of **sb2**s. On return for OUT bind variables, this points to size(s) of the received OUT indicator structures. *indszp* is ignored if the OCI_DATA_AT_EXEC mode is set. Then the indicator size is requested at runtime.

Comments

This function sets up additional attributes which binding a named data type or a REF. An error will be returned if this function is called when the OCI environment has been initialized in non-object mode.

This call takes as a parameter a type descriptor object (TDO) of datatype **OCIType** for the named data type being defined. The TDO can be retrieved with a call to OCITypeByName().

If the OCI_DATA_AT_EXEC mode was specified in OCIBindByName() or OCIBindByPos(), the pointers to the IN buffers are obtained either using the callback *icbfp* registered in the OCIBindDynamic() call or by the OCIStmtSetPieceInfo() call.

The buffers are dynamically allocated for the OUT data. The pointers to these buffers are returned either by

- calling ocbfp() registered by the OCIBindDynamic()

- or, by setting the pointer to the buffer in the buffer passed in by OCIStmtSetPieceInfo() called when OCIStmtExecute() returned OCI_NEED_DATA.

 The memory of these client library-allocated buffers must be freed when not in use anymore by using the OCIObjectFree() call.

Related Functions

OCIBindByName(), OCIBindByPos()

OCIDefineArrayOfStruct()

Purpose

This call specifies additional attributes necessary for a static array define, used in an array of structures (multi-row, multi-column) fetch.

Syntax

```
sword OCIDefineArrayOfStruct ( OCIDefine    *defnp,
                               OCIError     *errhp,
                               ub4          pvskip,
                               ub4          indskip,
                               ub4          rlskip,
                               ub4          rcskip );
```

Parameters

defnp (IN/OUT)
The handle to the define structure which was returned by a call to
OCIDefineByPos().

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

pvskip (IN)
Skip parameter for the next data value.

indskip (IN)
Skip parameter for the next indicator location.

rlskip (IN)
Skip parameter for the next return length value.

rcskip (IN)
Skip parameter for the next return code.

Comments

This call follows a call to OCIDefineByPos(). If the application is binding an array of structures involving objects, it must call OCIDefineObject() first, and then call OCIDefineArrayOfStruct().

> **See Also:** "Skip Parameters" on page 5-27.

Related Functions

OCIDefineByPos(), OCIDefineObject

OCIDefineByPos()

Purpose

Associates an item in a select-list with the type and output data buffer.

Syntax

```
sword OCIDefineByPos ( OCIStmt     *stmtp,
                       OCIDefine   **defnpp,
                       OCIError    *errhp,
                       ub4         position,
                       dvoid       *valuep,
                       sb4         value_sz,
                       ub2         dty,
                       dvoid       *indp,
                       ub2         *rlenp,
                       ub2         *rcodep,
                       ub4         mode );
```

Parameters

stmtp (IN/OUT)
A handle to the requested SQL query operation.

defnpp (IN/OUT)
A pointer to a pointer to a define handle. If this parameter is passed as null, this call implicitly allocates the define handle. In the case of a redefine, a non-null handle can be passed in this parameter. This handle is used to store the define information for this column.

> **Note:** The user must keep track of this pointer. If a second call to OCIDefineByPos() is made for the same column position, there is no guarantee that the same pointer is returned.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

position (IN)
The position of this value in the select list. Positions are 1-based and are numbered from left to right. For example, in the SELECT statement

```
SELECT empno, ssn, mgrno FROM employees;
```

empno is at position 1, ssn is at position 2, and mgrno is at position 3.

valuep (IN/OUT)
A pointer to a buffer or an array of buffers of the type specified in the *dty* parameter. A number of buffers can be specified when results for more than one row are desired in a single fetch call.

value_sz (IN)
The size of each *valuep* buffer in bytes. If the data is stored internally in VARCHAR2 format, the number of characters desired, if different from the buffer size in bytes, may be additionally specified by using OCIAttrSet().

In a multibyte conversion environment, a truncation error will be generated if the number of bytes specified is insufficient to handle the number of characters desired.

If the OCI_ATTR_CHARSET_ID attribute is set to OCI_UTF16ID (replaces the deprecated OCI_UCS2ID, which is retained for backward compatibility), all data passed to and received with the corresponding define call is assumed to be in UTF-16 encoding.

> **See Also:** OCI_ATTR_CHARSET_ID on page A-35

dty (IN)
The data type. Named data type (SQLT_NTY) and REF (SQLT_REF) are valid only if the environment has been initialized in object mode.

SQLT_CHAR and SQLT_LNG can be specified for CLOB columns, and SQLT_BIN sand SQLT_LBI for BLOB columns.

> **See Also:** For a listing of datatype codes and values, refer to Chapter 3, "Datatypes"

indp (IN)
pointer to an indicator variable or array. For scalar data types, pointer to **sb2** or an array of **sb2**s. Ignored for SQLT_NTY defines. For SQLT_NTY defines, a pointer to a named data type indicator structure or an array of named data type indicator structures is associated by a subsequent OCIDefineObject() call.

> **See Also:** "Indicator Variables" on page 2-36

rlenp (IN/OUT)
Pointer to array of length of data fetched. Each element in *rlenp* is the length of the data (in bytes, unless the data in *valuep* is in Unicode, when it is in codepoints) in the corresponding element in the row after the fetch.

rcodep (OUT)
Pointer to array of column-level return codes

mode (IN)
The valid modes are:

- OCI_DEFAULT - This is the default mode.

- OCI_DYNAMIC_FETCH - For applications requiring dynamically allocated data at the time of fetch, this mode must be used. The user may additionally call `OCIDefineDynamic()` to set up a callback function that will be invoked to receive the dynamically allocated buffers and. The *valuep* and *value_sz* parameters are ignored in this mode.

Comments

This call defines an output buffer which will receive data retrieved from Oracle. The define is a local step which is necessary when a SELECT statement returns data to your OCI application.

This call also implicitly allocates the define handle for the select-list item. If a non-null pointer is passed in **defnpp*, the OCI assumes that this points to a valid handle that has been previously allocated with a call to OCIHandleAlloc() or OCIDefineByPos(). This would be true in the case of an application which is redefining a handle to a different addresses so it can reuse the same define handle for multiple fetches.

Defining attributes of a column for a fetch is done in one or more calls. The first call is to OCIDefineByPos(), which defines the minimal attributes required to specify the fetch.

Following the call to OCIDefineByPos() additional define calls may be necessary for certain data types or fetch modes:

- A call to OCIDefineArrayOfStruct() is necessary to set up skip parameters for an array fetch of multiple columns.

- A call to OCIDefineObject is necessary to set up the appropriate attributes of a named data type (i.e., object or collection) or REF fetch. In this case the data buffer pointer in OCIDefineByPos() is ignored.

- Both OCIDefineArrayOfStruct() and OCIDefineObject() must be called after OCIDefineByPos() in order to fetch multiple rows with a column of named data types.

For a LOB define, the buffer pointer must be a pointer to a LOB locator of type **OCILobLocator**, allocated by the OCIDescriptorAlloc() call. LOB locators, and

not LOB values, are always returned for a LOB column. LOB values can then be fetched using OCI LOB calls on the fetched locator. This same mechanism is true for all descriptor datatypes.

For NCHAR (fixed and varying length), the buffer pointer must point to an array of bytes sufficient for holding the required NCHAR characters.

Nested table columns are defined and fetched like any other named data type.

When defining an array of descriptors or locators, you should pass in an array of pointers to descriptors or locators.

When doing an array define for character columns, you should pass in an array of character buffers.

If the *mode* parameter is this call is set to OCI_DYNAMIC_FETCH, the client application can fetch data dynamically at runtime. Runtime data can be provided in one of two ways:

- callbacks using a user-defined function which must be registered with a subsequent call to `OCIDefineDynamic()`. When the client library needs a buffer to return the fetched data, the callback will be invoked and the runtime buffers provided will return a piece or the whole data.

a polling mechanism using calls supplied by the OCI. This mode is assumed if no callbacks are defined. In this case, the fetch call returns the OCI_NEED_DATA error code, and a piecewise polling method is used to provide the data.

> **See Also:**
>
> - For more information about using the OCI_DYNAMIC_FETCH mode, see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44.
>
> - For more information about defines, see "Defining" on page 5-19.

Related Functions

`OCIDefineArrayOfStruct(), OCIDefineDynamic(), OCIDefineObject`

OCIDefineDynamic()

Purpose

This call is used to set the additional attributes required if the OCI_DYNAMIC_FETCH mode was selected in `OCIDefineByPos()`.

Syntax

```
sword OCIDefineDynamic ( OCIDefine    *defnp,
                         OCIError     *errhp,
                         dvoid        *octxp,
                         OCICallbackDefine    (ocbfp)(/*_
                                    dvoid        *octxp,
                                    OCIDefine    *defnp,
                                    ub4          iter,
                                    dvoid        **bufpp,
                                    ub4          **alenpp,
                                    ub1          *piecep,
                                    dvoid        **indpp,
                                    ub2          **rcodep _*/)  );
```

Parameters

defnp (IN/OUT)
The handle to a define structure returned by a call to `OCIDefineByPos()`.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

octxp (IN)
Points to a context for the callback function.

ocbfp (IN)
Points to a callback function. This is invoked at runtime to get a pointer to the buffer into which the fetched data or a piece of it will be retrieved. The callback also specifies the indicator, the return code and the lengths of the data piece and indicator.

> **Caution:** When working with callback parameters, it is important to keep in mind what is meant by IN and OUT for the parameter mode. Normally, in an OCI function, an IN parameter refers to data being passed to Oracle, and an OUT parameter refers to data coming back from Oracle. In the case of callbacks, this is reversed. IN means data is coming from Oracle into the callback, and OUT means data is coming out of the callback and going to Oracle.

The callback parameters are listed below:

octxp (IN/OUT)
A context pointer passed as an argument to all the callback functions.

defnp (IN)
The define handle.

iter (IN)
Which row of this current fetch; 0-based.

bufpp (OUT)
Returns a pointer to a buffer to store the column value, that is, *bufpp* points to some appropriate storage for the column value.

alenpp (IN/OUT)
Used by the application to set the size of the storage it is providing in *bufpp*. After data is fetched into the buffer, *alenpp* indicates the actual size of the data in bytes.

piecep (IN/OUT)
Returns a piece value from the callback (application) to Oracle, as follows:

- **IN** - The value can be OCI_ONE_PIECE or OCI_NEXT_PIECE.

- **OUT** - Depends on the IN value:

    If IN value is OCI_ONE_PIECE, then OUT value can be OCI_ONE_PIECE or OCI_FIRST_PIECE

    If IN value is OCI_NEXT_PIECE then OUT value can be OCI_NEXT_PIECE or OCI_LAST_PIECE

indpp (IN)
Indicator variable pointer

rcodep (IN)
Return code variable pointer

Comments

This call is used to set the additional attributes required if the OCI_DYNAMIC_FETCH mode has been selected in a call to OCIDefineByPos(). If OCI_DYNAMIC_FETCH mode was selected, and the call to OCIDefineDynamic() is skipped, then the application can fetch data piecewise using OCI calls (OCIStmtGetPieceInfo() and OCIStmtSetPieceInfo()). For more information about OCI_DYNAMIC_FETCH mode, see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44.

Related Functions

OCIDefineByPos()

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals,* the chapter on Establishing Security Policies

OCIDefineObject

Purpose

Sets up additional attributes necessary for a named data type or REF define.

Syntax

```
sword OCIDefineObject ( OCIDefine      *defnp,
                        OCIError       *errhp,
                        CONST OCIType  *type,
                        dvoid          **pgvpp,
                        ub4            *pvszsp,
                        dvoid          **indpp,
                        ub4            *indszp );
```

Parameters

defnp (IN/OUT)
A define handle previously allocated in a call to `OCIDefineByPos()`.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

type (IN) [optional]
Points to the Type Descriptor Object (TDO) which describes the type of the program variable. Only used for program variables of type SQLT_NTY. This parameter is optional, and may be passed as null if it is not being used.

pgvpp (IN/OUT)
Points to a pointer to a program variable buffer. For an array, *pgvpp* points to an array of pointers. Memory for the fetched named data type instance(s) is dynamically allocated in the object cache. At the end of the fetch when all the values have been received, *pgvpp* points to the pointer(s) to these newly allocated named data type instance(s). The application must call `OCIObjectFree()` to deallocate the named data type instance(s) when they are no longer needed.

> **Note:** If the application wants the buffer to be implicitly allocated in the cache, **pgvpp* should be passed in as null.

pvszsp (IN/OUT)
Points to the size of the program variable. For an array, it is an array of **ub4**s.

indpp (IN/OUT)
Points to a pointer to the program variable buffer containing the parallel indicator structure. For an array, points to an array of pointers. Memory is allocated to store the indicator structures in the object cache. At the end of the fetch when all values have been received, *indpp* points to the pointer(s) to these newly allocated indicator structure(s).

indszp (IN/OUT)
Points to the size(s) of the indicator structure program variable. For an array, it is an array of **ub4**s.

Comments

This function follows a call to OCIDefineByPos() to set initial define information. This call sets up additional attributes necessary for a Named Data Type define. An error will be returned if this function is called when the OCI environment has been initialized in non-Object mode.

This call takes as a parameter a type descriptor object (TDO) of datatype **OCIType** for the named data type being defined. The TDO can be retrieved with a call to OCIDescribeAny().

> **See Also:** See the description of OCIInitialize() on page 15-14 for more information about initializing the OCI process environment.

Related Functions
OCIDefineByPos()

OCIDescribeAny()

Purpose

Describes existing schema and sub-schema objects.

Syntax

```
sword OCIDescribeAny ( OCISvcCtx      *svchp,
                       OCIError       *errhp,
                       dvoid          *objptr,
                       ub4            objptr_len,
                       ub1            objptr_typ,
                       ub1            info_level,
                       ub1            objtyp,
                       OCIDescribe    *dschp );
```

Parameters

svchp (IN)
A service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the
event of an error.

objptr (IN)
This parameter can be:

1. A string containing the name of the object to be described. Must be UTF16
    encoded in OCI_UTF16 environment.

2. A pointer to a `REF` to the TDO (for a type).

3. A pointer to a TDO (for a type).

These cases are distinguished by passing the appropriate value for `objptr_typ`.
This parameter must be non-null.

In case 1, the string containing the object name should be in the format
`name1[.name2 ...][@linkname]`, such as `scott.emp.empno@mydb`.
Database links are only allowed to Oracle8*i* or later databases. The object name is
interpreted by the following SQL rules:

■   If only `name1` is entered and `objtyp` is equal to OCI_PTYPE_SCHEMA, then
    the name refers to the named schema. The Oracle database must be release 8.1
    or later.

- If only `name1` is entered and `objtyp` is equal to OCI_PTYPE_DATABASE, then the name refers to the named database. When describing a remote database with `database_name@db_link_name`, the remote Oracle database must be release 8.1 or later.

- If only `name1` is entered and `objtyp` is not equal to OCI_PTYPE_SCHEMA or OCI_PTYPE_DATABASE, then the name refers to the named object (of type table / view / procedure / function / package / type / synonym / sequence) in the current schema of the current user. When connected to an Oracle7 Server, the only valid types are procedure and function.

- If `name1.name2.name3 ...` is entered, the object name refers to a schema/sub-schema object in the schema named `name1`. For example, in the string `scott.emp.deptno`, `scott` is the name of the schema, `emp` is the name of a table in the schema, and `deptno` is the name of a column in the table.

objnm_len (IN)
The length of the name string pointed to by `objptr`. Must be nonzero if a name is passed. Can be zero if `objptr` is a pointer to a TDO or its REF.

objptr_typ (IN)
The type of object passed in `objptr`. Valid values are:

- OCI_OTYPE_NAME, if `objptr` points to the name of a schema object

- OCI_OTYPE_REF, if `objptr` is a pointer to a REF to a TDO

- OCI_OTYPE_PTR, if `objptr` is a pointer to a TDO

info_level (IN)
Reserved for future extensions. Pass OCI_DEFAULT.

objtyp (IN)
The type of schema object being described. Valid values are:

- OCI_PTYPE_TABLE, for tables

- OCI_PTYPE_VIEW, for views

- OCI_PTYPE_PROC, for procedures

- OCI_PTYPE_FUNC, for functions

- OCI_PTYPE_PKG, for packages

- OCI_PTYPE_TYPE, for types

- OCI_PTYPE_SYN, for synonyms

- OCI_PTYPE_SEQ, for sequences

- OCI_PTYPE_SCHEMA, for schemas

- OCI_PTYPE_DATABASE, for databases

- OCI_PTYPE_UNK, for unknown schema objects

dschp (IN/OUT)
A describe handle that is populated with describe information about the object after the call. Must be non-null.

Comments

This is a generic describe call that describes existing schema objects: tables, views, synonyms, procedures, functions, packages, sequences, types, schemas, and databases. This call also describes sub-schema objects, such as a column in a table. This call populates the describe handle with the object-specific attributes which can be obtained through an `OCIAttrGet()` call.

An `OCIParamGet()` on the describe handle returns a parameter descriptor for a specified position. Parameter positions begin with 1. Calling `OCIAttrGet()` on the parameter descriptor returns the specific attributes of a stored procedure or function parameter, or a table column descriptor. These subsequent calls do not need an extra round-trip to the server because the entire schema object description is cached on the client side by `OCIDescribeAny()`. Calling `OCIAttrGet()` on the describe handle also returns the total number of positions.

If the OCI_ATTR_DESC_PUBLIC attribute is set on the describe handle, then the object named is looked up as a public synonym when the object does not exist in the current schema and only *name1* is specified.

> **See Also:** For more information about describe operations, see
> Chapter 6, "Describing Schema Metadata"

Related Functions
OCIAttrGet(), OCIParamGet()

OCIStmtGetBindInfo()

Purpose

Gets the bind and indicator variable names.

Syntax

```
sword OCIStmtGetBindInfo ( OCIStmt     *stmtp,
                           OCIError    *errhp,
                           ub4         size,
                           ub4         startloc,
                           sb4         *found,
                           text        *bvnp[],
                           ub1         bvnl[],
                           text        *invp[],
                           ub1         inpl[],
                           ub1         dupl[],
                           OCIBind     *hndl[] );
```

Parameters

stmtp (IN)
The statement handle prepared by OCIStmtPrepare().

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

size (IN)
The number of elements in each array.

startloc (IN)
Position of the bind variable at which to start getting bind information.

found (IN)
abs(*found*) gives the total number of bind variables in the statement irrespective of the start position. Positive value if the number of bind variables returned is less than the size provided, otherwise negative.

bvnp (OUT)
Array of pointers to hold bind variable names. Must be in UTF-16 if the environment is set in OCI_UTF16 mode.

bvnl (OUT)
Array to hold the length of the each *bvnp* element. The length is in bytes.

invp (OUT)
Array of pointers to hold indicator variable names. Must be in UTF-16 if the environment is set in OCI_UTF16 mode.

inpl (OUT)
Array of pointers to hold the length of the each *invp* element. In number of bytes.

dupl (OUT)
An array whose element value is 0 or 1 depending on whether the bind position is duplicate of another.

hndl (OUT)
An array which returns the bind handle if binds have been done for the bind position. No handle is returned for duplicates.

Comments

This call returns information about bind variables after a statement has been prepared. This includes bind names, indicator names, and whether or not binds are duplicate binds. This call also returns an associated bind handle if there is one. The call sets the *found* parameter to the total number of bind variables and not just the number of distinct bind variables.

This function does not include SELECT INTO list variables, because they are not considered to be binds.

The statement must have been prepared with a call to OCIStmtPrepare() prior to this call. The encoding setting in the statement handle will determine whether Unicode strings will be retrieved.

This call is processed locally.

Related Functions

OCIStmtPrepare()

Statement Functions

This section describes the statement functions.

Table 15–4   Statement Functions

Function	Purpose
OCIStmtExecute() on page 15-84	Send statements to server for execution
OCIStmtFetch() on page 15-87	Fetch rows from a query (deprecated)
OCIStmtFetch2() on page 15-89	Fetch rows from a query
OCIStmtGetPieceInfo() on page 15-92	Get piece information for piecewise operations
OCIStmtPrepare() on page 15-94	Establish an application request
OCIStmtSetPieceInfo() on page 15-96	Set piece information for piecewise operations

OCIStmtExecute()

Purpose

This call associates an application request with a server.

Syntax

```
sword OCIStmtExecute ( OCISvcCtx          *svchp,
                       OCIStmt            *stmtp,
                       OCIError           *errhp,
                       ub4                iters,
                       ub4                rowoff,
                       CONST OCISnapshot  *snap_in,
                       OCISnapshot        *snap_out,
                       ub4                mode );
```

Parameters

svchp (IN/OUT)
Service context handle.

stmtp (IN/OUT)
An statement handle. It defines the statement and the associated data to be executed at the server. It is invalid to pass in a statement handle that has bind of data types only supported in release 8.x or later when *svchp* points to an Oracle7 server.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

iters (IN)
For non-SELECT statements, the number of times this statement is executed is equal to *iters - rowoff*.

For SELECT statements, if *iters* is nonzero, then defines must have been done for the statement handle. The execution fetches *iters* rows into these predefined buffers and prefetches more rows depending upon the prefetch row count. If you do not know how many rows the SELECT statement will retrieve, set *iters* to zero.

This function returns an error if *iters*=0 for non-SELECT statements.

rowoff (IN)
The starting index from which the data in an array bind is relevant for this multiple row execution.

snap_in (IN)
This parameter is optional. if supplied, must point to a snapshot descriptor of type OCI_DTYPE_SNAP. The contents of this descriptor must be obtained from the *snap_out* parameter of a previous call. The descriptor is ignored if the SQL is not a SELECT. This facility allows multiple service contexts to ORACLE to see the same consistent snapshot of the database's *committed* data. However, uncommitted data in one context is *not* visible to another context even using the same snapshot.

snap_out (OUT)
This parameter optional. if supplied, must point to a descriptor of type OCI_DTYPE_SNAP. This descriptor is filled in with an opaque representation which is the current ORACLE "system change number" suitable as a *snap_in* input to a subsequent call to OCIStmtExecute(). This descriptor should not be used longer than necessary in order to avoid "snapshot too old" errors.

mode (IN)
The modes are:

- OCI_BATCH_ERRORS - See "Batch Error Mode for OCIStmtExecute()" on page 4-9, for information about this mode.

- OCI_COMMIT_ON_SUCCESS - When a statement is executed in this mode, the current transaction is committed after execution, provided that execution completes successfully.

- OCI_DEFAULT - Calling OCIStmtExecute() in this mode executes the statement. It also implicitly returns describe information about the select-list.

- OCI_DESCRIBE_ONLY - This mode is for users who wish to describe a query prior to execution. Calling OCIStmtExecute() in this mode does not execute the statement, but it does return the select-list description. To maximize performance, it is recommended that applications execute the statement in default mode and use the implicit describe which accompanies the execution.

- OCI_EXACT_FETCH - Used when the application knows in advance exactly how many rows it will be fetching. This mode turns prefetching off for Oracle release 8 or later mode, and requires that defines be done before the execute call. Using this mode cancels the cursor after the desired rows are fetched and may result in reduced server-side resource usage.

- OCI_PARSE_ONLY - This mode allows the user to parse the query prior to execution. Executing in this mode parses the query and returns parse errors in the SQL, if any. Users must note that this will involve an addtional round-trip to the server. To maximize performance, it is recommended that the user

execute the statement in the default mode which, as part of a bundled operation, parses the statement.

■ OCI_STMT_SCROLLABLE_READONLY - Required for the result set to be scrollable. The result set will not be updateable. See "Fetching Results" on page 4-16. Cannot be used with any other mode.

The modes are not mutually exclusive and can be used together, except for OCI_STMT_SCROLLABLE_READONLY.

Comments

This function is used to execute a prepared SQL statement. Using an execute call, the application associates a request with a server.

If a SELECT statement is executed, the description of the select-list is available implicitly as a response. This description is buffered on the client side for describes, fetches and define type conversions. Hence it is optimal to describe a select list only after an execute.

> **See Also:** "Describing Select-List Items" on page 4-12

Also for SELECT statements, some results are available implicitly. Rows will be received and buffered at the end of the execute. For queries with small row count, a prefetch causes memory to be released in the server if the end of fetch is reached, an optimization that may result in memory usage reduction. Set attribute call has been defined to set the number of rows to be prefetched for each result set.

For SELECT statements, at the end of the execute, the statement handle implicitly maintains a reference to the service context on which it is executed. It is the user's responsibility to maintain the integrity of the service context. The implicit reference is maintained until the statement handle is freed or the fetch is cancelled or an end of fetch condition is reached.

Note: If output variables are defined for a SELECT statement before a call to OCIStmtExecute(), the number of rows specified by *iters* will be fetched directly into the defined output buffers and additional rows equivalent to the prefetch count will be prefetched. If there are no additional rows, then the fetch is complete without calling OCIStmtFetch().

Related Functions

OCIStmtPrepare()

OCIStmtFetch()

Purpose

Fetches rows from a query. Users are encouraged to use the new fetch call
`OCIStmtFetch2()`. This call is deprecated.

Syntax

```
sword OCIStmtFetch ( OCIStmt      *stmtp,
                     OCIError     *errhp,
                     ub4          nrows,
                     ub2          orientation,
                     ub4          mode );
```

Parameters

stmtp (IN)
A statement (application request) handle.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the
event of an error.

nrows (IN)
Number of rows to be fetched from the current position.

orientation (IN)
Prior to release 9.0, the only acceptable value is OCI_FETCH_NEXT, which is also
the default value.

mode (IN)
Pass as OCI_DEFAULT.

Comments

The fetch call is a local call, if prefetched rows suffice. However, this is transparent
to the application.

If LOB columns are being read, LOB locators are fetched for subsequent LOB
operations to be performed on these locators. Prefetching is turned off if LONG
columns are involved.

This function can return OCI_NO_DATA on EOF and OCI_SUCCESS_WITH_INFO
when one of the following errors occur:

- ORA-24344 Success with compilation error

- ORA-23445 A truncation or null fetch error occurred

■ ORA-23447 Warning of a NULL column in an aggregate function

If you call `OCIStmtFetch()` with the *nrows* parameter set to zero, this cancels the cursor.

Use OCI_ATTR_ROWS_FETCHED to find the number of rows that were successfully fetched into the user's buffers in the last fetch call.

Related Functions

OCIStmtExecute()

OCIStmtFetch2()

Purpose

This fetches a row from the (scrollable) result set. You are encouraged to use this fetch call instead of the deprecated call OCIStmtFetch().

Syntax

```
sword OCIStmtFetch2 ( OCIStmt    *stmthp,
                      OCIError   *errhp,
                      ub4        nrows,
                      ub2        orientation,
                      sb4        fetchOffset,
                      ub4        mode );
```

Parameters

stmthp (IN/OUT)
This is the statement handle of the (scrollable) result set.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in event of an error.

nrows (IN)
Number of rows to be fetched from the current position.

orientation (IN)
The acceptable values are:

- OCI_DEFAULT - has the same effect as OCI_FETCH_NEXT

- OCI_FETCH_CURRENT - gets the current row.

- OCI_FETCH_NEXT - gets the next row from the current position. It is the default (has the same effect as OCI_DEFAULT). Use for a non-scrollable statement handle.

- OCI_FETCH_FIRST - gets the first row in the result set.

- OCI_FETCH_LAST - gets the last row in the result set.

- OCI_FETCH_PRIOR - gets the previous row from the current row in the result set.

- OCI_FETCH_ABSOLUTE will fetch the row number (specified by *fetchOffset* parameter) in the result set using absolute positioning.

■ OCI_FETCH_RELATIVE will fetch the row number (specified by *fetchOffset* parameter) in the result set using relative positioning.

fetchOffset (IN)
The offset to be used with the orientation parameter for changing the current row position.

mode (IN)
Pass in OCI_DEFAULT.

Comments

The fetch call works similar to the `OCIStmtFetch()` call with the addition of the *fetchOffset* parameter. It can be used on any statement handle, whether it is scrollable or not. For a *non-scrollable statement handle*, the only acceptable value of *orientation* is OCI_FETCH_NEXT, and the *fetchOffset* parameter will be ignored.

For new applications you are encouraged to use this new call, `OCIStmtFetch2()`.

A *fetchOffset* with *orientation* set to OCI_FETCH_RELATIVE is equivalent to all of the following:

■ OCI_FETCH_CURRENT with a value of *fetchOffset* equal to 0,

■ OCI_FETCH_NEXT with a value of *fetchOffset* equal to 1,

■ OCI_FETCH_PRIOR with a value of *fetchOffset* equal to -1.

OCI_ATTR_ROW_COUNT contains the highest absolute row value that was fetched.

All other orientation modes besides OCI_FETCH_ABSOLUTE and OCI_FETCH_RELATIVE will ignore the *fetchOffset* value.

This call can also be used to find out the number of rows in the result set by using OCI_FETCH_LAST, and then calling `OCIAttrGet()` on OCI_ATTR_CURRENT_POSITION. But the response time of this call can be high.

The return codes are the same as for `OCIStmtFetch()`, except that OER(1403) with return code OCI_NO_DATA will be returned every time a fetch on a scrollable statement handle (or execute) is made and not all rows requested by the application could be fetched.

The scrollable statement handle will need to be explicitly cancelled (that is, fetch with 0 rows) or freed in order to release server-side resources for this scrollable cursor. A non-scrollable statement handle is implicitly cancelled on receiving the OER(1403).

Use OCI_ATTR_ROWS_FETCHED to find the number of rows that were successfully fetched into the user's buffers in the last fetch call.

> **See Also:** "Scrollable Cursors" on page 4-18 for more information on this topic

Related Functions

OCIStmtExecute(), OCIBindByPos()

OCIStmtGetPieceInfo()

Purpose

Returns piece information for a piecewise operation.

Syntax

```
sword OCIStmtGetPieceInfo( CONST OCIStmt  *stmtp,
                           OCIError       *errhp,
                           dvoid          **hndlpp,
                           ub4            *typep,
                           ub1            *in_outp,
                           ub4            *iterp,
                           ub4            *idxp,
                           ub1            *piecep );
```

Parameters

stmtp (IN)
The statement when executed returned OCI_NEED_DATA.

errhp (OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

hndlpp (OUT)
Returns a pointer to the bind or define handle of the bind or define whose runtime data is required or is being provided.

typep (OUT)
The type of the handle pointed to by `hndlpp`: OCI_HTYPE_BIND (for a bind handle) or OCI_HTYPE_DEFINE (for a define handle).

in_outp (OUT)
Returns OCI_PARAM_IN if the data is required for an IN bind value. Returns OCI_PARAM_OUT if the data is available as an OUT bind variable or a define position value.

iterp (OUT)
Returns the row number of a multiple row operation.

idxp (OUT)
The index of an array element of a PL/SQL array bind operation.

piecep (OUT)
Returns one of the following defined values OCI_ONE_PIECE, OCI_FIRST_PIECE, OCI_NEXT_PIECE and OCI_LAST_PIECE.

Comments

When an execute/fetch call returns OCI_NEED_DATA to get/return a dynamic bind/define value or piece, `OCIStmtGetPieceInfo()` returns the relevant information: bind/define handle, iteration, index number and which piece.

> **See Also:** See the section "Runtime Data Allocation and Piecewise Operations" on page 5-44 for more information about using `OCIStmtGetPieceInfo()`.

Related Functions

`OCIAttrGet()`, `OCIAttrSet()`, `OCIStmtExecute()`, `OCIStmtFetch()`, `OCIStmtSetPieceInfo()`

OCIStmtPrepare()

Purpose

This call prepares a SQL or PL/SQL statement for execution.

Syntax

```
sword OCIStmtPrepare ( OCIStmt       *stmtp,
                       OCIError      *errhp,
                       CONST text    *stmt,
                       ub4           stmt_len,
                       ub4           language,
                       ub4           mode );
```

Parameters

stmtp (IN)

A statement handle associated with the statement to be executed. By default, it contains the encoding setting in the environment handle from which it is derived. A statement can be prepared in UTF-16 encoding only in a UTF-16 environment.

errhp (IN)

An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

stmt (IN)

SQL or PL/SQL statement to be executed. Must be a null-terminated string. That is, the ending character is a number of NULL bytes, depending on the encoding. The statement should be in UTF-16 if the environment handle is created with OCI_UTF16 mode.

Always cast the parameter to (**text** *). After a statement has been prepared in UTF-16, the character set for the bind and define buffers will default to UTF-16. The pointer to the text of the statement must be available as long as the statement is executed, or data is fetched from it.

stmt_len (IN)

Length of the statement in characters or in number of bytes, depending on the encoding. Must not be zero.

language (IN)

Specifies V7, or native syntax. Possible values are:

- OCI_V7_SYNTAX - V7 ORACLE parsing syntax

- OCI_NTV_SYNTAX - syntax depends upon the version of the server.

mode (IN)

Similar to the *mode* in the OCIEnvCreate() call, but this one has higher priority because it can override the "naturally" inherited mode setting.

The possible values are:

- OCI_DEFAULT - default mode. The statement handle *stmtp* uses whatever is specified by its parent environment handle.

- OCI_NO_SHARING - disables sharing mode for the SQL statement. See "Shared Data Mode" on page 2-22.

Comments

An OCI application uses this call to prepare a SQL or PL/SQL statement for execution. The OCIStmtPrepare() call defines an application request.

The *mode* parameter determines whether the statement content is encoded as UTF-16 or not. The statement length is in number of codepoints or in number of bytes, depending on the encoding.

While the statement handle inherits the encoding setting from the parent environment handle, the *mode* for this call can also change the encoding setting for the statement handle itself.

Data values for this statement initialized in subsequent bind calls will be stored in a bind handle which use settings in this statement handle as default.

This call does not create an association between this statement handle and any particular server.

> **See Also:** See the section "Preparing Statements" on page 4-4 for more information about using this call.

Related Functions

OCIAttrGet(), OCIStmtExecute()

OCIStmtSetPieceInfo()

Purpose

Sets piece information for a piecewise operation.

Syntax

```
sword OCIStmtSetPieceInfo ( dvoid           *hndlp,
                            ub4             type,
                            OCIError        *errhp,
                            CONST dvoid     *bufp,
                            ub4             *alenp,
                            ub1             piece,
                            CONST dvoid     *indp,
                            ub2             *rcodep );
```

Parameters

hndlp (IN/OUT)
The bind/define handle.

type (IN)
Type of the handle.

errhp (OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

bufp (IN/OUT)
A pointer to a storage containing the data value or the piece when it is an IN bind variable, otherwise *bufp* is a pointer to storage for getting a piece or a value for OUT binds and define variables. For named data types or REFs, a pointer to the object or REF is returned.

alenp (IN/OUT)
The length of the piece or the value.

piece (IN)
The piece parameter. Valid values:

- OCI_ONE_PIECE

- OCI_FIRST_PIECE

- OCI_NEXT_PIECE

- OCI_LAST_PIECE

This parameter is used for IN bind variables only.

indp (IN/OUT)
Indicator. A pointer to a **sb2** value or pointer to an indicator structure for named data types (SQLT_NTY) and REFs (SQLT_REF), i.e., *indp* is either an **sb2** or a **dvoid** * depending upon the data type.

rcodep (IN/OUT)
Return code.

Comments

When an execute call returns OCI_NEED_DATA to get a dynamic IN/OUT bind value or piece, OCIStmtSetPieceInfo() sets the piece information: the buffer, the length, which piece is currently being processed, the indicator, and the return code for this column.

> **See Also:** For more information about using OCIStmtSetPieceInfo() see the section "Runtime Data Allocation and Piecewise Operations" on page 5-44

Related Functions

OCIAttrGet(), OCIAttrSet(), OCIStmtExecute(), OCIStmtFetch(), OCIStmtGetPieceInfo()

16

More OCI Relational Functions

This chapter completes description of the OCI relational functions started in the last chapter. It includes information about calling OCI functions in your application, along with detailed descriptions of each function call. This chapter contains the following sections:

- Introduction to the Relational Functions
- LOB Functions
- Advanced Queuing and Publish-Subscribe Functions
- Direct Path Loading Functions
- Thread Management Functions
- Transaction Functions
- Miscellaneous Functions

Introduction to the Relational Functions

This chapter completes descriptions of the OCI relational function calls. It continues from the last chapter. The function calls for manipulating objects are described in the next three chapters.

> **See Also:** For information about return codes and error handling, refer to the section "Error Handling" on page 2-31

Function Syntax

For each function, the following information is listed:

Purpose

A brief description of the action performed by the function.

Syntax

The function declaration.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below.

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

Comments

More detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Example

A complete or partial code example demonstrating the use of the function call being described. Not all function descriptions include an example.

Related Functions

A list of related function calls.

Calling OCI Functions

Unlike earlier versions of the OCI, in release 8 you cannot pass -1 for the string length parameter of a null-terminated string.

When you pass string lengths as parameters, do not include the null terminator byte in the length. The OCI does not expect strings to be null-terminated.

Buffer lengths that are OCI parameters are in bytes, except the amount parameters in some LOB calls, which are in characters.

Server Round-trips for LOB Functions

For a table showing the number of server round- trips required for individual OCI LOB functions, refer to Appendix C, "OCI Function Server Round-trips".

LOB Functions

This section describes the LOB functions.

Table 16–1   LOB Functions

Function	Purpose
OCIDurationBegin() on page 16-7	Start user duration for temporary LOB
OCIDurationEnd() on page 16-9	End user duration for temporary LOB
OCILobAppend() on page 16-10	Append one LOB to another
OCILobAssign() on page 16-12	Assign one LOB locator to another
OCILobCharSetForm() on page 16-14	Get character set form from LOB locator
OCILobCharSetId() on page 16-15	Get character set ID from LOB locator
OCILobClose() on page 16-16	Close a previously opened LOB
OCILobCopy() on page 16-18	Copy all or part of one LOB to another
OCILobCreateTemporary() on page 16-20	Create a temporary LOB
OCILobDisableBuffering() on page 16-22	Turn LOB buffering off
OCILobEnableBuffering() on page 16-23	Turn LOB buffering on
OCILobErase() on page 16-24	Erase a portion of a LOB
OCILobFileClose() on page 16-26	Close a previously opened FILE
OCILobFileCloseAll() on page 16-27	Close all previously opened files
OCILobFileExists() on page 16-28	Check if a file exists on the server
OCILobFileGetName() on page 16-29	Get directory alias and file NaMe from the LOB locator
OCILobFileIsOpen() on page 16-31	Check if file on server is open using this locator
OCILobFileOpen() on page 16-32	Open a FILE
OCILobFileSetName() on page 16-33	Set directory alias and file name in the LOB locator
OCILobFlushBuffer() on page 16-35	Flush the LOB buffer
OCILobFreeTemporary() on page 16-37	Free a temporary LOB
OCILobGetChunkSize() on page 16-38	Get the chunk size of a LOB
OCILobGetLength() on page 16-40	Get length of a LOB
OCILobIsEqual() on page 16-42	Compare two LOB locators for Equality

Table 16–1   LOB Functions

Function	Purpose
`OCILobIsOpen()` on page 16-43	Check to see if a LOB is open
`OCILobIsTemporary()` on page 16-45	Determine if a given LOB is a temporary LOB
`OCILobLoadFromFile()` on page 16-46	Load a LOB from a FILE
`OCILobLocatorAssign()` on page 16-48	Assigns one LOB locator to another
`OCILobLocatorIsInit()` on page 16-50	Check to see if a LOB locator is initialized
`OCILobOpen()` on page 16-52	Open a LOB
`OCILobRead()` on page 16-54	Read a portion of a LOB
`OCILobTrim()` on page 16-59	Truncate a LOB
`OCILobWrite()` on page 16-61	Write into a LOB
`OCILobWriteAppend()` on page 16-66	Write data beginning at the end of a LOB

Note the following for parameters in the OCI LOB calls:

- For fixed-width client-side character sets, the offset and amount parameters are always in characters for CLOBs and NCLOBs, and in bytes for BLOBs and BFILEs.

- For varying-width client-side character sets, these rules generally apply:

    - amount (*amtp*) parameter - When the amount parameter refers to the server-side LOB, the amount is in characters. When the amount parameter refers to the client-side buffer, the amount is in bytes. For more information, see individual LOB calls, such as `OCILobGetLength()`, `OCILobRead()`, and `OCILobWrite()`.

    - offset (*offset*) parameter - Regardless of whether the client-side character set is varying-width, the offset parameter is always in characters for CLOBs and NCLOBs and in bytes for BLOBs and BFILEs.

- For many of the LOB operations, regardless of the client-side character set, the amount parameter is in characters for CLOBs and NCLOBs. These LOB operations include OCILobCopy(), OCILobErase(), OCILobGetLength(), OCILobLoadFromFile(), and OCILobTrim(). All these operations refer to the amount of LOB data on the server.

A *streaming operation* means that the LOB is read or written in pieces. Streaming can be implemented using a polling mechanism or by registering a user-defined callback.

OCIDurationBegin()

Purpose

Starts a user duration for a temporary LOB.

Syntax

```
sword OCIDurationBegin ( OCIEnv            *env,
                         OCIError          *err,
                         CONST OCISvcCtx   *svc,
                         OCIDuration       parent,
                         OCIDuration       *duration );
```

Parameters

env (IN/OUT)
Pass as a null pointer.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
OCIErrorGet().

svc (IN)
An OCI service context handle. Must be non-null.

parent (IN)
The duration number of the parent duration. One of these:

- OCI_DURATION_STATEMENT

- OCI_DURATION_SESSION

duration (OUT)
An identifier unique to the newly created user duration.

Comments

This function starts an user duration. In release 8.1 or later, user durations can be
used when creating temporary LOBs. An user can have multiple active user
durations simultaneously. The user durations do not have to be nested. The *dur*
parameter is used to return a number which uniquely identifies the duration
created by this call.

See Also: "Temporary LOB Durations" on page 7-19

Related Functions

OCIDurationEnd()

OCIDurationEnd()

Purpose

Terminates a user duration for a temporary LOB.

Syntax

```
sword OCIDurationEnd ( OCIEnv            *env,
                       OCIError          *err,
                       CONST OCISvcCtx   *svc,
                       OCIDuration       duration,
                       CONST OCISvcCtx   *svc );
```

Parameters

env (IN/OUT)
Pass as a null pointer.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
`OCIErrorGet()`.

duration (IN)
A number to identify the user duration.

svc (IN)
OCI service context (this should be passed as null for cartridge services, otherwise
non-null)

Comments

This function terminates an user duration. Temporary LOBs that are allocated for
the user duration are freed.

> **See Also:** "Temporary LOB Durations" on page 7-19

Related Functions

OCIDurationBegin()

OCILobAppend()

Purpose

Appends a LOB value at the end of another LOB as specified.

Syntax

```
sword OCILobAppend ( OCISvcCtx          *svchp,
                     OCIError           *errhp,
                     OCILobLocator      *dst_locp,
                     OCILobLocator      *src_locp );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

dst_locp (IN/OUT)
An internal LOB locator uniquely referencing the destination LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

src_locp (IN)
An internal LOB locator uniquely referencing the source LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

Comments

Appends a LOB value at the end of another LOB as specified. The data is copied from the source to the end of the destination. The source and destination LOBs must already exist. The destination LOB is extended to accommodate the newly written data. It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes) or to try to copy from a NULL LOB.

The source and the destination LOB locators must be of the same type (i.e., they must both be BLOBs or both be CLOBs). LOB buffering must not be enabled for either type of locator. This function does not accept a FILE locator as the source or the destination.

It is not mandatory that you wrap this LOB operation inside the Open/Close calls. If you did not open the LOB prior to performing this operation, then the functional and domain indexes on the LOB column are updated during this call. However, if you did open the LOB prior to performing this operation, then you must close it

before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. If you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

Related Functions

`OCILobTrim()`, `OCILobWrite()`, `OCILobCopy()`, `OCIErrorGet()`, `OCILobWriteAppend()`

OCILobAssign()

Purpose

Assigns one LOB/FILE locator to another.

Syntax

```
sword OCILobAssign ( OCIEnv              *envhp,
                     OCIError            *errhp,
                     CONST OCILobLocator *src_locp,
                     OCILobLocator       **dst_locpp );
```

Parameters

envhp (IN/OUT)
OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

src_locp (IN)
LOB/FILE locator to copy from.

dst_locpp (IN/OUT)
LOB/FILE locator to copy to. The caller must have allocated space for the destination locator by calling OCIDescriptorAlloc().

Comments

Assign *source* locator to *destination* locator. After the assignment, both locators refer to the same LOB value. For internal LOBs, the source locator's LOB value gets copied to the *destination* locator's LOB value only when the *destination* locator gets stored in the table. Therefore, issuing a flush of the object containing the *destination* locator will copy the LOB value.

OCILobAssign() cannot be used for temporary LOBs; it will generate an OCI_INVALID_HANDLE error. For temporary LOBs, use OCILobLocatorAssign().

For FILEs, only the locator that refers to the file is copied to the table. The operating system file itself is not copied.

It is an error to assign a FILE locator to an internal LOB locator, and vice versa.

If the source locator is for an internal LOB that was enabled for buffering, and the source locator has been used to modify the LOB data through the LOB buffering subsystem, and the buffers have not been flushed since the write, then the source

locator may not be assigned to the destination locator. This is because only one locator per LOB may modify the LOB data through the LOB buffering subsystem.

The value of the input destination locator must have already been allocated with a call to OCIDescriptorAlloc(). For example, assume the following declarations:

```
OCILobLocator     *source_loc = (OCILobLocator *) 0;
OCILobLocator     *dest_loc = (OCILobLocator *) 0;
```

An application could allocate the source_loc locator as follows:

```
if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &source_loc,
    (ub4) OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    handle_error;
```

Assume that it then selects a LOB from a table into the source_loc in order to initialize it. The application must allocate the destination locator, dest_loc, before issuing the OCILobAssign() call to assign the value of source_loc to dest_loc. For example:

```
    if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &dest_loc,
        (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
        handle_error;
    if (OCILobAssign(envhp, errhp, source_loc, &dest_loc))
        handle_error;
```

Related Functions

OCIErrorGet(), OCILobIsEqual(), OCILobLocatorAssign(), OCILobLocatorIsInit(), OCILobEnableBuffering()

OCILobCharSetForm()

Purpose

Gets the LOB locator's character set form, if any.

Syntax

```
sword OCILobCharSetForm ( OCIEnv              *envhp,
                          OCIError            *errhp,
                          CONST OCILobLocator *locp,
                          ub1                 *csfrm );
```

Parameters

envhp (IN/OUT)
OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN)
LOB locator for which to get the character set form.

csfrm (OUT)
Character set form of the input LOB locator. If the input locator, *locp,* is for a BLOB or a BFILE, *csfrm* is set to 0 since there is no concept of a character set for binary LOBs and FILEs. The caller must allocate space for the *csfrm* (a **ub1**).

csfrm has two possible non-zero values:

- SQLCS_IMPLICIT - database character set ID

- SQLCS_NCHAR - NCHAR character set ID

The default value is SQLCS_IMPLICIT.

Comments

Returns the character set form of the input LOB locator in the *csfrm* output parameter. This function makes sense only for character LOBs (i.e., CLOBs and NCLOBs).

Related Functions

OCIErrorGet(), OCILobCharSetId(), OCILobLocatorIsInit()

OCILobCharSetId()

Purpose

Gets the LOB locator's database character set ID, if any.

Syntax

```
sword OCILobCharSetId ( OCIEnv                *envhp,
                        OCIError              *errhp,
                        CONST OCILobLocator   *locp,
                        ub2                   *csid );
```

Parameters

envhp (IN/OUT)
OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN)
LOB locator for which to get the character set ID.

csid (OUT)
Database character set ID of the input LOB locator. If the input locator is for a BLOB or a BFILE, *csid* is set to 0 since there is no concept of a character set for binary LOBs/FILEs. The caller must allocate space for the *csid* **ub2**.

Comments

Returns the character set ID of the input LOB locator in the *csid* output parameter.

This function makes sense only for character LOBs (i.e., CLOBs, NCLOBs).

Related Functions

OCIErrorGet(), OCILobCharSetForm(), OCILobLocatorIsInit()

OCILobClose()

Purpose

Closes a previously opened LOB or FILE.

Syntax

```
sword OCILobClose ( OCISvcCtx      *svchp,
                    OCIError       *errhp,
                    OCILobLocator  *locp );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

locp (IN/OUT)
The LOB to close. The locator can refer to an internal or external LOB.

Comments

Closes a previously opened internal or external LOB. No error is returned if the `BFILE` exists but is not opened. An error is returned if the internal LOB is not open.

Closing a LOB requires a round- trip to the server for both internal and external LOBs. For internal LOBs, close will trigger other code that relies on the close call and for external LOBs (`BFILE`s), close actually closes the server-side operating system file.

It is not mandatory that you wrap all LOB operations inside the Open/Close calls. However, if you open a LOB, then you must close it before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column. It is an error to commit the transaction before closing all opened LOBs that were opened by the transaction.

When the error is returned, the LOB is no longer marked as open, but the transaction is successfully committed. Hence, all the changes made to the LOB and non-LOB data in the transaction are committed but the domain and functional indexing are not updated. If this happens, please rebuild your functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This

can adversely affect performance, so if you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

> **See Also:** "Functions for Opening and Closing LOBs" on page 7-12

Related Functions

OCIErrorGet(), OCILobOpen(), OCILobIsOpen()

OCILobCopy()

Purpose

Copies all or a portion of a LOB value into another LOB value

Syntax

```
sword OCILobCopy ( OCISvcCtx        *svchp,
                   OCIError         *errhp,
                   OCILobLocator    *dst_locp,
                   OCILobLocator    *src_locp,
                   ub4              amount,
                   ub4              dst_offset,
                   ub4              src_offset );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

dst_locp (IN/OUT)
An internal LOB locator uniquely referencing the destination LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

src_locp (IN)
An internal LOB locator uniquely referencing the source LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

amount (IN)
The number of characters for `CLOBs`/`NCLOBs` or bytes for `BLOBs` to be copied from the source LOB to the destination LOB.

dst_offset (IN)
This is the absolute offset for the destination LOB. For character LOBs it is the number of characters from the beginning of the LOB at which to begin writing. For binary LOBs it is the number of bytes from the beginning of the LOB from which to begin writing. The offset starts at 1.

src_offset (IN)
This is the absolute offset for the source LOB. For character LOBs it is the number of characters from the beginning of the LOB, for binary LOBs it is the number of bytes. Starts at 1.

Comments

Copies all or a portion of an internal LOB value into another internal LOB as specified. The data is copied from the source to the destination. The source (*src_locp*) and the destination (*dst_locp*) LOBs must already exist.

If the data already exists at the destination's start position, it is overwritten with the source data. If the destination's start position is beyond the end of the current data, zero-byte fillers (for BLOBs) or spaces (for CLOBs) are written into the destination LOB from the end of the current data to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend the destination LOB beyond the maximum length allowed (i.e., 4 gigabytes) or to try to copy from a NULL LOB.

Both the source and the destination LOB locators must be of the same type (i.e., they must both be BLOBs or both be CLOBs). LOB buffering must not be enabled for either locator.

This function does not accept a FILE locator as the source or the destination.

It is not mandatory that you wrap this LOB operation inside the Open/Close calls. If you did not open the LOB prior to performing this operation, then the functional and domain indexes on the LOB column are updated during this call. However, if you did open the LOB prior to performing this operation, then you must close it before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. If you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

> **Note:** You can call `OCILobGetLength()` to determine the length of the source LOB.

Related Functions

`OCIErrorGet()`, `OCILobRead()`, `OCILobAppend()`, `OCILobCopy()`, `OCILobWrite()`, `OCILobWriteAppend()`

OCILobCreateTemporary()

Purpose

Create a temporary LOB

Syntax

```
sword OCILobCreateTemporary(OCISvcCtx         *svchp,
                            OCIError          *errhp,
                            OCILobLocator     *locp,
                            ub2               csid,
                            ub1               csfrm,
                            ub1               lobtype,
                            boolean           cache,
                            OCIDuration       duration);
```

Parameters

svchp (IN)
The OCI service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

locp (IN/OUT)
A locator which points to the temporary LOB. You must allocate the locator using `OCIDescriptorAlloc()` before passing it to this function. It does not matter whether or not this locator already points to a LOB, it will get overwritten either way.

csid (IN)
The LOB character set ID. For Oracle8*i* or later, pass as OCI_DEFAULT.

csfrm (IN)
The LOB character set form of the buffer data. For Oracle8*i* or later, pass as OCI_DEFAULT.

lobtype (IN)
The type of LOB to create. Valid values include:

- OCI_TEMP_BLOB - for a temporary BLOB
- OCI_TEMP_CLOB - for a temporary CLOB
- OCI_TEMP_NCLOB - for a temporary NCLOB

cache (IN)

Pass TRUE if the temporary LOB should be read into the cache; FALSE, if it should not. The default is FALSE for NOCACHE functionality.

duration (IN)

The duration of the temporary LOB. The following are valid values:

- OCI_DURATION_SESSION

- OCI_DURATION_CALL

Comments

This function creates a temporary LOB and its corresponding index in the user's temporary tablespace.

When this function is complete, the *locp* parameter points to an empty temporary LOB whose length is zero.

The lifetime of the temporary LOB is determined by the *duration* parameter. At the end of its duration the temporary LOB is freed. An application can free a temporary LOB sooner with the OCILobFreeTemporary() call.

If the LOB is a BLOB, the *csid* and *csfrm* parameters are ignored.

> **See Also:** For more information about temporary LOBs and their durations, refer to "Temporary LOB Support" on page 7-18.

Related functions

OCILobFreeTemporary(), OCILobIsTemporary(), OCIDescriptorAlloc(), OCIErrorGet()

OCILobDisableBuffering()

Purpose

Disable LOB buffering for the input locator.

Syntax

```
sword OCILobDisableBuffering ( OCISvcCtx      *svchp,
                               OCIError       *errhp,
                               OCILobLocator  *locp );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
An internal LOB locator uniquely referencing the LOB.

Comments

Disables LOB buffering for the input internal LOB locator. The next time data is read from or written to the LOB through the input locator, the LOB buffering subsystem is *not* used. Note that this call does *not* implicitly flush the changes made in the buffering subsystem. The user must explicitly call OCILobFlushBuffer() to do this.

This function does not accept a FILE locator.

Related Functions

OCILobEnableBuffering(), OCIErrorGet(), OCILobFlushBuffer()

OCILobEnableBuffering()

Purpose

Enable LOB buffering for the input locator.

Syntax

```
sword OCILobEnableBuffering ( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCILobLocator  *locp );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
An internal LOB locator uniquely referencing the LOB.

Comments

Enables LOB buffering for the input internal LOB locator. The next time data is read from or written to the LOB through the input locator, the LOB buffering subsystem is used.

If LOB buffering is enabled for a locator and that locator is passed to one of the following routines, an error is returned: OCILobAppend(), OCILobCopy(), OCILobErase(), OCILobGetLength(), OCILobLoadFromFile(), OCILobTrim(), or OCILobWriteAppend().

This function does not accept a FILE locator.

Related Functions

OCILobDisableBuffering(), OCIErrorGet(), OCILobWrite(), OCILobRead(), OCILobFlushBuffer(), OCILobWriteAppend()

OCILobErase()

Purpose

Erases a specified portion of the internal LOB data starting at a specified offset.

Syntax

```
sword OCILobErase ( OCISvcCtx        *svchp,
                    OCIError         *errhp,
                    OCILobLocator    *locp,
                    ub4              *amount,
                    ub4              offset );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
An internal LOB locator that uniquely references the LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

amount (IN/OUT)
The number of characters for CLOBs/NCLOBs or bytes for BLOBs to erase. On IN, the value signifies the number of characters or bytes to erase. On OUT, the value identifies the actual number of characters or bytes erased.

offset (IN)
Absolute offset in characters for CLOBs/NCLOBs or bytes for BLOBs from the beginning of the LOB value from which to start erasing data. Starts at 1.

Comments

The actual number of characters/bytes erased is returned. For BLOBs, erasing means that zero-byte fillers overwrite the existing LOB value. For CLOBs, erasing means that spaces overwrite the existing LOB value.

This function is valid only for internal LOBs; FILEs are not allowed.

It is not mandatory that you wrap this LOB operation inside the Open/Close calls. If you did not open the LOB prior to performing this operation, then the functional and domain indexes on the LOB column are updated during this call. However, if you did open the LOB prior to performing this operation, then you must close it

before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. If you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

Related Functions

`OCIErrorGet()`, `OCILobRead()`, `OCILobAppend()`, `OCILobCopy()`, `OCILobWrite()`, `OCILobWriteAppend()`

OCILobFileClose()

Purpose

Closes a previously opened FILE.

Syntax

```
sword OCILobFileClose ( OCISvcCtx           *svchp,
                        OCIError            *errhp,
                        OCILobLocator       *filep );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

filep (IN/OUT)
A pointer to a FILE locator that refers to the FILE to be closed.

Comments

Closes a previously opened FILE. It is an error if this function is called for an internal LOB. No error is returned if the FILE exists but is not opened.

This function is only meaningful the first time it is called for a particular FILE locator. Subsequent calls to this function using the same FILE locator have no effect.

> **See Also:** For more information about FILEs, refer to the description of BFILEs in the *Oracle8i Application Developer's Guide - Large Objects (LOBs)*.

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobOpen(), OCILobIsOpen()

OCILobFileCloseAll()

Purpose

Closes all open FILEs on a given service context.

Syntax

```
sword OCILobFileCLoseAll ( OCISvcCtx   *svchp,
                           OCIError    *errhp );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

Comments

Closes all open FILEs on a given service context.

> **See Also:** For more information about FILEs, refer to the description of BFILEs in the *Oracle8i Application Developer's Guide - Large Objects (LOBs).*

Related Functions

OCILobFileClose(), OCIErrorGet(), OCILobFileExists(), OCILobFileIsOpen()

OCILobFileExists()

Purpose

Tests to see if the FILE exists on the server's operating system.

Syntax

```
sword OCILobFileExists ( OCISvcCtx        *svchp,
                         OCIError         *errhp,
                         OCILobLocator    *filep,
                         boolean          *flag );
```

Parameters

svchp (IN)
The OCI service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

filep (IN)
Pointer to the FILE locator that refers to the file.

flag (OUT)
Returns TRUE if the FILE exists on the server; FALSE if it does not.

Comments

Checks to see if the FILE exists on the server's file system. It is an error to call this function for an internal LOB.

> **See Also:** For more information about FILEs, refer to the description of BFILEs in the *Oracle8i Application Developer's Guide - Large Objects (LOBs)*.

Related Functions

OCIErrorGet(), OCILobFileClose(), OCILobFileCloseAll(), OCILobFileIsOpen(), OCILobOpen(), OCILobIsOpen()

OCILobFileGetName()

Purpose

Gets the FILE locator's directory alias and file name.

Syntax

```
sword OCILobFileGetName ( OCIEnv               *envhp,
                          OCIError             *errhp,
                          CONST OCILobLocator  *filep,
                          text                 *dir_alias,
                          ub2                  *d_length,
                          text                 *filename,
                          ub2                  *f_length );
```

Parameters

envhp (IN/OUT)
OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

filep (IN)
FILE locator for which to get the directory alias and file name.

dir_alias (OUT)
Buffer into which the directory alias name is placed. This can be in UTF-16. You must allocate enough space for the directory alias name. The maximum length for the directory alias is 30 bytes.

d_length (IN/OUT)
Serves the following purposes (can be in codepoint for Unicode, or bytes):

- IN: length of the input *dir_alias* string

- OUT: length of the returned *dir_alias* string

filename (OUT)
Buffer into which the file name is placed. You must allocate enough space for the file name. The maximum length for the file name is 255 bytes.

f_length (IN/OUT)
Serves the following purposes (in number of bytes):

- IN: length of the input *filename* buffer

■ OUT: length of the returned *filename* string

Comments

Returns the directory alias and file name associated with this FILE locator. The environment handle determines whether or not it is in Unicode. It is an error to call this function for an internal LOB.

> **See Also:** For more information about FILEs, refer to the description of BFILEs in the *Oracle8i Application Developer's Guide - Large Objects (LOBs).*

Related Functions

OCILobFileSetName(), OCIErrorGet()

OCILobFileIsOpen()

Purpose

Tests to see if the FILE is open

Syntax

```
sword OCILobFileIsOpen ( OCISvcCtx      *svchp,
                         OCIError       *errhp,
                         OCILobLocator  *filep,
                         boolean        *flag );
```

Parameters

svchp (IN)
The OCI service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the
event of an error.

filep (IN)
Pointer to the FILE locator being examined.

flag (OUT)
Returns TRUE if the FILE was opened using this particular locator; FALSE if it was
not.

Comments

Checks to see if a file on the server was opened with the *filep* FILE locator. It is an
error to call this function for an internal LOB.

If the input FILE locator was never passed to the OCILobFileOpen() or
OCILobOpen() command, the file is considered not to be opened by this locator.
However, a different locator may have the file open. Openness is associated with a
particular locator.

> **See Also:** For more information about FILEs, refer to the
> description of BFILEs in the *Oracle8i Application Developer's Guide -
> Large Objects (LOBs).*

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(),
OCILobFileExists(), OCILobFileClose(), OCILobFileOpen(),
OCILobOpen(), OCILobIsOpen()

OCILobFileOpen()

Purpose

Opens a FILE on the file system of the server for read-only access.

Syntax

```
sword OCILobFileOpen ( OCISvcCtx           *svchp,
                       OCIError            *errhp,
                       OCILobLocator       *filep,
                       ub1                 mode );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

filep (IN/OUT)
The FILE to open. It is an error if the locator does not refer to a FILE.

mode (IN)
Mode in which to open the file. The only valid mode is OCI_FILE_READONLY.

Comments

Opens a FILE on the file system of the server. The FILE can be opened for read-only access. FILEs may not be written through Oracle. It is an error to call this function for an internal LOB.

This function is only meaningful the first time it is called for a particular FILE locator. Subsequent calls to this function using the same FILE locator have no effect.

> **See Also:** For more information about FILEs, refer to the description of BFILEs in the *Oracle8i Application Developer's Guide - Large Objects (LOBs)*.

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileIsOpen(), OCILobOpen(), OCILobIsOpen()

OCILobFileSetName()

Purpose

Sets the directory alias and file name in the FILE locator.

Syntax

```
sword OCILobFileSetName ( OCIEnv            *envhp,
                          OCIError          *errhp,
                          OCILobLocator     **filepp,
                          CONST text        *dir_alias,
                          ub2               d_length,
                          CONST text        *filename,
                          ub2               f_length );
```

Parameters

envhp (IN/OUT)
OCI environment handle. Contains UTF-16 setting.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

filepp (IN/OUT)
Pointer to the FILE locator for which to set the directory alias and file name.

dir_alias (IN)
Buffer that contains the directory alias name (can be in UTF-16 in OCI_UTF16 environment) to set in the FILE locator.

d_length (IN)
Length of the input *dir_alias* parameter. In bytes.

filename (IN)
Buffer that contains the file name (can be in UTF-16 in OCI_UTF16 environment) to set in the FILE locator.

f_length (IN)
Length of the input *filename* parameter. In bytes.

Comments

It is an error to call this function for an internal LOB.

> **See Also:** For more information about FILEs, refer to the
> description of BFILEs in the *Oracle8i Application Developer's Guide -
> Large Objects (LOBs).*

Related Functions

OCILobFileGetName(), OCIErrorGet()

OCILobFlushBuffer()

Purpose

Flush/write all buffers for this lob to the server.

Syntax

```
sword OCILobFlushBuffer ( OCISvcCtx        *svchp,
                          OCIError         *errhp,
                          OCILobLocator    *locp
                          ub4              flag );
```

Parameters

svchp (IN/OUT)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
An internal locator uniquely referencing the LOB.

flag (IN)
When set to OCI_LOB_BUFFER_FREE, the buffer resources for the LOB are freed after the flush. See comments below.

Comments

Flushes to the server, changes made to the buffering subsystem that are associated with the LOB referenced by the input locator. This routine will actually write the data in the buffer to the LOB in the database. LOB buffering must have already been enabled for the input LOB locator.

The flush operation, by default, does not free the buffer resources for reallocation to another buffered LOB operation. However, if you want to free the buffer explicitly, you can set the flag parameter to OCI_LOB_BUFFER_FREE.

If the client application intends to read the buffer value after the flush and knows in advance that the current value in the buffer is the desired value, there is no need to reread the data from the server.

The effects of freeing the buffer are mostly transparent to the user, except that the next access to the same range in the LOB involves a round- trip to the server, and also the cost of acquiring buffer resources and initializing it with the data read from

the LOB. This option is intended for client environments that have low on-board memory.

Related Functions

OCILobEnableBuffering(), OCIErrorGet(), OCILobWrite(),
OCILobRead(), OCILobDisableBuffering(), OCILobWriteAppend()

OCILobFreeTemporary()

Purpose

Free a temporary LOB

Syntax

```
sword OCILobFreeTemporary( OCISvcCtx        *svchp,
                           OCIError         *errhp,
                           OCILobLocator    *locp);
```

Parameters

svchp (IN/OUT)
The OCI service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the
event of an error.

locp (IN/OUT)
A locator uniquely referencing the LOB to be freed.

Comments

This function frees the contents of the temporary LOB to which this locator points.
Note that the locator itself is not freed until OCIDescriptorFree()is called.

This function returns an error if the LOB locator passed in the *locp* parameter does
not point to a temporary LOB, which might be due to any of the following:

- It points to a permanent LOB

- It pointed to a temporary LOB which has already been freed

- It has never pointed to anything

Related functions

OCILobCreateTemporary(), OCILobIsTemporary(), OCIErrorGet()

OCILobGetChunkSize()

Purpose

Gets the chunk size of a LOB.

Syntax

```
sword OCILobGetChunkSize ( OCISvcCtx        *svchp,
                           OCIError         *errhp,
                           OCILobLocator    *locp,
                           ub4              *chunk_size );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
The internal LOB for which to get the usable chunk size.

chunk_size (OUT)
The amount of a chunk's space that is used to store the internal LOB value. This is the amount that users should use when reading/writing the LOB value. If possible, users should start their writes at chunk boundaries, such as the beginning of a chunk, and write a chunk at a time.

chunk_size will be returned in terms of bytes for BLOBs and in terms of characters for CLOBs and NCLOBs. For varying width character sets, the value will be the number of Unicode characters that fit in a chunk.

Comments

When creating a table that contains an internal LOB, the user can specify the chunking factor, which can be a multiple of Oracle blocks. This corresponds to the chunk size used by the LOB data layer when accessing/modifying the LOB value. Part of the chunk is used to store system-related information and the rest stores the LOB value. This function returns the amount of space used in the LOB chunk to store the LOB value. Performance will be improved if the application issues read/write requests using a multiple of this chunk size. For writes, there is an added benefit since LOB chunks are versioned and, if all writes are done on a chunk basis, no extra/excess versioning is done nor duplicated. Users could batch up the

write until they have enough for a chunk instead of issuing several write calls for the same chunk.

> **See Also:** "Functions for Improving LOB Read/Write Performance" on page 7-11

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobWrite(), OCILobWriteAppend()

OCILobGetLength()

Purpose

Gets the length of a LOB.

Syntax

```
sword OCILobGetLength ( OCISvcCtx       *svchp,
                        OCIError        *errhp,
                        OCILobLocator   *locp,
                        ub4             *lenp );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

locp (IN)
A LOB locator that uniquely references the LOB. For internal LOBs, this locator must be a locator that was obtained from the server specified by *svchp*. For FILEs, the locator can be set by `OCILobFileSetName()`, by a SELECT statement, or by `OCIObjectPin()`.

lenp (OUT)
On output, it is the length of the LOB if the LOB is not NULL. For character LOBs, it is the number of characters, for binary LOBs and BFILEs it is the number of bytes in the LOB.

Comments

Gets the length of a LOB. If the LOB is NULL, the length is undefined. The length of a FILE includes the EOF, if it exists. The length of an empty internal LOB is zero.

Regardless of whether the client-side character set is varying-width, the output length is in characters for CLOBs and NCLOBs, and in bytes for BLOBs and BFILEs.

> **Note:** Any zero-byte or space fillers in the LOB written by previous calls to `OCILobErase()` or `OCILobWrite()` are also included in the length count.

Related Functions

`OCIErrorGet()`, `OCILobFileSetName()`, `OCILobRead()`,
`OCILobWrite()`, `OCILobCopy()`, `OCILobAppend()`,
`OCILobLoadFromFile()`, `OCILobWriteAppend()`

OCILobIsEqual()

Purpose

Compares two LOB/FILE locators for equality.

Syntax

```
sword OCILobIsEqual ( OCIEnv              *envhp,
                      CONST OCILobLocator *x,
                      CONST OCILobLocator *y,
                      boolean             *is_equal );
```

Parameters

envhp (IN)
The OCI environment handle.

x (IN)
LOB locator to compare.

y (IN)
LOB locator to compare.

is_equal (OUT)
TRUE, if the LOB locators are equal; FALSE if they are not.

Comments

Compares the given LOB/FILE locators for equality. Two LOB/FILE locators are equal if and only if they both refer to the same LOB/FILE value.

Two NULL locators are considered *not* equal by this function.

Related Functions

OCILobAssign(), OCILobLocatorIsInit()

OCILobIsOpen()

Purpose

Tests whether a LOB/FILE is open.

Syntax

```
sword OCILobIsOpen ( OCISvcCtx        *svchp,
                     OCIError         *errhp,
                     OCILobLocator    *locp,
                     boolean          *flag );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle which can be passed to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN)
Pointer to the LOB locator being examined. The locator can refer to an internal or external LOB.

flag (OUT)
Returns TRUE if the internal LOB is open or if the BFILE was opened using the input locator. Returns FALSE if it was not.

Comments

Checks to see if the internal LOB is open or if the BFILE was already opened using the input locator.

For BFILES
If the input BFILE locator was never passed to OCILobOpen() or OCILobFileOpen(), the BFILE is considered not to be opened by this BFILE locator. However, a different BFILE locator may have opened the BFILE. More than one open can be performed on the same BFILE using different locators. In other words, openness is associated with a specific locator for BFILEs.

For internal LOBs
Openness is associated with the LOB, not with the locator. If *locator1* opened the LOB then *locator2* also sees the LOB as open.

For internal LOBs, this call requires a server round-trip because it checks the state on the server to see if the LOB is indeed open. For external LOBs (BFILEs), this call

also requires a round- trip because the actual operating system file on the server side must be checked to see if it is actually open.

> **See Also:** "Functions for Opening and Closing LOBs" on page 7-12

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobOpen()

OCILobIsTemporary()

Purpose

Tests if a locator points to a temporary LOB

Syntax

```
sword OCILobIsTemporary(OCIEnv           *envhp,
                        OCIError         *errhp,
                        OCILobLocator    *locp,
                        boolean          *is_temporary);
```

Parameters

envhp (IN)
The OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN)
The locator to test.

is_temporary (OUT)
Returns TRUE if the LOB locator points to a temporary LOB; FALSE if it does not.

Comments

This function tests a locator to determine if it points to a temporary LOB. If so, *is_temporary* is set to TRUE. If not, *is_temporary* is set to FALSE.

Related Functions

OCILobCreateTemporary(), OCILobFreeTemporary()

OCILobLoadFromFile()

Purpose

Load/copy all or a portion of the file into an internal LOB.

Syntax

```
sword OCILobLoadFromFile ( OCISvcCtx        *svchp,
                           OCIError         *errhp,
                           OCILobLocator    *dst_locp,
                           OCILobLocator    *src_locp,
                           ub4              amount,
                           ub4              dst_offset,
                           ub4              src_offset );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

dst_locp (IN/OUT)
A locator uniquely referencing the destination internal LOB which may be of type BLOB, CLOB, or NCLOB.

src_locp (IN/OUT)
A locator uniquely referencing the source FILE.

amount (IN)
The number of bytes to be loaded.

dst_offset (IN)
This is the absolute offset for the destination LOB. For character LOBs it is the number of characters from the beginning of the LOB at which to begin writing. For binary LOBs it is the number of bytes from the beginning of the LOB from which to begin reading. The offset starts at 1.

src_offset (IN)
This is the absolute offset for the source FILE. It is the number of bytes from the beginning of the FILE. The offset starts at 1.

Comments

Loads/copies a portion or all of a FILE value into an internal LOB as specified. The data is copied from the source FILE to the destination internal LOB (BLOB/CLOB). No character set conversions are performed when copying the FILE data to a CLOB/NCLOB. Also, when binary data is loaded into a BLOB, no character set conversions are performed. Therefore, the FILE data must already be in the same character set as the LOB in the database. No error checking is performed to verify this.

The source (*src_locp*) and the destination (*dst_locp*) LOBs must already exist. If the data already exists at the destination's start position, it is overwritten with the source data. If the destination's start position is beyond the end of the current data, zero-byte fillers (for BLOBs) or spaces (for CLOBs) are written into the destination LOB from the end of the data to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB.

It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes) or to try to copy from a NULL FILE.

It is not mandatory that you wrap this LOB operation inside the Open/Close calls. If you did not open the LOB prior to performing this operation, then the functional and domain indexes on the LOB column are updated during this call. However, if you did open the LOB prior to performing this operation, then you must close it before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. If you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

Related Functions

OCIErrorGet(), OCILobAppend(), OCILobWrite(), OCILobTrim(), OCILobCopy(), OCILobGetLength(), OCILobWriteAppend()

OCILobLocatorAssign()

Purpose

Assigns one LOB/FILE locator to another.

Syntax

```
sword OCILobLocatorAssign ( OCISvcCtx           *svchp,
                            OCIError            *errhp,
                            CONST OCILobLocator  *src_locp,
                            OCILobLocator         **dst_locpp );
```

Parameters

svchp (IN/OUT)
The OCI service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

src_locp (IN)
The LOB/FILE locator to copy from.

dst_locpp (IN/OUT)
The LOB/FILE locator to copy to. The caller must allocate space for the OCILobLocator by calling `OCIDescriptorAlloc()`.

Comments

This call assigns the source locator to the destination locator. After the assignment, both locators refer to the same LOB data. For internal LOBs, the source locator's LOB data gets copied to the destination locator's LOB data only when the destination locator gets stored in the table. Therefore, issuing a flush of the object containing the destination locator copies the LOB data. For FILEs only the locator that refers to the OS file is copied to the table; the OS file is not copied.

Note that this call is similar to `OCILobAssign()` but `OCILobLocatorAssign()` takes an OCI service handle pointer instead of an OCI environment handle pointer. Also, `OCILobLocatorAssign()` can be used for temporary LOBs and `OCILobAssign()` cannot be used for temporary LOBs.

> **Note:** If the `OCILobLocatorAssign()` function fails, the target
> locator will not be restored to its previous state. The target locator
> should not be used in subsequent operations unless it is
> reinitialized.

If the destination locator is for a temporary LOB, the destination temporary LOB is
freed before assigning the source LOB locator to it. If the source LOB locator refers
to a temporary LOB, the source temporary LOB is deep copied and a destination
locator is created to refer to the new deep copy of the temporary LOB. To avoid this
deep copy, the user should use the equal sign to ensure that two LOB locator
pointers refer to the same LOB locator.

Related Functions

OCIErrorGet(), OCILobAssign(), OCILobIsEqual(),
OCILobLocatorIsInit()

OCILobLocatorIsInit()

Purpose

Tests to see if a given LOB/FILE locator is initialized.

Syntax

```
sword OCILobLocatorIsInit ( OCIEnv                *envhp,
                            OCIError              *errhp,
                            CONST OCILobLocator   *locp,
                            boolean               *is_initialized );
```

Parameters

envhp (IN/OUT)
OCI environment handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN)
The LOB/FILE locator being tested

is_initialized (OUT)
Returns TRUE if the given LOB/FILE locator is initialized; FALSE if it is not.

Comments

Tests to see if a given LOB/FILE locator is initialized.

Internal LOB locators can be initialized by one of the following methods:

- selecting a non-NULL LOB into the locator,

- pinning an object that contains a non-NULL LOB attribute by OCIObjectPin()

- setting the locator to empty by OCIAttrSet()

  See Also: "LOB Locator Attributes" on page A-39

FILE locators can be initialized by one of the following methods:

- selecting a non-NULL FILE into the locator

- pinning an object that contains a non-NULL FILE attribute by OCIObjectPin()

- calling OCILobFileSetName()

Related Functions

OCIErrorGet(), OCILobIsEqual()

OCILobOpen()

Purpose

Opens a LOB, internal or external, in the indicated mode.

Syntax

```
sword OCILobOpen ( OCISvcCtx      *svchp,
                   OCIError       *errhp,
                   OCILobLocator  *locp,
                   ub1            mode );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
The LOB to open. The locator can refer to an internal or external LOB.

mode (IN)
The mode in which to open the LOB/BFILE. In Oracle8*i* or later, valid modes for LOBs are OCI_LOB_READONLY and OCI_LOB_READWRITE. Note that OCI_FILE_READONLY exists as input to OCILobFileOpen(). OCI_FILE_READONLY can be used with OCILobOpen() if the input locator is for a BFILE.

Comments

It is an error to open the same LOB twice. BFILEs cannot be opened in read-write mode. Note that if the LOB/BFILE was opened in read-only mode and the user tries to write to the LOB/BFILE, an error will be returned.

Opening a LOB requires a round- trip to the server for both internal and external LOBs. For internal LOBs, the open will trigger other code that relies on the open call. For external LOBs (BFILEs), open requires a round- trip because the actual operating system file on the server side is being opened.

It is not necessary to open a LOB in order to perform operations on it. When using functional indexes, extensible indexes or context, and making more than one call to update or write to the LOB, you should first call OCILobOpen(), then update the LOB as many times as you want, and finally call OCILobClose(). This sequence

of operations will ensure that the indexes are only updated once at the end of all the write operations instead of once per write operation.

It is not mandatory that you wrap all LOB operations inside the Open/Close calls. However, if you open a LOB, then you must close it before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column. It is an error to commit the transaction before closing all opened LOBs that were opened by the transaction.

When the error is returned, the LOB is no longer marked as open, but the transaction is successfully committed. Hence, all the changes made to the LOB and non-LOB data in the transaction are committed but the domain and functional indexing are not updated. If this happens, please rebuild your functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance, so if you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

> **See Also:** "Functions for Opening and Closing LOBs" on page 7-12

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobIsOpen()

OCILobRead()

Purpose

Reads a portion of a LOB/FILE, as specified by the call, into a buffer.

Syntax

```
sword OCILobRead ( OCISvcCtx          *svchp,
                   OCIError           *errhp,
                   OCILobLocator      *locp,
                   ub4                *amtp,
                   ub4                offset,
                   dvoid              *bufp,
                   ub4                bufl,
                   dvoid              *ctxp,
                   OCICallbackLobRead (cbfp)
                                      ( dvoid      *ctxp,
                                        CONST dvoid *bufp,
                                        ub4        len,
                                        ub1        piece )
                   ub2                csid,
                   ub1                csfrm );
```

Parameters

svchp (IN/OUT)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

locp (IN)
A LOB/FILE locator that uniquely references the LOB/FILE. This locator must be a locator that was obtained from the server specified by *svchp*.

amtp (IN/OUT)
The value in *amtp* is the amount in either bytes or characters, as shown in this table:

LOB/FILE	Input	Output with fixed-width client-side character set	Output with varying-width client-side character set
BLOBs and BFILEs	bytes	bytes	bytes

LOB/FILE	Input	Output with fixed-width client-side character set	Output with varying-width client-side character set
`CLOB`s and `NCLOB`s	characters	characters	bytes (1)

(1) The input amount refers to the number of characters to be read from the server-side `CLOB`/`NCLOB`. The output amount indicates how many bytes were read into the buffer *bufp*.

amtp is the total amount of data read if:

- data is not read in streamed mode (only one piece read and there is no polling or callback)

- data is read in streamed mode with a callback

amtp is the length of the last piece read if the data is read in streamed mode using polling.

If the amount to be read is larger than the buffer length it is assumed that the LOB is being read in a streamed mode from the input offset until the end of the LOB, or until the specified number of bytes have been read, whichever comes first. On input if this value is 0, then the data shall be read in streamed mode from the input offset until the end of the LOB.

The streamed mode (implemented with either polling or callbacks) reads the LOB value sequentially from the input offset.

If the data is read in pieces, *amtp* always contains the length of the piece just read.

If a callback function is defined, then this callback function will be invoked each time *bufl* bytes are read off the pipe. Each piece will be written into *bufp*.

If the callback function is not defined, then the OCI_NEED_DATA error code will be returned. The application must call OCILobRead() over and over again to read more pieces of the LOB until the OCI_NEED_DATA error code is not returned. The buffer pointer and the length can be different in each call if the pieces are being read into different sizes and locations.

offset (IN)
On input, this is the absolute offset from the beginning of the LOB value. For character LOBs (`CLOB`s, `NCLOB`s) it is the number of characters from the beginning of the LOB, for binary LOBs/FILEs it is the number of bytes. The first position is 1.

If you use streaming (by polling or a callback), then, specify the offset in the first call and in subsequent polling calls the offset parameter is ignored. When using a callback there is no offset parameter.

bufp (IN/OUT)
The pointer to a buffer into which the piece will be read. The length of the allocated memory is assumed to be *bufl*.

bufl (IN)
The length of the buffer in octets. This value will differ from the *amtp* value for CLOBs and for NCLOBs (*csfrm*=SQLCS_NCHAR) if the *amtp* parameter is specified in terms of characters, while the *bufl* parameter is specified in terms of bytes.

ctxp (IN)
The context pointer for the callback function. Can be null.

cbfp (IN)
A callback that may be registered to be called for each piece. If this is null, then OCI_NEED_DATA will be returned for each piece.

The callback function must return OCI_CONTINUE for the read to continue. If any other error code is returned, the LOB read is aborted.

ctxp (IN)
    The context for the callback function. Can be null.

bufp (IN/OUT)
    A buffer pointer for the piece.

len (IN)
    The length in bytes of the current piece in *bufp*.

piece (IN)
    Which piece: OCI_FIRST_PIECE, OCI_NEXT_PIECE or OCI_LAST_PIECE.

csid (IN)
The character set ID of the buffer data. If this value is 0 then *csid* is set to the client's NLS_LANG or NLS_CHAR value, depending on the value of *csfrm*. It is never assumed to be the server's character set, unless the server and client have the same settings.

csfrm (IN)
The character set form of the buffer data. The *csfrm* parameter must be consistent with the type of the LOB.

csfrm has two possible non-zero values:

- SQLCS_IMPLICIT - database character set ID

- SQLCS_NCHAR - NCHAR character set ID

The default value is SQLCS_IMPLICIT. If `csfrm` is not specified, the default is assumed.

Comments

Reads a portion of a LOB/FILE as specified by the call into a buffer. It is an error to try to read from a NULL LOB/FILE.

> **Note:** When reading or writing LOBs, the character set form (`csfrm`) specified should match the form of the locator itself.

For FILEs, the operating system file must already exist on the server, and it must have been opened by OCILobFileOpen() or OCILobOpen() using the input locator. Oracle must have permission to read the operating system file, and the user must have read permission on the directory object.

When using the polling mode for OCILobRead(), the first call needs to specify values for `offset` and `amtp`, but on subsequent polling calls to OCILobRead(), the user need not specify these values.

If the LOB is a BLOB, the `csid` and `csfrm` parameters are ignored.

> **Note:** To abort an OCILobRead() operation and free the statement handle, use the OCIBreak() call.

The following apply to client-side varying-width character sets for CLOBs and NCLOBs:

- When using polling mode, be sure to specify the `amtp` and `offset` parameters only in the first call to OCILobRead(). On subsequent polling calls, these parameters are ignored.

- When using callbacks, the `len` parameter, which is input to the callback, indicates how many bytes are filled in the buffer. Check the `len` parameter during your callback processing since the entire buffer may not be filled with data.

The following applies to client-side fixed-width character sets and server-side varying-width character sets for CLOBs and NCLOBs:

■ When reading a CLOB or NCLOB value, look at the *amtp* parameter after every call to OCILobRead() to see how much of the buffer is filled. When the return value is in characters (as when the client-side character set is fixed-width) then convert this value to bytes and determine how much of the buffer is filled. When using callbacks, always check the *len* parameter to see how much of the buffer is filled. This value is always in bytes.

To read data in UTF-16 format, set the *csid* parameter to OCI_UTF16ID. If the *csid* parameter is set, it overrides the NLS_LANG environment variable.

See Also:

■ For additional information on Unicode format, see "PL/SQL REF CURSORs and Nested Tables" on page 5-43

■ For more information about BFILEs, refer to the description of BFILEs in the *Oracle8i Application Developer's Guide - Large Objects (LOBs)*

■ For a code sample showing the use of LOB reads and writes, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs"

■ For general information about piecewise OCI operations, refer to "Runtime Data Allocation and Piecewise Operations" on page 5-44

Related Functions

OCIErrorGet(), OCILobWrite(), OCILobFileSetName(), OCILobWriteAppend()

OCILobTrim()

Purpose

Trims/truncates the LOB value to a shorter length.

Syntax

```
sword OCILobTrim ( OCISvcCtx        *svchp,
                   OCIError         *errhp,
                   OCILobLocator    *locp,
                   ub4              newlen );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
An internal LOB locator that uniquely references the LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

newlen (IN)
The new length of the LOB value, which must be less than or equal to the current length. For character LOBs, it is the number of characters, for binary LOBs and BFILEs it is the number of bytes in the LOB.

Comments

This function trims the LOB data to a specified shorter length. The function returns an error if *newlen* is greater than the current LOB length. This function is valid only for internal LOBs. FILEs are not allowed.

It is not mandatory that you wrap this LOB operation inside the Open/Close calls. If you did not open the LOB prior to performing this operation, then the functional and domain indexes on the LOB column are updated during this call. However, if you did open the LOB prior to performing this operation, then you must close it before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. If you have functional or domain indexes, we

recommend that you enclose write operations to the LOB within the open/close statements.

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobErase(), OCILobWrite(), OCILobWriteAppend()

OCILobWrite()

Purpose

Writes a buffer into a LOB

Syntax

```
sword OCILobWrite ( OCISvcCtx        *svchp,
                    OCIError         *errhp,
                    OCILobLocator    *locp,
                    ub4              *amtp,
                    ub4              offset,
                    dvoid            *bufp,
                    ub4              buflen,
                    ub1              piece,
                    dvoid            *ctxp,
                    OCICallbackLobWrite    (cbfp)
                                     (/*_
                                     dvoid  *ctxp,
                                     dvoid  *bufp,
                                     ub4    *lenp,
                                     ub1    *piecep */)
                    ub2              csid,
                    ub1              csfrm );
```

Parameters

svchp (IN/OUT)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

locp (IN/OUT)
An internal LOB locator that uniquely references the LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

amtp (IN/OUT)
The value in *amtp* is the amount in either bytes or characters, as shown in this table:

LOB/FILE	Input with fixed-width client-side character set	Input with varying-width client-side character set	Output
BLOBs and BFILEs	bytes	bytes	bytes
CLOBs and NCLOBs	characters	bytes (1)	characters

(1) The input amount refers to the number of bytes of data that the user wants to write into the LOB and not the number of bytes in the `bufp`, which is specified by `buflen`. In the case where data is written in pieces, the amount of bytes to write may be larger than the `buflen`. The output amount refers to the number of characters written into the server-side CLOB/NCLOB.

This should *always* be a non-null pointer. If you want to specify write-until-end-of-file, then you must declare a variable, set it equal to zero, and pass its address for this parameter.

If the amount is specified on input, and the data is written in pieces, `*amtp` will contain the total length of the pieces written at the end of the call (last piece written) and is undefined in between. Note that it is different from the piecewise read case. An error is returned if that amount is not sent to the server.

If `amtp` is zero, then streaming mode is assumed, and data is written until the user specifies OCI_LAST_PIECE.

offset (IN)
On input, it is the absolute offset from the beginning of the LOB value. For character LOBs it is the number of characters from the beginning of the LOB, for binary LOBs it is the number of bytes. The first position is 1.

If you use streaming (by polling or a callback), then, specify the offset in the first call and in subsequent polling calls the offset parameter is ignored. When using a callback there is no offset parameter.

bufp (IN)
The pointer to a buffer from which the piece will be written. The length of the data in the buffer is assumed to be the value passed in `buflen`. Even if the data is being written in pieces using the polling method, `bufp` must contain the first piece of the LOB when this call is invoked. If a callback is provided, `bufp` must not be used to provide data or an error will result.

buflen (IN)
The length, in bytes, of the data in the buffer. This value will differ from the `amtp` value for CLOBs and NCLOBs if the `amtp` parameter is specified in terms of characters, while the `buflen` parameter is specified in terms of bytes.

> **Note:** This parameter assumes an 8-bit byte. If your platform uses a longer byte, you must adjust the value of *buflen* accordingly.

piece (IN)
Which piece of the buffer is being written. The default value for this parameter is OCI_ONE_PIECE, indicating the buffer will be written in a single piece.

The following other values are also possible for piecewise or callback mode: OCI_FIRST_PIECE, OCI_NEXT_PIECE and OCI_LAST_PIECE.

ctxp (IN)
The context for the callback function. Can be null.

cbfp (IN)
A callback that may be registered to be called for each piece in a piecewise write. If this is null, the standard polling method will be used.

The callback function must return OCI_CONTINUE for the write to continue. If any other error code is returned, the LOB write is aborted. The callback takes the following parameters:

ctxp (IN)
The context for the callback function. Can be null.

bufp (IN/OUT)
A buffer pointer for the piece. This is the same as the *bufp* passed as an input to the OCILobWrite() routine.

lenp (IN/OUT)
The length, in bytes, of the data in the buffer (IN), and the length in bytes of current piece in *bufp* (OUT).

piecep (OUT)
Which piece: OCI_NEXT_PIECE or OCI_LAST_PIECE.

csid (IN)
The character set ID of the data in the buffer. If this value is 0 then *csid* is set to the client's NLS_LANG or NLS_CHAR value, depending on the value of *csfrm*.

csfrm (IN)
The character set form of the buffer data. The *csfrm* parameter must be consistent with the type of the LOB.

csfrm has two possible non-zero values:

- SQLCS_IMPLICIT - database character set ID
- SQLCS_NCHAR - NCHAR character set ID

The default value is SQLCS_IMPLICIT.

Comments

Writes a buffer into an internal LOB as specified. If LOB data already exists it is overwritten with the data stored in the buffer. The buffer can be written to the LOB in a single piece with this call, or it can be provided piecewise using callbacks or a standard polling method.

> **Note:** When reading or writing LOBs, the character set form (*csfrm*) specified should match the form of the locator itself.

When using the polling mode for OCILobWrite(), the first call needs to specify values for *offset* and *amtp*, but on subsequent polling calls to OCILobWrite(), the user need not specify these values.

If the value of the *piece* parameter is OCI_FIRST_PIECE, data may need to be provided through callbacks or polling.

If a callback function is defined in the *cbfp* parameter, then this callback function will be invoked to get the next piece after a piece is written to the pipe. Each piece will be written from *bufp*. If no callback function is defined, then OCILobWrite() returns the OCI_NEED_DATA error code. The application must call OCILobWrite() again to write more pieces of the LOB. In this mode, the buffer pointer and the length can be different in each call if the pieces are of different sizes and from different locations.

A *piece* value of OCI_LAST_PIECE terminates the piecewise write, regardless of whether the polling or callback method is used.

If the amount of data passed to Oracle (through either input mechanism) is less than the amount specified by the *amtp* parameter, an ORA-22993 error is returned.

This function is valid for internal LOBs only. FILEs are not allowed, since they are read-only. If the LOB is a BLOB, the *csid* and *csfrm* parameters are ignored.

If the client-side character set is varying-width, then the input amount is in bytes and the output amount is in characters for CLOBs and NCLOBs. The input amount refers to the number of bytes of data that the user wants to write into the LOB and not the number of bytes in the *bufp*, which is specified by *buflen*. In the case where data is written in pieces, the amount of bytes to write may be larger than the

buflen. The output amount refers to the number of characters written into the server-side CLOB/NCLOB.

To write data in UTF16 format, set the *csid* parameter to OCI_UTF16ID. If the *csid* parameter is set, it overrides the NLS_LANG environment variable.

It is not mandatory that you wrap this LOB operation inside the Open/Close calls. If you did not open the LOB prior to performing this operation, then the functional and domain indexes on the LOB column are updated during this call. However, if you did open the LOB prior to performing this operation, then you must close it before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. If you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

See Also:

■ For additional information on Unicode format, see "PL/SQL REF CURSORs and Nested Tables" on page 5-43

■ For a code sample showing the use of LOB reads and writes, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs"

■ For general information about piecewise OCI operations, refer to "Runtime Data Allocation and Piecewise Operations" on page 5-44

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobWriteAppend()

OCILobWriteAppend()

Purpose

Writes data starting at the end of a LOB.

Syntax

```
sword   OCILobWriteAppend ( OCISvcCtx *svchp,
                            OCIError *errhp,
                            OCILobLocator *locp,
                            ub4 *amtp,
                            dvoid *bufp,
                            ub4 buflen,
                            ub1 piece,
                            dvoid *ctxp,
                            OCICallbackLobWrite    (cbfp)
                                        (/*_
                                            dvoid   *ctxp,
                                            dvoid   *bufp,
                                            ub4     *lenp,
                                            ub1     *piecep */)
                            ub2 csid,
                            ub1 csfrm);
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

locp (IN/OUT)
An internal LOB locator that uniquely references a LOB.

amtp (IN/OUT)
The value in *amtp* is the amount in either bytes or characters, as shown in this table:

LOB/FILE	Input with fixed-width client-side character set	Input with varying-width client-side character set	Output
BLOBs and BFILEs	bytes	bytes	bytes
CLOBs and NCLOBs	characters	bytes (1)	characters

(1) The input amount refers to the number of bytes of data that the user wants to write into the LOB and not the number of bytes in the *bufp*, which is specified by *buflen*. In the case where data is written in pieces, the amount of bytes to write may be larger than the *buflen*. The output amount refers to the number of characters written into the server-side CLOB/NCLOB.

If the amount specified on input, and the data is written in pieces, **amtp* will contain the total length of the pieces written at the end of the call (last piece written) and is undefined in between. (Note it is different from the piecewise read case). An error is returned if that amount is not sent to the server. If amtp is zero, then streaming mode is assumed, and data is written until the user specifies OCI_LAST_PIECE.

If the client-side character set is varying-width, then the input amount is in bytes, not characters, for CLOBs/NCLOBs.

bufp (IN)
The pointer to a buffer from which the piece will be written. The length of the data in the buffer is assumed to be the value passed in *buflen*. Even if the data is being written in pieces, *bufp* must contain the first piece of the LOB when this call is invoked. If a callback is provided, bufp must not be used to provide data or an error will result.

buflen (IN)
The length, in bytes, of the data in the buffer. Note that this parameter assumes an 8-bit byte. If your platform uses a longer byte, the value of *buflen* must be adjusted accordingly.

piece (IN)
Which piece of the buffer is being written. The default value for this parameter is OCI_ONE_PIECE, indicating the buffer will be written in a single piece. The following other values are also possible for piecewise or callback mode: OCI_FIRST_PIECE, OCI_NEXT_PIECE and OCI_LAST_PIECE.

ctxp (IN)
The context for the call back function. Can be null.

cbfp (IN)
A callback that may be registered to be called for each piece in a piecewise write. If this is null, the standard polling method will be used. The callback function must return OCI_CONTINUE for the write to continue. If any other error code is returned, the LOB write is aborted. The callback takes the following parameters:

ctxp (IN)
The context for the callback function. Can be null.

bufp (IN/OUT)
A buffer pointer for the piece.

lenp (IN/OUT)
The length, in bytes, of the data in the buffer (IN), and the length in bytes of current piece in *bufp* (OUT).

piecep (OUT)
Which piece: OCI_NEXT_PIECE or OCI_LAST_PIECE.

csid (IN)
The character set ID of the buffer data.

csfrm (IN)
The character set form of the buffer data.

csfrm has two possible non-zero values:

- SQLCS_IMPLICIT - database character set ID
- SQLCS_NCHAR - NCHAR character set ID

The default value is SQLCS_IMPLICIT.

Comments

The buffer can be written to the LOB in a single piece with this call, or it can be provided piecewise using callbacks or a standard polling method. If the value of the piece parameter is OCI_FIRST_PIECE, data must be provided through callbacks or polling. If a callback function is defined in the *cbfp* parameter, then this callback function will be invoked to get the next piece after a piece is written to the pipe. Each piece will be written from *bufp*. If no callback function is defined, then OCILobWriteAppend() returns the OCI_NEED_DATA error code.

The application must call OCILobWriteAppend() again to write more pieces of the LOB. In this mode, the buffer pointer and the length can be different in each call if the pieces are of different sizes and from different locations. A piece value of OCI_LAST_PIECE terminates the piecewise write.

OCILobWriteAppend() is not supported if LOB buffering is enabled.

If the LOB is a BLOB, the *csid* and *csfrm* parameters are ignored.

If the client-side character set is varying-width, then the input amount is in bytes, not characters, for CLOBs/NCLOBs.

It is not mandatory that you wrap this LOB operation inside the Open/Close calls. If you did not open the LOB prior to performing this operation, then the functional and domain indexes on the LOB column are updated during this call. However, if

you did open the LOB prior to performing this operation, then you must close it before you commit or rollback your transaction. When an internal LOB is closed, it updates the functional and domain indexes on the LOB column.

If you do not wrap your LOB operations inside the Open/Close API, then the functional and domain indexes are updated each time you write to the LOB. This can adversely affect performance. If you have functional or domain indexes, we recommend that you enclose write operations to the LOB within the open/close statements.

> **See Also:** "Functions for Improving LOB Read/Write Performance" on page 7-11

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobWrite()

Advanced Queuing and Publish-Subscribe Functions

This section describes the Advanced Queuing and publish-subscribe functions.

Table 16–2   Advanced Queuing and Publish-Subscribe Functions

Function	Purpose
OCIAQDeq() on page 16-71	Advanced Queuing dequeue
OCIAQEnq() on page 16-73	Advanced Queuing enqueue
OCIAQListen() on page 16-85	Listens on one or more queues on behalf of a list of agents
OCISubscriptionEnable() on page 16-88	Enables notifications on a subscription
OCISubscriptionPost() on page 16-89	Posts to a subscription to receive notifications
OCISubscriptionRegister() on page 16-91	Registers a subscription
OCISubscriptionUnRegister() on page 16-94	Unregisters a subscription

OCIAQDeq()

Purpose

This call is used for an Advanced Queuing dequeue operation using the OCI.

Syntax

```
sword OCIAQDeq ( OCISvcCtx          *svch,
                 OCIError           *errh,
                 text               *queue_name,
                 OCIAQDeqOptions    *dequeue_options,
                 OCIAQMsgProperties *message_properties,
                 OCIType            *payload_tdo,
                 dvoid              **payload,
                 dvoid              **payload_ind,
                 OCIRaw             **msgid,
                 ub4                flags );
```

Parameters

svch (IN)
OCI service context.

errh (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

queue_name (IN)
The target queue for the dequeue operation.

dequeue_options (IN)
The options for the dequeue operation; stored in an **OCIAQDeqOptions** descriptor.

message_properties (OUT)
The message properties for the message; stored in an **OCIAQMsgProperties** descriptor.

payload_tdo (IN)
The TDO (type descriptor object) of an object type. For a raw queue, this parameter should point to the TDO of SYS.RAW.

payload (IN/OUT)
A pointer to a pointer to a program variable buffer that is an instance of an object type. For a raw queue, this parameter should point to an instance of **OCIRaw**.

Memory for the payload is dynamically allocated in the object cache. The application can optionally call OCIObjectFree() to deallocate the payload

instance when it is no longer needed. If the pointer to the program variable buffer (*`payload`*) is passed as null, the buffer is implicitly allocated in the cache.

The application may choose to pass null for `payload` the first time `OCIAQDeq()` is called, and let the OCI allocate the memory for the payload. It can then use a pointer to that previously allocated memory in subsequent calls to `OCIAQDeq()`.

To obtain a TDO for the payload, use `OCITypeByName()`, or `OCITypeByRef()`.

The OCI provides functions which allow the user to set attributes of the payload, such as its text. For information about setting these attributes, refer to "Manipulating Object Attributes" on page 10-14.

payload_ind (IN/OUT)
A pointer to a pointer to the program variable buffer containing the parallel indicator structure for the object type.

The memory allocation rules for `payload_ind` are the same as those for `payload`, above.

msgid (OUT)
The message ID.

flags (IN)
Not currently used; pass as OCI_DEFAULT.

Comments

Users must have the `aq_user_role` or privileges to execute the `dbms_aq` package in order to use this call. The OCI environment must be initialized in object mode (using `OCIInitialize()`) to use this call.

> **See Also:**
>
> ■ For more information about OCI and Advanced Queuing, refer to "OCI and Advanced Queuing" on page 9-42
>
> ■ For additional information about Advanced Queuing, refer to *Oracle8i Application Developer's Guide - Advanced Queuing*

Examples

For code examples, refer to the description of `OCIAQEnq()` on page 16-73.

Related Functions

`OCIAQEnq()`, `OCIAQListen()`, `OCIInitialize()`

OCIAQEnq()

Purpose

This call is used for an Advanced Queuing enqueue.

Syntax

```
sword OCIAQEnq ( OCISvcCtx          *svch,
                 OCIError           *errh,
                 text               *queue_name,
                 OCIAQEnqOptions    *enqueue_options,
                 OCIAQMsgProperties *message_properties,
                 OCIType            *payload_tdo,
                 dvoid              **payload,
                 dvoid              **payload_ind,
                 OCIRaw             **msgid,
                 ub4                flags );
```

Parameters

svch (IN)
OCI service context.

errh (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

queue_name (IN)
The target queue for the enqueue operation.

enqueue_options (IN)
The options for the enqueue operation; stored in an **OCIAQEnqOptions** descriptor.

message_properties (IN)
The message properties for the message; stored in an **OCIAQMsgProperties** descriptor.

payload_tdo (IN)
The TDO (type descriptor object) of an object type. For a raw queue, this parameter should point to the TDO of SYS.RAW.

payload (IN)
A pointer to a pointer to an instance of an object type. For a raw queue, this parameter should point to an instance of **OCIRaw**.

The OCI provides functions which allow the user to set attributes of the payload, such as its text.

> **See Also:** For information about setting these attributes, refer to
> "Manipulating Object Attributes" on page 10-14

payload_ind (IN)
A pointer to a pointer to the program variable buffer containing the parallel
indicator structure for the object type.

msgid (OUT)
The message ID.

flags (IN)
Not currently used; pass as OCI_DEFAULT.

Comments

Users must have the `aq_user_role` or privileges to execute the `dbms_aq` package
in order to use this call.

The OCI environment must be initialized in object mode (using
`OCIInitialize()`) to use this call.

> **See Also:**
>
> - For more information about OCI and Advanced Queuing, refer
>   to "OCI and Advanced Queuing" on page 9-42
>
> - For more information about Advanced Queuing, refer to
>   *Oracle8i Application Developer's Guide - Advanced Queuing*

To obtain a TDO for the payload, use `OCITypeByName()`, or `OCITypeByRef()`.

Examples

The following four examples demonstrate the use of `OCIAQEnq()` and
`OCIAQDeq()` in several different situations.

> **See Also:** These examples assume that the database is set up as
> illustrated in the section "Oracle Advanced Queuing By Example"
> in the Advanced Queuing chapter of the *Oracle8i Application
> Developer's Guide - Advanced Queuing*

Example 1 - Enqueue And Dequeue Of A Payload Object.

```
struct message
{
  OCIString   *subject;
  OCIString   *data;
```

```
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message   msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                  52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                   52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                   52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                  52, (dvoid **) &tmp);
  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
                  (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
```

```
                 OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

                 /* obtain TDO of message_type */
                 OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
                               (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
                               (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

                 /* prepare the message payload */
                 mesg->subject = (OCIString *)0;
                 mesg->data = (OCIString *)0;
                 OCIStringAssignText(envhp, errhp, (CONST text *)"NORMAL MESSAGE",
                               strlen("NORMAL MESSAGE"), &mesg->subject);
                 OCIStringAssignText(envhp, errhp,(CONST text *)"OCI ENQUEUE",
                               strlen("OCI ENQUEUE"), &mesg->data);
                 nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

                 /* enqueue into the msg_queue */
                 OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
                               mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
                 OCITransCommit(svchp, errhp, (ub4) 0);

                 /* dequeue from the msg_queue */
                 OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
                               mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
                 printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
                 printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
                 OCITransCommit(svchp, errhp, (ub4) 0);
                 }
```

Example 2 - Enqueue and Dequeue Using Correlation Ids.

```
struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;
```

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;
  OCIRaw*firstmsg = (OCIRaw *)0;
  OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
  OCIAQDeqOptions *deqopt = (OCIAQDeqOptions *)0;
  text correlation1[30], correlation2[30];

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );
  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);
  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
                (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* allocate message properties descriptor */
  OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
                  OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
  strcpy(correlation1, "1st message");
```

```
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)&correlation1,
                strlen(correlation1), OCI_ATTR_CORRELATION, errhp);

/* obtain TDO of message_type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
            (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
            (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIString *)0;
mesg->data = (OCIString *)0;
OCIStringAssignText(envhp, errhp, (CONST text *)"NORMAL ENQUEUE1",
                strlen("NORMAL ENQUEUE1"), &mesg->subject);
OCIStringAssignText(envhp, errhp,(CONST text *)"OCI ENQUEUE",
                strlen("OCI ENQUEUE"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue into the msg_queue, store the message id into firstmsg */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, msgprop,
                mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, &firstmsg, 0);

/* enqueue into the msg_queue with a different correlation id */
strcpy(correlation2, "2nd message");
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid*)&correlation2,
                strlen(correlation2), OCI_ATTR_CORRELATION, errhp);
OCIStringAssignText(envhp, errhp, (CONST text *)"NORMAL ENQUEUE2",
                strlen("NORMAL ENQUEUE2"), &mesg->subject);
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, msgprop,
                mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

OCITransCommit(svchp, errhp, (ub4) 0);

/* first dequeue by correlation id "2nd message" */
/* allocate dequeue options descriptor and set the correlation option */
OCIDescriptorAlloc(envhp, (dvoid **)&deqopt,
                OCI_DTYPE_AQDEQ_OPTIONS, 0, (dvoid **)0);
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)correlation2,
                strlen(correlation2),  OCI_ATTR_CORRELATION, errhp);

/* dequeue from the msg_queue */
OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", deqopt, 0,
                mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
OCITransCommit(svchp, errhp, (ub4) 0);
```

```
     /* second dequeue by message id */
     OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&firstmsg,
     OCIRawSize(envhp, firstmsg), OCI_ATTR_DEQ_MSGID, errhp);
     /* clear correlation id option */
     OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
                 (dvoid *)correlation2, 0, OCI_ATTR_CORRELATION, errhp);

     /* dequeue from the msg_queue */
     OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", deqopt, 0,
                 mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
   printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
   printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
   OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Example 3 - Enqueue and Dequeue Of A Raw Queue.

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid     *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  char  msg_text[100];
  OCIRaw  *mesg = (OCIRaw *)0;
  OCIRaw*deqmesg = (OCIRaw *)0;
  OCIInd   ind = 0;
  dvoid  *indptr = (dvoid *)&ind;
  inti;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
              (dvoid * (*)()) 0,  (void (*)()) 0 );
  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                  52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                  52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                  52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
```

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
                (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* obtain the TDO of the RAW data type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"SYS", strlen("SYS"),
              (CONST text *)"RAW", strlen("RAW"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
strcpy(msg_text, "Enqueue to a RAW queue");
OCIRawAssignBytes(envhp, errhp, msg_text, strlen(msg_text), &mesg);

/* enqueue the message into raw_msg_queue */
OCIAQEnq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
              mesg_tdo, (dvoid **)&mesg, (dvoid **)&indptr, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue the same message into C variable deqmesg */
OCIAQDeq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
 mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&indptr, 0, 0);
for (i = 0; i < OCIRawSize(envhp, deqmesg); i++)
  printf("%c", *(OCIRawPtr(envhp, deqmesg) + i));
OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Example 4 - Enqueue and Dequeue Using OCIAQAgent.

```
struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;
```

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  dvoid      *tmp;
  OCIType *mesg_tdo = (OCIType *) 0;
  message  msg;
  null_message nmsg;
  message *mesg = &msg;
  null_message *nmesg = &nmsg;
  message *deqmesg = (message *)0;
  null_message *ndeqmesg = (null_message *)0;
  OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
  OCIAQAgent *agents[2];
  OCIAQDeqOptions *deqopt = (OCIAQDeqOptions *)0;
  ub4wait = OCI_DEQ_NO_WAIT;
  ub4 navigation = OCI_DEQ_FIRST_MSG;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                  52, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                  52, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                  52, (dvoid **) &tmp);

  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
                  (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* obtain TDO of message_type */
```

```
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
      (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
      (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIString *)0;
mesg->data = (OCIString *)0;
OCIStringAssignText(envhp, errhp,
                    (CONST text *)"MESSAGE 1", strlen("MESSAGE 1"),
                    &mesg->subject);
OCIStringAssignText(envhp, errhp,
                    (CONST text *)"mesg for queue subscribers",
                    strlen("mesg for queue subscribers"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue MESSAGE 1 for subscribers to the queue i.e. for RED and GREEN */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, 0,
                       mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

/* enqueue MESSAGE 2 for specified recipients i.e. for RED and BLUE */
/* prepare message payload */
OCIStringAssignText(envhp, errhp,
                    (CONST text *)"MESSAGE 2", strlen("MESSAGE 2"),
                    &mesg->subject);
OCIStringAssignText(envhp, errhp,
                    (CONST text *)"mesg for two recipients",
                    strlen("mesg for two recipients"), &mesg->data);

/* allocate AQ message properties and agent descriptors */
OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
                OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
OCIDescriptorAlloc(envhp, (dvoid **)&agents[0],
                  OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
OCIDescriptorAlloc(envhp, (dvoid **)&agents[1],
                  OCI_DTYPE_AQAGENT, 0, (dvoid **)0);

/* prepare the recipient list, RED and BLUE */
OCIAttrSet(agents[0], OCI_DTYPE_AQAGENT, "RED", strlen("RED"),
                    OCI_ATTR_AGENT_NAME, errhp);
OCIAttrSet(agents[1], OCI_DTYPE_AQAGENT, "BLUE", strlen("BLUE"),
                    OCI_ATTR_AGENT_NAME, errhp);
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)agents, 2,
                    OCI_ATTR_RECIPIENT_LIST, errhp);

OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, msgprop,
```

```
                        mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
        OCITransCommit(svchp, errhp, (ub4) 0);

        /* now dequeue the messages using different consumer names */
        /* allocate dequeue options descriptor to set the dequeue options */
        OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
                        (dvoid **)0);

        /* set wait parameter to NO_WAIT so that the dequeue returns immediately */
        OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&wait, 0,
                        OCI_ATTR_WAIT, errhp);

        /* set navigation to FIRST_MESSAGE so that the dequeue resets the position */
        /* after a new consumer_name is set in the dequeue options      */
        OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&navigation, 0,
                        OCI_ATTR_NAVIGATION, errhp);

        /* dequeue from the msg_queue_multiple as consumer BLUE */
        OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"BLUE", strlen("BLUE"),
                        OCI_ATTR_CONSUMER_NAME, errhp);
        while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
        mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
        == OCI_SUCCESS)
        {
          printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
          printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
        }
        OCITransCommit(svchp, errhp, (ub4) 0);

        /* dequeue from the msg_queue_multiple as consumer RED */
        OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"RED", strlen("RED"),
                        OCI_ATTR_CONSUMER_NAME, errhp);
        while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
                        mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
        == OCI_SUCCESS)
        {
          printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
          printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
        }
        OCITransCommit(svchp, errhp, (ub4) 0);

        /* dequeue from the msg_queue_multiple as consumer GREEN */
        OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,(dvoid *)"GREEN",strlen("GREEN"),
                        OCI_ATTR_CONSUMER_NAME, errhp);
        while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
```

```
                        mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
 == OCI_SUCCESS)
  {
    printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
  }
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Related Functions

OCIAQDeq(), OCIAQListen(), OCIInitialize()

OCIAQListen()

Purpose

Listens on one or more queues on behalf of a list of agents.

Syntax

```
sword OCIAQListen (OCISvcCtx        *svchp,
                   OCIError         *errhp,
                   OCIAQAgent       **agent_list,
                   ub4              num_agents,
                   sb4              wait,
                   OCIAQAgent       **agent,
                   ub4              flags);
```

Parameters

svchpp (IN/OUT)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

agent_list (IN)
List of agents for which to monitor messages.

num_agents (IN)
Number of agents in the agent list.

wait (IN)
Time-out for the listen call.

agent (OUT)
Agent for which there is a message. OCIAgent is an OCI descriptor.

flags (IN)
Not currently used; pass as OCI_DEFAULT.

Comments

This is a blocking call that returns when there is a message ready for consumption for an agent in the list. If there are no messages found when the wait time expires, an error is returned.

Related Functions

OCIAQEnq(), OCIAQDeq(), OCISvcCtxToLda(),
OCISubscriptionEnable(), OCISubscriptionPost(),
OCISubscriptionRegister(),OCISubscriptionUnRegister()

OCISubscriptionDisable()

Purpose

Disables a subscription registration which turns off all notifications.

Syntax

```
ub4 OCISubscriptionDisable ( OCISubscription   *subscrhp,
                             OCIError          *errhp
                             ub4               mode );
```

Parameters

subscrhp (IN)
A subscription handle with the OCI_ATTR_SUBSCR_NAME and
OCI_ATTR_SUBSCR_NAMESPACE attributes set.

> **See Also:** For information, see Subscription Handle Attributes on
> page A-53

errhp (OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the
event of an error.

mode (IN)
Call-specific mode. Valid values:

- OCI_DEFAULT - executes the default call which discards all notifications on
  this subscription until the subscription is enabled

Comments

This call is used to temporarily turn off notifications. This is useful when the
application is running a critical section of the code and should not be interrupted.

The user need not be connected or authenticated to perform this operation. A
registration must have been performed to the subscription specified by the
subscription handle before this call is made.

All notifications subsequent to an `OCISubscriptionDisable()` are discarded by
the system until an `OCISubscriptionEnable()` is performed.

Related Functions

OCIAQListen(), OCISubscriptionEnable(), OCISubscriptionPost(),
OCISubscriptionRegister(), OCISubscriptionUnRegister()

OCISubscriptionEnable()

Purpose

Enables a subscription registration that has been disabled. This turns on all notifications.

Syntax

```
ub4 OCISubscriptionEnable ( OCISubscription    *subscrhp,
                            OCIError           *errhp
                            ub4                mode );
```

Parameters

subscrhp (IN)
A subscription handle with the OCI_ATTR_SUBSCR_NAME and
OCI_ATTR_SUBSCR_NAMESPACE attributes set.

> **See Also:** For information, see Subscription Handle Attributes on
> page A-53

errhp (OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the
event of an error.

mode (IN)
Call-specific mode. Valid value:

- OCI_DEFAULT - executes the default call which buffers all notifications on this
  subscription until a subsequent enable is performed

Comments

This call is used to turn on notifications after a subscription registration has been
disabled.

The user need not be connected or authenticated to perform this operation. A
registration must have been done for the specified subscription before this call is
made.

Related Functions

OCIAQListen(), OCISvcCtxToLda(), OCISubscriptionPost(),
OCISubscriptionRegister(), OCISubscriptionUnRegister()

OCISubscriptionPost()

Purpose

Posts to a subscription which allows all clients who are registered for the subscription to get notifications.

Syntax

```
ub4 OCISubscriptionPost ( OCISvcCtx        *svchp,
                          OCISubscription  **subscrhpp,
                          ub2              count,
                          OCIError         *errhp
                          ub4              mode );
```

Parameters

svchp (IN)
An OCI service context (after release 7). This service context should have a valid authenticated user handle.

subscrhpp (IN)
An array of subscription handles. Each element of this array should be a subscription handle with the OCI_ATTR_SUBSCR_NAME and OCI_ATTR_SUBSCR_NAMESPACE attributes set.

> **See Also:** For information, see Subscription Handle Attributes on page A-53

The OCI_ATTR_SUBSCR_PAYLOAD attribute has to be set for each subscription handle prior to this call. If it is not set, the payload is assumed to be NULL and no payload is delivered when the notification is received by the clients that have registered interest. Note that the caller will have to preserve the payload until the post is done as the OCIAttrSet() call keeps track of the reference to the payload but does not copy the contents.

count (IN)
The number of elements in the subscription handle array.

errhp (OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

mode (IN)
Call-specific mode. Valid value:

- OCI_DEFAULT - executes the default call

Comments

Posting to a subscription involves identifying the subscription name and the payload if desired. If no payload is associated, the payload length can be set to 0.

This call provides a *best-effort* guarantee. A notification does to registered clients at most once.

This call is primarily used for light-weight notification and is useful in the case of several system events. If the application needs more rigid guarantees, it can use the Advanced Queuing functionality by enqueuing to queue.

Related Functions

OCIAQListen(), OCISvcCtxToLda(), OCISubscriptionEnable(), OCISubscriptionRegister(), OCISubscriptionUnRegister()

OCISubscriptionRegister()

Purpose

Registers a callback for message notification.

Syntax

```
ub4 OCISubscriptionRegister ( OCISvcCtx        *svchp,
                              OCISubscription  **subscrhpp,
                              ub2              count,
                              OCIError         *errhp
                              ub4              mode );
```

Parameters

svchp (IN)
An OCI service context (after release 7). This service context should have a valid
authenticated user handle.

subscrhpp (IN)
An array of subscription handles. Each element of this array should be a
subscription handle with all of the following attributes set:

- OCI_ATTR_SUBSCR_NAME,

- OCI_ATTR_SUBSCR_NAMESPACE,

- OCI_ATTR_SUBSCR_RECPTPROTO.

Otherwise, an error will be returned.

One of attributes

- OCI_ATTR_SUBSCR_CBACK,

- OCI_ATTR_SUBSCR_CTX,

- OCI_ATTR_SUBSCR_RECPT,

must also be set.

> **See Also:** For information about the handle attributes, see
> Subscription Handle Attributes on page A-53

When a notification is received for the registration denoted by *subscrhpp[i]*,
either the user-defined callback function (OCI_ATTR_SUBSCR_CBACK) set for
subscrhpp[i] will be invoked with the context (OCI_ATTR_SUBSCR_CTX) set
for *subscrhpp[i]*, or an e-mail will be sent to (OCI_ATTR_SUBSCR_RECPT) set

for *subscrhpp[i],* or the PL/SQL procedure (OCI_ATTR_SUBSCR_RECPT) set for *subscrhpp[i],* will be invoked in the database, provided the subscriber of *subscrhpp[i]* has the appropriate permissions on the procedure.

count (IN)
The number of elements in the subscription handle array.

errhp (OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

mode (IN)
Call-specific mode. Valid values:

- OCI_DEFAULT - executes the default call which specifies that the registration is treated as disconnected

- OCI_NOTIFY_CONNECTED - notifications are received only if the client is connected (not supported in this release)

Whenever a new client process comes up, or an old one goes down and comes back up, it needs to register for all subscriptions of interest. If the client stays up and the server first goes down and then comes back up, the client will continue to receive notifications for registrations that are DISCONNECTED. However, the client will not receive notifications for CONNECTED registrations as they will be lost once the server goes down and comes back up.

Comments

This call is invoked for registration to a subscription which identifies the subscription name of interest and the associated callback to be invoked. Interest in several subscriptions can be registered at one time.

This interface is only valid for the asynchronous mode of message delivery. In this mode, a subscriber issues a registration call which specifies a callback. When messages are received that match the subscription criteria, the callback is invoked. The callback may then issue an explicit message_receive (dequeue) to retrieve the message.

The user must specify a subscription handle at registration time with the namespace attribute set to OCI_SUBSCR_NAMESPACE_AQ.

The subscription name is the string '*SCHEMA.QUEUE*' if the registration is for a single consumer queue and '*CONSUMER_NAME:SCHEMA.QUEUE*' if the registration is for a multi-consumer queue. The string should be in uppercase.

Each namespace will have its own privilege model. If the user performing the register is not entitled to register in the namespace for the specified subscription, an error is returned.

Related Functions

`OCIAQListen()`, `OCISvcCtxToLda()`, `OCISubscriptionEnable()`, `OCISubscriptionPost()`, `OCISubscriptionUnRegister()`

OCISubscriptionUnRegister()

Purpose

Unregisters a subscription which turns off notifications.

Syntax

```
ub4 OCISubscriptionUnRegister ( OCISvcCtx       *svchp,
                                OCISubscription *subscrhp,
                                OCIError        *errhp
                                ub4             mode );
```

Parameters

svchp (IN)
An OCI service context (after release 7). This service context should have a valid authenticated user handle.

subscrhp (IN)
A subscription handle with the OCI_ATTR_SUBSCR_NAME and OCI_ATTR_SUBSCR_NAMESPACE attributes set.

> **See Also:** For information, see Subscription Handle Attributes on page A-53

errhp (OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

mode (IN)
Call-specific mode. Valid value:

- OCI_DEFAULT - executes the default call

Comments

Unregistering to a subscription is going to ensure that the user will not receive notifications regarding the specified subscription in future. If the user wishes to resume notification, then the only option is to re-register to the subscription.

All notifications that would otherwise have been delivered are not delivered after a subsequent register is performed because the user is no longer in the list of interested clients.

Related Functions

`OCIAQListen()`, `OCISvcCtxToLda()`, `OCISubscriptionEnable()`,
`OCISubscriptionPost()`, `OCISubscriptionRegister()`.

Direct Path Loading Functions

This section describes the direct path loading functions.

Table 16–3   Direct Path Loading Functions

Function	Purpose
OCIDirPathAbort() on page 16-97	Aborts a direct path operation
OCIDirPathColArrayEntryGet() on page 16-98	Gets a specified entry in a column array
OCIDirPathColArrayEntrySet() on page 16-100	Sets a specified entry in a column array to a specific value
OCIDirPathColArrayRowGet() on page 16-102	Gets the base row pointers for a specified row number
OCIDirPathColArrayReset() on page 16-104	Resets the row array state
OCIDirPathColArrayToStream() on page 16-105	Converts from a column array to a direct path stream format
OCIDirPathDataSave() on page 16-107	Does a data savepoint, or commits the loaded data and finishes the load operation
OCIDirPathFinish() on page 16-108	Finishes and commits the loaded data
OCIDirPathFlushRow() on page 16-109	Loads data that has been converted to direct path stream format
OCIDirPathPrepare() on page 16-112	Prepares direct path interface to convert or load rows

OCIDirPathAbort()

Purpose

Aborts a direct path operation.

Syntax

```
sword OCIDirPathAbort  ( OCIDirPathCtx           *dpctx,
                         OCIError                *errhp );
```

Parameters

dpctx (IN)
Direct path context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the
event of an error.

Comments

All state maintained by the server on behalf of the direct path operation is
destroyed by an abort. For a direct path load, the data loaded prior to the abort will
not be visible to any queries. However, the data may still consume space in the
segments that are being loaded. Any load completion operations, such as index
maintenance operations, are not performed.

Related Functions

OCIDirPathFinish(), OCIDirPathFlushRow(), OCIDirPathPrepare(),
OCIDirPathLoadStream(), OCIDirPathStreamReset(),
OCIDirPathDataSave()

OCIDirPathColArrayEntryGet()

Purpose

Gets a specified entry in a column array.

Syntax

```
sword OCIDirPathColArrayEntryGet ( OCIDirPathColArray   *dpca,
                                   OCIError             *errhp,
                                   ub4                  rownum,
                                   ub2                  colIdx,
                                   ub1                  **cvalpp,
                                   ub4                  *clenp,
                                   ub1                  *cflgp );
```

Parameters

dpca (IN/OUT)
Direct path column array handle.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

rownum (IN)
Zero-based row offset

colIdx (IN)
Column identifier (index), the column ID is returned by `OCIDirPathColAttrSet()`

cvalpp (IN/OUT)
Pointer to pointer to column data

clenp (IN/OUT)
Pointer to length of column data

cflgp (IN/OUT)
Pointer to column flag.

One of the following values is returned:

- OCI_DIRPATH_COL_COMPLETE - all data for column is present

- OCI_DIRPATH_COL_NULL - column is NULL

- OCI_DIRPATH_COL_PARTIAL - partial column data is being supplied

Comments

If *cflgp* is set to OCI_DIRPATH_COL_NULL, the *cvalp* and *clenp* parameters are not set by this operation.

Related Functions

OCIDirPathColArrayEntrySet(), OCIDirPathColArrayRowGet(), OCIDirPathColArrayReset(), OCIDirPathColArrayToStream()

OCIDirPathColArrayEntrySet()

Purpose

Sets a specified entry in a column array to the supplied values.

Syntax

```
sword OCIDirPathColArrayEntrySet ( OCIDirPathColArray   *dpca,
                                   OCIError             *errhp,
                                   ub4                  rownum,
                                   ub2                  colIdx,
                                   ub1                  *cvalp,
                                   ub4                  clen,
                                   ub1                  cflg );
```

Parameters

dpca (IN/OUT)
Direct path column array handle.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

rownum (IN)
Zero-based row offset

colIdx (IN)
Column identifier (index), the column ID is returned by `OCIDirPathColAttrSet()`

cvalp (IN)
Pointer to column data

clen (IN)
Length of column data

cflg (IN)
Column flag. One of the following values is returned:

- OCI_DIRPATH_COL_COMPLETE - all data for column is present

- OCI_DIRPATH_COL_NULL - column is `NULL`

- OCI_DIRPATH_COL_PARTIAL - partial column data is being supplied

Comments

If `cflg` is set to OCI_DIRPATH_COL_NULL, the `cval` and `clen` parameters are not used.

Example

This example sets the source of data for the first row in a column array to `addr`, with a length of len. In this example, the column is identified by `colId`.

```
err = OCIDirPathColArrayEntrySet(dpca, errhp, (ub2)0, colId, addr, len,
        OCI_DIRPATH_COL_COMPLETE);
```

Related Functions

OCIDirPathColArrayRowGet(), OCIDirPathColArrayRowGet(),
OCIDirPathColArrayReset(), OCIDirPathColArrayToStream()

OCIDirPathColArrayRowGet()

Purpose

Gets the column array row pointers for a given row number

Syntax

```
sword OCIDirPathColArrayRowGet ( OCIDirPathColArray   *dpca,
                                 OCIError             *errhp,
                                 ub4                  rownum,
                                 ub1                  ***cvalppp,
                                 ub4                  **clenpp,
                                 ub1                  **cflgpp );
```

Parameters

dpca (IN/OUT)
Direct path column array handle.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

rownum (IN)
Zero-based row offset

cvalppp (IN/OUT)
Pointer to vector of pointers to column data

clenpp (IN/OUT)
Pointer to vector of column data lengths

cflgpp (IN/OUT)
Pointer to vector of column flags

Comments

Returns pointers to column array entries for the given row. This allows the application to do simple pointer arithmetic to iterate across the columns of the specific row. This interface can be used to efficiently get or set the column array entries of a row, as opposed to calling `OCIDirPathColArrayEntrySet()` for every column. The application is also responsible for not de-referencing memory beyond the column array boundaries. The dimensions of the column array are available as attributes of the column array.

Related Functions

OCIDirPathColArrayRowGet(), OCIDirPathColArrayEntrySet(),
OCIDirPathColArrayReset(), OCIDirPathColArrayToStream()

OCIDirPathColArrayReset()

Purpose

Resets the column array state.

Syntax

```
sword OCIDirPathColArrayReset ( OCIDirPathColArray   *dpca,
                                OCIError             *errhp );
```

Parameters

dpca (IN)
Direct path column array handle.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

Comments

Resetting the column array state is necessary when piecing in a large column and an error occurs in the middle of loading the column.

Related Functions

OCIDirPathColArrayEntryGet(), OCIDirPathColArrayEntrySet(),
OCIDirPathColArrayRowGet(), OCIDirPathColArrayToStream()

OCIDirPathColArrayToStream()

Purpose

Converts from column array format to a direct path stream format.

Syntax

```
sword OCIDirPathColArrayToStream ( OCIDirPathColArray    *dpca,
                                   OCIDirPathCtx  const  *dpctx,
                                   OCIDirPathStream      *dpstr,
                                   OCIError              *errhp,
                                   ub4                   rowcnt,
                                   ub4                   rowoff );
```

Parameters

dpca (IN)
Direct path column array handle.

dpctx (IN)
Direct path context handle for the object being loaded.

dpstr (IN/OUT)
Direct path stream handle.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

rowcnt (IN)
Number of rows in the column array.

rowoff (IN)
Starting index in the column array.

Comments

This interface is used to convert a column array representation of data in its external format as specified by OCIDirPathColAttrSet() to a direct path stream format. The converted format is suitable for loading with OCIDirPathLoadStream().

The column data in direct path stream format is converted to its Oracle internal representation. All conversions are done on the client side of the two-task interface, all conversion errors occur synchronously with the call to this interface. Information concerning which row and column that an error occurred on is available as an attribute of the column array handle.

Note that in a threaded environment concurrent `OCIDirPathColArrayToStream()` operations can be referencing the same direct path context handle. However, the direct path context handle is not modified by this interface.

The return codes for this call are:

- OCI_SUCCESS - All data in the column array was successfully converted to stream format. The column array attribute OCI_ATTR_ROW_COUNT is the number of rows processed.

- OCI_ERROR - An error occurred during conversion, the error handle contains the error information. The column array attribute OCI_ATTR_ROW_COUNT, is the number of rows processed. The attribute OCI_ATTR_COL_COUNT contains the column index into the column array for the column which caused the error.

- OCI_CONTINUE - Not all of the data in the column array could be converted to stream format. The stream buffer is not large enough to contain all of the column array data. The caller should either load the data, save the data to a file, or use another stream and call `OCIDirPathArrayToStream()` again to convert the remainder of the column array data. Note that the column array has internal state to know where to resume conversion from. The column array attribute OCI_ATTR_ROW_COUNT is the number of rows processed.

- OCI_NEED_DATA - All of the data in the column array was successfully converted, but a partial column was encountered. The caller should load the resulting stream, and supply the remainder of the row, iteratively if necessary. The column array attribute OCI_ATTR_ROW_COUNT, is the number of rows processed. The attribute OCI_ATTR_COL_COUNT contains the column index into the column array for the column which is marked partial.

Related Functions

`OCIDirPathColArrayEntryGet()`, `OCIDirPathColArrayEntrySet()`, `OCIDirPathColArrayRowGet()`, `OCIDirPathColArrayReset()`

OCIDirPathDataSave()

Purpose

Depending on the action requested, does a data savepoint, or commits the loaded data and finishes the direct path load operation.

Syntax

```
sword OCIDirPathDataSave ( OCIDirPathCtx        *dpctx,
                           OCIError             *errhp,
                           ub4                  action );
```

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

action (IN)
Values for action parameter to `OCIDirPathDataSave()`:

- OCI_DIRPATH_DATASAVE_SAVEONLY - to execute a data savepoint only.

- OCI_DIRPATH_DATASAVE_FINISH - to commit the loaded data and call the direct finishing function.

Comments

A return value of OCI_SUCCESS indicates that the back-end has properly executed a data savepoint or executed the finishing logic.

Executing a data savepoint is not allowed for LOBs.

Executing the finishing logic is not the same as properly terminating the load, because resources allocated are not freed.

Related Functions

OCIDirPathAbort(), OCIDirPathFinish(), OCIDirPathFlushRow(), OCIDirPathPrepare(), OCIDirPathStreamReset()

OCIDirPathFinish()

Purpose

Finishes the direct path load operation.

Syntax

```
sword OCIDirPathFinish (   OCIDirPathCtx          *dpctx,
                           OCIError               *errhp );
```

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

Comments

After the load has completed, and the loaded data is to be committed, the direct path finishing function is called.

A return value of OCI_SUCCESS indicates that the back-end has properly terminated the load.

Related Functions

OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathFlushRow(), OCIDirPathPrepare(), OCIDirPathStreamReset()

OCIDirPathFlushRow()

Purpose

Flushes a partially loaded row from server.

Syntax

```
sword OCIDirPathFlushRow (    OCIDirPathCtx          *dpctx,
                              OCIError               *errhp );
```

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

Comments

This function is necessary when part of a row is loaded, but a conversion error occurs on the next piece being processed by the application. Only the row currently in partial state is discarded. If the server is not currently processing a partial row for the object associated with the direct path context, this function is basically does nothing.

Related Functions

OCIDirPathAbort(), OCIDirPathFinish(), OCIDirPathPrepare(), OCIDirPathLoadStream()

OCIDirPathLoadStream()

Purpose

Loads the data converted to direct path stream format.

Syntax

```
sword OCIDirPathLoadStream (    OCIDirPathCtx          *dpctx,
                                OCIDirPathStream       *dpstr,
                                OCIError               *errhp );
```

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

dpstr (IN)
Direct path stream handle for the stream to load.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

Comments

When the interface returns an error, information concerning the row in the column array that sourced the stream can be obtained as an attribute of the direct path stream. Also, the offset into the stream where the error occurred can be obtained as an attribute of the stream.

Return codes for this function are:

- OCI_SUCCESS - All data in the stream was successfully loaded.

- OCI_ERROR - An error occurred while loading the data. The problem could be a partition mapping error, a null constraint violation, functional index evaluation error, or an out of space condition, such as cannot allocate extent. The attribute OCI_ATTR_STREAM_OFFSET of the direct path stream is the offset into the stream which corresponds to the offending row. OCI_ATTR_ROW_COUNT is the number of rows processed.

- OCI_NEED_DATA - Last row was not complete. The caller needs to supply another row piece. If the stream was sourced from a column array, the attribute OCI_ATTR_ROW_COUNT is the number of complete rows processed.

- OCI_NO_DATA - Attempt to load an empty stream, or a stream which has been completely processed.

Related Functions

`OCIDirPathAbort()`, `OCIDirPathDataSave()`, `OCIDirPathFinish()`, `OCIDirPathPrepare()`, `OCIDirPathStreamReset()`

OCIDirPathPrepare()

Purpose

Prepares the direct path load interface before any rows can be converted or loaded.

Syntax

```
sword OCIDirPathPrepare (   OCIDirPathCtx           *dpctx,
                            OCISvcCtx               *svchp,
                            OCIError                *errhp );
```

Parameters

dpctx (IN)
Direct path context handle for the object loaded.

svchp (IN)
Service context.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

Comments

After the name of the object to be operated on is set, the external attributes of the column data is set, and all load options are set, the direct path interface must be prepared with `OCIDirPathPrepare()` before any rows can be converted or loaded.

A return value of OCI_SUCCESS indicates that the back-end has been properly initialized for a direct path load operation. A non-zero return indicates an error. Possible errors are:

- invalid context

- not connected to a server

- object name not set

- already prepared (cannot prepare twice)

- object not suitable for a direct path operation

Related Functions

`OCIDirPathAbort()`, `OCIDirPathDataSave()`, `OCIDirPathFinish()`, `OCIDirPathFlushRow()`, `OCIDirPathStreamReset()`

OCIDirPathStreamReset()

Purpose

Resets the direct path stream state.

Syntax

```
sword OCIDirPathStreamReset ( OCIDirPathStream        *dpstr,
                              OCIError                *errhp );
```

Parameters

dpstr (IN)
Direct path stream handle.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

Comments

A direct path stream maintains the state that indicates where the next OCIDirPathColArrayToStream() call should start writing into the stream. Normally, data is appended to the end of the stream. When the caller wants to start a new stream after a stream is successfully loaded, or discard the data in a stream, the stream must be reset with this call.

Related Functions

OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathFinish(), OCIDirPathFlushRow(), OCIDirPathPrepare()

Thread Management Functions

This section describes the thread management functions.

Table 16–4   Thread Management functions

Function	Purpose
OCIThreadClose() on page 16-116	Closes a thread handle
OCIThreadCreate() on page 16-117	Creates a new thread
OCIThreadHandleGet() on page 16-119	Retrieves the OCIThreadHandle of the thread in which it is called
OCIThreadHndDestroy() on page 16-120	Destroys and deallocates the thread handle
OCIThreadHndInit() on page 16-121	Allocates and initializes the thread handle
OCIThreadIdDestroy() on page 16-122	Destroys and deallocates a thread id
OCIThreadIdGet() on page 16-123	Retrieves the **OCIThreadId** of the thread in which it is called
OCIThreadIdInit() on page 16-124	Allocate and initialize the thread id
OCIThreadIdNull() on page 16-125	Determines whether or not a given **OCIThreadId** is the NULL thread ID
OCIThreadIdSame() on page 16-126	Determines whether or not two **OCIThreadId**s represent the same thread
OCIThreadIdSet() on page 16-127	Sets one **OCIThreadId** to another
OCIThreadIdSetNull() on page 16-128	Sets the NULL thread ID to a given **OCIThreadId**
OCIThreadInit() on page 16-129	Initializes OCIThread context
OCIThreadIsMulti() on page 16-130	Tells the caller whether the application is running in a multi-threaded environment or a single-threaded environment
OCIThreadJoin() on page 16-131	Allows the calling thread to join with another thread
OCIThreadKeyDestroy() on page 16-132	Destroy and deallocate the key pointed to by *key*
OCIThreadKeyGet() on page 16-133	Gets the calling threads current value for a key
OCIThreadKeyInit() on page 16-134	Creates a key
OCIThreadKeySet() on page 16-135	Sets the calling threads value for a key
OCIThreadMutexAcquire() on page 16-136	Acquires a mutex for the thread in which it is called
OCIThreadMutexDestroy() on page 16-137	Destroys and deallocate a mutex
OCIThreadMutexInit() on page 16-138	Allocates and initializes a mutex

Table 16–4   Thread Management functions

Function	Purpose
`OCIThreadMutexRelease()` on page 16-139	Releases a mutex
`OCIThreadProcessInit()` on page 16-140	Performs OCIThread process initialization
`OCIThreadTerm()` on page 16-141	Releases the OCIThread context

OCIThreadClose()

Purpose

Closes a thread handle.

Syntax

```
sword OCIThreadClose ( dvoid            *hndl,
                       OCIError         *err,
                       OCIThreadHandle  *tHnd );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
`OCIErrorGet()`.

tHnd (IN/OUT)
The OCIThread thread handle to close.

Comments

tHnd should be initialized by `OCIThreadHndInit()`. Both thread handle and the
thread ID that was returned by the same call to `OCIThreadCreate()` are invalid
after the call to `OCIThreadClose()`.

Related Functions

OCIThreadCreate()

OCIThreadCreate()

Purpose

Creates a new thread.

Syntax

```
sword OCIThreadCreate ( dvoid            *hndl,
                        OCIError         *err,
                        void (*start)    (dvoid
                        dvoid            *arg,
                        OCIThreadId      *tid,
                        OCIThreadHandle  *tHnd );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Diagnostic information can be obtained by calling `OCIErrorGet()`.

start (IN)
The function in which the new thread should begin execution.

arg (IN)
The argument to give the function pointed to by `start`.

tid (IN/OUT)
If not NULL, gets the ID for the new thread.

tHnd (IN/OUT)
If not NULL, gets the handle for the new thread.

Comments

The new thread starts by executing a call to the function pointed to by `start` with the argument given by `arg`. When that function returns, the new thread will terminate. The function should not return a value and should accept one parameter, a **dvoid**. The call to OCIThreadCreate() must be matched by a call to OCIThreadClose() if and only if `tHnd` is non-NULL.

If `tHnd` is NULL, a thread ID placed in `*tid` will not be valid in the calling thread because the timing of the spawned threads termination is unknown.

tid should be initialized by OCIThreadIdInit() and *tHnd* should be initialized by OCIThreadHndInit().

Related Functions

OCIThreadClose(), OCIThreadIdInit(), OCIThreadHndInit()

OCIThreadHandleGet()

Purpose

Retrieves the **OCIThreadHandle** of the thread in which it is called.

Syntax

```
sword OCIThreadHandleGet ( dvoid           *hndl,
                           OCIError        *err,
                           OCIThreadHandle *tHnd );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling OCIErrorGet().

tHnd (IN/OUT)
If not NULL, the location to place the thread handle for the thread.

Comments

tHnd should be initialized by OCIThreadHndInit().

The thread handle *tHnd* retrieved by this function must be closed with OCIThreadClose() and destroyed by OCIThreadHndDestroy() after it is used.

Related Functions

OCIThreadHndDestroy(), OCIThreadHndInit()

OCIThreadHndDestroy()

Purpose

Destroys and deallocates the thread handle.

Syntax

```
sword OCIThreadHndDestroy ( dvoid           *hndl,
                            OCIError        *err,
                            OCIThreadHandle **thnd );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
`OCIErrorGet()`.

thnd (IN/OUT)
The address of pointer to the thread handle to destroy.

Comments

thnd should be initialized by `OCIThreadHndInit()`.

Related Functions

OCIThreadHandleGet(), OCIThreadHndInit()

OCIThreadHndInit()

Purpose

Allocates and initializes the thread handle.

Syntax

```
sword OCIThreadHndInit ( dvoid          *hndl,
                         OCIError       *err,
                         OCIThreadHandle **thnd );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling OCIErrorGet().

thnd (OUT)
The address of pointer to the thread handle to initialize.

Related Functions

OCIThreadHandleGet(), OCIThreadHndDestroy()

OCIThreadIdDestroy()

Purpose

Destroys and deallocates a thread Id.

Syntax

```
sword OCIThreadIdDestroy (dvoid       *hndl,
                          OCIError     *err,
                          OCIThreadId  **tid );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling `OCIErrorGet()`.

tid (IN/OUT)
Pointer to the thread ID to destroy.

Comments

tid should be initialized by `OCIThreadIdInit()`.

Related Functions

OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(),
OCIThreadIdSame(), OCIThreadIdSet(), OCIThreadIdSetNull()

OCIThreadIdGet()

Purpose

Retrieves the **OCIThreadId** of the thread in which it is called.

Syntax

```
sword OCIThreadIdGet ( dvoid      *hndl,
                       OCIError   *err,
                       OCIThreadId *tid );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
OCIErrorGet().

tid (OUT)
This should point to the location in which to place the ID of the calling thread.

Comments

tid should be initialized by OCIThreadIdInit(). When OCIThread is used in a
single-threaded environment, OCIThreadIdGet() will always place the same
value in the location pointed to by *tid*. The exact value itself is not important. The
important thing is that it is not the same as the NULL thread ID and that it is always
the same value.

Related Functions

OCIThreadIdDestroy(), OCIThreadIdInit(), OCIThreadIdNull(),
OCIThreadIdSame(), OCIThreadIdSet(), OCIThreadIdSetNull()

OCIThreadIdInit()

Purpose

Allocate and initialize the thread Id *tid*.

Syntax

```
sword OCIThreadIdInit ( dvoid      *hndl,
                        OCIError    *err,
                        OCIThreadId **tid );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling OCIErrorGet().

tid (OUT)
Pointer to the thread ID to initialize.

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSet(), OCIThreadIdSetNull()

OCIThreadIdNull()

Purpose

Determines whether or not a given **OCIThreadId** is the NULL thread Id.

Syntax

```
sword OCIThreadIdNull ( dvoid      *hndl,
                        OCIError   *err,
                        OCIThreadId *tid,
                        boolean    *result );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
OCIErrorGet().

tid (IN)
Pointer to the **OCIThreadId** to check.

result (IN/OUT)
Pointer to the result.

Comments

If *tid* is the NULL thread ID, *result* is set to TRUE. Otherwise, *result* is set to
FALSE. *tid* should be initialized by OCIThreadIdInit().

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(),
OCIThreadIdSame(), OCIThreadIdSet(), OCIThreadIdSetNull()

OCIThreadIdSame()

Purpose

Determines whether or not two **OCIThreadId**s represent the same thread.

Syntax

```
sword OCIThreadIdSame ( dvoid        *hndl,
                        OCIError      *err,
                        OCIThreadId   *tid1,
                        OCIThreadId   *tid2,
                        boolean       *result );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
`OCIErrorGet()`.

tid1 (IN)
Pointer to the first **OCIThreadId**.

tid2 (IN)
Pointer to the second **OCIThreadId**.

result (IN/OUT)
Pointer to the result.

Comments

If `tid1` and `tid2` represent the same thread, `result` is set to TRUE. Otherwise,
`result` is set to FALSE. `result` is set to TRUE if both `tid1` and `tid2` are the NULL
thread ID. `ti1d` and `tid2` should be initialized by OCIThreadIdInit().

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(),
OCIThreadIdNull(), OCIThreadIdSet(), OCIThreadIdSetNull()

OCIThreadIdSet()

Purpose

Sets one **OCIThreadId** to another.

Syntax

```
sword OCIThreadIdSet ( dvoid       *hndl,
                       OCIError    *err,
                       OCIThreadId *tidDest,
                       OCIThreadId *tidSrc );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling OCIErrorGet().

tidDest (OUT)
This should point to the location of the **OCIThreadId** to be set to.

tidSrc (IN)
This should point to the **OCIThreadId** to set from.

Comments

tid should be initialized by OCIThreadIdInit().

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSetNull()

OCIThreadIdSetNull()

Purpose

Sets the NULL thread ID to a given **OCIThreadId**.

Syntax

```
sword OCIThreadIdSetNull ( dvoid       *hndl,
                           OCIError     *err,
                           OCIThreadId  *tid );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
OCIErrorGet().

tid (OUT)
This should point to the **OCIThreadId** in which to put the NULL thread Id.

Comments

tid should be initialized by OCIThreadIdInit().

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(),
OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSet()

OCIThreadInit()

Purpose

Initializes the OCIThread context.

Syntax

```
sword OCIThreadInit ( dvoid    *hndl,
                      OCIError *err );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling OCIErrorGet().

Comments

It is illegal for OCIThread clients to try an examine the memory pointed to by the returned pointer. It is safe to make concurrent calls to OCIThreadInit(). Unlike OCIThreadProcessInit(), there is no need to have a first call that occurs before all the others.

The first time OCIThreadInit() is called, it initializes the OCI Thread context. It also saves a pointer to the context in some system dependent manner. Subsequent calls to OCIThreadInit() will return the same context.

Each call to OCIThreadInit() must eventually be matched by a call to OCIThreadTerm().

Related Functions

OCIThreadTerm()

OCIThreadIsMulti()

Purpose

Tells the caller whether the application is running in a multi-threaded environment or a single-threaded environment.

Syntax

```
boolean OCIThreadIsMulti ( );
```

Returns

TRUE if the environment is multi-threaded;

FALSE if the environment is single-threaded.

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSet()

OCIThreadJoin()

Purpose

Allows the calling thread to join with another thread.

Syntax

```
sword OCIThreadJoin ( dvoid           *hndl,
                      OCIError        *err,
                      OCIThreadHandle *tHnd );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Diagnostic information can be obtained by calling `OCIErrorGet()`.

tHnd (IN)
The **OCIThreadHandle** of the thread to join with.

Comments

This function blocks the caller until the specified thread terminates.

`tHnd` should be initialized by `OCIThreadHndInit()`. The result of multiple threads all trying to join with the same thread is undefined.

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSet()

OCIThreadKeyDestroy()

Purpose

Destroy and deallocate the key pointed to by *key*.

Syntax

```
sword OCIThreadKeyDestroy ( dvoid        *hndl,
                            OCIError      *err,
                            OCIThreadKey  **key );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling `OCIErrorGet()`.

key (IN/OUT)
The **OCIThreadKey** in which to destroy the key.

Comments

This is different from the destructor function callback passed to the key create routine. This new destroy function `OCIThreadKeyDestroy()` is used to terminate any resources OCI THREAD acquired when it created *key*. The *OCIThreadKeyDestFunc* callback of `OCIThreadKeyInit()` is a key VALUE destructor; it does in no way operate on the key itself.

This must be called once the user has finished using the key. Not calling the key destroy function may result in memory leaks.

Related Functions

OCIThreadKeyGet(), OCIThreadKeyInit(), OCIThreadKeySet()

OCIThreadKeyGet()

Purpose

Gets the calling threads current value for a key.

Syntax

```
sword OCIThreadKeyGet ( dvoid        *hndl,
                        OCIError     *err,
                        OCIThreadKey *key,
                        dvoid        **pValue );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling OCIErrorGet().

key (IN)
The key.

pValue (IN/OUT)
The location in which to place the thread-specific key value.

Comments

It is illegal to use this function on a key that has not been created using OCIThreadKeyInit().

If the calling thread has not yet assigned a value to the key, NULL is placed in the location pointed to by *pValue*.

Related Functions

OCIThreadKeyDestroy(), OCIThreadKeyInit(), OCIThreadKeySet()

OCIThreadKeyInit()

Purpose

Creates a key.

Syntax

```
sword OCIThreadKeyInit (dvoid              *hndl,
                        OCIError           *err,
                        OCIThreadKey       **key,
                        OCIThreadKeyDestFunc destFn );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling OCIErrorGet().

key (OUT)
The **OCIThreadKey** in which to create the new key.

destFn (IN)
The destructor for the key. NULL is permitted.

Comments

Each call to this routine allocate and generates a new key that is distinct from all other keys. After this function executes successfully, a pointer to an allocated and initialized key is return. That key can be used with OCIThreadKeyGet() and OCIThreadKeySet(). The initial value of the key will be NULL for all threads.

It is illegal for this function to be called more than once with the same value for the *key* parameter.

If the *destFn* parameter is not NULL, the routine pointed to by *destFn* will be called whenever a thread that has a non-NULL value for the key terminates. The routine will be called with one parameter. The parameter will be the keys value for the thread at the time at which the thread terminated. If the key does not need a destructor function, pass NULL for *destFn*.

Related Functions

OCIThreadKeyDestroy(), OCIThreadKeyGet(), OCIThreadKeySet()

OCIThreadKeySet()

Purpose

Sets the calling threads value for a key.

Syntax

```
sword OCIThreadKeySet ( dvoid         *hndl,
                        OCIError      *err,
                        OCIThreadKey  *key,
                        dvoid         *value );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in *err* and diagnostic information can be obtained by calling
OCIErrorGet().

key (IN/OUT)
The key.

value (IN)
The thread-specific value to set in the key.

Comments

It is illegal to use this function on a key that has not been created using
OCIThreadKeyInit().

Related Functions

OCIThreadKeyDestroy(), OCIThreadKeyGet(), OCIThreadKeyInit()

OCIThreadMutexAcquire()

Purpose

Acquires a mutex for the thread in which it is called.

Syntax

```
sword OCIThreadMutexAcquire ( dvoid          *hndl,
                              OCIError       *err,
                              OCIThreadMutex *mutex );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
`OCIErrorGet()`.

mutex (IN/OUT)
The mutex to acquire.

Comments

If the mutex is held by another thread, the calling thread is blocked until it can
acquire the mutex.

It is illegal to attempt to acquire an uninitialized mutex.

This functions behavior is undefined if it is used by a thread to acquire a mutex that
is already held by that thread.

Related Functions

`OCIThreadMutexDestroy()`, `OCIThreadMutexInit()`,
`OCIThreadMutexRelease()`

OCIThreadMutexDestroy()

Purpose

Destroys and deallocate a mutex.

Syntax

```
sword OCIThreadMutexDestroy ( dvoid          *hndl,
                              OCIError       *err,
                              OCIThreadMutex **mutex );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in err and diagnostic information can be obtained by calling OCIErrorGet().

mutex (IN/OUT)
The mutex to destroy.

Comments

Each mutex must be destroyed once it is no longer needed.

It is not legal to destroy a mutex that is uninitialized or is currently held by a thread. The destruction of a mutex must not occur concurrently with any other operations on the mutex. A mutex must not be used after it has been destroyed.

Related Functions

OCIThreadMutexAcquire(), OCIThreadMutexInit(), OCIThreadMutexRelease()

OCIThreadMutexInit()

Purpose

Allocates and initializes a mutex.

Syntax

```
sword OCIThreadMutexInit ( dvoid          *hndl,
                           OCIError       *err,
                           OCIThreadMutex **mutex );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in err and diagnostic information can be obtained by calling
OCIErrorGet().

mutex (OUT)
The mutex to initialize.

Comments

All mutexes must be initialized prior to use.

Multiple threads must not initialize the same mutex simultaneously. Also, a mutex must not be reinitialized until it has been destroyed (see
OCIThreadMutexDestroy()).

Related Functions

OCIThreadMutexDestroy(), OCIThreadMutexAcquire(),
OCIThreadMutexRelease()

OCIThreadMutexRelease()

Purpose

Releases a mutex.

Syntax

```
sword OCIThreadMutexRelease ( dvoid         *hndl,
                              OCIError      *err,
                              OCIThreadMutex *mutex );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in err and diagnostic information can be obtained by calling OCIErrorGet().

mutex (IN/OUT)
The mutex to release.

Comments

If there are any threads blocked on the mutex, one of them will acquire it and become unblocked.

It is illegal to attempt to release an uninitialized mutex. It is also illegal for a thread to release a mutex that it does not hold.

Related Functions

OCIThreadMutexDestroy(), OCIThreadMutexInit(), OCIThreadMutexAcquire()

OCIThreadProcessInit()

Purpose

Performs OCIThread process initialization.

Syntax

```
void OCIThreadProcessInit ( );
```

Comments

Whether or not this function needs to be called depends on how OCI Thread is going to be used.

In a single-threaded application, calling this function is optional. If it is called at all, the first call to it must occur before calls to any other OCIThread functions. Subsequent calls can be made without restriction; they will not have any effect.

In a multi-threaded application, this function must be called. The first call to it must occur strictly before any other OCIThread calls; i.e., no other calls to OCIThread functions (including other calls to this one) can be concurrent with the first call.

Subsequent calls to this function can be made without restriction; they will not have any effect.

Related Functions

OCIThreadIdDestroy(), OCIThreadIdGet(), OCIThreadIdInit(), OCIThreadIdNull(), OCIThreadIdSame(), OCIThreadIdSet()

OCIThreadTerm()

Purpose

Releases the OCIThread context.

Syntax

```
sword OCIThreadTerm ( dvoid    *hndl,
                      OCIError *err );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error and OCI_ERROR is returned, the error is recorded in err and diagnostic information can be obtained by calling
`OCIErrorGet()`.

Comments

This function should be called exactly once for each call made to
`OCIThreadInit()`.

It is safe to make concurrent calls to `OCIThreadTerm()`. `OCIThreadTerm()` will not do anything until it has been called as many times as `OCIThreadInit()` has been called. When that happens, it terminates the OCIThread layer and frees the memory allocated for the context. Once this happens, the context should not be re-used. It will be necessary to obtain a new one by calling `OCIThreadInit()`.

Related Functions

OCIThreadInit()

Transaction Functions

This section describes the transaction functions.

Table 16–5   Transaction Functions

Function	Purpose
OCITransCommit() on page 16-143	Commit a transaction on a service context
OCITransDetach() on page 16-146	Detach a transaction from a service context
OCITransForget() on page 16-147	Forget a prepared global transaction
OCITransMultiPrepare() on page 16-148	Prepare a transaction with multiple branches in a single cell
OCITransPrepare() on page 16-149	Prepare a global transaction for commit
OCITransRollback() on page 16-150	Roll back a transaction
OCITransStart() on page 16-151	Start a transaction on a service context

OCITransCommit()

Purpose

Commits the transaction associated with a specified service context.

Syntax

```
sword OCITransCommit ( OCISvcCtx    *svchp,
                       OCIError     *errhp,
                       ub4          flags );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

flags (IN)
A flag used for one-phase commit optimization in global transactions.

If the transaction is non-distributed, the flags parameter is ignored, and OCI_DEFAULT can be passed as its value. OCI applications managing global transactions should pass a value of OCI_TRANS_TWOPHASE to the flags parameter for a two-phase commit. The default is one-phase commit.

Comments

The transaction currently associated with the service context is committed. If it is a global transaction that the server cannot commit, this call additionally retrieves the state of the transaction from the database to be returned to the user in the error handle.

If the application has defined multiple transactions, this function operates on the transaction currently associated with the service context. If the application is working with only the implicit local transaction created when database changes are made, that implicit transaction is committed.

If the application is running in the object mode, then the modified or updated objects in the object cache for this transaction are also flushed and committed.

Under normal circumstances, OCITransCommit() returns with a status indicating that the transaction has either been committed or rolled back. With global transactions, it is possible that the transaction is now in-doubt, meaning that it is

neither committed nor aborted. In this case, `OCITransCommit()` attempts to retrieve the status of the transaction from the server. The status is returned.

Example

The following example demonstrates the use of a simple local transaction, as described in the section "Simple Local Transactions" on page 8-3.

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCIStmt *stmthp;
  dvoid     *tmp;
  text sqlstmt[128];

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
              (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
              0, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, (size_t)0, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                  (size_t)0, (dvoid **) &tmp);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                  (size_t)0, (dvoid **) &tmp);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                  (size_t)0, (dvoid **) &tmp);

  OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0);

  OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
                    OCI_ATTR_SERVER, errhp);

  OCILogon(envhp, errhp, &svchp, "SCOTT", strlen("SCOTT"),
                  "TIGER", strlen("TIGER"), 0, 0);

  /* update scott.emp empno=7902, increment salary */
  sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
```

```
OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* update scott.emp empno=7902, increment salary again, but rollback */
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
OCITransRollback(svchp, errhp, (ub4) 0);
}
```

Related Functions

OCITransRollback()

OCITransDetach()

Purpose

Detaches a transaction.

Syntax

```
sword OCITransDetach ( OCISvcCtx    *svchp,
                       OCIError     *errhp,
                       ub4          flags );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

flags (IN)
You must pass a value of OCI_DEFAULT for this parameter.

Comments

Detaches a global transaction from the service context handle. The transaction currently attached to the service context handle becomes inactive at the end of this call. The transaction may be resumed later by calling `OCITransStart()`, specifying a flags value of OCI_TRANS_RESUME.

When a transaction is detached, the value which was specified in the timeout parameter of `OCITransStart()` when the transaction was started is used to determine the amount of time the branch can remain inactive before being deleted by the server's PMON process.

> **Note:** The transaction can be resumed by a different process than the one that detached it, provided that the transaction has the same authorization. If this function is called before a transaction is actually started, this function is a no-op.

For example code demonstrating the use of `OCITransDetach()` see the description of `OCITransStart()`.

Related Functions

`OCITransStart()`

OCITransForget()

Purpose

Causes the server to forget a heuristically completed global transaction.

Syntax

```
sword OCITransForget ( OCISvcCtx     *svchp,
                       OCIError      *errhp,
                       ub4           flags );
```

Parameters

svchp (IN)
The service context handle in which the transaction resides.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

flags (IN)
You must pass OCI_DEFAULT for this parameter.

Comments

Forgets a heuristically completed global transaction. The server deletes the status of the transaction from the system's pending transaction table.

You set the XID of the transaction to be forgotten as an attribute of the transaction handle (OCI_ATTR_XID).

Related Functions

OCITransCommit(), OCITransRollback()

OCITransMultiPrepare()

Purpose

Prepares a transaction with multiple branches in a single call.

Syntax

```
sword OCITransMultiPrepare ( OCISvcCtx    *svchp,
                             ub4         numBranches,
                             OCITrans    **txns,
                             OCIError    **errhp);
```

Parameters

srvchp (IN)
The service context handle.

numBranches (IN)
The number of branches expected. It is also the array size for the next two parameters.

txns (IN)
The array of transaction handles for the branches to prepare. They should all have the OCI_ATTR_XID set. The global transaction ID should be the same.

errhp (IN)
The array of error handles. If OCI_SUCCESS is not returned, then these will indicate which branches received which errors.

Comments

Prepares the specified global transaction for commit. This call is valid only for distributed transactions. This call is an advanced performance feature intended for use only in situations where the caller is responsible for preparing all the branches in a transaction.

Related Functions

OCITransPrepare()

OCITransPrepare()

Purpose

Prepares a transaction for commit.

Syntax

```
sword OCITransPrepare ( OCISvcCtx    *svchp,
                        OCIError     *errhp,
                        ub4          flags );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

flags (IN)
You must pass OCI_DEFAULT for this parameter.

Comments

Prepares the specified global transaction for commit.

This call is valid only for global transactions.

The call returns OCI_SUCCESS_WITH_INFO if the transaction has not made any changes. The error handle will indicate that the transaction is read-only. The flag parameter is not currently used.

Related Functions

`OCITransCommit()`, `OCITransForget()`

OCITransRollback()

Purpose

Rolls back the current transaction.

Syntax

```
sword OCITransRollback ( dvoid      *svchp,
                         OCIError   *errhp,
                         ub4        flags );
```

Parameters

svchp (IN)
A service context handle. The transaction currently set in the service context handle is rolled back.

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

flags (IN)
You must pass a value of OCI_DEFAULT for this parameter.

Comments

The current transaction— defined as the set of statements executed since the last `OCITransCommit()` or since `OCISessionBegin()`—is rolled back.

If the application is running under object mode then the modified or updated objects in the object cache for this transaction are also rolled back.

Attempting to roll back a global transaction that is not currently active causes an error.

Examples

For example code demonstrating the use of `OCITransRollback()` see the description of `OCITransCommit()`.

Related Functions

`OCITransCommit()`

OCITransStart()

Purpose

Sets the beginning of a transaction.

Syntax

```
sword OCITransStart ( OCISvcCtx    *svchp,
                      OCIError     *errhp,
                      uword        timeout,
                      ub4          flags );
```

Parameters

svchp (IN/OUT)
The service context handle. The transaction context in the service context handle is initialized at the end of the call if the flag specified a new transaction to be started.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling OCIErrorGet().

timeout (IN)
The time, in seconds, to wait for a transaction to become available for resumption when OCI_TRANS_RESUME is specified. When OCI_TRANS_NEW is specified, the timeout parameter indicates the number of seconds the transaction can be inactive before it is automatically aborted by the system. A transaction is inactive between the time it is detached (with OCITransDetach()) and the time it is resumed with OCITransStart().

flags (IN)
Specifies whether a new transaction is being started or an existing transaction is being resumed. Also specifies serializiability or read-only status. More than a single value can be specified. By default, a read/write transaction is started. The flag values are:

- OCI_TRANS_NEW - starts a new transaction branch. By default starts a tightly coupled and migratable branch.

- OCI_TRANS_TIGHT - explicitly specifies a tightly coupled branch

- OCI_TRANS_LOOSE - specifies a loosely coupled branch

- OCI_TRANS_RESUME - resumes an existing transaction branch.

- OCI_TRANS_READONLY - start a read-only transaction

- OCI_TRANS_SERIALIZABLE - start a serializable transaction

- OCI_TRANS_SEPARABLE - the transaction will be separated after each call.

  This flag results in a warning that the transaction was started using *regular* transactions. Separated transactions are not supported through release 9.0.1 of the server.

  An error message results if there is an error in your code or the transaction service. The error indicates that you attempted an action on a transaction that has already been prepared.

Comments

This function sets the beginning of a global or serializable transaction. The transaction context currently associated with the service context handle is initialized at the end of the call if the flags parameter specifies that a new transaction should be started.

The XID of the transaction is set as an attribute of the transaction handle (OCI_ATTR_XID)

Examples

The following examples demonstrate the use of OCI transactional calls for manipulating global transactions.

Example 1 - A Single Session Operating On Different Branches.
This concept is illustrated by Figure 8–2, "Session Operating on Multiple Branches" on page 8-6.

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCISession *usrhp;
  OCIStmt *stmthp1, *stmthp2;
  OCITrans *txnhp1, *txnhp2;
  dvoid     *tmp;
  XID gxid;
  text sqlstmt[128];

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
              (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
```

```
             0, (dvoid **) &tmp);

OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
             52, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
             52, (dvoid **) &tmp);

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
             52, (dvoid **) &tmp);

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp1, OCI_HTYPE_STMT, 0, 0);
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp2, OCI_HTYPE_STMT, 0, 0);

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
              OCI_ATTR_SERVER, errhp);

/* set the external name and internal name in server handle */
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "demo", 0,
              OCI_ATTR_EXTERNAL_NAME, errhp);
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "txn demo", 0,
                   OCI_ATTR_INTERNAL_NAME, errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
             (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"scott",
           (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"tiger",
           (ub4)strlen("tiger"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
             (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp1,  OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
                   OCI_ATTR_TRANS, errhp);
```

More OCI Relational Functions   **16-153**

```
/* start a transaction with global transaction id = [1000, 123, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;

OCIAttrSet((dvoid *)txnhp1, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
                          OCI_ATTR_XID, errhp);

/* start global transaction 1 with 60 second time to live when detached */
OCITransStart(svchp, errhp, 60, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp1, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp1, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);

/* allocate transaction handle 2 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp2,  OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
                          OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 124, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 124 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 4;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;

OCIAttrSet((dvoid *)txnhp2, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
                          OCI_ATTR_XID, errhp);

/* start global transaction 2 with 90 second time to live when detached */
OCITransStart(svchp, errhp, 90, OCI_TRANS_NEW);

/* update scott.emp empno=7934, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7934");
OCIStmtPrepare(stmthp2, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp2, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
```

```
                    OCITransDetach(svchp, errhp, 0);

                    /* Resume transaction 1, increment salary and commit it */
                    /* Set transaction handle 1 into the service handle */
                    OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
                                        OCI_ATTR_TRANS, errhp);

                    /* attach to transaction 1, wait for 10 seconds if the transaction is busy */
                    /* The wait is clearly not required in this example because no other      */
                    /* process/thread is using the transaction. It is only for illustration    */
                    OCITransStart(svchp, errhp, 10, OCI_TRANS_RESUME);
                    OCIStmtExecute(svchp, stmthp1, errhp, 1, 0, 0, 0, 0);
                    OCITransCommit(svchp, errhp, (ub4) 0);

                    /* attach to transaction 2 and commit it */
                    /* set transaction handle2 into the service handle */
                    OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
                                        OCI_ATTR_TRANS, errhp);
                    OCITransCommit(svchp, errhp, (ub4) 0);
                  }
```

Example 2 - A Single Session Operating On Multiple Branches That Share The Same Transaction.

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCISession *usrhp;
  OCIStmt *stmthp;
  OCITrans *txnhp1, *txnhp2;
  dvoid     *tmp;
  XID gxid;
  text sqlstmt[128];

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
            (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
            0, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
```

```
                   52, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                   52, (dvoid **) &tmp);

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                   52, (dvoid **) &tmp);

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0);

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
                     OCI_ATTR_SERVER, errhp);

/* set the external name and internal name in server handle */
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "demo", 0,
                     OCI_ATTR_EXTERNAL_NAME, errhp);
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "txn demo2", 0,
                         OCI_ATTR_INTERNAL_NAME, errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                   (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"scott",
              (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"tiger",
              (ub4)strlen("tiger"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
                   (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp1,  OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
                         OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;
```

```
OCIAttrSet((dvoid *)txnhp1, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
                      OCI_ATTR_XID, errhp);

/* start global transaction 1 with 60 second time to live when detached */
OCITransStart(svchp, errhp, 60, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);

/* allocate transaction handle 2 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp2,  OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
                      OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 2] */
/* The global transaction will be tightly coupled with earlier transaction */
/* There is not much practical value in doing this but the example */
/* illustrates the use of tightly-coupled transaction branches */
/* In a practical case the second transaction that tightly couples with */
/* the first can be executed from a different process/thread */

gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 2 */
gxid.data[3] = 2;

OCIAttrSet((dvoid *)txnhp2, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
                      OCI_ATTR_XID, errhp);

/* start global transaction 2 with 90 second time to live when detached */
OCITransStart(svchp, errhp, 90, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
/* This is possible even if the earlier transaction has locked this row */
/* because the two global transactions are tightly coupled */
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
```

```
OCITransDetach(svchp, errhp, 0);

/* Resume transaction 1 and prepare it. This will return */
/* OCI_SUCCESS_WITH_INFO because all branches except the last branch */
/* are treated as read-only transactions for tightly-coupled transactions */

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
                          OCI_ATTR_TRANS, errhp);
if (OCITransPrepare(svchp, errhp, (ub4) 0) == OCI_SUCCESS_WITH_INFO)
{
  text errbuf[512];
  ub4 buflen;
  sb4 errcode;

  OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
  errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
  printf("OCITransPrepare - %s\n", errbuf);
}

/* attach to transaction 2 and commit it */
/* set transaction handle2 into the service handle */
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
                          OCI_ATTR_TRANS, errhp);
OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Related Functions

OCITransDetach()

Miscellaneous Functions

This section describes the miscellaneous OCI functions.

Table 16–6   Miscellaneous Functions

Function	Purpose
OCIBreak() on page 16-160	Perform an immediate asynchronous break
OCIErrorGet() on page 16-161	Return error message and Oracle error
OCILdaToSvcCtx() on page 16-163	Toggle Lda_Def to service context handle
OCIPasswordChange() on page 16-165	Change password
OCIReset() on page 16-167	Called after OCIBreak() to reset asynchronous operation and protocol
OCIRowidToChar() on page 16-168	Converts a Universal ROWID to character extended (base 64) representation.
OCIServerVersion() on page 16-169	Get the Oracle version string
OCISvcCtxToLda() on page 16-170	Toggle service context handle to Lda_Def
OCIUserCallbackGet() on page 16-171	Identifies the callback that is registered for handle
OCIUserCallbackRegister() on page 16-173	Registers a user-created callback function

OCIBreak()

Purpose

This call performs an immediate (asynchronous) abort of any currently executing OCI function that is associated with a server.

Syntax

```
sword OCIBreak ( dvoid      *hndlp,
                 OCIError   *errhp );
```

Parameters

hndlp (IN/OUT)
The service context handle or the server context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

Comments

This call performs an immediate (asynchronous) abort of any currently executing OCI function that is associated with a server. It is normally used to stop a long-running OCI call being processed on the server.

This call can take either the service context handle or the server context handle as a parameter to identify the function to be aborted.

Related Functions

OCIReset()

OCIErrorGet()

Purpose

Returns an error message in the buffer provided and an Oracle error.

Syntax

```
sword OCIErrorGet ( dvoid      *hndlp,
                    ub4        recordno,
                    text       *sqlstate,
                    sb4        *errcodep,
                    text       *bufp,
                    ub4        bufsiz,
                    ub4        type );
```

Parameters

hndlp (IN)
The error handle, in most cases, or the environment handle (for errors on
OCIEnvCreate(), OCIHandleAlloc()).

recordno (IN)
Indicates the status record from which the application seeks info. Starts from 1.

sqlstate (OUT)
Not supported in release 8.x or later.

errcodep (OUT)
The error code returned.

bufp (OUT)
The error message text returned.

bufsiz (IN)
The size of the buffer provide to get the error message. In number of bytes.

type (IN)
The type of the handle (OCI_HTYPE_ERR or OCI_HTYPE_ENV).

Comments

Returns an error message in the buffer provided and an ORACLE error code. This
function does not support SQL statements. In most cases, *hndlp* is actually the
error handle, or the environment handle. You should always get the message in the
encoding that was set in the environment handle.This function can be called
multiple times if there are more than one diagnostic record for an error.

The error handle is originally allocated with a call to OCIHandleAlloc().

Example

The following sample code demonstrates how you can use `OCIErrorGet()` in an error-handling routine. This routine prints out the type of status code returned by an OCI function, and if an error occurred, `OCIErrorGet()` retrieves the text of the message, which is printed.

```
static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{ text errbuf[512];
  ub4 buflen;
  ub4 errcode;
switch (status)
{ case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    printf("ErrorOCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    printf("ErrorOCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    printf("ErrorOCI_NO_DATA\n");
    break;
  case OCI_ERROR:
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
            errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error%s\n", errbuf);
    break;
  case OCI_INVALID_HANDLE:
    printf("ErrorOCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    printf("ErrorOCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    printf("ErrorOCI_CONTINUE\n");
    break;
  default:
    break;
  }
}
```

Related Functions

OCIHandleAlloc()

OCILdaToSvcCtx()

Purpose

Converts a V7 Lda_Def to a V8 or later service context handle.

Syntax

```
sword OCILdaToSvcCtx ( OCISvcCtx  **svchpp,
                       OCIError   *errhp,
                       Lda_Def    *ldap );
```

Parameters

svchpp (IN/OUT)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

ldap (IN/OUT)
The Oracle7 logon data area returned by OCISvcCtxToLda() from this service context.

Comments

Converts an Oracle7 **Lda_Def** to a release 8 or later service context handle. The action of this call can be reversed by passing the resulting service context handle to the OCISvcCtxToLda() function.

The OCILdaToSvcCtx() call should be used only for resetting an **Lda_Def** obtained from OCISvcCtxToLda() back to a service context handle. It cannot be used to transform an **Lda_def** which started as an **Lda_def** back to a service context handle.

If the service context has been converted to an **Lda_Def**, only Oracle7 calls may be used. It is illegal to make OCI release 8 or later calls without first resetting the **Lda_Def** to a service context.

The OCI_ATTR_IN_V8_MODE attribute of the server handle or service context handle enables an application to determine whether the application is currently in Oracle release 7 mode or Oracle release 8 or later mode.

> **See Also:** Appendix A, "Handle and Descriptor Attributes"

Related Functions

OCISvcCtxToLda()

OCIPasswordChange()

Purpose

This call allows the password of an account to be changed.

Syntax

```
sword OCIPasswordChange ( OCISvcCtx      *svchp,
                          OCIError       *errhp,
                          CONST text     *user_name,
                          ub4            usernm_len,
                          CONST text     *opasswd,
                          ub4            opasswd_len,
                          CONST text     *npasswd,
                          sb4            npasswd_len,
                          ub4            mode );
```

Parameters

svchp (IN/OUT)
A handle to a service context. The service context handle must be initialized and have a server context handle associated with it.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

user_name (IN)
Specifies the user name, which can be in UTF-16 encoding. It must be terminated with a NULL character if the service context has been initialized with an authentication handle.

usernm_len (IN)
The length of the user name string specified in *user_name*, in number of bytes regardless of the encoding. *usernm_len* must be non-zero.

opasswd (IN)
Specifies the user's old password, which can be in UTF-16 encoding.

opasswd_len (IN)
The length of the old password string specified in *opasswd*, in bytes. *opasswd_len* must be non-zero.

npasswd (IN)
Specifies the user's new password, which can be in UTF-16 encoding. If the password complexity verification routine is specified in the user's profile to verify

the new password's complexity, the new password must meet the complexity requirements of the verification function.

npasswd_len (IN)
Then length in bytes of the new password string specified in `npasswd.` For a valid password string, `npasswd_len` must be non-zero.

mode (IN)

- OCI_DEFAULT - use the setting in the environment handle.

- OCI_UTF16. - use UTF-16 encoding, regardless of the setting of the environment handle.

  There is only one encoding allowed, either UTF-16 or not, for `user_name`, `opasswd`, and `npasswd`.

- OCI_AUTH - If a user session context is not created, this call creates the user session context and changes the password. At the end of the call, the user session context is not cleared. Hence the user remains logged in.

  If the user session context is already created, this call just changes the password and the flag has no effect on the session. Hence the user still remains logged in.

Comments

This call allows the password of an account to be changed. This call is similar to `OCISessionBegin()` with the following differences:

- If the user session is already established, it authenticates the account using the old password and then changes the password to the new password

- If the user session is not established, it establishes a user session and authenticates the account using the old password, then changes the password to the new password.

This call is useful when the password of an account has expired and `OCISessionBegin()` returns an error (ORA-28001) or warning that indicates that the password has expired.

The `mode` or the environment handle determines if UTF-16 is being used.

Related Functions

OCISessionBegin()

OCIReset()

Purpose

Resets the interrupted asynchronous operation and protocol. Must be called if a OCIBreak call had been issued while a non-blocking operation was in progress.

Syntax

```
sword OCIReset ( dvoid      *hndlp,
                 OCIError   *errhp );
```

Parameters

hndlp (IN)
The service context handle or the server context handle.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

Comments

This call is called in non-blocking mode only. Resets the interrupted asynchronous operation and protocol. Must be called if an OCIBreak() call had been issued while a non-blocking operation was in progress.

Related Functions

OCIBreak()

OCIRowidToChar()

Purpose

Converts a Universal ROWID to character extended (base 64) representation.

Syntax

```
sword OCIRowidToChar ( OCIRowid      *rowidDesc,
                       OraText       *outbfp,
                       ub2           *outbflp
                       OCIError      *errhp );
```

Parameters

rowidDesc (IN)
The ROWID descriptor which is allocated by OCIDescriptorAlloc() and populated by a prior execution of a SQL statement.

outbfp (OUT)
Pointer to the buffer where the character representaion is stored after successful execution of this call.

outbflp (IN/OUT)
Pointer to the output buffer length. Before execution, the buffer length contains the size of *outbfp*. After execution it contains the number of bytes converted.

In the event of truncation during conversion, *outbfp* contains the length required to make conversion successful. An error is also returned.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

Comments

After this conversion, the ROWID in character format can be bound with the OCIBindByPos() or OCIBindByName() calls, and used to query a row at the given ROWID.

OCIServerVersion()

Purpose

Returns the version string of the Oracle server.

Syntax

```
sword OCIServerVersion ( dvoid      *hndlp,
                         OCIError   *errhp,
                         text       *bufp,
                         ub4        bufsz
                         ub1        hndltype );
```

Parameters

hndlp (IN)
The service context handle or the server context handle.

errhp (IN)
An error handle you can pass to OCIErrorGet() for diagnostic information in the event of an error.

bufp (IN)
The buffer in which the version information is returned.

bufsz (IN)
The length of the buffer. In number of bytes.

hndltype (IN)
The type of handle passed to the function.

Comments

This call returns the version string of the Oracle server. It can be in Unicode if the environment handle so determines.

For example, the following might be returned as the version string if an application is running on an 8.1.5 SunOS server:

```
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production
```

Related Functions

OCIErrorGet()

OCISvcCtxToLda()

Purpose

Toggles between a V8 or later service context handle and a V7 Lda_Def.

Syntax

```
sword OCISvcCtxToLda ( OCISvcCtx    *srvhp,
                       OCIError     *errhp,
                       Lda_Def      *ldap );
```

Parameters

svchp (IN/OUT)
The service context handle.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

ldap (IN/OUT)
A Logon Data Area for Oracle7-style OCI calls which is initialized by this call.

Comments

Toggles between an OCI release 8 or later service context handle and an Oracle7 **Lda_Def**.

This function can only be called after a service context has been properly initialized.

Once the service context has been translated to an **Lda_Def**, it can be used in release 7.x OCI calls (e.g., `obindps()`, `ofen()`).

> **Note:** If there are multiple service contexts which share the same server handle, only one can be in Oracle7 mode at any time.

The action of this call can be reversed by passing the resulting **Lda_Def** to the `OCILdaToSvcCtx()` function.

The OCI_ATTR_IN_V8_MODE attribute of the server handle or service context handle enables an application to determine whether the application is currently in Oracle release 7 mode or Oracle release 8 or later mode.

See Also: Appendix A, "Handle and Descriptor Attributes"

Related Functions

OCILdaToSvcCtx()

OCIUserCallbackGet()

Purpose

Determines the callback that is registered for a handle.

Syntax

```
sword OCIUserCallbackGet ( dvoid    *hndlp,
                           ub4    type,
                           dvoid    *ehndlp,
                           ub4    fcode,
                           ub4    when
                           OCIUserCallback (*callbackp)
                                            (/*_
                                              dvoid  *ctxp,
                                              dvoid  *hndlp,
                                              ub4  type,
                                              ub4  fcode,
                                              ub1  when,
                                              sword  returnCode,
                                              ub4  *errnop,
                                              va_list arglist
                                              _*/),
                           dvoid    **ctxpp
                           OCIUcb  *ucbDesc );
```

Parameters

hndlp (IN)
This is the handle whose type is specified by the type parameter.

type (IN)
The handle type. The valid handle type is:

■   OCI_HTYPE_ENV - The callback is registered for all calls of the function
specified by *fcode* made on the environment handle.

ehndlp (IN)
The OCI error or environment handle. If there is an error, it is recorded in *ehndlp*
and this function returns OCI_ERROR. Diagnostic information can be obtained by
calling OCIErrorGet().

fcode (IN)
A unique function code of an OCI function. These are listed in Table 16–7, "OCI
Function Codes" on page 16-177.

when (IN)
Defines when the callback is invoked. Valid modes are:

- OCI_CBTYPE_ENTRY - the callback is invoked on entry into the OCI function.

- OCI_CBTYPE_EXIT - the callback is invoked before exit from the OCI function.

- OCI_UCBTYPE_REPLACE - if it returns anything other than an OCI_CONTINUE, then the next replacement callback and the OCI code for the OCI function is not called. Instead, processing jumps to the exit callbacks. For information about this parameter see `OCIUserCallbackRegister()` on page 16-173.

callbackp (OUT)
A pointer to a callback function pointer. This returns the function that is currently registered for these values of `fcode`, `when`, and `hndlp`. The value returned would be NULL if no callback is registered for this case.

> **See Also:** For information about the parameters of `callbackp` see the description of `OCIUserCallbackRegister()` on page 16-173

ctxpp (OUT)
A pointer to return context for the currently registered callback.

ucbDesc (IN)
An OCI provided descriptor. This descriptor is passed by OCI in the environment callback. It contains the priority at which the callback would be registered at. If the `ucbDesc` parameter is specified as NULL, then this callback has the highest priority.

User callbacks registered statically (as opposed to those registered dynamically in a package) use a null descriptor because they do not have a ucb descriptor to use.

Comments

This function finds out what callback is registered for a particular handle.

> **See Also:** For information on the restrictions of the use of callback functions, see "Restrictions on Callback Functions" on page 9-32.

Related Functions

OCIUserCallbackRegister()

OCIUserCallbackRegister()

Purpose

Register a user-created callback function

Syntax

```
sword OCIUserCallbackRegister ( dvoid    *hndlp,
                                ub4       type,
                                dvoid    *ehndlp,
                                OCIUserCallback  (callback)
                                                 (/*_
                                                    dvoid  *ctxp,
                                                    dvoid  *hndlp,
                                                    ub4  type,
                                                    ub4  fcode,
                                                    ub1  when,
                                                    sword  returnCode,
                                                    ub4  *errnop,
                                                    va_list arglist
                                                  _*/),
                                dvoid    *ctxp,
                                ub4       fcode,
                                ub4       when
                                OCIUcb   *ucbDesc );
```

Parameters

hndlp (IN)
This is the handle whose type is specified by the type parameter.

type (IN)
The handle type. The valid handle type is:

- OCI_HTYPE_ENV - The callback is registered for all calls of the function specified by *fcode* made on the environment handle.

ehndlp (IN)
The OCI error or environment handle. If there is an error, it is recorded in *ehndlp* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling OCIErrorGet(). Note that the because an error handle is not available within OCIEnvCallback, so the environment handle is passed in as a *ehndlp*.

callback (IN)

A callback function pointer. The variable argument list in the OCIUserCallback function prototype are the parameters passed to the OCI function. The typedef for OCIUserCallback is described below later.

If an entry callback returns anything other than OCI_CONTINUE, then the return code is passed to the subsequent entry or replacement callback, if there is one. If this is the last entry callback and there is no replacement callback, then the OCI code is executed and the return code is ignored.

If a replacement callback returns anything other than OCI_CONTINUE, then subsequent replacement callbacks and the OCI code are bypassed, and processing jumps to the exit callbacks.

If the exit callback returns anything other than OCI_CONTINUE, then that returned value is returned by the OCI function; otherwise, the return value from the OCI code or the replacement callback (if the replacement callback did not return OCI_CONTINUE and essentially bypassed the OCI code) is returned by the call.

If a null value is passed in for callback, then the callback is removed for the *when* value and the specified handle. This is the way to de-register a callback for a given *ucbDesc* value, including the null *ucbDesc.*

ctxp (IN)

A context pointer for the callback.

fcode (IN)

A unique function code of an OCI function. These are listed in Table 16–7, "OCI Function Codes" on page 16-177.

when (IN)

Defines when the callback is invoked. Valid modes are:

- OCI_CBTYPE_ENTRY - the callback is invoked on entry into the OCI function.

- OCI_CBTYPE_EXIT - the callback is invoked before exit from the OCI function.

- OCI_UCBTYPE_REPLACE - if it returns anything other than OCI_CONTINUE, then the next replacement callback and the OCI code for the OCI function is not called. Instead, processing jumps to the exit callbacks.

ucbDesc (IN)

An OCI provided descriptor. This descriptor is passed by OCI in the environment callback. It contains the priority at which the callback would be registered at. If the *ucbDesc* parameter is specified as null, then this callback has the highest priority.

User callbacks registered statically (as opposed to those registered dynamically in a package) use a null descriptor as they do not have a ucb descriptor to use.

Comments

This function is used to register a user-created callback functions.s with the OCI environment.

> **See Also:** For a more complete discussion, see "User-Defined Callback Functions" on page 16-2.

Such callbacks allow an application to:

1. Trace OCI calls for debugging and performance measurements.

2. Perform additional pre- or post-processing after selected OCI calls.

3. Substitute the body of a given function with proprietary code to execute on a foreign data source.

The OCI supports these kinds of callbacks: *entry callbacks, replacement callbacks,* and *exit callbacks.*

The three types of callbacks are identified by the modes OCI_UCBTYPE_ENTRY, OCI_UCBTYPE_REPLACE, and OCI_UCBTYPE_EXIT.

The control flow now is:

- Execute entry callbacks.

- Execute replacement callbacks.

- Execute OCI code.

- Execute exit callbacks.

Entry callbacks are executed when a program enters an OCI function.

Replacement callbacks are executed after entry callbacks. If the replacement callback returns a value of OCI_CONTINUE, then subsequent replacement callbacks or the normal OCI-specific code is executed. If the callback returns anything other than OCI_CONTINUE, then subsequent replacement callbacks and the OCI code do not execute.

After an OCI function successfully executes, or after a replacement callback returns something other than OCI_CONTINUE, program control transfers to the exit callback (if one is registered).

If a replacement or exit callback returns anything other than OCI_CONTINUE, then the return code from the callback is returned from the associated OCI call.

To find out the callback that is registered for the handle, you can use OCIUserCallbackGet().

The prototype of the *OCIUserCallback* typedef is:

```
typedef sword (*OCIUserCallback)
         (dvoid   *ctxp,
          dvoid   *hndlp,
          ub4      type,
          ub4      fcode,
          ub4      when,
          sword   returnCode,
          ub4     *errnop,
          va_list arglist
          );
```

The parameters to the OCIUserCallback function prototype are:

ctxp (IN)
The context passed in as ctxp in the register callback function.

hndlp (IN)
This is the handle whose type is specified in the type parameter. It is the handle on which the callback is invoked. Because we only allow a type of OCI_HTYPE_ENV, therefore, the environment handle, *env*, would be passed-in here.

type (IN)
The type registered for the *hndlp*. The valid handle type is:

■ OCI_HTYPE_ENV - The callback is registered for all calls of the function specified by *fcode* made on the environment handle.

fcode (IN)
The function code of the OCI call. These are listed in Table 16–7, "OCI Function Codes". Please note that callbacks can be registered for only the OCI calls listed in Table 16–2, "Advanced Queuing and Publish-Subscribe Functions".

when (IN)
The when value of the callback.

returnCode (IN)
This is the return code from the previous callback or the OCI code. For the first entry callback, OCI_SUCCESS will always be passed in. For the subsequent callbacks, the return code from the OCI code or the previous callback is passed in.

errnop (IN/OUT)
When the first entry callback is called, the input value of *errnop* is 0. If the callback is returning any value other than an OCI_CONTINUE, then it must also set an error number in *errnop*. This value is the set in the error handle passed in the OCI call.

For all subsequent callbacks, the input value of *errnop* is the value of error number in the error handle. Therefore, if the previous callback did not return OCI_CONTINUE, then the out value of *errnop* from the previous callback would be the one in the error handle, and that value would be passed in here to the subsequent callback. If, on the other hand, the previous callback returned OCI_CONTINUE, then whatever value that is in the error handle would be passed in here.

Note that if a non-Oracle error number is returned in *errnop*, then a callback must also be registered for the OCIErrorGet() function to return appropriate text for the error number.

arglist (IN)
These are the parameters to the OCI call passed in here as variable number of arguments. They should be de-referenced using *va_arg*, as illustrated in the user callback demonstration programs.

> **See Also:** See Appendix B, "OCI Demonstration Programs" for a list of the available demonstration programs.

Table 16–7 OCI Function Codes

#	OCI Routine	#	OCI Routine	#	OCI Routine
1	OCIInitialize	33	OCITransStart	65	OCIDefineByPos
2	OCIHandleAlloc	34	OCITransDetach	66	OCIBindByPos
3	OCIHandleFree	35	OCITransCommit	67	OCIBindByName
4	OCIDescriptorAlloc	36	(not used)	68	OCILobAssign
5	OCIDescriptorFree	37	OCIErrorGet	69	OCILobIsEqual
6	OCIEnvInit	38	OCILobFileOpen	70	OCILobLocatorIsInit
7	OCIServerAttach	39	OCILobFileClose	71	OCILobEnableBuffering
8	OCIServerDetach	40	(not used)	72	OCILobCharSetID
9	(not used)	41	(not used)	73	OCILobCharSetForm

Table 16–7   OCI Function Codes (Cont.)

#	OCI Routine	#	OCI Routine	#	OCI Routine
10	OCISessionBegin	42	OCILobCopy	74	OCILobFileSetName
11	OCISessionEnd	43	OCILobAppend	75	OCILobFileGetName
12	OCIPasswordChange	44	OCILobErase	76	OCILogon
13	OCIStmtPrepare	45	OCILobGetLength	77	OCILogoff
14	(not used)	46	OCILobTrim	78	OCILobDisableBuffering
15	(not used)	47	OCILobRead	79	OCILobFlushBuffer
16	(not used)	48	OCILobWrite	80	OCILobLoadFromFile
17	OCIBindDynamic	49	(not used)	81	OCILobOpen
18	OCIBindObject	50	OCIBreak	82	OCILobClose
19	(not used)	51	OCIServerVersion	83	OCILobIsOpen
20	OCIBindArrayOfStruct	52	(not used)	84	OCILobFileIsOpen
21	OCIStmtExecute	53	(not used)	85	OCILobFileExists
22	(not used)	54	OCIAttrGet	86	OCILobFileCloseAll
23	(not used)	55	OCIAttrSet	87	OCILobCreateTemporary
24	(not used)	56	OCIParamSet	88	OCILobFreeTemporary
25	OCIDefineObject	57	OCIParamGet	89	OCILobIsTemporary
26	OCIDefineDynamic	58	OCIStmtGetPieceInfo	90	OCIAQEnq
27	OCIDefineArrayOfStruct	59	OCILdaToSvcCtx	91	OCIAQDeq
28	OCIStmtFetch	60	(not used)	92	OCIReset
29	OCIStmtGetBindInfo	61	OCIStmtSetPieceInfo	93	OCISvcCtxToLda
30	(not used)	62	OCITransForget	94	OCILobLocatorAssign
31	(not used)	63	OCITransPrepare	95	(not used)
32	OCIDescribeAny	64	OCITransRollback	96	OCIAQListen

Related Functions

OCIUserCallbackGet()

17

OCI Navigational and Type Functions

This chapter describes the OCI navigational functions which are used to navigate through objects retrieved from an Oracle database server. It also contains the descriptions of the functions which are used to obtain type descriptor objects (TDOs). The chapter contains the following sections:

- Introduction to the Navigational and Type Functions

- OCI Flush or Refresh Functions

- OCI Mark or Unmark Object and Cache Functions

- OCI Get Object Status Functions

- OCI Miscellaneous Object Functions

- OCI Pin, Unpin, and Free Functions

- OCI Type Information Accessor Functions

Introduction to the Navigational and Type Functions

In an object navigational paradigm, data is represented as a graph of objects connected by references. Objects in the graph are reached by following the references. The OCI provides a navigational interface to objects in the Oracle server. Those calls are described in this chapter.

The OCI object environment is initialized when the application calls OCIInitialize() in OCI_OBJECT mode.

> **See Also:** For more information about using the calls in this chapter, refer to Chapter 10, "OCI Object-Relational Programming", and Chapter 13, "Object Cache and Object Navigation".

Object Types and Lifetimes

An object instance is an occurrence of a type defined in an Oracle database. This section describes how an object instance can be represented in OCI. See Figure 17–1 on page 17-3. In OCI, an object instance can be classified based on the type, the lifetime and referenceability:

- A persistent object is an instance of an object type. A persistent object resides in a row of a table in the server and can exist longer than the duration of a session (connection). Persistent objects can be identified by object references which contain the object identifiers. A persistent object is obtained by pinning its object reference.

- A transient object is an instance of an object type. A transient object cannot exist longer than the duration of a session, and it is used to contain temporary computing results. Transient objects can also be identified by references which contain transient object identifiers.

- A value is an instance of an user-defined type (object type or collection type) or any built-in Oracle type. Unlike objects, values of object types are identified by memory pointers, rather than by references.

A value can be standalone or embedded. A standalone value is usually obtained by issuing a select statement. OCI also allows the client program to select a row of object table into a value by issuing a SQL statement. A referenceable object in the database can be represented as a value which cannot be identified by a reference. A standalone value can also be an out-of-line attribute in an object, such as VARCHAR or RAW, or an out-of-line element in a collection, such as VARCHAR, RAW, or object.

An embedded value is physically included in a containing instance. An embedded value can be an in-line attribute in an object. such as number or nested object, or an in-line element in a collection.

All values are considered to be transient by OCI, which means that OCI does not support automatic flushing a value to the database, and the client has to explicitly execute a SQL statement to store a value into the database. For embedded values, they are flushed when their containing instance are flushed.

Figure 17–1 shows how instances can be classified according to their type and lifetime:

Figure 17–1   Classification of Instances by Type and Lifetime
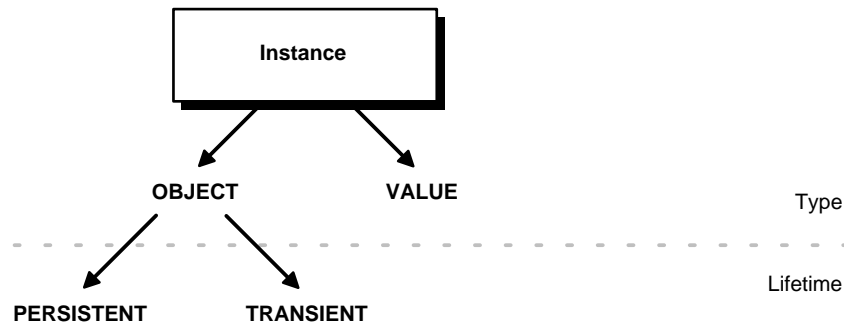
The distinction between various instances is further illustrated by the following table:

	Persistent Object	Transient Object	Value
Type	object type	object type	object type, built-in, collection
Maximum Lifetime	until object is deleted	session	session
Referenceable	yes	yes	no
Embeddable	no	no	yes

Terminology

In the remainder of this chapter, the following terms will be used:

- An *object* can be generally used to refer to a persistent object, a transient object, a standalone value of object type, or an embedded value of object type.

- A *referenceable object* refers to a persistent object or a transient object.

- A *standalone object* refers to a persistent object, a transient object or a standalone value of object type.

- An *embedded object* refers to a embedded value of object type.

- An object is *dirty* if it has been created (*newed*), or marked updated or deleted.

> **See Also:** For a further discussion of the terms used to refer to different types of objects, please see "Persistent Objects, Transient Objects, and Values" on page 10-6.

The Function Syntax

The entries for each function contain the following information:

Purpose

A brief description of what the function does.

Syntax

The function declaration.

Comments

Detailed information about the function if available. This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below:

Mode	Description
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.

Returns

A description of what value is returned by the function if the function returns something other than the standard return codes listed in Table 18–1, "Function Return Values".

Related Functions

A list of related calls which may provide additional useful information.

Navigational Function Return Values

The OCI navigational functions typically return one of the following values:

Return Value	Meaning
OCI_SUCCESS	The operation succeeded
OCI_ERROR	The operation failed. The specific error can be retrieved by calling OCIErrorGet() on the error handle passed to the function.
OCI_INVALID_HANDLE	The environment or error handle passed to the function is null.

Function-specific return information follows the description of each function in this chapter. Information about specific error codes returned by each function is presented in the following section.

> **See Also:**   For more information about return codes and error handling, see the section "Error Handling" on page 2-31.

Server Roundtrips for Cache and Object Functions

For a table showing the number of server roundtrips required for individual OCI cache and object functions, refer to Appendix C, "OCI Function Server Round-trips".

Navigational Function Error Codes

Table 17–1 lists the external Oracle error codes which can be returned by each of the OCI navigational functions. The list following the table identifies what each error represents.

Table 17–1   OCI Navigational Functions Error Codes

Function	Possible ORA Errors
OCICacheFlush()	24350, 21560, 21705
OCICacheFree()	24350, 21560, 21705
OCICacheRefresh()	24350, 21560, 21705
OCICacheUnmark()	24350, 21560, 21705
OCICacheUnpin()	24350, 21560, 21705
OCIObjectArrayPin()	24350, 21560
OCIObjectCopy()	24350, 21560, 21705, 21710
OCIObjectExists()	24350, 21560, 21710
OCIObjectFlush()	24350, 21560, 21701, 21703, 21708, 21710
OCIObjectFree()	24350, 21560, 21603, 21710
OCIObjectGetAttr()	21560, 21600, 22305
OCIObjectGetInd()	24350, 21560, 21710
OCIObjectGetTypeRef()	24350, 21560, 21710
OCIObjectIsDirty()	24350, 21560, 21710
OCIObjectIsLocked()	24350, 21560, 21710
OCIObjectLock()	24350, 21560, 21701, 21708, 21710
OCIObjectLockNoWait()	24350, 21560, 21701, 21708, 21710
OCIObjectMarkDelete()	24350, 21560, 21700, 21701, 21702, 21710
OCIObjectMarkDeleteByRef()	24350, 21560
OCIObjectMarkUpdate()	24350, 21560, 21700, 21701, 21710
OCIObjectNew()	24350, 21560, 21705, 21710
OCIObjectPin()	24350, 21560, 21700, 21702
OCIObjectPinCountReset()	24350, 21560, 21710
OCIObjectPinTable()	24350, 21560, 21705
OCIObjectRefresh()	24350, 21560, 21709, 21710
OCIObjectSetAttr()	21560, 21600, 22305, 22279, 21601
OCIObjectUnmark()	24350, 21560, 21710

Table 17–1 OCI Navigational Functions Error Codes (Cont.)

Function	Possible ORA Errors
OCIObjectUnmarkByRef()	24350, 21560
OCIObjectUnpin()	24350, 21560, 21710
OCIOjectGetObjectRef()	24350, 21560, 21710

The ORA errors in Table 17–1 have the following meanings.

- ORA-21560 - name argument should not be null
- ORA-21600 - path expression too long
- ORA-21601 - attribute is not an instance of user-defined type
- ORA-21603 - cannot free a dirtied persistent object
- ORA-21700 - object does not exist or has been deleted
- ORA-21701 - invalid object
- ORA-21702 - object is not instantiated in the cache
- ORA-21703 - cannot flush an object that is not modified
- ORA-21704 - terminate cache or connection without flushing
- ORA-21705 - service context is invalid
- ORA-21708 - operations cannot be performed on a transient object
- ORA-21709 - operations can only be performed on a current object
- ORA-21710 - invalid pointer or value passed to the function
- ORA-22279 - cannot perform operation with LOB buffering enabled
- ORA-22305 - name argument is invalid
- ORA-24350 - this OCI call is not allowed from external subroutines

OCI Flush or Refresh Functions

This section describes the OCI flush or refresh functions.

Table 17–2   Flush or Refresh Functions

Function/Page	Purpose
OCICacheFlush() on page 17-9	Flush modified persistent objects in cache to server
OCICacheRefresh() on page 17-11	Refresh pinned persistent objects
OCIObjectFlush() on page 17-13	Flush a modified persistent object to the server
OCIObjectRefresh() on page 17-14	Refresh a persistent object

OCICacheFlush()

Purpose

Flushes modified persistent objects to the server

Syntax

```
sword OCICacheFlush ( OCIEnv            *env,
                      OCIError          *err,
                      CONST OCISvcCtx   *svc,
                      dvoid             *context,
                      OCIRef            *(*get)
                                        ( dvoid   *context,
                                          ub1     *last ),
                      OCIRef            **ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
OCI service context.

context (IN) [optional]
Specifies an user context that is an argument to the client callback function *get*.
This parameter is set to null if there is no user context.

get (IN) [optional]
A client-defined function which acts an iterator to retrieve a batch of dirty objects
that need to be flushed. If the function is not null, this function will be called to get
a reference of a dirty object. This is repeated until a null reference is returned by the
client function or the parameter *last* is set to TRUE. The parameter *context* is
passed to get() for each invocation of the client function. This parameter should
be null if user callback is not given. If the object that is returned by the client
function is not a dirtied persistent object, the object is ignored.

All the objects that are returned from the client function must be newed or pinned
using the same service context, otherwise an error is signalled. Note that the cache
flushes the returned objects in the order in which they were marked dirty.

If this parameter is passed as null (e.g., no client-defined function is provided), then all dirty persistent objects for the given service context are flushed in the order in which they were dirtied.

ref (OUT) [optional]
If there is an error in flushing the objects (*`ref`) will point to the object that is causing the error. If `ref` is null, then the object will not be returned. If *`ref` is null, then a reference will be allocated and set to point to the object. If *`ref` is not null, then the reference of the object is copied into the given space. If the error is not caused by any of the dirtied object, the given REF is initialized to be a null reference (OCIRefIsNull(*ref) is TRUE).

The REF is allocated for session duration (OCI_DURATION_SESSION). The application can free the allocated REF using the OCIObjectFree() function.

Comments

This function flushes the modified persistent objects from the object cache to the server. The objects are flushed in the order that they are newed or marked updated or deleted.

> **See Also:** OCIObjectFlush() *on page 17-13*

This function incurs at most one network round-trip.

Related Functions

OCIObjectFlush()

OCICacheRefresh()

Purpose

Refreshes all pinned persistent objects in the cache.

Syntax

```
sword OCICacheRefresh ( OCIEnv          *env,
                        OCIError        *err,
                        CONST OCISvcCtx *svc,
                        OCIRefreshOpt   option,
                        dvoid           *context,
                        OCIRef          *(*get)(dvoid *context),
                        OCIRef          **ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
OCI service context.

option (IN) [optional]
If OCI_REFRESH_LOADED is specified, all objects that are loaded within the
transaction are refreshed. If the option is OCI_REFRESH_LOADED and the
parameter *get* is not null, this function will ignore the parameter.

context (IN) [optional]
Specifies an user context that is an argument to the client callback function *get*.
This parameter is set to null if there is no user context.

get (IN) [optional]
A client-defined function which acts an iterator to retrieve a batch of objects that
need to be refreshed. If the function is not null, this function will be called to get a
reference of an object. If the reference is not null, then the object will be refreshed.
These steps are repeated until a null reference is returned by this function. The
parameter *context* is passed to get() for each invocation of the client function.
This parameter should be null if user callback is not given.

ref (OUT) [optional]

If there is an error in refreshing the objects, (*`ref`*) will point to the object that is causing the error. If `ref` is null, then the object will not be returned. If *`ref`* is null, then a reference will be allocated and set to point to the object. If *`ref`* is not null, then the reference of the object is copied into the given space. If the error is not caused by any of the object, the given ref is initialized to be a null reference (`OCIRefIsNull(*ref`) is TRUE).

Comments

This function refreshes all pinned persistent objects and all unpinned persistent objects are freed from the object cache.

> **See Also:** For more information about refreshing, see the description of `OCIObjectRefresh()`, and the section "Refreshing an Object Copy" on page 13-11.

> **Caution:** When objects are refreshed, the secondary-level memory of those objects could potentially move to a different place in memory. As a result, any pointers to attributes which were saved prior to this call may be invalidated. Examples of attributes using secondary-level memory include **OCIString** *, **OCIColl** *, and **OCIRaw** *.

Related Functions

`OCIObjectRefresh()`

OCIObjectFlush()

Purpose

Flushes a modified persistent object to the server.

Syntax

```
sword OCIObjectFlush ( OCIEnv      *env,
                       OCIError    *err,
                       dvoid       *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the descriptions of
`OCIEnvCreate()` and *OCIInitialize()* for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
A pointer to the persistent object. The object must be pinned before this call.

Comments

This function flushes a modified persistent object to the server. An exclusive lock is
obtained implicitly for the object when it is flushed. When the object is written to
the server, triggers may be fired. This function returns an error for transient objects
and values, and for unmodified persistent objects.

Objects can be modified by triggers at the server. To keep objects in the cache
consistent with the database, an application can free or refresh objects in the cache.

If the object to flush contains an internal LOB attribute and the LOB attribute was
modified due to an OCIObjectCopy(), OCILobAssign(), or
OCILobLocatorAssign() or by assigning another LOB locator to it, then the
flush makes a copy of the LOB value that existed in the source LOB at the time of
the assignment or copy of the internal LOB locator or object.

> **See Also:** For more information on LOB functions, see"LOB
> Functions" on page 16-4.

Related Functions

OCIObjectPin(), OCICacheFlush()

OCIObjectRefresh()

Purpose

Refreshes a persistent object from the most current database snapshot.

Syntax

```
sword OCIObjectRefresh ( OCIEnv       *env,
                         OCIError     *err,
                         dvoid        *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
A pointer to the persistent object, which must already be pinned.

Comments

This function refreshes an object with data retrieved from the latest snapshot in the
server. An object should be refreshed when the objects in the object cache are
inconsistent with the objects at the server.

Note: When an object is flushed to the server, triggers can be fired
to modify more objects in the server. The same objects (modified by
the triggers) in the object cache become out-of-date, and must be
refreshed before they can be locked or flushed.

This occurs when the user issues a SQL statement or PL/SQL
procedure to modify any object in the server.

Caution: Modifications made to objects (dirty objects) since the
last flush are lost if object are refreshed by this function.

The various meta-attribute flags and durations of an object are modified after being
refreshed:

Object Attribute	Status After Refresh
existent	set to appropriate value
pinned	unchanged
allocation duration	unchanged
pin duration	unchanged

The object that is refreshed will be *replaced-in-place*. When an object is replaced-in-place, the top-level memory of the object will be reused so that new data can be loaded into the same memory address. The top level memory of the null structure is also reused. Unlike the top-level memory chunk, the secondary memory chunks will be freed and reallocated.

You should be careful when writing functionality that holds on to a pointer to the secondary memory chunk, such as assigning the address of a secondary memory to a local variable, because this pointer can become invalid after the object is refreshed.

This function does nothing for transient objects or values.

Related Functions

OCICacheRefresh()

OCI Mark or Unmark Object and Cache Functions

This section describe the OCI mark or unmark Object and Cache functions.

Table 17–3   Mark or Unmark Object and Cache Functions

Function/Page	Purpose
OCICacheUnmark() on page 17-17	Unmarks objects in the cache
OCIObjectMarkDelete() on page 17-18	Mark an object deleted / delete a value instance
OCIObjectMarkDeleteByRef() on page 17-19	Mark an object deleted given a ref
OCIObjectMarkUpdate() on page 17-20	Mark an object as updated/dirty
OCIObjectUnmark() on page 17-22	Unmarks an object
OCIObjectUnmarkByRef() on page 17-23	Unmarks an object, given a ref to it

OCICacheUnmark()

Purpose

Unmarks all dirty objects in the object cache.

Syntax

```
sword OCICacheUnmark ( OCIEnv          *env,
                       OCIError        *err,
                       CONST OCISvcCtx *svc );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
`OCIEnvCreate()` and `OCIInitialize()` for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

svc (IN)
OCI service context.

Comments

If a connection is specified, this function unmarks all dirty objects in that
connection. Otherwise, all dirty objects in the cache are unmarked.

> **See Also:**   See `OCIObjectUnmark()` on page 17-22 for more
> information about unmarking an object.

Related Functions

`OCIObjectUnmark()`

OCIObjectMarkDelete()

Purpose

Marks a standalone instance as deleted, given a pointer to the instance.

Syntax

```
sword OCIObjectMarkDelete ( OCIEnv       *env,
                            OCIError     *err,
                            dvoid        *instance );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

instance (IN)
Pointer to the instance. It must be standalone, and if it is an object it must be
pinned.

Comments

This function accepts a pointer to a standalone instance and marks the object as
deleted. The object is freed according to the following rules:

For Persistent Objects
The object is marked deleted. The memory of the object is not freed. The object is
deleted in the server when the object is flushed.

For Transient Objects
The object is marked deleted. The memory of the object is not freed.

For Values
This function frees a value immediately.

Related Functions
OCIObjectMarkDeleteByRef(), OCIObjectGetProperty()

OCIObjectMarkDeleteByRef()

Purpose

Marks an object as deleted, given a reference to the object.

Syntax

```
sword OCIObjectMarkDeleteByRef ( OCIEnv        *env,
                                 OCIError      *err,
                                 OCIRef        *object_ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object_ref (IN)
Reference to the object to be deleted.

Comments

This function accepts a reference to an object, and marks the object designated by
object_ref as deleted. The object is marked and freed as follows:

For Persistent Objects
If the object is not loaded, then a temporary object is created and is marked deleted.
Otherwise, the object is marked deleted.

The object is deleted in the server when the object is flushed.

For Transient Objects
The object is marked deleted. The object is not freed until it is unpinned.

Related Functions

OCIObjectMarkDelete(), OCIObjectGetProperty()

OCIObjectMarkUpdate()

Purpose

Marks a persistent object as updated, or *dirty.*

Syntax

```
sword OCIObjectMarkUpdate ( OCIEnv      *env,
                            OCIError    *err,
                            dvoid       *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
A pointer to the persistent object, which must already be pinned.

Comments

This function marks a persistent object as updated, or *dirty.* The following special
rules apply to different types of objects. The *dirty* status of an object may be checked
by calling OCIObjectIsLocked().

For Persistent Objects
This function marks the specified persistent object as updated.

The persistent objects will be written to the server when the object cache is flushed.
The object is not locked or flushed by this function. It is an error to update a deleted
object.

After an object is marked updated and flushed, this function must be called again to
mark the object as updated if it has been dirtied after it is being flushed.

For Transient Objects
This function marks the specified transient object as updated. The transient objects
will not be written to the server. It is an error to update a deleted object.

For Values
This function is an no-op for values.

> **See Also:** For more information about the use of this function, see
> "Marking Objects and Flushing Changes" on page 10-15.

Related Functions

`OCIObjectPin()`, `OCIObjectGetProperty()`,
`OCIObjectIsDirty()`, `OCIObjectUnmark()`.

OCIObjectUnmark()

Purpose

Unmarks an object as dirty.

Syntax

```
sword OCIObjectUnmark ( OCIEnv      *env,
                        OCIError    *err,
                        dvoid       *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
Pointer to the persistent object. It must be pinned.

Comments

For Persistent Objects and Transient Objects
This function unmarks the specified persistent object as dirty. Changes that are
made to the object will not be written to the server. If the object is marked locked, it
remains marked locked. The changes that have already made to the object will not
be undone implicitly.

For Values
This function is an no-op for values. This means that the function will have no effect
if called on a value.

Related Functions
OCIObjectUnmarkByRef()

OCIObjectUnmarkByRef()

Purpose

Unmarks an object as dirty, given a REF to the object.

Syntax

```
sword OCIObjectUnmarkByRef ( OCIEnv     *env,
                             OCIError   *err,
                             OCIRef     *ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

ref (IN)
Reference of the object. It must be pinned.

Comments

This function unmarks an object as dirty. This function is identical to
OCIObjectUnmark(), except that it takes a REF to the object as an argument.

For Persistent Objects and Transient Objects
This function unmarks the specified persistent object as dirty. Changes that are
made to the object will not be written to the server. If the object is marked locked, it
remains marked locked. The changes that have already made to the object will not
be undone implicitly.

For Values
This function is a no-op for values.

Related Functions

OCIObjectUnmark()

OCI Get Object Status Functions

This section describes the OCI get object status functions.

Table 17–4   Get Object Status functions

Function/Page	Purpose
OCIObjectExists() on page 17-25	Get the existent status of an instance
OCIObjectGetProperty() on page 17-26	Get the status of a particular object property
OCIObjectIsDirty() on page 17-31	Get the dirtied status of an instance
OCIObjectIsLocked() on page 17-31	Get the locked status of an instance

OCIObjectExists()

Purpose

Returns the existence meta-attribute of a standalone instance.

Syntax

```
sword OCIObjectExists ( OCIEnv      *env,
                        OCIError    *err,
                        dvoid       *ins,
                        boolean     *exist );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

ins (IN)
Pointer to an instance. If it is an object, it must be pinned.

exist (OUT)
Return value for the existence status.

Comments

This function returns the existence of an instance. If the instance is a value, this function always returns TRUE. The instance must be a standalone persistent or transient object.

> **See Also:** For more information about object meta-attributes, see "Object Meta-Attributes" on page 10-18.

Related Functions

OCIObjectPin()

OCIObjectGetProperty()

Purpose

Retrieve a given property of an object.

Syntax

```
sword OCIObjectGetProperty ( OCIEnv           *envh,
                             OCIError         *errh,
                             CONST dvoid      *obj,
                             OCIObjectPropId  propertyId,
                             dvoid            *property,
                             ub4              *size );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

obj (IN)
The object whose property is returned.

propertyId (IN)
The identifier which identifies the desired property.

property (OUT)
The buffer into which the desired property is copied.

size (IN/OUT)
On input, this parameter specifies the size of the property buffer passed by caller.

On output it contains the size in bytes of the property returned. This parameter is
required for string-type properties only, such as OCI_OBJECTPROP_SCHEMA,
OCI_OBJECTPROP_TABLE). For non-string properties this parameter is ignored
since the size is fixed.

Comments

This function returns the specified property of the object. The desired property is
identified by *propertyId*. The property value is copied into *property* and for
string typed properties the string size is returned by *size*.

Objects are classified as persistent, transient and value depending upon the lifetime and referenceability of the object. Some of the properties are applicable only to persistent objects and some others only apply to persistent and transient objects. An error is returned if the user tries to get a property which in not applicable to the given object. To avoid such an error, the user should first check whether the object is persistent or transient or value (OCI_OBJECTPROP_LIFETIME property) and then appropriately query for other properties.

The different property ids and the corresponding type of *property* argument are given below.

OCI_OBJECTPROP_LIFETIME
This identifies whether the given object is a persistent object or a transient object or a value instance. The *property* argument must be a pointer to a variable of type **OCIObjectLifetime**. Possible values include:

- OCI_OBJECT_PERSISTENT

- OCI_OBJECT_TRANSIENT

- OCI_OBJECT_VALUE

OCI_OBJECTPROP_SCHEMA
This returns the schema name of the table in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the schema name an error is returned, the error message will communicate the required size. Upon success, the size of the returned schema name in bytes is returned by *size*. The *property* argument must be an array of type **text** and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_TABLE
This returns the table name in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the table name an error is returned, the error message will communicate the required size. Upon success, the size of the returned table name in bytes is returned by *size*. The *property* argument must be an array of type text and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_PIN_DURATION
This returns the pin duration of the object. An error is returned if the given object points to a value instance. The *property* argument must be a pointer to a variable of type **OCIDuration**. Valid values include

- OCI_DURATION_SESSION

- OCI_DURATION_TRANS

> **See Also:** For more information about durations, see "Object Duration" on page 13-15.

OCI_OBJECTPROP_ALLOC_DURATION

This returns the allocation duration of the object. The *property* argument must be a pointer to a variable of type **OCIDuration**. Valid values include:

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

For more information about durations, see "Object Duration" on page 13-15.

OCI_OBJECTPROP_LOCK

This returns the lock status of the object. The possible lock statuses are enumerated by **OCILockOpt**. An error is returned if the given object points to a transient or value instance. The *property* argument must be a pointer to a variable of type **OCILockOpt**. Note, the lock status of an object can also be retrieved by calling OCIObjectIsLocked(). Valid values include:

- OCI_LOCK_NONE - for no lock
- OCI_LOCK_X - for an exclusive lock
- OCI_LOCK_X_NOWAIT - for an exclusive lock with the NOWAIT option.

  > **See Also:** For information about the NOWAIT option, see "Locking With the NOWAIT Option" on page 13-13.

OCI_OBJECTPROP_MARKSTATUS

This returns the dirty status and indicates whether the object is a new object, updated object or deleted object. An error is returned if the given object points to a transient or value instance. The *property* argument must be of type **OCIObjectMarkStatus**. Valid values include:

- OCI_OBJECT_NEW
- OCI_OBJECT_DELETED
- OCI_OBJECT_UPDATED

The following macros are available to test the object mark status:

- OCI_OBJECT_IS_UPDATED (flag)
- OCI_OBJECT_IS_DELETED (flag)
- OCI_OBJECT_IS_NEW (flag)

■ OCI_OBJECT_IS_DIRTY (flag)

OCI_OBJECTPROP_VIEW
This identifies whether the specified object is a view object or not. If the property value returned is TRUE, it indicates the object is a view otherwise it is not. An error is returned if the given object points to a transient or value instance. The *property* argument must be of type boolean.

Related Functions

`OCIObjectLock()`, `OCIObjectMarkDelete()`, `OCIObjectMarkUpdate()`, `OCIObjectPin()`, `OCIObjectPin()`

OCIObjectIsDirty()

Purpose

Checks to see if an object is marked as *dirty*.

Syntax

```
sword OCIObjectIsDirty ( OCIEnv      *env,
                         OCIError    *err,
                         dvoid       *ins,
                         boolean     *dirty );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

ins (IN)
Pointer to an instance.

dirty (OUT)
Return value for the dirty status.

Comments

The instance passed to this function must be standalone. If the instance is an object,
the instance must be pinned.

This function returns the dirty status of an instance. If the instance is a value, this
function always returns FALSE for the dirty status.

Related Functions

OCIObjectMarkUpdate(), OCIObjectGetProperty()

OCIObjectIsLocked()

Purpose

Get lock status of an object.

Syntax

```
sword OCIObjectIsLocked ( OCIEnv      *env,
                          OCIError    *err,
                          dvoid       *ins,
                          boolean     *lock );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

ins (IN)
Pointer to an instance. The instance must be standalone, and if it is an object it must
be pinned.

lock (OUT)
Return value for the lock status.

Comments

This function returns the lock status of an instance. If the instance is a value, this
function always returns FALSE.

Related Functions

OCIObjectLock(), OCIObjectGetProperty()

OCI Miscellaneous Object Functions

This section describes the miscellaneous object functions.

Table 17–5   Miscellaneous Object functions

Function/Page	Purpose
OCIObjectCopy() on page 17-33	Copy one instance to another
OCIObjectGetAttr() on page 17-35	Gets an object attribute
OCIObjectGetInd() on page 17-37	Get null structure of an instance
OCIObjectGetObjectRef() on page 17-38	Return reference to a given object
OCIObjectGetTypeRef() on page 17-39	Get a reference to a TDO of an instance
OCIObjectLock() on page 17-40	Lock a persistent object
OCIObjectLockNoWait() on page 17-41	Lock a persistent object but do not wait for the lock
OCIObjectPin() on page 17-56	Create a new instance
OCIObjectSetAttr() on page 17-47	Sets an object attribute

OCIObjectCopy()

Purpose

Copies a source instance to a destination.

Syntax

```
sword OCIObjectCopy ( OCIEnv            *env,
                      OCIError          *err,
                      CONST OCISvcCtx   *svc,
                      dvoid             *source,
                      dvoid             *null_source,
                      dvoid             *target,
                      dvoid             *null_target,
                      OCIType           *tdo,
                      OCIDuration       duration,
                      ub1               option );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
`OCIEnvCreate()` and `OCIInitialize()` in Chapter 15 for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
An OCI service context handle, specifying the service context on which the copy
operation is taking place

source (IN)
A pointer to the source instance; if it is an object, it must be pinned.

> **See Also:** `OCIObjectPin()` on page 17-56

null_source (IN)
Pointer to the null structure of the source object.

target (IN)
A pointer to the target instance; if it is an object is must be pinned.

null_target (IN)
A pointer to the null structure of the target object.

tdo (IN)
The TDO for both the source and the target. Can be retrieved with
`OCIDescribeAny()`.

duration (IN)
Allocation duration of the target memory.

option (IN)
This parameter is currently unused. Pass as zero or OCI_DEFAULT.

Comments

This function copies the contents of the *source* instance to the *target* instance.
This function performs a deep-copy such that all of the following is copied:

- all the top level attributes (see the exceptions below)

- all secondary memory (of the source) reachable from the top level attributes

- the null structure of the instance

Memory is allocated with the duration specified in the *duration* parameter.

Certain data items are not copied:

- If the option OCI_OBJECTCOPY_NOREF is specified in the *option* parameter,
  then all references in the source are not copied. Instead, the references in the
  target are set to null.

- If the attribute is an internal LOB, then only the LOB locator from the source
  object is copied. A copy of the LOB data is not made until `OCIObjectFlush()`
  is called. Before the target object is flushed, both the source and the target
  locators refer to the same LOB value.

The target or the containing instance of the target must be already have been
created. This may be done with `OCIObjectNew()` or `OCIObjectPin()`
depending on whether or not the target object already exists.

The *source* and *target* instances must be of the same type. If the source and
target are located in a different databases, then the same type must exist in both
databases.

Related Functions

`OCIObjectPin()`

OCIObjectGetAttr()

Purpose

Retrieves an object attribute.

Syntax

```
sword OCIObjectGetAttr ( OCIEnv          *env,
                         OCIError        *err,
                         dvoid           *instance,
                         dvoid           *null_struct,
                         struct OCIType  *tdo,
                         CONST OraText   **names,
                         CONST ub4       *lengths,
                         CONST ub4       name_count,
                         CONST ub4       *indexes,
                         CONST ub4       index_count,
                         OCIInd          *attr_null_status,
                         dvoid           **attr_null_struct,
                         dvoid           **attr_value,
                         struct OCIType  **attr_tdo );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the descriptions of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

instance (IN)
Pointer to an object.

null_struct (IN)
The null structure of the object or array.

tdo (IN)
Pointer to the TDO.

names (IN)
Array of attribute names. This is used to specify the names of the attributes in the
path expression.

lengths (IN)
Array of lengths of attribute names.

name_count (IN)
Number of element in the array *names*.

indexes (IN) [optional]
Not currently supported. Pass as (ub4 *)0.

index_count (IN) [optional]
Not currently supported. Pass as (ub4)0.

attr_null_status (OUT)
The null status of the attribute if the type of attribute is primitive.

attr_null_struct (OUT)
The null structure of an object or collection attribute.

attr_value (OUT)
Pointer to the attribute value.

attr_tdo (OUT)
Pointer to the TDO of the attribute.

Comments

This function gets a value from an object or from an array. If the parameter *instance* points to an object, then the path expression specifies the location of the attribute in the object. It is assumed that the object is pinned and that the value returned is valid until the object is unpinned.

Related Functions

OCIObjectSetAttr()

OCIObjectGetInd()

Purpose

Gets the null structure of a standalone instance.

Syntax

```
sword OCIObjectGetInd ( OCIEnv        *env,
                        OCIError      *err,
                        dvoid         *instance,
                        dvoid         **null_struct );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

instance (IN)
A pointer to the instance whose null structure is being retrieved. The instance must
be standalone. If *instance* is an object, it must already be pinned.

null_struct (OUT)
The null structure for the instance.

Comments

None.

Related Functions

OCIObjectPin()

OCIObjectGetObjectRef()

Purpose

Returns a reference to a given persistent object.

Syntax

```
sword OCIObjectGetObjectRef ( OCIEnv        *env,
                              OCIError      *err,
                              dvoid         *object,
                              OCIRef        *object_ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
Pointer to a persistent object. It must already be pinned.

object_ref (OUT)
A reference to the object specified in *object*. The reference must already be allocated. This can be accomplished with OCIObjectNew().

Comments

This function returns a reference to the given persistent object, given a pointer to the object. Passing a value (rather than an object) to this function causes an error.

> **See Also:** For more information about object meta-attributes, see "Object Meta-Attributes" on page 10-18.

Related Functions

OCIObjectPin(), OCIObjectPin()

OCIObjectGetTypeRef()

Purpose

Returns a reference to the type descriptor object (TDO) of a standalone instance.

Syntax

```
sword OCIObjectGetTypeRef ( OCIEnv      *env,
                            OCIError    *err,
                            dvoid       *instance,
                            OCIRef      *type_ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

instance (IN)
A pointer to the standalone instance. It must be standalone, and if it is an object, it must already be pinned.

type_ref (OUT)
A reference to the type of the object. The reference must already be allocate. This can be accomplished with OCIObjectNew().

Comments

None.

Related Functions

OCIObjectPin(), OCIObjectPin()

OCIObjectLock()

Purpose

Locks a persistent object at the server.

Syntax

```
sword OCIObjectLock ( OCIEnv        *env,
                      OCIError      *err,
                      dvoid         *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
A pointer to the persistent object being locked. It must already be pinned.

Comments

This function will return an error for transient objects and values. It also returns an error if the object does not exist.

> **See Also:** For more information about object locking, see "Locking Objects For Update" on page 13-13.

Related Functions

OCIObjectPin(), OCIObjectIsLocked(), OCIObjectGetProperty(), OCIObjectLockNoWait()

OCIObjectLockNoWait()

Purpose

Locks a persistent object at the server but does not wait for the lock. and returns an error if the lock is unavailable.

Syntax

```
sword OCIObjectLockNoWait ( OCIEnv      *env,
                            OCIError    *err,
                            dvoid       *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
A pointer to the persistent object being locked. It must already be pinned.

Comments

This function locks a persistent object at the server. However, unlike OCIObjectLock(), this function does not wait if another user holds the lock on the desired object and an error is returned if the object is currently locked by another user. This function also returns an error for transient objects and values, or objects that do not exist.

The lock of an object is released at the end of a transaction.

> **See Also:** For more information about object locking, see "Locking Objects For Update" on page 13-13.

OCIObjectLockNoWait() returns the following values:

- OCI_INVALID_HANDLE, if the environment handle or error handle is null.

- OCI_SUCCESS, if the operation suceeds.

- OCI_ERROR, if the operation fails.

Related Functions

`OCIObjectPin(), OCIObjectIsLocked(), OCIObjectGetProperty(), OCIObjectLock()`

OCIObjectNew()

Purpose

Creates a standalone instance

Syntax

```
sword OCIObjectNew ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCISvcCtx *svc,
                     OCITypeCode     typecode,
                     OCIType         *tdo,
                     dvoid           *table,
                     OCIDuration     duration,
                     boolean         value,
                     dvoid           **instance );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode with Unicode setting. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN) [optional]
OCI service handle. It must be given if the program wants to associate the duration of an instance with an OCI service (e.g. free a string when the transaction is committed). This parameter is ignored if the TDO is given.

typecode (IN)
The typecode of the type of the instance.

> **See Also:** "Typecodes" on page 3-31

tdo (IN) [optional]
Pointer to the type descriptor object. The TDO describes the type of the instance that is to be created. Refer to OCITypeByName() for obtaining a TDO. The TDO is required for creating a named type, such as an object or a collection.

table (IN) [optional]
Pointer to a table object which specifies a table in the server. `This` parameter can be set to null if no table is given. See the following description to find out how the table object and the TDO are used together to determine the kind of instances (persistent, transient, value) to be created. Also see `OCIObjectPinTable()` for retrieving a table object.

duration (IN)
This is an overloaded parameter. The use of this parameter is based on the kind of the instance that is to be created.

- Persistent object. This parameter specifies the pin duration.

- Transient object. This parameter specifies the allocation duration and pin duration.

- Value. This parameter specifies the allocation duration.

value (IN)
Specifies whether the created object is a value. If TRUE, then a value is created. Otherwise, a referenceable object is created. If the instance is not an object, then this parameter is ignored.

instance (OUT)
Address of the newly created instance. The instance can be a character string in Unicode if the environment handle has the appropriate setting and the object is OCIString. In this case, the instance will have a flag to indicate its Unicode setting.

Comments

This function creates a new instance of the type specified by the typecode or the TDO. It can create an OCIString object with a Unicode buffer if the typecode indicates the object to be created is OCIString.

> **See Also:** "Typecodes" on page 3-31

Based on the parameters *typecode* (or *tdo*), *value* and *table*, different instances are created:

	Value of *table* Parameter	
TYPE	Not null	null
object type (*value*=TRUE)	value	value
object type (*value*=FALSE)	persistent object	transient object

	Value of *table* Parameter	
built-in type	value	value
collection type	value	value

This function allocates the top-level memory chunk of an instance. The attributes in the top-level memory are initialized which means that an attribute of **varchar2** is initialized to a **OCIString** of 0 length. If the instance is an object, the object is marked existed but is atomically null.

> **See Also:** For information about creating new objects based on object views or user-created OIDs, see "Creating Objects Based on Object Views or User-defined OIDs" on page 10-36.

For Persistent Objects
The object is marked dirty and existed. The allocation duration for the object is session. The object is pinned and the pin duration is specified by the given parameter *duration*. Creating a persistent object does not cause any entries to be made into a database table until the object is flushed to the server.

For Transient Objects
The object is pinned. The allocation duration and the pin duration are specified by the given parameter *duration*.

For Values
The allocation duration is specified by the given parameter *duration*.

Attribute Values of New Objects
By default, all attributes of a newly created objects have null values. After initializing attribute data, the user must change the corresponding null status of each attribute to non-null.

It is possible to have attributes set to non-null values when an object is created. This is accomplished by setting the OCI_ATTR_OBJECT_NEWNOTNULL attribute of the environment handle to TRUE using OCIAttrSet(). This mode can later be turned off by setting the attribute to FALSE. If OCI_ATTR_OBJECT_NEWNOTNULL is set to TRUE, then OCIObjectNew() creates a non-null object.

> **See Also:** "Attribute Values of New Objects" on page 10-34

Objects with LOB Attributes

If the object contains an internal LOB attribute, the LOB is set to empty. The object must be marked as dirty and flushed (in order to insert the object into the table) and repinned before the user can start writing data into the LOB. When pinning the object after creating it, you must use the OCI_PIN_LATEST pin option in order to retrieve the newly updated LOB locator from the server.

If the object contains an external LOB attribute (FILE), the FILE locator is allocated but not initialized. The user must call `OCILobFileSetName()` to initialize the FILE attribute before flushing the object to the database. It is an error to `INSERT` or `UPDATE` a FILE without first indicating a directory alias and filename. Once the filename is set, the user can start reading from the FILE.

> **Note:** Oracle now supports only binary FILEs (`BFILE`s).

Related Functions

`OCIObjectPinTable()`, `OCIObjectFree()`

OCIObjectSetAttr()

Purpose

Set an object attribute.

Syntax

```
sword OCIObjectSetAttr ( OCIEnv              *env,
                         OCIError            *err,
                         dvoid               *instance,
                         dvoid               *null_struct,
                         struct OCIType      *tdo,
                         CONST OraText       **names,
                         CONST ub4           *lengths,
                         CONST ub4           name_count,
                         CONST ub4           *indexes,
                         CONST ub4           index_count,
                         CONST OCIInd        null_status,
                         CONST dvoid         *attr_null_struct,
                         CONST dvoid         *attr_value );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
`OCIEnvCreate()` and `OCIInitialize()` for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

instance (IN)
Pointer to an object instance.

null_struct (IN)
The null structure of the object instance or array.

tdo (IN)
Pointer to the TDO.

names (IN)
Array of attribute names. This is used to specify the names of the attributes in the
path expression.

lengths (IN)
Array of lengths of attribute names.

name_count (IN)
Number of element in the array *names*.

indexes (IN) [optional]
Not currently supported. Pass as `(ub4 *)0`.

index_count (IN) [optional]
Not currently supported. Pass as `(ub4)0`.

attr_null_status (IN)
The null status of the attribute if the type of attribute is primitive.

attr_null_struct (IN)
The null structure of an object or collection attribute.

attr_value (IN)
Pointer to the attribute value.

Comments

This function sets the attribute of the given object with the given value. The position of the attribute is specified as a path expression which is an array of names and an array of indexes.

Example

For the path expression stanford.cs.stu[5].addr, the arrays will look like:

names = {"stanford", "cs", "stu", "addr"}

lengths = {8, 2, 3, 4}

indexes = {5}

Related Functions

OCIObjectGetAttr()

OCI Pin, Unpin, and Free Functions

This section describes the OCI pin unpin, and free functions.

Table 17–6   Pin, Unpin, and Free Functions

Function/Page	Purpose
OCICacheFree() on page 17-50	Free objects in the cache
OCICacheUnpin() on page 17-51	Unpin persistent objects in cache or connection
OCIObjectArrayPin() on page 17-52	Pin an array of references
OCIObjectFree() on page 17-54	Free a previously allocated object
OCIObjectPin() on page 17-56	Pin an object
OCIObjectPinCountReset() on page 17-59	Unpin an object to zero pin count
OCIObjectPinTable() on page 17-61	Pin a table object with a given duration
OCIObjectUnpin() on page 17-63	Unpin an object

OCICacheFree()

Purpose

Frees all objects and values in the cache for the specified connection.

Syntax

```
sword OCICacheFree ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCISvcCtx *svc );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
An OCI service context.

Comments

If a connection is specified, this function frees the persistent objects, transient objects and values allocated for that connection. Otherwise, all persistent objects, transient objects and values in the object cache are freed. Objects are freed regardless of their pin count.

> **See Also:**   See OCIObjectFree() *on page 17-54* for more information about freeing an instance.

Related Functions

OCIObjectFree()

OCICacheUnpin()

Purpose

Unpins persistent objects.

Syntax

```
sword OCICacheUnpin ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCISvcCtx  *svc );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
An OCI service context handle. The objects on the specified connection are
unpinned.

Comments

This function completely unpins all of the persistent objects for the given
connection. The pin count for the objects is reset to zero.

> **See Also:** For more information about pinning and unpinning,
> see "Pinning an Object" on page 10-12, and "Pin Count and
> Unpinning" on page 10-30.

Related Functions

OCIObjectUnpin()

OCIObjectArrayPin()

Purpose

Pins an array of references.

Syntax

```
sword OCIObjectArrayPin ( OCIEnv          *env,
                          OCIError        *err,
                          OCIRef          **ref_array,
                          ub4             array_size,
                          OCIComplexObject **cor_array,
                          ub4             cor_array_size,
                          OCIPinOpt       pin_option,
                          OCIDuration     pin_duration,
                          OCILockOpt      lock,
                          dvoid           **obj_array,
                          ub4             *pos );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

ref_array (IN)
Array of references to be pinned

array_size (IN)
Number of elements in the array of references

cor_array
An array of COR handles corresponding to the objects being pinned.

cor_array_size
The number of elements in *cor_array*.

pin_option (IN)
Pin option.

> **See Also:** OCIObjectPin() on page 17-56

pin_duration (IN)
Pin duration. See `OCIObjectPin()`.

lock (IN)
Lock option. See `OCIObjectPin()`.

obj_array (OUT)
If this argument is not null, the pinned objects will be returned in the array. The user must allocate this array with element type being **dvoid** *. The size of this array is identical to `array_size`.

pos (OUT)
If there is an error, this argument indicates the element that is causing the error. Note that this argument is set to 1 for the first element in the `ref_array`.

Comments

All the pinned objects are retrieved from the database in one network roundtrip. If the user specifies an output array (`obj_array`), then the address of the pinned objects will be assigned to the elements in the array.

Related Functions

`OCIObjectPin()`

OCIObjectFree()

Purpose

Frees and unpins an object instance.

Syntax

```
sword OCIObjectFree ( OCIEnv        *env,
                      OCIError      *err,
                      dvoid         *instance,
                      ub2           flags );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

instance (IN)
Pointer to a standalone instance. If it is an object, it must be pinned.

flags (IN)
If OCI_OBJECTFREE_FORCE is passed, free the object even if it is pinned or dirty. If OCI_OBJECTFREE_NONULL is passed, the null structure is not freed.

Comments

This function deallocates all the memory allocated for an object instance, including the null structure. The following rules apply for different instance types:

For Persistent Objects
This function returns an error if the client is attempting to free a dirty persistent object that has not been flushed. The client should either flush the persistent object, unmark it, or set the parameter *flags* to OCI_OBJECTFREE_FORCE.

This function calls OCIObjectUnpin() once to check if the object can be completely unpin. If it succeeds, the rest of the function proceeds to free the object. If it fails, then an error is returned unless the parameter *flags* is set to OCI_OBJECTFREE_FORCE.

Freeing a persistent object in memory does not change the persistent state of that object at the server. For example, the object remains locked after the object is freed.

For Transient Objects

This function will call `OCIObjectUnpin()` once to check if the object can be completely unpin. If it succeeds, the rest of the function will proceed to free the object. If it fails, then an error is returned unless the parameter *flags* is set to OCI_OBJECTFREE_FORCE.

For Values

The memory of the object is freed immediately.

Related Functions

[OCICacheFree()](#)

OCIObjectPin()

Purpose

Pin a referenceable object.

Syntax

```
sword OCIObjectPin ( OCIEnv            *env,
                     OCIError          *err,
                     OCIRef            *object_ref,
                     OCIComplexObject  *corhdl,
                     OCIPinOpt         pin_option,
                     OCIDuration       pin_duration,
                     OCILockOpt        lock_option,
                     dvoid             **object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object_ref (IN)
The reference to the object.

corhdl (IN)
Handle for complex object retrieval.

pin_option (IN)
Used to specify the copy of the object that is to be retrieved.

pin_duration (IN)
The duration of which the object is being accessed by a client. The object is implicitly unpinned at the end of the pin duration. If OCI_DURATION_NULL is passed, there is no pin promotion if the object is already loaded into the cache. If the object is not yet loaded, then the pin duration is set to OCI_DURATION_DEFAULT in the case of OCI_DURATION_NULL.

lock_option (IN)
Lock option (e.g., exclusive). If a lock option is specified, the object is locked in the server. Note, the lock status of an object can also be retrieved by calling OCIObjectIsLocked(). Valid values include:

- OCI_LOCK_NONE - for no lock

- OCI_LOCK_X - for an exclusive lock

- OCI_LOCK_X_NOWAIT - for an exclusive lock with the NOWAIT option.

    See Also: For information about the NOWAIT option, see "Locking With the NOWAIT Option" on page 13-13.

object (OUT)
The pointer to the pinned object.

Comments

This function pins a referenceable object instance given the object reference. The process of pinning serves two purposes:

- locate an object given its reference. This is done by the object cache which keeps track of the objects in the object cache.

- notify the object cache that a persistent object is being in use such that the persistent object cannot be aged out. Since a persistent object can be loaded from the server whenever is needed, the memory utilization can be increased if a completely unpinned persistent object can be freed (aged out), even before the allocation duration is expired. An object can be pinned many times. A pinned object will remain in memory until it is completely unpinned.

    See Also: See OCIObjectUnpin() on page 17-63.

For Persistent Objects
When pinning a persistent object, if it is not in the cache, the object will be fetched from the persistent store. The allocation duration of the object is session. If the object is already in the cache, it is returned to the client. The object will be locked in the server if a lock option is specified.

This function will return an error for a non-existent object.

A pin option is used to specify the copy of the object that is to be retrieved:

- If *pin_option* is OCI_PIN_ANY (pin any), then if the object is already in the object cache, return this object. Otherwise, the object is retrieved from the database. In this case, it is the same as OCI_PIN_LATEST. This option is useful when the client knows that he has the exclusive access to the data in a session.

- If *pin_option* is OCI_PIN_LATEST (pin latest), if the object is not locked, it is retrieved from the database. If the object is cached, it is refreshed with the latest version. See `OCIObjectRefresh()` for more information about refreshing.

- If *pin_option* is OCI_PIN_RECENT (pin recent), if the object is loaded into the cache in the current transaction, the object is returned. If the object is not loaded in the current transaction, the object is refreshed from the server.

For Transient Objects

This function will return an error if the transient object has already been freed. This function does not return an error if an exclusive lock is specified in the lock option.

Related Functions

OCIObjectUnpin(), OCIObjectPinCountReset()

OCIObjectPinCountReset()

Purpose

Completely unpins an object, setting its pin count to zero.

Syntax

```
sword OCIObjectPinCountReset ( OCIEnv      *env,
                               OCIError    *err,
                               dvoid       *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
A pointer to an object, which must already be pinned.

Comments

This function completely unpins an object, setting its pin count to zero. When an
object is completely unpinned, it can be freed implicitly by the OCI at any time
without error. The following rules apply for specific object types:

For Persistent Objects
When a persistent object is completely unpinned, it becomes a candidate for aging.
The memory of an object is freed when it is aged out. Aging is used to maximize the
utilization of memory. An dirty object cannot be aged out unless it is flushed.

For Transient Objects
The pin count of the object is decremented. A transient can be freed only at the end
of its allocation duration or when it is explicitly freed by calling
OCIObjectFree().

For Values
This function will return an error for value.

> **See Also:**

Related Functions

`OCIObjectPin()`, `OCIObjectUnpin()`

OCIObjectPinTable()

Purpose

Pins a table object for a specified duration.

Syntax

```
sword OCIObjectPinTable ( OCIEnv           *env,
                          OCIError         *err,
                          CONST OCISvcCtx  *svc,
                          CONST OraText    *schema_name,
                          ub4              s_n_length,
                          CONST OraText    *object_name,
                          ub4              o_n_length,
                          dvoid            *not_used,
                          OCIDuration      pin_duration,
                          dvoid            **object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of `OCIEnvCreate()` and `OCIInitialize()` for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

svc (IN)
The OCI service context handle.

schema_name (IN) [optional]
The schema name of the table.

s_n_length (IN) [optional]
The length of the schema name indicated in *schema_name*.

object_name (IN)
The name of the table.

o_n_length (IN)
The length of the table name specified in *object_name*.

not_used (IN/OUT)
This parameter is not currently used. Pass as null.

pin_duration (IN)
The pin duration.

> **See Also:**   See description in `OCIObjectPin()` on page 17-56.

object (OUT)
The pinned table object.

Comments

This function pins a table object with the specified pin duration. The client can unpin the object by calling `OCIObjectUnpin()`.

The table object pinned by this call can be passed as a parameter to `OCIObjectNew()` to create a standalone persistent object.

Related Functions

`OCIObjectPin()`, `OCIObjectUnpin()`

OCIObjectUnpin()

Purpose

Unpins an object.

Syntax

```
sword OCIObjectUnpin ( OCIEnv      *env,
                       OCIError    *err,
                       dvoid       *object );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

object (IN)
A pointer to an object, which must already be pinned.

Comments

There is a pin count associated with each object which is incremented whenever an
object is pinned. When the pin count of the object is zero, the object is said to be
completely unpinned. An unpinned object can be freed implicitly by the OCI at any
time without error.

This function unpins an object. An object is completely unpinned when any of the
following is true:

1. The object's pin count reaches zero (i.e., it is unpinned a total of N times after
   being pinned a total of N times).

2. It is the end of the object's pin duration.

3. The function OCIObjectPinCountReset() is called on the object.

When an object is completely unpinned, it can be freed implicitly by the OCI at any
time without error.

The following rules apply for unpinning different types of objects:

For Persistent Objects
When a persistent object is completely unpinned, it becomes a candidate for aging. The memory of an object is freed when it is aged out. Aging is used to maximize the utilization of memory. An dirty object cannot be aged out unless it is flushed.

For Transient Objects
The pin count of the object is decremented. A transient can be freed only at the end of its allocation duration or when it is explicitly deleted by calling `OCIObjectFree()`.

For Values
This function returns an error for values.

Related Functions

OCIObjectPin(), OCIObjectPinCountReset()

OCI Type Information Accessor Functions

This section describes the OCI type information accessor functions.

Table 17–7   Type Information Accessor Functions

Function/Page	Purpose
OCITypeArrayByName() on page 17-66	Get an array of TDOs given an array of object names
OCITypeArrayByRef() on page 17-69	Get an array of TDOs given an array of object references
OCITypeByName() on page 17-71	Get a TDO given an object name
OCITypeByRef() on page 17-74	Get a TDO given an object reference

OCITypeArrayByName()

Purpose

Get an array of types given an array of names.

Syntax

```
sword OCITypeArrayByName ( OCIEnv           *envhp,
                           OCIError         *errhp,
                           CONST OCISvcCtx  *svc,
                           ub4              array_len,
                           CONST text       *schema_name[],
                           ub4              s_length[],
                           CONST text       *type_name[],
                           ub4              t_length[],
                           CONST text       *version_name[],
                           ub4              v_length[],
                           OCIDuration      pin_duration,
                           OCITypeGetOpt    get_option,
                           OCIType          *tdo[] );
```

Parameters

envhp (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
`OCIEnvCreate()` and `OCIInitialize()` for more information.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
OCI service handle.

array_len (IN)
Number of *schema_name/type_name/version_name* entries to be retrieved.

schema_name (IN, optional)
Array of schema names associated with the types to be retrieved. The array must
have *array_len* elements if specified. If 0 is supplied, the default schema is
assumed, otherwise it must have *array_len* number of elements. 0 can be
supplied for one or more of the entries to indicate that the default schema is desired
for those entries.

s_length (IN)
Array of *schema_name* lengths with each entry corresponding to the length of the corresponding *schema_name* entry in the *schema_name* array in bytes. The array must either have *array_len* number of elements or it must be 0 if *schema_name* is not specified.

type_name (IN)
Array of the names of the types to retrieve. This must have *array_len* number of elements.

t_length (IN)
Array of the lengths of type names in the *type_name* array in bytes.

version_name (IN)
The version name is ignored and the latest version of the requested type is returned. Because type evolution is available starting in release 9.0, pre-9.0 applications attempting to access an altered type will generate an error. These applications must be modified, re-compiled, and re-linked using the new type definition.

Array of the version names of the types to retrieve corresponding. This can be 0 to indicate retrieval of the most current versions, or it must have *array_len* number of elements.

If 0 is supplied, the most current version is assumed, otherwise it must have *array_len* number of elements. 0 can be supplied for one or more of the entries to indicate that the current version is desired for those entries.

v_length (IN)
Array of the lengths of version names in the *version_name* array in bytes.

pin_duration (IN)
Pin duration (e.g. until the end of current transaction) for the types retrieved. See `oro.h` for a description of each option.

get_option (IN)
Options for loading the types. It can be one of two values:

- OCI_TYPEGET_HEADER - for only the header to be loaded, or

- OCI_TYPEGET_ALL - for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)
Output array for the pointers to each pinned type in the object cache. It must have space for *array_len* pointers. Use `OCIObjectGetObjectRef()` to obtain the CREF to each pinned type descriptor.

Comments

Gets pointers to the existing types associated with the schema/type name array.

The *get_option* parameter can be used to control the portion of the TDO that gets loaded per roundtrip.

This function returns an error if any of the required parameters is null or any object types associated with a schema/type name entry do not exist.

To retrieve a single type, rather than an array, use OCITypeByName().

Related Functions

OCITypeArrayByRef(), OCITypeByName(), OCITypeByRef()

OCITypeArrayByRef()

Purpose

Get an array of types given an array of references.

Syntax

```
sword OCITypeArrayByRef ( OCIEnv          *envhp,
                          OCIError        *errhp,
                          ub4             array_len,
                          CONST OCIRef    *type_ref[],
                          OCIDuration     pin_duration,
                          OCITypeGetOpt   get_option,
                          OCIType         *tdo[] );
```

Parameters

envhp (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

array_len (IN)
Number of schema_name/type_name/version_name entries to be retrieved.

type_ref (IN)
Array of **OCIRef** * pointing to the particular version of the type descriptor object to
obtain. The array must have *array_len* elements if specified.

pin_duration (IN)
Pin duration (e.g. until the end of current transaction) for the types retrieved. See
oro.h for a description of each option.

get_option (IN)
Options for loading the types. It can be one of two values:

- OCI_TYPEGET_HEADER - for only the header to be loaded

- OCI_TYPEGET_ALL - for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)
Output array for the pointers to each pinned type in the object cache. It must have
space for *array_len* pointers. Use OCIObjectGetObjectRef() to obtain the
CREF to each pinned type descriptor.

Comments

Gets pointers to the with the schema/type name array.

This function returns an error if:

- any of the required parameters is null.

- one or more object types associated with a schema/type name entry does not exist.

To retrieve a single type, rather than an array of types, use `OCITypeByRef()`.

Related Functions

`OCITypeArrayByName()`, `OCITypeByRef()`, `OCITypeByName()`

OCITypeByName()

Purpose

Get the most current version of an existing type by name.

Syntax

```
sword OCITypeByName ( OCIEnv            *env,
                      OCIError          *err,
                      CONST OCISvcCtx   *svc,
                      CONST text        *schema_name,
                      ub4               s_length,
                      CONST text        *type_name,
                      ub4               t_length,
                      CONST text        *version_name,
                      ub4               v_length,
                      OCIDuration       pin_duration,
                      OCITypeGetOpt     get_option
                      OCIType           **tdo );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
OCI service handle.

schema_name (IN, optional)
Name of schema associated with the type. By default, the user's schema name is
used. This string must be all in upper-case, or else OCI throws an internal error and
the program stops.

s_length (IN)
Length of the *schema_name* parameter.

type_name (IN)
Name of the type to get. This string must be all in upper-case, or else OCI throws an
internal error and the program stops.

t_length (IN)
Length of the *type_name* parameter.

version_name (IN)
The version name is ignored and the latest version of the requested type is returned. Because type evolution is available starting in release 9.0, pre-9.0 applications attempting to access an altered type will generate an error. These applications must be modified, re-compiled, and re-linked using the new type definition.

User-readable version of the type. Pass as (text *)0 to retrieve the most current version.

v_length (IN)
Length of *version_name* in bytes.

pin_duration (IN)
Pin duration.

> **See Also:** "Object Duration" on page 13-15

get_option ((IN)
Options for loading the types. It can be one of two values:

- OCI_TYPEGET_HEADER for only the header to be loaded, or

- OCI_TYPEGET_ALL for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)
Pointer to the pinned type in the object cache.

Comments

This function gets a pointer to the existing type associated with schema/type name. It returns an error if any of the required parameters is null, or if the object type associated with schema/type name does not exist, or if *version_name* does not exist.

> **Note:** Schema and type names are case-sensitive. If they have been created with SQL, you need to use strings all in upper-case, or the program will stop.

This function always makes a round-trip to the server and hence calling this function repeatedly to get the type can significantly drag down performance. To minimize the round-trips, the application may call the function for each type and cache the type objects.

To free the type obtained by this function, `OCIObjectUnpin()` or `OCIObjectPinCountReset()` may be called.

An application can retrieve an array of TDOs by calling `OCITypeArrayByName()`, or `OCITypeArrayByRef()`.

Related Functions

`OCITypeByRef()`, `OCITypeArrayByName()`, `OCITypeArrayByRef()`

OCITypeByRef()

Purpose

Get a type given a reference.

Syntax

```
sword OCITypeByRef ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCIRef    *type_ref,
                     OCIDuration     pin_duration,
                     OCITypeGetOpt   get_option,
                     OCIType         *tdo );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode. See the description of
OCIEnvCreate() and OCIInitialize() for more information.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type_ref (IN)
An **OCIRef** * pointing to the version of the type descriptor object to obtain.

pin_duration (IN)
Pin duration until the end of current transaction for the type to retrieve. See oro.h
for a description of each option.

get_option (IN)
Options for loading the type. It can be one of two values:

- OCI_TYPEGET_HEADER - for only the header to be loaded, or

- OCI_TYPEGET_ALL - for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)
Pointer to the pinned type in the object cache.

Comments

OCITypeByRef() returns an error if any of the required parameters is null.

Related Functions

OCITypeByName(), OCITypeArrayByName(), OCITypeArrayByRef()

18

OCI Datatype Mapping and Manipulation Functions

This chapter describes the OCI datatype mapping and manipulation functions, which is Oracle's external C Language interface to Oracle predefined types. The following sections are included in this chapter:

- Introduction to Datatype Mapping and Manipulation Functions

- OCI Collection and Iterator Functions

- OCI Date, Datetime, and Interval Functions

- OCI Number Functions

- OCI Raw Functions

- OCI Ref Functions

- OCI String Functions

- OCI Table Functions

Introduction to Datatype Mapping and Manipulation Functions

This chapter describes the OCI datatype mapping and manipulation functions in detail.

> **See Also:** For more information about the functions listed in this chapter, refer to Chapter 11, "Object-Relational Datatypes"

The Function Syntax

The entries for each function contain the following information:

Purpose

A brief statement of the purpose of the function.

Syntax

The function declaration.

Comments

Detailed information about the function if available. This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below:

Mode	Description
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.

Returns

A description of what value is returned by the function if the function returns something other than the standard return codes listed in Table 18–1, "Function Return Values" on page 18-3.

Related Functions

A list of related functions.

Datatype Mapping and Manipulation Function Return Values

The OCI datatype mapping and manipulation functions typically return one of the following values:

Table 18–1 Function Return Values

Return Value	Meaning
OCI_SUCCESS	The operation succeeded
OCI_ERROR	The operation failed. The specific error can be retrieved by calling OCIErrorGet() on the error handle passed to the function.
OCI_INVALID_HANDLE	The environment or error handle passed to the function is null.

Function-specific return information follows the description of each function in this chapter.

> **See Also:**   For more information about return codes and error handling, see the section "Error Handling" on page 2-31

Functions Returning Other Values

Some functions return values other than those listed in Table 18–1. When using these function be sure to take into account that they return a value directly from the function call, rather than through an OUT parameter.

- OCICollMax()
- OCIRawPtr()
- OCIRawSize()
- OCIRefHexSize()
- OCIRefIsEqual()
- OCIRefIsNull()
- OCIStringPtr()
- OCIStringSize()

Server Roundtrips for Datatype Mapping and Manipulation Functions

For a table showing the number of server roundtrips required for individual OCI datatype mapping and manipulation functions, refer to Appendix C, "OCI Function Server Round-trips".

Examples

For more information about these functions, including some code examples, refer to Chapter 11, "Object-Relational Datatypes".

OCI Collection and Iterator Functions

This section describes the Collection and Iterator functions.

Table 18–2   Collection and Iterator Functions

Function/Page	Purpose
OCICollAppend() on page 18-6	Collection appends element
OCICollAssign() on page 18-8	Assigns collection
OCICollAssignElem() on page 18-10	Collection assigns element
OCICollGetElem() on page 18-12	Gets pointer to an element
OCICollIsLocator() on page 18-15	Indicates whether a collection is locator-based or not
OCICollMax() on page 18-16	Returns maximum number of elements in collection
OCICollSize() on page 18-17	Gets current size of collection (in number of elements)
OCICollTrim() on page 18-19	Trims elements from the collection
OCIIterCreate() on page 18-20	Creates iterator to scan the varray elements
OCIIterDelete() on page 18-21	Deletes iterator
OCIIterGetCurrent() on page 18-22	Gets current collection element
OCIIterInit() on page 18-23	Initializes iterator to scan the given collection
OCIIterNext() on page 18-24	Gets next collection element
OCIIterPrev() on page 18-26	Gets previous collection element,

OCICollAppend()

Purpose

Appends an element to the end of a collection.

Syntax

```
sword OCICollAppend ( OCIEnv           *env,
                      OCIError         *err,
                      CONST dvoid      *elem,
                      CONST dvoid      *elemind,
                      OCIColl          *coll );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

elem (IN)
Pointer to the element which is appended to the end of the given collection.

elemind (IN) [optional]
Pointer to the element's null indicator information. If (*elemind* == NULL) then the
null indicator information of the appended element will be set to non-null.

coll (IN/OUT)
Updated collection.

Comments

Appending an element is equivalent to increasing the size of the collection by 1
element and updating (deep-copying) the last element's data with the given
element's data. Note that the pointer to the given element *elem* is not saved by this
function, which means that *elem* is strictly an input parameter.

This function returns an error if the current size of the collection is equal to the max
size (upper-bound) of the collection prior to appending the element. This function
also returns an error if any of the input parameters is null.

Related Functions

OCIErrorGet()

OCICollAssign()

Purpose

Assigns (deep-copies) one collection to another.

Syntax

```
sword OCICollAssign ( OCIEnv            *env,
                      OCIError          *err,
                      CONST OCIColl     *rhs,
                      OCIColl           *lhs );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

rhs (IN)
Right-hand side (source) collection to be assigned from.

lhs (OUT)
Left-hand side (target) collection to be assigned to.

Comments

Assigns *rhs* (source) to *lhs* (target). The *lhs* collection may be decreased or
increased depending upon the size of *rhs*. If the *lhs* contains any elements then
the elements will be deleted prior to the assignment. This function performs a deep
copy. The memory for the elements comes from the object cache.

An error is returned if the element types of the *lhs* and *rhs* collections do not
match. Also, an error is returned if the upper-bound of the *lhs* collection is less
than the current number of elements in the *rhs* collection. An error is also returned
if:

- any of the input parameters is null

- there is a type mismatch between the *lhs* and *rhs* collections

- the upper bound of *lhs* collection is less than the current number of elements in the *rhs* collection

Related Functions

OCIErrorGet(), OCICollAssignElem()

OCICollAssignElem()

Purpose

Assigns the given element value *elem* to the element at *coll[index]*.

Syntax

```
sword OCICollAssignElem ( OCIEnv        *env,
                          OCIError      *err,
                          sb4           index,
                          CONST dvoid   *elem,
                          CONST dvoid   *elemind,
                          OCIColl       *coll );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

index (IN)
Index of the element whose is assigned to.

elem (IN)
Element which is assigned from (source element).

elemind (IN) [optional]
Pointer to the element's null indicator information; if (*elemind* == NULL) then the null indicator information of the assigned element will be set to non-null.

coll (IN/OUT)
Collection to be updated.

Comments

If the collection is of type nested table, the element at the given index may not exist, as in the case when an element has been deleted. In this case, the given element is inserted at *index*. Otherwise, the element at *index* is updated with the value of *elem*.

Note that the given element is deep-copied and *elem* is strictly an input parameter.

This function returns an error if any input parameter is null or if the given index is beyond the bounds of the given collection.

Related Functions

OCIErrorGet(), OCICollAssign()

OCICollGetElem()

Purpose

Gets a pointer to the element at the given index.

Syntax

```
sword OCICollGetElem ( OCIEnv          *env,
                       OCIError        *err,
                       CONST OCIColl   *coll,
                       sb4             index,
                       boolean         *exists,
                       dvoid           **elem,
                       dvoid           **elemind );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

coll (IN)
Pointer to the element in this collection is returned.

index (IN)
Index of the element whose pointer is returned.

exists (OUT)
Set to FALSE if the element at the specified index does not exist; otherwise, set to
TRUE.

elem (OUT)
Address of the desired element is returned.

elemind (OUT) [optional]
Address of the null indicator information is returned. If (*elemind* == NULL), then
the null indicator information will not be returned.

Comments

Gets the address of the element at the given position. Optionally this function also returns the address of the element's null indicator information.

The following table describes for each collection element type what the corresponding element pointer type is. The element pointer is returned with the `elem` parameter of `OCICollGetElem()`.

Element Type	*elem is set to
Oracle Number (OCINumber)	OCINumber*
Date (OCIDate)	OCIDate*
Datetime (OCIDateTime)	OCIDateTime*
Interval (OCIInterval)	OCIInterval*
Variable-length string (OCIString*)	OCIString**
Variable-length raw (OCIRaw*)	OCIRaw**
object reference (OCIRef*)	OCIRef**
lob locator (OCILobLocator*)	OCILobLocator**
object type (such as person)	person*

The element pointer returned by `OCICollGetElem()` is in a form such that it can not only be used to access the element data but also is in a form that can be used as the target (left-hand-side) of an assignment statement.

For example, assume the user is iterating over the elements of a collection whose element type is object reference (**OCIRef***). A call to `OCICollGetElem()` returns pointer to a reference handle (**OCIRef****). After getting, the pointer to the collection element, the user may wish to modify it by assigning a new reference.

This can be accomplished via the ref assignment function as follows:

```
sword OCIRefAssign( OCIEnv      *env,
                    OCIError    *err,
                    CONST OCIRef *source,
                    OCIRef       **target );
```

Note that the `target` parameter of `OCIRefAssign()` is of type **OCIRef****. Hence `OCICollGetElem()` returns **OCIRef****. If `*target` equals NULL, a new REF will be allocated by `OCIRefAssign()` and returned via the `target` parameter.

Similarly, if the collection element was of type string (**OCIString***),
`OCICollGetElem()` returns pointer to string handle (i.e. **OCIString****). If a new
string is assigned, via `OCIStringAssign()` or `OCIStringAssignText()` the
type of the target must be **OCIString ****.

If the collection element is of type Oracle number, `OCICollGetElem()` returns
OCINumber*. The prototype of `OCINumberAssign()` is:

```
sword OCINumberAssign(OCIError      *err,
                      CONST OCINumber *from,
                      OCINumber       *to);
```

This function returns an error if any of the input parameters is null.

Related Functions

OCIErrorGet(), OCICollAssignElem()

OCICollIsLocator()

Purpose

Indicates whether a collection is locator-based or not.

Syntax

```
sword OCICollIsLocator ( OCIEnv *env,
                         OCIError *err,
                         CONST OCIColl *coll,
                         boolean *result );
```

Parameters

env (IN)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

coll (IN)
A collection item.

result (OUT)
Returns TRUE if the collection item is locator-based, FALSE otherwise.

Comments

This function tests to see whether or not a collection is locator-based. Returns TRUE
in the *result* parameter if the collection item is locator-based, otherwise it returns
FALSE.

Related Functions

OCIErrorGet()

OCICollMax()

Purpose

Gets the maximum size in number of elements of the given collection.

Syntax

```
sb4 OCICollMax ( OCIEnv        *env,
                 CONST OCIColl *coll );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

coll (IN)
Collection whose number of elements is returned. *coll* must point to a valid collection descriptor.

Comments

Returns the maximum number of elements that the given collection can hold. A value of zero indicates that the collection has no upper bound.

Returns

The upper bound of the given collection.

Related Functions

OCIErrorGet(), OCICollSize()

OCICollSize()

Purpose

Gets the current size in number of elements of the given collection.

Syntax

```
sword OCICollSize ( OCIEnv        *env,
                    OCIError      *err,
                    CONST OCIColl *coll
                    sb4           *size );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

coll (IN)
Collection whose number of elements is returned. Must point to a valid collection
descriptor.

size (OUT)
Current number of elements in the collection.

Comments

Returns the current number of elements in the given collection. For the case of
nested table, this count will not be decremented upon deleting elements. So, this
count includes any *holes* created by deleting elements. A trim operation
(OCICollTrim()) will decrement the count by the number of trimmed elements.
To get the count minus the deleted elements use OCITableSize().

The following pseudocode shows some examples:

```
OCICollSize(...);
// assume 'size' returned is equal to 5
OCITableDelete(...); // delete one element
OCICollSize(...);
// 'size' returned is still 5
```

To get the count minus the deleted elements use `OCITableSize()`. Continuing the above example:

```
OCITableSize(...)
// 'size' returned is equal to 4
```

A trim operation (`OCICollTrim()`) decrements the count by the number of trimmed elements. Continuing the above example:

```
OCICollTrim(..,1..); // trim one element
OCICollSize(...);
// 'size' returned is equal to 4
```

This function returns an error if an error occurs during the loading of the collection into object cache or if any of the input parameters is null.

Related Functions

OCIErrorGet(), OCICollMax()

OCICollTrim()

Purpose

Trims the given number of elements from the end of the collection.

Syntax

```
sword OCICollTrim ( OCIEnv        *env,
                    OCIError      *err,
                    sb4           trim_num,
                    OCIColl       *coll );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

trim_num (IN)
Number of elements to trim.

coll (IN/OUT)
This function removes (frees) *trim_num* elements from the end of *coll*.

Comments

The elements are removed from the end of the collection. An error is returned if *trim_num* is greater than the current size of the collection.

Related Functions

OCIErrorGet(), OCICollSize()

OCIIterCreate()

Purpose

Creates an iterator to scan the elements or the collection.

Syntax

```
sword OCIIterCreate ( OCIEnv              *env,
                      OCIError            *err,
                      CONST OCIColl       *coll,
                      OCIIter             **itr );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

coll (IN)

Collection which will be scanned. For this release, valid collection types include
varrays and nested tables.

itr (OUT)

Address to the allocated collection iterator is returned by this function.

Comments

The iterator is created in the object cache. The iterator is initialized to point to the
beginning of the collection.

If OCIIterNext() is called immediately after creating the iterator then the first
element of the collection is returned. If OCIIterPrev() is called immediately after
creating the iterator then a "at beginning of collection" error is returned.

This function returns an error if any of the input parameters is null.

Related Functions

OCIErrorGet(), OCIIterDelete()

OCIIterDelete()

Purpose

Deletes a collection iterator.

Syntax

```
sword OCIIterDelete ( OCIEnv          *env,
                      OCIError        *err,
                      OCIIter         **itr );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

itr (IN/OUT)
The allocated collection iterator which is destroyed and set to null prior to
returning.

Comments

Deletes an iterator which was previously created by a call to OCIIterCreate().

This function returns an error if any of the input parameters is null.

Related Functions

OCIErrorGet(), OCIIterCreate()

OCIIterGetCurrent()

Purpose

Gets a pointer to the current iterator collection element.

Syntax

```
sword OCIIterGetCurrent ( OCIEnv          *env,
                          OCIError        *err,
                          CONST OCIIter   *itr,
                          dvoid           **elem,
                          dvoid           **elemind );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

itr (IN)
Iterator which points to the current element.

elem (OUT)
Address of the element pointed by the iterator is returned.

elemind (OUT) [optional]
Address of the element's null indicator information is returned; if (*elem_ind* ==
null) then the null indicator information will *not* be returned.

Comments

Returns pointer to the current iterator collection element and its corresponding null
information. This function returns an error if any input parameter is null.

Related Functions

OCIErrorGet(), OCIIterNext(), OCIIterPrev()

OCIIterInit()

Purpose

Initializes an iterator to scan a collection.

Syntax

```
sword OCIIterInit ( OCIEnv          *env,
                    OCIError        *err,
                    CONST OCIColl   *coll,
                    OCIIter         *itr );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

coll (IN)

Collection which will be scanned. For Oracle8*i* or later, valid collection types include varrays and nested tables.

itr (IN/OUT)

Pointer to an allocated collection iterator.

Comments

Initializes given iterator to point to the beginning of given collection. Returns an error if any input parameter is null. This function can be used to:

- reset an iterator to point back to the beginning of the collection, or

- reuse an allocated iterator to scan a different collection.

Related Functions

OCIErrorGet()

OCIIterNext()

Purpose

Gets a pointer to the next iterator collection element.

Syntax

```
sword OCIIterNext ( OCIEnv         *env,
                    OCIError       *err,
                    OCIIter        *itr,
                    dvoid          **elem,
                    dvoid          **elemind,
                    boolean        *eoc);
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

itr (IN/OUT)
Iterator is updated to point to the next element.

elem (OUT)
After updating the iterator to point to the next element, address of the element is
returned.

elemind (OUT) [optional]
Address of the element's null indicator information is returned; if (*elem_ind* ==
NULL) then the null indicator information will *not* be returned.

eoc (OUT)
TRUE if iterator is at End of Collection (i.e. next element does not exist); otherwise,
FALSE.

Comments

This function returns a pointer to the next iterator collection element and its
corresponding null information. It also updates the iterator to point to the next
element.

If the iterator is pointing to the last element of the collection prior to executing this function, then calling this function will set the *eoc* flag to TRUE. The iterator will be left unchanged in that case.

This function returns an error if any input parameter is null.

Related Functions

OCIErrorGet(), OCIIterGetCurrent(), OCIIterPrev()

OCIIterPrev()

Purpose

Gets a pointer to the previous iterator collection element.

Syntax

```
sword OCIIterPrev ( OCIEnv          *env,
                    OCIError        *err,
                    OCIIter         *itr,
                    dvoid           **elem,
                    dvoid           **elemind,
                    boolean         *boc );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

itr (IN/OUT)
Iterator which is updated to point to the previous element.

elem (OUT)
Address of the previous element; returned after the iterator is updated to point to it.

elemind (OUT) [optional]
Address of the element's null indicator; if (*elemind* == NULL) then the null
indicator will *not* be returned.

boc (OUT)
TRUE if iterator is at beginning of collection (i.e. previous element does not exist);
otherwise, FALSE.

Comments

This function returns a pointer to the previous iterator collection element and its
corresponding null information. The iterator is updated to point to the previous
element.

If the iterator is pointing to the first element of the collection prior to executing this function, then calling this function will set *boc* to TRUE. The iterator is left unchanged in that case.

This function returns an error if any input parameter is null.

Related Functions

OCIErrorGet(), OCIIterGetCurrent(), OCIIterNext()

OCI Date, Datetime, and Interval Functions

This section describes the OCI Date and Interval functions.

Table 18–3   Date Functions

Function/Page	Purpose
OCIDateAddDays() on page 18-31	Adds or subtracts days
OCIDateAddMonths() on page 18-32	Adds or subtracts months
OCIDateAssign() on page 18-33	Assigns date
OCIDateCheck() on page 18-34	Checks if the given date is valid
OCIDateCompare() on page 18-36	Compares dates
OCIDateDaysBetween() on page 18-37	Gets number of days between two dates
OCIDateFromText() on page 18-38	Converts string to date
OCIDateGetDate() on page 18-40	Gets the date portion of a date
OCIDateGetTime() on page 18-41	Gets the time portion of a date
OCIDateLastDay() on page 18-42	Gets date of last day of month
OCIDateNextDay() on page 18-43	Gets date of next day
OCIDateSetDate() on page 18-44	Sets the date portion of a date
OCIDateSetTime() on page 18-45	Sets the time portion of a date
OCIDateSysDate() on page 18-46	Gets current system date and time
OCIDateToText() on page 18-47	Converts date to string
OCIDateTimeAssign() on page 18-49	Performs datetime assignment
OCIDateTimeCheck() on page 18-50	Checks if the given date is valid
OCIDateTimeCompare() on page 18-52	Compares two datetime values
OCIDateTimeConstruct() on page 18-53	Constructs a datetime descriptor
OCIDateTimeConvert() on page 18-55	Converts one datetime type to another
OCIDateTimeFromArray() on page 18-56	Converts an array of size OCI_DT_ARRAYLEN to an **OCIDateTime** descriptor
OCIDateTimeFromText() on page 18-58	Converts the given string to Oracle datetime type in the **OCIDateTime** descriptor, according to the specified format
OCIDateTimeGetDate() on page 18-60	Gets the date (year, month, day) portion of a datetime value

Table 18–3   Date Functions

Function/Page	Purpose
OCIDateTimeGetTime() on page 18-61	Gets the time (hour, min, second, fractional second) out of a datetime value
OCIDateTimeGetTimeZoneName() on page 18-63	Gets the time zone name portion of a datetime value
OCIDateTimeGetTimeZoneOffset() on page 18-64	Gets the time zone (hour, minute) portion of a datetime value
OCIDateTimeIntervalAdd() on page 18-65	Adds an interval to a datetime to produce a resulting datetime
OCIDateTimeIntervalSub() on page 18-66	Subtracts an interval from a datetime and stores the result in a datetime
OCIDateTimeSubtract() on page 18-67	Takes two datetimes as input and stores their difference in an interval
OCIDateTimeSysTimeStamp() on page 18-68	Gets the system current date and time as a timestamp with time zone
OCIDateTimeToArray() on page 18-69	Converts a **OCIDateTime** descriptor to an array
OCIDateTimeToText() on page 18-71	Converts the given date to a string according to the specified format
OCIDateZoneToZone() on page 18-73	Converts date from one time zone to another zone
OCIIntervalAdd() on page 18-75	Adds two intervals to produce a resulting interval
OCIIntervalAssign() on page 18-76	Copies one interval to another
OCIIntervalCheck() on page 18-77	Checks the validity of an interval
OCIIntervalCompare() on page 18-79	Compares two intervals
OCIIntervalDivide() on page 18-81	Divides an interval by an Oracle Number to produce an interval
OCIIntervalFromNumber() on page 18-82	Converts an Oracle Number to an interval
OCIIntervalFromText() on page 18-83	Given an interval string, returns the interval represented by the string
OCIIntervalFromTZ() on page 18-85	Returns an OCI_DTYPE_INTERVAL_DS.
OCIIntervalGetDaySecond() on page 18-86	Gets values of day and second from an interval
OCIIntervalGetYearMonth() on page 18-88	Gets year and month from an interval

Table 18–3   Date Functions

Function/Page	Purpose
OCIIntervalMultiply() on page 18-89	Multiplies an interval by an Oracle Number to produce an interval
OCIIntervalSetDaySecond() on page 18-90	Sets day and second in an interval
OCIIntervalSetYearMonth() on page 18-92	Sets year and month in an interval
OCIIntervalSubtract() on page 18-93	Subtracts two intervals and stores the result in an interval
OCIIntervalToNumber() on page 18-94	Converts an interval to an Oracle Number
OCIIntervalToText() on page 18-95	Given an interval, produces a string representing the interval

OCIDateAddDays()

Purpose

Adds or subtracts days from a given date.

Syntax

```
sword OCIDateAddDays ( OCIError          *err,
                       CONST OCIDate     *date,
                       sb4               num_days,
                       OCIDate           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date (IN)
The given date from which to add or subtract.

num_days (IN)
Number of days to be added or subtracted. A negative value is subtracted.

result (IN/OUT)
Result of adding days to, or subtracting days from, *date*.

Comments

This function returns and error if an invalid date is passed to it.

Related Functions

OCIErrorGet(), OCIDateAddMonths()

OCIDateAddMonths()

Purpose

Adds or subtracts months from a given date.

Syntax

```
sword OCIDateAddMonths ( OCIError          *err,
                         CONST OCIDate     *date,
                         sb4               num_months,
                         OCIDate           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date (IN)
The given date from which to add or subtract.

num_months (IN)
Number of months to be added or subtracted. A negative value is subtracted.

result (IN/OUT)
Result of adding days to, or subtracting days from, *date*.

Comments

If the input *date* is the last day of a month, then the appropriate adjustments are made to ensure that the output date is also the last day of the month. For example, Feb. 28 + 1 month = March 31, and November 30 - 3 months = August 31. Otherwise the *result* date has the same day component as *date*.

This function returns an error if invalid date is passed to it.

Related Functions

OCIErrorGet(), OCIDateAddDays()

OCIDateAssign()

Purpose

Performs a date assignment.

Syntax

```
sword OCIDateAssign ( OCIError        *err,
                      CONST OCIDate   *from,
                      OCIDate         *to );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

from (IN)
Date to be assigned.

to (OUT)
Target of assignment.

Comments

This function assigns a value from one **OCIDate** variable to another.

Related Functions

OCIErrorGet(), OCIDateCheck()

OCIDateCheck()

Purpose

Checks if the given date is valid.

Syntax

```
sword OCIDateCheck ( OCIError        *err,
                     CONST OCIDate   *date,
                     uword           *valid );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

date (IN)
Date to be checked

valid (OUT)
Returns zero for a valid date. Otherwise, the ORed combination of all error bits
specified as follows:

Macro Name	Bit Number	Error
OCI_DATE_INVALID_DAY	0x1	Bad day
OCI_DATE_DAY_BELOW_VALID	0x2	Bad day low/high bit (1=low)
OCI_DATE_INVALID_MONTH	0x4	Bad month
OCI_DATE_MONTH_BELOW_VALID	0x8	Bad month low/high bit (1=low)
OCI_DATE_INVALID_YEAR	0x10	Bad year
OCI_DATE_YEAR_BELOW_VALID	0x20	Bad year low/high bit (1=low)
OCI_DATE_INVALID_HOUR	0x40	Bad hour
OCI_DATE_HOUR_BELOW_VALID	0x80	Bad hour low/high bit (1=low)
OCI_DATE_INVALID_MINUTE	0x100	Bad minute
OCI_DATE_MINUTE_BELOW_VALID	0x200	Bad minute low/high bit (1=low)
OCI_DATE_INVALID_SECOND	0x400	Bad second
OCI_DATE_SECOND_BELOW_VALID	0x800	Bad second low/high bit (1=low)

Macro Name	Bit Number	Error
OCI_DATE_DAY_MISSING_FROM_1582	0x1000	Day is one of those missing from 1582
OCI_DATE_YEAR_ZERO	0x2000	Year may not equal zero
OCI_DATE_INVALID_FORMAT	0x8000	Bad date format input

For example, if the date passed in was 2/0/1990 25:61:10 in (month/day/year hours:minutes:seconds format), the error returned would be:

```
OCI_DATE_INVALID_DAY | OCI_DATE_DAY_BELOW_VALID | OCI_DATE_INVALID_HOUR |
    OCI_DATE_INVALID_MINUTE.
```

Comments

This function returns an error if *date* or *valid* pointer is null.

Related Functions

OCIErrorGet(), OCIDateCompare()

OCIDateCompare()

Purpose

Compares two dates.

Syntax

```
sword OCIDateCompare ( OCIError        *err,
                       CONST OCIDate   *date1,
                       CONST OCIDate   *date2,
                       sword           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date1, date2 (IN)
Dates to be compared.

result (OUT)
Comparison result:

Comparison result	Output in *result* parameter
date1 < date2	-1
date1 = date2	0
date1 > date2	1

Comments

This function returns and error if an invalid date is passed to it.

Related Functions

OCIErrorGet(), OCIDateCheck()

OCIDateDaysBetween()

Purpose

Gets the number of days between two dates.

Syntax

```
sword OCIDateDaysBetween ( OCIError          *err,
                           CONST OCIDate     *date1,
                           CONST OCIDate     *date2,
                           sb4               *num_days );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date1 (IN)
Input date.

date2 (IN)
Input date.

num_days (OUT)
Number of days between *date1* and *date2*.

Comments

When the number of days between *date1* and *date2* is computed, the time is
ignored.

This function returns an error if invalid date is passed to it.

Related Functions

OCIErrorGet(), OCIDateCheck()

OCIDateFromText()

Purpose

Converts a character string to a date type according to the specified format.

Syntax

```
sword OCIDateFromText ( OCIError          *err,
                        CONST text        *date_str,
                        ub4               d_str_length,
                        CONST text        *fmt,
                        ub1               fmt_length,
                        CONST text        *lang_name,
                        ub4               lang_length,
                        OCIDate           *date );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

date_str (IN)
Input string to be converted to Oracle date.

d_str_length (IN)
Size of the input string, if the length is -1 then `date_str` is treated as a null-terminated string.

fmt (IN)
Conversion format. If `fmt` is a null pointer, then the string is expected to be in 'DD-MON-YY' format.

fmt_length (IN)
Length of the `fmt` parameter.

lang_name (IN)
Language in which the names and abbreviations of days and months are specified. If `lang_name` is a null string, `(text *) 0`, then the default language of the session is used.

lang_length (IN)
Length of the `lang_name` parameter.

date (OUT)
Given string converted to date.

Comments

Refer to the `TO_DATE` conversion function described in Chapter 3 of the *Oracle8i SQL Reference* for a description of format and multilingual arguments.

This function returns an error if it receives an invalid format, language, or input string.

Related Functions

`OCIErrorGet()`, `OCIDateToText()`

OCIDateGetDate()

Purpose

Get the year, month, and day stored in an Oracle date.

Syntax

```
void OCIDateGetDate ( CONST OCIDate    *date,
                      sb2              *year,
                      ub1              *month,
                      ub1              *day );
```

Parameters

date (IN)
Oracle date whose year, month, day data is retrieved.

year (OUT)
Year value returned.

month (OUT)
Month value returned.

day (OUT)
Day value returned.

Comments

None.

Related Functions

OCIDateSetDate(), OCIDateGetTime()

OCIDateGetTime()

Purpose

Gets the time stored in an Oracle date.

Syntax

```
void OCIDateGetTime ( CONST OCIDate    *date,
                      ub1              *hour,
                      ub1              *min,
                      ub1              *sec );
```

Parameters

date (IN)
Oracle date whose time data is retrieved.

hour (OUT)
Hour value returned.

min (OUT)
Minute value returned.

sec (OUT)
Second value returned.

Comments

Returns the time information returned in the form: hour, minute and seconds.

Related Functions

OCIDateSetTime(), OCIDateGetDate()

OCIDateLastDay()

Purpose

Gets the date of the last day of the month in a specified date.

Syntax

```
sword OCIDateLastDay ( OCIError        *err,
                       CONST OCIDate    *date,
                       OCIDate          *last_day );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date (IN)
Input date.

last_day (OUT)
Last day of the month in *date*.

Comments

This function returns an error if invalid date is passed to it.

Related Functions

OCIErrorGet(), OCIDateGetDate()

OCIDateNextDay()

Purpose

Gets the date of next day of the week, after a given date.

Syntax

```
sword OCIDateNextDay ( OCIError        *err,
                       CONST OCIDate   *date,
                       CONST OraText   *day,
                       ub4             day_length,
                       OCIDate         *next_day );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date (IN)
Returned date should be later than this date.

day (IN)
First day of week named by this is returned.

day_length (IN)
Length in bytes of string *day*.

next_day (OUT)
First day of the week named by *day* later than *date*.

Comments

Returns the date of the first day of the week named by *day* that is later than *date*.

Example

Get the date of the next Monday after April 18, 1996 (a Thursday).

```
OCIDateNextDay(&err, '18-APR-96', 'MONDAY', strlen('MONDAY'), &next_day)
```

OCIDateNextDay() returns '22-APR-96'.

This function returns and error if an invalid date or day is passed to it.

Related Functions

OCIErrorGet(), OCIDateGetDate()

OCIDateSetDate()

Purpose

Set the values in an Oracle date.

Syntax

```
void OCIDateSetDate ( OCIDate    *date,
                      sb2        year,
                      ub1        month,
                      ub1        day );
```

Parameters

date (OUT)
Oracle date whose time data is set.

year (IN)
Year value to be set.

month (IN)
Month value to be set.

day (IN)
Day value to be set.

Comments

None.

Related Functions

OCIDateGetDate()

OCIDateSetTime()

Purpose

Sets the time information in an Oracle date.

Syntax

```
void OCIDateSetTime ( OCIDate    *date,
                      ub1        hour,
                      ub1        min,
                      ub1        sec );
```

Parameters

date (OUT)
Oracle date whose time data is set.

hour (IN)
Hour value to be set.

min (IN)
Minute value to be set.

sec (IN)
Second value to be set.

Comments

None.

Related Functions

OCIDateGetTime()

OCIDateSysDate()

Purpose

Gets the current system date and time of the client.

Syntax

```
sword OCIDateSysDate ( OCIError      *err,
                       OCIDate       *sys_date );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sys_date (OUT)
Current system date and time of the client.

Comments

None.

Related Functions

OCIErrorGet()

OCIDateToText()

Purpose

Converts a date type to a character string.

Syntax

```
sword OCIDateToText ( OCIError              *err,
                      CONST OCIDate         *date,
                      CONSTOraText OraText   *fmt,
                      ub1                    fmt_length,
                      CONST OraText         *lang_name,
                      ub4                    lang_length,
                      ub4                   *buf_size,
                      OraText               *buf );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date (IN)
Oracle date to be converted.

fmt (IN)
Conversion format, if NULL, (text *) 0, then the date is converted to a character string in the default date format, DD-MON-YY.

fmt_length (IN)
Length of the *fmt* parameter.

lang_name (IN)
Specifies the language in which names and abbreviations of months and days are returned; default language of session is used if *lang_name* is null ((text *) 0).

lang_length (IN)
Length of the *lang_name* parameter.

buf_size (IN/OUT)
- Size of the buffer (IN);
- Size of the resulting string is returned with this parameter (OUT).

buf (OUT)
Buffer into which the converted string is placed.

Comments

Converts the given date to a string according to the specified format. The converted null-terminated date string is stored in *buf*.

Refer to the TO_DATE conversion function described in Chapter 3 of the *Oracle8i SQL Reference* for a description of format and multilingual arguments.

This function returns an error if the buffer is too small, or if the function is passed an invalid format or unknown language. Overflow also causes an error. For example, converting a value of 10 into format '9' causes an error.

Related Functions

OCIErrorGet(), OCIDateFromText()

OCIDateTimeAssign()

Purpose

Performs datetime assignment.

Syntax

```
sword OCIDateTimeAssign ( dvoid            *hndl,
                          OCIError         *err,
                          CONST OCIDateTime *from,
                          OCIDateTime      *to );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle. If a user session handle is passed, the conversion takes place in the session's NLS_LANGUAGE and the session's NLS_CALENDAR, otherwise the default is used.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

from (IN)
Source (rhs) datetime to be assigned.

to (OUT)
Target (lhs) of assignment.

Comments

This function performs an assignment from the *from* datetime to the *to* datetime for any of the datetime types listed in the description of the *type* parameter.

The *type* of the output is the same as that of the input.

Returns

OCI_SUCCESS,

OCI_ERROR

Related Functions

OCIDateTimeCheck(), OCIDateTimeConstruct()

OCIDateTimeCheck()

Purpose

Checks if the given date is valid.

Syntax

```
sword OCIDateTimeCheck ( dvoid              *hndl,
                         OCIError           *err,
                         CONST OCIDateTime  *date,
                         ub4                *valid );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle. If a user session handle is passed, the conversion takes place in the session's NLS_LANGUAGE and the session's NLS_CALENDAR, otherwise the default is used.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

date (IN)
The date to be checked.

valid (OUT)
Returns zero for a valid date, otherwise it returns the ORed combination of all the error bits specified below:

Macro Name	Bit Number	Error
OCI_DT_INVALID_DAY	0x1	Bad day
OCI_DT_DAY_BELOW_VALID	0x2	Bad day low/high bit (1=low)
OCI_DT_INVALID_MONTH	0x4	Bad month
OCI_DT_MONTH_BELOW_VALID	0x8	Bad month low/high bit (1=low)
OCI_DT_INVALID_YEAR	0x10	Bad year
OCI_DT_YEAR_BELOW_VALID	0x20	Bad year low/high bit (1=low)
OCI_DT_INVALID_HOUR	0x40	Bad hour
OCI_DT_HOUR_BELOW_VALID	0x80	Bad hour low/high bit (1=low)

Macro Name	Bit Number	Error
OCI_DT_INVALID_MINUTE	0x100	Bad minute
OCI_DT_MINUTE_BELOW_VALID	0x200	Bad minute low/high bit (1=low)
OCI_DT_INVALID_SECOND	0x400	Bad second
OCI_DT_SECOND_BELOW_VALID	0x800	Bad second low/high bit (1=low)
OCI_DT_DAY_MISSING_FROM_1582	0x1000	Day is one of those missing from 1582
OCI_DT_YEAR_ZERO	0x2000	Year may not equal zero
OCI_DT_INVALID_TIMEZONE	0x4000	Bad time zone
OCI_DT_INVALID_FORMAT	0x8000	Bad date format input

So, for example, if the date passed in was 2/0/1990 25:61:10 in (month/day/year hours:minutes:seconds format), the error returned would be:

```
OCI_DT_INVALID_DAY | OCI_DT_DAY_BELOW_VALID |
    OCI_DT_INVALID_HOUR | OCI_DT_INVALID_MINUTE.
```

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if err is a null pointer,

OCI_ERROR, if *date* or *valid* is a null pointer.

Related Functions

OCIDateTimeAssign()

OCIDateTimeCompare()

Purpose

Compares two datetime values.

Syntax

```
sword OCIDateTimeCompare ( dvoid            *hndl,
                           OCIError         *err,
                           CONST OCIDateTime *date1,
                           CONST OCIDateTime *date2,
                           sword            *result );
```

Parameters

hndl (IN/OUT)

The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date1, date2 (IN)
Dates to be compared.

result (OUT)
Comparison result:

Comparison result	Output in *result* parameter
date1 < date2	-1
date1 = date2	0
date1 > date2	1

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if err is a null pointer,

OCI_ERROR, if an invalid date is used, or if the input date arguments are not of
mutually comparable types.

Related Functions

OCIDateTimeConstruct()

OCIDateTimeConstruct()

Purpose

Constructs a datetime descriptor.

Syntax

```
sword OCIDateTimeConstruct ( dvoid       *hndl,
                             OCIError    *err,
                             OCIDateTime *datetime,
                             sb2         year,
                             ub1         month,
                             ub1         day,
                             ub1         hour,
                             ub1         min,
                             ub1         sec,
                             ub4         fsec,
                             OraText     *timezone,
                             size_t      timezone_length );
```

Parameters

hndl (IN)

The OCI user session handle or environment handle.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)

Pointer to an **OCIDateTime** descriptor.

year (IN)

Year value.

month (IN)

Month value.

day (IN)

Day value.

hour (IN)

Hour value.

min (IN)

Minute value.

sec (IN)
Second value

fsec (IN)
Fractional second value.

timezone (IN)
Time zone string.

timezone_length (IN)
Length of the time zone string.

Comments

The type of the datetime is the type of the **OCIDateTime** descriptor. Only the relevant fields based on the type are used. For types with time zone, the date and time fields are assumed to be in the local time of the specified time zone.

If time zone is not specified, then session default time zone is assumed.

Returns

OCI_SUCCESS,

OCI_ERROR, if datetime is not valid.

Related Functions

OCIDateTimeAssign(), OCIDateTimeConvert()

OCIDateTimeConvert()

Purpose

Converts one datetime type to another.

Syntax

```
sword OCIDateTimeConvert ( dvoid      *hndl,
                           OCIError   *err,
                           OCIDateTime *indate,
                           OCIDateTime *outdate );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

indate (IN)
A pointer to the input date.

outdate (OUT)
A pointer to the output datetime.

Comments

This function converts one datetime type to another. The result type is the type of
the *outdate* descriptor. The session default time zone (ORA_SDTZ) is used when
converting a datetime without time zone to one with time zone.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE if *err* is null,

OCI_ERROR, if the conversion is not possible with the given input values.

Related Functions

OCIDateTimeCheck()

OCIDateTimeFromArray()

Purpose

Converts an array of size OCI_DT_ARRAYLEN to an **OCIDateTime** descriptor.

Syntax

```
sword OCIDateTimeFromArray ( dvoid           *hndl,
                             OCIError         *err,
                             CONST ub1        *inarray,
                             ub4              *len
                             ub1              type,
                             OCIDateTime      *datetime,
                             CONST OCIInterval *reftz,
                             ub1              fsprec );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

inarray(IN)
Array of ub1 of size OCI_DT_ARRAYLEN, containing the date.

len (IN)
Length of *inarray*.

type (IN)
Type of the resulting datetime. The array will be converted to the specific SQLT type.

datetime (OUT)
Pointer to an **OCIDateTime** descriptor.

reftz (IN)
Descriptor for **OCIInterval** used as a reference when converting a SQLT_TIMESTAMP_LTZ type.

fsprec (IN)
Fractional second precision of the resulting datetime.

Returns

OCI_SUCCESS,

OCI_ERROR if *type* is invalid.

Related Functions

OCIDateTimeFromText(),OCIDateTimeToArray().

OCIDateTimeFromText()

Purpose

Converts the given string to Oracle datetime type in the **OCIDateTime** descriptor, according to the specified format.

Syntax

```
sword OCIDateTimeFromText ( dvoid            *hndl,
                            OCIError         *err,
                            CONST OraText    *date_str,
                            size_t           dstr_length,
                            CONST OraText    *fmt,
                            ub1              fmt_length,
                            CONST OraText    *lang_name,
                            size_t           lang_length,
                            OCIDateTime      *datetime );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle. If a user session handle is passed, the conversion takes place in the session's NLS_LANGUAGE and the session's NLS_CALENDAR, otherwise the default is used.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date_str (IN)
The input string to be converted to an Oracle datetime.

dstr_length (IN)
The size of the input string. If the length is -1 then *date_str* is treated as a null-terminated string.

fmt (IN)
The conversion format. If *fmt* is a null pointer, then the string is expected to be in the default format for the datetime type.

fmt_length (IN)
The length of the *fmt* parameter.

lang_name (IN)
Specifies the language in which the names and abbreviations of months and days are specified. The default language of the session is used if *lang_name* is null (*lang_name* = (text *)0).

lang_length (IN)
The length of the *lang_name* parameter.

datetime (OUT)
The given string converted to a date.

Comments

Refer to the description of the TO_DATE conversion function in the Oracle8*i* or later SQL Reference for a description of the format argument.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE if *err* is null,

OCI_ERROR, if any of the following is true:

- An invalid format is used.

- An unknown language is used.

- An invalid input string is used.

Related Functions

OCIDateTimeToText(), OCIDateTimeFromArray().

OCIDateTimeGetDate()

Purpose

Gets the date (year, month, day) portion of a datetime value.

Syntax

```
void OCIDateTimeGetDate ( dvoid             *hndl,
                          OCIError          *err,
                          CONST OCIDateTime *datetime,
                          sb2               *year,
                          ub1               *month,
                          ub1               *day );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)
Pointer to an **OCIDateTime** descriptor from which date information is retrieved.

year (OUT)
month (OUT)
day (OUT)
The retrieved year, month, and day values.

Comments

This function gets the date (year, month, day) portion of a datetime value.

Returns

OCI_SUCCESS,

OCI_ERROR if the input type is SQLT_TIME or OCI_TIME_TZ.

Related Functions

OCIDateTimeGetTime()

OCIDateTimeGetTime()

Purpose

Gets the time (hour, min, second, fractional second) out of a datetime value.

Syntax

```
void OCIDateTimeGetTime ( dvoid       *hndl,
                          OCIError    *err,
                          OCIDateTime *datetime,
                          ub1         *hour,
                          ub1         *min,
                          ub1         *sec,
                          ub4         *fsec );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)
Pointer to an **OCIDateTime** descriptor from which time information will be
retrieved.

hour (OUT)
The retrieved hour value.

min (OUT)
The retrieved minute value.

sec (OUT)
The retrieved second value.

fsec (OUT)
The retrieved fractional second value.

Comments

This function gets the time portion (hour, min, second, fractional second) out of a
given datetime value.

This function returns an error if the given datetime does not contain time
information.

Returns

OCI_SUCCESS,

OCI_ERROR if datetime does not contain time (SQLT_DATE).

Related Functions

OCIDateTimeGetDate()

OCIDateTimeGetTimeZoneName()

Purpose

Gets the time zone name portion of a datetime value.

Syntax

```
void OCIDateTimeGetTimeZoneName ( dvoid          *hndl,
                                  OCIError        *err,
                                  CONST OCIDateTime *datetime,
                                  ub1             *buf,
                                  ub4             *buflen, );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)
Pointer to an **OCIDateTime** descriptor.

buf (OUT)
Buffer to store the retrieved time zone name.

buflen (IN/OUT)
The size of the buffer (IN). The size of the name field (OUT)

Comments

This function gets the time portion (hour, min, second, fractional second) out of a
given datetime value.

This function returns an error if the given datetime does not contain time
information.

Returns

OCI_SUCCESS,

OCI_ERROR if datetime does not contain time zone (SQLT_DATE,
SQLT_TIMESTAMP).

Related Functions

OCIDateTimeGetDate(), OCIDateTimeGetTimeZoneOffset().

OCIDateTimeGetTimeZoneOffset()

Purpose

Gets the time zone (hour, minute) portion of a datetime value.

Syntax

```
void OCIDateTimeGetTimeZoneOffset ( dvoid          *hndl,
                                    OCIError        *err,
                                    CONST OCIDateTime  *datetime,
                                    sb1             *hour,
                                    sb1             *min, );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)
Pointer to an **OCIDateTime** descriptor.

hour (OUT)
The retrieved time zone hour value.

min (OUT)
The retrieved time zone minute value.

Comments

This function gets the time zone hour and the time zone minute portion out of a
given datetime value.

This function returns an error if the given datetime does not contain time
information.

Returns

OCI_SUCCESS,

OCI_ERROR if datetime does not contain time zone (SQLT_DATE,
SQLT_TIMESTAMP).

Related Functions

OCIDateTimeGetDate(), OCIDateTimeGetTimeZoneName().

OCIDateTimeIntervalAdd()

Purpose

Adds an interval to a datetime to produce a resulting datetime.

Syntax

```
sword OCIDateTimeIntervalAdd ( dvoid      *hndl,
                               OCIError    *err,
                               OCIDateTime *datetime,
                               OCIInterval *inter,
                               OCIDateTime *outdatetime );
```

Parameters

hndl (IN)
The user session or environment handle. If a session handle is passed, the addition takes place in the session default calendar.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)
Pointer to the input datetime.

inter (IN)
Pointer to the input interval.

outdatetime (OUT)
Pointer to the output datetime. The output datetime will be of same type as the input datetime.

Returns

OCI_SUCCESS if the function completes successfully,

OCI_INVALID_HANDLE, if *err* is a null pointer,

OCI_ERROR, if the resulting date is before Jan 1, -4713, or is after Dec 31, 9999.

Related Functions

OCIDateTimeIntervalSub()

OCIDateTimeIntervalSub()

Purpose

Subtracts an interval from a datetime and stores the result in a datetime.

Syntax

```
sword OCIDateTimeIntervalSub ( dvoid       *hndl,
                               OCIError    *err,
                               OCIDateTime *datetime,
                               OCIInterval *inter,
                               OCIDateTime *outdatetime );
```

Parameters

hndl (IN)
The user session or environment handle. If a session handle is passed, the subtraction takes place in the session default calendar. The interval is assumed to be in the session calendar.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)
Pointer to the input datetime value.

inter (IN)
Pointer to the input interval.

outdatetime (OUT)
Pointer to the output datetime. The output datetime will be of same type as the input datetime.

Returns

OCI_SUCCESS if the function completes successfully,

OCI_INVALID_HANDLE, if *err* is a null pointer,

OCI_ERROR, if the resulting date is before Jan 1, -4713, or is after Dec 31, 9999.

Related Functions

OCIDateTimeIntervalAdd()

OCIDateTimeSubtract()

Purpose

Takes two datetimes as input and stores their difference in an interval.

Syntax

```
sword OCIDateTimeSubtract ( dvoid       *hndl,
                            OCIError    *err,
                            OCIDateTime *indate1,
                            OCIDateTime *indate2,
                            OCIInterval *inter );
```

Parameters

hndl (IN)

The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

indate1(IN)
Pointer to the subtrahend.

indate2(IN)
Pointer to the minuend.

inter (OUT)
Pointer to the output interval.

Returns

OCI_SUCCESS if the function completes successfully,

OCI_INVALID_HANDLE if *err* is null pointer,

OCI_ERROR, if the input datetimes are not of comparable types.

Related Functions

OCIDateTimeCompare()

OCIDateTimeSysTimeStamp()

Purpose

Gets the system current date and time as a timestamp with time zone.

Syntax

```
sword OCIDateTimeSysTimeStamp ( dvoid        *hndl,
                                OCIError     *err,
                                OCIDateTime  *sys_date );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sys_date (OUT)
Pointer to the output timestamp.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer.

Related Functions

OCIDateSysDate()

OCIDateTimeToArray()

Purpose

Converts a **OCIDateTime** descriptor to an array.

Syntax

```
sword OCIDateTimeToArray ( dvoid           *hndl,
                           OCIError        *err,
                           CONST OCIDateTime *datetime,
                           CONST OCIInterval *reftz,
                           ub1             *outarray,
                           ub4             *len
                           ub1             fsprec );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

datetime (IN)
Pointer to an **OCIDateTime** descriptor.

reftz (IN)
Descriptor for the **OCIInterval** used as a reference when converting
SQL_TIMESTAMP_LTZ type.

outarray(OUT)
Array of bytes containing the date.

len (OUT)
Length of *outarray*.

fsprec (IN)
Fractional second precision in the array.

Comments

The array is allocated by OCI and its length is returned.

Returns

OCI_SUCCESS,

OCI_ERROR if datetime is invalid.

Related Functions

`OCIDateTimeToText(), OCIDateTimeFromArray()`.

OCIDateTimeToText()

Purpose

Converts the given date to a string according to the specified format.

Syntax

```
sword OCIDateTimeToText ( dvoid              *hndl,
                          OCIError           *err,
                          CONST OCIDateTime  *date,
                          CONST OraText      *fmt,
                          ub1                fmt_length,
                          ub1                fsprec,
                          CONST OraText      *lang_name,
                          size_t             lang_length,
                          ub4                *buf_size,
                          OraText            *buf );
```

Parameters

hndl (IN)
The OCI user session handle or environment handle. If a user session handle is passed, the conversion takes place in the session's NLS_LANGUAGE and the session's NLS_CALENDAR, otherwise the default is used.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

date (IN)
Oracle datetime value to be converted

fmt (IN)
The conversion format. If it is a null string pointer (text*)0, then the date is converted to a character string in the default format for that type.

fmt_length (IN)
The length of the *fmt* parameter.

fsprec (IN)
Specifies the precision in which the fractional seconds is returned.

lang_name (IN)
Specifies the language in which the names and abbreviations of months and days are returned. The default language of the session is used if *lang_name* is null (*lang_name* = (OraText *)0).

lang_length (IN)
The length of the *lang_name* parameter.

buf_size (IN/OUT)
The size of the *buf* buffer (IN).

The size of the resulting string after the conversion (OUT).

buf (OUT)
The buffer into which the converted string is placed.

Comments

Refer to the description of the TO_DATE conversion function in the Oracle8*i* or later SQL Reference for a description of format and multilingual arguments. The converted null-terminated date string is stored in the buffer *buf*.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is null,

OCI_ERROR, if any of the following is true:

- The buffer is too small.

- An invalid format is used.

- An unknown language is used.

- There is an overflow error.

Related Functions

OCIDateTimeFromText()

OCIDateZoneToZone()

Purpose

Converts a date from one time zone to another.

Syntax

```
sword OCIDateZoneToZone ( OCIError          *err,
                          CONST OCIDate     *date1,
                          CONST OraText     *zon1,
                          ub4               zon1_length,
                          CONST OraText     *zon2,
                          ub4               zon2_length,
                          OCIDate           *date2 );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

date1 (IN)
Date to convert.

zon1 (IN)
Zone of input date.

zon1_length (IN)
Length in bytes of `zon1`.

zon2 (IN)
Zone to be converted to.

zon2_length (IN)
Length in bytes of `zon2`.

date2 (OUT)
Converted date (in `zon2`).

Comments

Converts a given date `date1` in time zone `zon1` to a date `date2` in time zone `zon2`. Works only with North American time zones.

For a list of valid zone strings, refer to the description of the NEW_TIME function in Chapter 3 of the *Oracle8i SQL Reference*. Examples of valid zone strings include:

- AST, Atlantic Standard Time

- ADT, Atlantic Daylight Time
- BST, Bering Standard Time
- BDT, Bering Daylight Time

This function returns an error if an invalid date or time zone is passed to it.

Related Functions

OCIErrorGet(), OCIDateCheck()

OCIIntervalAdd()

Purpose

Adds two intervals to produce a resulting interval.

Syntax

```
sword OCIIntervalAdd ( dvoid       *hndl,
                       OCIError     *err,
                       OCIInterval  *addend1,
                       OCIInterval  *addend2,
                       OCIInterval  *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

addend1 (IN)
Interval to be added.

addend2 (IN)
Interval to be added.

result (OUT)
The resulting interval (*addend1* + *addend2*).

Returns

OCI_SUCCESS,

OCI_ERROR, if:

- the two input intervals are not mutually comparable,

- or, the resulting year would go above SB4MAXVAL,

- or, he resulting year would go below SB4MINVAL,

OCI_INVALID_HANDLE, if *err* is a null pointer.

Related Functions

OCIIntervalSubtract()

OCIIntervalAssign()

Purpose

Copies one interval to another.

Syntax

```
void OCIIntervalAssign ( dvoid         *hndl,
                         OCIError       *err,
                         CONST OCIInterval *inpinter,
                         OCIInterval     *outinter );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

inpinter (IN)
Input interval.

outinter (OUT)
Output interval.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer.

Related Functions

OCIIntervalCompare()

OCIIntervalCheck()

Purpose

Checks the validity of an interval.

Syntax

```
sword OCIIntervalCheck ( dvoid              *hndl,
                         OCIError           *err,
                         CONST OCIInterval  *interval,
                         ub4                *valid );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

interval (IN)
Interval to be checked.

valid (OUT)
Zero if the interval is valid, else returns an ORed combination of the following
codes:

Macro Name	Bit Number	Error
OCI_INTER_INVALID_DAY	0x1	Bad day
OCI_INTER_DAY_BELOW_VALID	0x2	Bad day low/high bit (1=low)
OCI_INTER_INVALID_MONTH	0x4	Bad month
OCI_INTER_MONTH_BELOW_VALID	0x8	Bad month low/high bit (1=low)
OCI_INTER_INVALID_YEAR	0x10	Bad year
OCI_INTER_YEAR_BELOW_VALID	0x20	Bad year low/high bit (1=low)
OCI_INTER_INVALID_HOUR	0x40	Bad hour
OCI_INTER_HOUR_BELOW_VALID	0x80	Bad hour low/high bit (1=low)
OCI_INTER_INVALID_MINUTE	0x100	Bad minute
OCI_INTER_MINUTE_BELOW_VALID	0x200	Bad minute low/high bit (1=low)

Macro Name	Bit Number	Error
OCI_INTER_INVALID_SECOND	0x400	Bad second
OCI_INTER_SECOND_BELOW_VALID	0x800	Bad second low/high bit (1=low)
OCI_INTER_INVALID_FRACSEC	0x1000	Bad fractional second
OCI_INTER_FRACSEC_BELOW_VALID	0x2000	Bad fractional second low/high bit (1=low)

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer,

OCI_ERROR, on error.

Related Functions

OCIIntervalCompare()

OCIIntervalCompare()

Purpose

Compares two intervals.

Syntax

```
sword OCIIntervalCompare( dvoid       *hndl,
                          OCIError     *err,
                          OCIInterval  *inter1,
                          OCIInterval  *inter2,
                          sword        *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

inter1 (IN)
Interval to be compared.

inter2 (IN)
Interval to be compared.

result (OUT)
Comparison result:

Comparison result	Output in *result* parameter
inter1 < inter2	-1
inter1 = inter2	0
inter1 > inter2	1

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer,

OCI_ERROR, if the input values are not mutually comparable.

Related Functions

OCIIntervalAssign()

OCIIntervalDivide()

Purpose

Divides an interval by an Oracle Number to produce an interval.

Syntax

```
sword OCIIntervalDivide ( dvoid      *hndl,
                          OCIError    *err,
                          OCIInterval *dividend,
                          OCINumber   *divisor,
                          OCIInterval *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

dividend (IN)
Interval to be divided.

divisor (IN)
Oracle Number dividing *dividend*.

result (OUT)
The resulting interval (*dividend / divisor*).

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer.

Related Functions

OCIIntervalMultiply()

OCIIntervalFromNumber()

Purpose

Converts an Oracle Number to an interval.

Syntax

```
sword OCIIntervalFromNumber ( dvoid          *hndl,
                              OCIError        *err,
                              OCIInterval     *interval,
                              OCINumber       *number );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

interval (OUT)
Interval result.

number (IN)
Oracle number to be converted (in years for YEAR TO MONTH intervals and in
days for DAY TO SECOND intervals).

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer.

Related Functions

OCIIntervalToNumber()

OCIIntervalFromText()

Purpose

Given an interval string, returns the interval represented by the string. The type of the interval is the type of the *result* descriptor.

Syntax

```
sword OCIIntervalFromText ( dvoid          *hndl,
                            OCIError       *err,
                            CONST OraText  *inpstring,
                            size_t         str_len,
                            OCIInterval    *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

inpstring (IN)
Input string.

str_len (IN)
Length of the input string.

result (OUT)
Resultant interval.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer,

OCI_ERROR, if any of the following are true:

- there are too many fields in the literal string
- the year is out of range (-4713 to 9999)
- the month is out of range (1 to 12)
- the day of month is out of range (1 to 28...31)
- the hour is out of range (0 to 23)

- if hour is out of range (0 to 11)

- if minute is out of range (0 to 59)

- if seconds in minute out of range (0 to 59)

- if seconds in day out of range (0 to 86399)

- if the interval is invalid

Related Functions

OCIIntervalToText()

OCIIntervalFromTZ()

Purpose

Returns an OCI_DTYPE_INTERVAL_DS of datatype **OCIInterval** with the region id set (if the region is specified in the input string) and the current absolute offset, or an absolute offset with the region id set to 0.

Syntax

```
sword OCIIntervalFromTZ ( dvoid        *hndl,
                          OCIError      *err,
                          CONST oratext *inpstring,
                          size_t        str_len,
                          OCIInterval   *result ) ;
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

inpstring (IN)
Pointer to the input string.

str_len (IN)
Length of *inpstring*.

result (OUT)
Output interval.

Returns

OCI_SUCCESS, on success,

OCI_INVALID_HANDLE, if err is null,

OCI_ERROR, if there is a bad interval type or time zone errors.

Comments

The input string must be of the form [+/-]TZH:TZM or 'TZR [TZD]'

Related Functions

OCIIntervalFromText()

OCIIntervalGetDaySecond()

Purpose

Gets values of day and second from an interval.

Syntax

```
sword OCIIntervalGetDaySecond (dvoid             *hndl,
                               OCIError           *err,
                               sb4                *dy,
                               sb4                *hr,
                               sb4                *mm,
                               sb4                *ss,
                               sb4                *fsec,
                               CONST OCIInterval  *interval );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

dy (OUT)
Number of days.

hr (OUT)
Number of hours.

mm (OUT)
Number of minutes.

ss (OUT)
Number of seconds.

fsec (OUT)
Number of fractional seconds.

interval (IN)
The input interval.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if `err` is a null pointer,

Related Functions

OCIIntervalSetDaySecond()

OCIIntervalGetYearMonth()

Purpose

Gets year and month from an interval.

Syntax

```
sword OCIIntervalGetYearMonth ( dvoid          *hndl,
                                OCIError        *err,
                                sb4             *yr,
                                sb4             *mnth,
                                CONST OCIInterval  *interval );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

yr (OUT)
Year value.

mnth (OUT)
Month value.

interval (IN)
The input interval.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer.

Related Functions

OCIIntervalSetYearMonth()

OCIIntervalMultiply()

Purpose

Multiplies an interval by an Oracle Number to produce an interval.

Syntax

```
sword OCIIntervalMultiply ( dvoid            *hndl,
                            OCIError         *err,
                            CONST OCIInterval *inter,
                            OCINumber        *nfactor,
                            OCIInterval      *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

inter (IN)
Interval to be multiplied.

nfactor (IN)
Oracle Number to be multiplied.

result (OUT)
The resulting interval (*inter* * *nfactor*).

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer,

OCI_ERROR, if any of the following are true:

- the resulting year would go above SB4MAXVAL

- the resulting year would go below SB4MINVAL

Related Functions

OCIIntervalDivide()

OCIIntervalSetDaySecond()

Purpose

Sets day and second in an interval.

Syntax

```
sword OCIIntervalSetDaySecond ( dvoid          *hndl,
                                OCIError       *err,
                                sb4            dy,
                                sb4            hr,
                                sb4            mm,
                                sb4            ss,
                                sb4            fsec,
                                OCIInterval    *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

dy (IN)
Number of days.

hr (IN)
Number of hours.

mm (IN)
Number of minutes.

ss (IN)
Number of seconds.

fsec (IN)
Number of fractional seconds.

result (OUT)
The resulting interval.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if `err` is a null pointer,

Related Functions

OCIIntervalGetDaySecond()

OCIIntervalSetYearMonth()

Purpose

Sets year and month in an interval.

Syntax

```
sword OCIIntervalSetYearMonth ( dvoid          *hndl,
                                OCIError       *err,
                                sb4            yr,
                                sb4            mnth,
                                OCIInterval    *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

yr (IN)
Year value.

mnth (IN)
Month value.

result (OUT)
The resulting interval.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer,

Related Functions

- the resulting year would go above SB4MAXVAL

- the resulting year would go below SB4MINVAL

Related Functions

OCIIntervalGetYearMonth()

OCIIntervalSubtract()

Purpose

Subtracts two intervals and stores the result in an interval.

Syntax

```
sword OCIIntervalSubtract ( dvoid      *hndl,
                            OCIError    *err,
                            OCIInterval *minuend,
                            OCIInterval *subtrahend,
                            OCIInterval *result );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in err and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

minuend (IN)
The interval to be subtracted from.

subtrahend (IN)
The interval subtracted from $minuend$.

result (OUT)
The resulting interval ($minuend$ - $subtrahend$).

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if err is a null pointer,

OCI_ERROR, if any of the following are true:

- the resulting year would go above SB4MAXVAL

- the resulting year would go below SB4MINVAL

- the two input intervals are not mutually comparable

Related Functions

OCIIntervalAdd()

OCIIntervalToNumber()

Purpose

Converts an interval to an Oracle Number.

Syntax

```
sword OCIIntervalToNumber ( dvoid           *hndl,
                            OCIError        *err,
                            OCIInterval     *interval,
                            OCINumber       *number );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

interval (IN)
Interval to be converted.

number (OUT)
Oracle number result (in years for YEARMONTH interval and in days for DAYSECOND).

Comments

Fractional portions of the date (for instance, minutes and seconds if the unit chosen is hours) are included in the Oracle number produced. Excess precision is truncated.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if *err* is a null pointer.

Related Functions

OCIIntervalFromNumber()

OCIIntervalToText()

Purpose

Given an interval, produces a string representing the interval.

Syntax

```
sword OCIIntervalToText ( dvoid             *hndl,
                          OCIError          *err,
                          CONST OCIInterval *interval,
                          ub1               lfprec,
                          ub1               fsprec,
                          OraText           *buffer,
                          size_t            buflen,
                          size_t            *resultlen );
```

Parameters

hndl (IN)
The OCI user session handle or the environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

interval (IN)
Interval to be converted.

lfprec (IN)
Leading field precision. (The number of digits used to represent the leading field.)

fsprec (IN)
Fractional second precision of the interval (the number of digits used to represent the fractional seconds).

buffer (OUT)
Buffer to hold the result.

buflen (IN)
The length of *buffer*.

resultlen (OUT)
The length of the result placed into *buffer*.

Comments

The interval literal is output as 'year' or '[year-]month' for INTERVAL YEAR TO MONTH intervals and as 'seconds' or 'minutes[:seconds]' or

'hours[:minutes[:seconds]]' or 'days[hours[:minutes[:seconds]]]' for INTERVAL DAY TO SECOND intervals (where optional fields are surrounded by brackets).

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE, if `err` is a null pointer,

OCI_ERROR, if the buffer is not large enough to hold the result.

Related Functions

OCIIntervalFromText()

OCI Number Functions

This section describes the OCI Number functions.

Table 18–4   Number Functions

Function/Page	Purpose
`OCINumberAbs()` on page 18-99	Computes the absolute value
`OCINumberAdd()` on page 18-100	Adds numbers
`OCINumberArcCos()` on page 18-101	Computes the arc cosine
`OCINumberArcSin()` on page 18-102	Computes the arc sine
`OCINumberArcTan()` on page 18-103	Computes the arc tangent
`OCINumberArcTan2()` on page 18-104	Computes the arc tangent of two numbers
`OCINumberAssign()` on page 18-105	Assigns one number to another
`OCINumberCeil()` on page 18-106	Computes the ceiling of number
`OCINumberCmp()` on page 18-107	Compares numbers
`OCINumberCos()` on page 18-108	Computes the cosine
`OCINumberDec()` on page 18-109	Decrements an OCI number
`OCINumberDiv()` on page 18-110	Divides two numbers
`OCINumberExp()` on page 18-111	Raises *e* to the specified Oracle number power
`OCINumberFloor()` on page 18-112	Computes the floor of a number
`OCINumberFromInt()` on page 18-113	Converts an integer to an Oracle number
`OCINumberFromReal()` on page 18-114	Convert a real to an Oracle number
`OCINumberFromText()` on page 18-115	Convert a string to an Oracle number
`OCINumberHypCos()` on page 18-117	Computes the hyperbolic cosine
`OCINumberHypSin()` on page 18-118	Computes the hyperbolic sine
`OCINumberHypTan()` on page 18-119	Computes the hyperbolic tangent
`OCINumberInc()` on page 18-120	Increments an Oracle number
`OCINumberIntPower()` on page 18-121	Raises a given base to an integer power
`OCINumberIsInt()` on page 18-122	Tests if a number is an integer
`OCINumberIsZero()` on page 18-123	Tests if a number is zero
`OCINumberLn()` on page 18-124	Computes the natural logarithm

Table 18–4   Number Functions

Function/Page	Purpose
OCINumberLog() on page 18-125	Computes the logarithm to an arbitrary base
OCINumberMod() on page 18-126	Modulo division
OCINumberMul() on page 18-127	Multiplies numbers
OCINumberNeg() on page 18-128	Negates a number
OCINumberPower() on page 18-129	Exponentiation to base *e*
OCINumberPrec() on page 18-130	Rounds a number to a specified number of decimal places
OCINumberRound() on page 18-131	Rounds an Oracle number to a specified decimal place
OCINumberSetPi() on page 18-132	Initializes a number to Pi
OCINumberSetZero() on page 18-133	Initializes a number to zero
OCINumberShift() on page 18-134	Multiplies by 10, shifting specified number of decimal places
OCINumberSign() on page 18-135	Obtains the sign of an Oracle number
OCINumberSin() on page 18-136	Computes the sine
OCINumberSqrt() on page 18-137	Computes the square root of a number
OCINumberSub() on page 18-138	Subtracts numbers
OCINumberTan() on page 18-139	Computes the tangent
OCINumberToInt() on page 18-140	Converts an Oracle number to an integer
OCINumberToReal() on page 18-141	Converts an Oracle number to a real
OCINumberToText() on page 18-142	Converts an Oracle number to a string
OCINumberTrunc() on page 18-144	Truncates an Oracle number at a specified decimal place

OCINumberAbs()

Purpose

Computes the absolute value of an Oracle number.

Syntax

```
sword OCINumberAbs ( OCIError          *err,
                     CONST OCINumber   *number,
                     OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Input number.

result (OUT)
The absolute value of the input number.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberSign()

OCINumberAdd()

Purpose

Adds two Oracle numbers together.

Syntax

```
sword OCINumberAdd ( OCIError          *err,
                     CONST OCINumber   *number1,
                     CONST OCINumber   *number2,
                     OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1, number2 (IN)
Numbers to be added.

result (OUT)
Result of adding *number1* to *number2*.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberSub()

OCINumberArcCos()

Purpose

Takes the arc cosine in radians of an Oracle number.

Syntax

```
sword OCINumberArcCos ( OCIError        *err,
                        CONST OCINumber  *number,
                        OCINumber        *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the arc cosine.

result (OUT)
Result of the arc cosine in radians.

Comments

This function returns an error if any of the number arguments is null, or if *number*
is less than -1 or if *number* is greater than 1.

Related Functions

OCIErrorGet(), OCINumberCos()

OCINumberArcSin()

Purpose

Takes the arc sine in radians of an Oracle number.

Syntax

```
sword OCINumberArcSin ( OCIError              *err,
                        CONST OCINumber       *number,
                        OCINumber             *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the arc sine.

result (OUT)
Result of the arc sine in radians.

Comments

This function returns an error if any of the number arguments is null, or if *number* is less than -1 or if *number* is greater than 1.

Related Functions

OCIErrorGet(), OCINumberSin()

OCINumberArcTan()

Purpose

Takes the arc tangent in radians of an Oracle number.

Syntax

```
sword OCINumberArcTan ( OCIError          *err,
                        CONST OCINumber   *number,
                        OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the arc tangent.

result (OUT)
Result of the arc tangent in radians.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberTan()

OCINumberArcTan2()

Purpose

Takes the arc tangent of two Oracle numbers.

Syntax

```
sword OCINumberArcTan2 ( OCIError            *err,
                         CONST OCINumber     *number1,
                         CONST OCINumber     *number2,
                         OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1 (IN)
Argument 1 of the arc tangent.

number2 (IN)
Argument 2 of the arc tangent.

result (OUT)
Result of the arc tangent in radians.

Comments

This function returns an error if any of the number arguments is null, or if *number2*
is equal to 0.

Related Functions

OCIErrorGet(), OCINumberTan()

OCINumberAssign()

Purpose

Assigns one Oracle number to another Oracle number.

Syntax

```
sword OCINumberAssign ( OCIError          *err,
                        CONST OCINumber   *from,
                        OCINumber         *to );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

from (IN)
Number to be assigned.

to (OUT)
Number copied into.

Comments

Assigns the number identified by *from* to the number identified by *to*.

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberCmp()

OCINumberCeil()

Purpose

Computes the ceiling value of an Oracle number.

Syntax

```
sword OCINumberCeil ( OCIError            *err,
                      CONST OCINumber     *number,
                      OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Input number.

result (OUT)
Output which will contain the ceiling value of the input number.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberFloor()

OCINumberCmp()

Purpose

Compares two Oracle numbers.

Syntax

```
sword OCINumberCmp ( OCIError        *err,
                     CONST OCINumber  *number1,
                     CONST OCINumber  *number2,
                     sword            *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1, number2 (IN)
Numbers to compare.

result (OUT)
Comparison result:

Comparison result	Output in *result* parameter
number1 < number2	negative
number1 = number2	0
number1 > number2	positive

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberAssign()

OCINumberCos()

Purpose

Computes the cosine in radians of an Oracle number.

Syntax

```
sword OCINumberCos ( OCIError           *err,
                     CONST OCINumber    *number,
                     OCINumber          *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the cosine in radians.

result (OUT)
Result of the cosine.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberArcCos()

OCINumberDec()

Purpose

Decrements an OCINumber.

Syntax

```
OCINumberInc ( OCIError *err,
               OCINumber *number );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN/OUT)
A positive Oracle number to be decremented.

Comments

Decrements an Oracle number in place. It is assumed that the input is an integer between 0 and 100^21-2. If the is input too large, it will be treated as 0 - the result will be an Oracle number 1. If the input is not a positive integer, the result will be unpredictable.

This function returns an error if the input number is null.

Related Functions

OCINumberInc(), OCINumberInc()

OCINumberDiv()

Purpose

Divides two Oracle numbers.

Syntax

```
sword OCINumberDiv ( OCIError              *err,
                     CONST OCINumber       *number1,
                     CONST OCINumber       *number2,
                     OCINumber             *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1 (IN)
Pointer to the numerator.

number2 (IN)
Pointer to the denominator.

result (OUT)
Division result.

Comments

Divides *number1* by *number2* and returns result in *result*.

This function returns an error if:

- any of the number arguments is null

- there is an underflow error

- there is a divide-by-zero error

Related Functions

OCIErrorGet(), OCINumberMul()

OCINumberExp()

Purpose

Raises *e* to the specified Oracle number power.

Syntax

```
sword OCINumberExp ( OCIError            *err,
                     CONST OCINumber     *number,
                     OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
This function raises *e* to this Oracle number power.

result (OUT)
Output of exponentiation.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberLn()

OCINumberFloor()

Purpose

Computes the floor value of an Oracle number.

Syntax

```
sword OCINumberFloor ( OCIError              *err,
                       CONST OCINumber       *number,
                       OCINumber             *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

number (IN)
Input number.

result (OUT)
The floor value of the input number.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberCeil()

OCINumberFromInt()

Purpose

Converts an integer to an Oracle number.

Syntax

```
sword OCINumberFromInt ( OCIError         *err,
                         CONST dvoid      *inum,
                         uword            inum_length,
                         uword            inum_s_flag,
                         OCINumber        *number );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

inum (IN)
Pointer to the integer to convert.

inum_length (IN)
Size of the integer.

inum_s_flag (IN)
Flag that designates the sign of the integer, as follows:

- OCI_NUMBER_UNSIGNED - Unsigned values

- OCI_NUMBER_SIGNED - Signed values

number (OUT)
Given integer converted to Oracle number.

Comments

This is a native type conversion function. It converts any Oracle standard
machine-native integer type, such as **ub4** or **sb2**, to an Oracle number.

This function returns an error if the number is too big to fit into an Oracle number, if
number or *inum* is null, or if an invalid sign flag value is passed in *inum_s_flag*.

Related Functions

OCIErrorGet(), OCINumberToInt()

OCINumberFromReal()

Purpose

Converts a real (floating-point) type to an Oracle number.

Syntax

```
sword OCINumberFromReal ( OCIError        *err,
                          CONST dvoid     *rnum,
                          uword           rnum_length,
                          OCINumber       *number );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

rnum (IN)
Pointer to the floating point number to convert.

rnum_length (IN)
The size of the desired result, which equals sizeof({**float** | **double** | **long
double**}).

number (OUT)
Given float converted to Oracle number.

Comments

This is a native type conversion function. It converts a machine-native floating point
type to an Oracle number.

This function returns an error if *number* or *rnum* is null, or if *rnum_length* equals
zero.

Related Functions

OCIErrorGet(), OCINumberToReal()

OCINumberFromText()

Purpose

Converts character string to Oracle number.

Syntax

```
sword OCINumberFromText ( OCIError        *err,
                          CONST OraText   *str,
                          ub4             str_length,
                          CONST OraText   *fmt,
                          ub4             fmt_length,
                          CONST OraText   *nls_params,
                          ub4             nls_p_length,
                          OCINumber       *number );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

str (IN)
Input string to convert to Oracle number.

str_length (IN)
Size of the input string.

fmt (IN)
Conversion format.

fmt_length (IN)
Length of the *fmt* parameter.

nls_params (IN)
Global Support format specification. If it is the null string (""), then the default parameters for the session is used.

nls_p_length (IN)
Length of the *nls_params* parameter.

number (OUT)
Given string converted to number.

Comments

Converts the given string to a number according to the specified format. Refer to the TO_NUMBER conversion function described in the *Oracle8i SQL Reference* for a description of format and multilingual parameters.

This function returns an error if there is an invalid format, an invalid multibyte format, or an invalid input string, if *number* or *str* is null, or if *str_length* is zero.

Related Functions

OCIErrorGet(), OCINumberToText()

OCINumberHypCos()

Purpose

Computes the hyperbolic cosine of an Oracle number.

Syntax

```
sword OCINumberHypCos ( OCIError            *err,
                        CONST OCINumber     *number,
                        OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the cosine hyperbolic.

result (OUT)
Result of the cosine hyperbolic.

Comments

This function returns an error if any of the number arguments is null.

Caution: An Oracle number overflow causes an unpredictable result value.

Related Functions

OCIErrorGet(), OCINumberHypSin(), OCINumberHypTan()

OCINumberHypSin()

Purpose

Computes the hyperbolic sine of an Oracle number.

Syntax

```
sword OCINumberHypSin ( OCIError              *err,
                        CONST OCINumber       *number,
                        OCINumber             *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the sine hyperbolic.

result (OUT)
Result of the sine hyperbolic.

Comments

This function returns an error if any of the number arguments is null.

Caution: An Oracle number overflow causes an unpredictable
result value.

Related Functions

OCIErrorGet(), OCINumberHypCos(), OCINumberHypTan()

OCINumberHypTan()

Purpose

Computes the hyperbolic tangent of an Oracle number.

Syntax

```
sword OCINumberHypTan ( OCIError            *err,
                        CONST OCINumber     *number,
                        OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the tangent hyperbolic.

result (OUT)
Result of the tangent hyperbolic.

Comments

This function returns an error if any of the number arguments is null.

Caution: An Oracle number overflow causes an unpredictable
result value.

Related Functions

OCIErrorGet(), OCINumberHypCos(), OCINumberHypSin()

OCINumberInc()

Purpose

Increments an OCINumber.

Syntax

```
OCINumberInc ( OCIError   *err,
               OCINumber  *number );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

number (IN/OUT)
A positive Oracle number to be incremented.

Comments

Increments an Oracle number in place. It is assumed that the input is an integer between 0 and 100^21-2. If the is input too large, it will be treated as 0 - the result will be an Oracle number 1. If the input is not a positive integer, the result will be unpredictable.

This function returns an error if the input number is null.

Related Functions

OCINumberDec()

OCINumberIntPower()

Purpose

Raises a given base to a given integer power.

Syntax

```
sword OCINumberIntPower ( OCIError        *err,
                          CONST OCINumber *base,
                          CONST sword     exp,
                          OCINumber       *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

base (IN)
Base of the exponentiation.

exp (IN)
Exponent to which the base is raised.

result (OUT)
Output of exponentiation.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberPower()

OCINumberIsInt()

Purpose

Tests if an OCINumber is an integer.

Syntax

```
sword OCINumberIsInt ( OCIError          *err,
                       CONST OCINumber   *number,
                       boolean           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
number to be tested

result (OUT)
set to TRUE if integer value else FALSE

Comments

This function returns an error if *number* or *result* is null.

Related Functions

OCIErrorGet(), OCINumberRound(), OCINumberTrunc()

OCINumberIsZero()

Purpose

Tests if the given number is equal to zero.

Syntax

```
sword OCINumberIsZero ( OCIError          *err,
                        CONST OCINumber   *number,
                        boolean           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Number to compare.

result (OUT)
Set to TRUE if equal to zero; otherwise, set to FALSE.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberSetZero()

OCINumberLn()

Purpose

Takes the natural logarithm (base *e*) of an Oracle number.

Syntax

```
sword OCINumberLn ( OCIError          *err,
                    CONST OCINumber   *number,
                    OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Logarithm of this number is computed.

result (OUT)
Logarithm result.

Comments

This function returns an error if any of the number arguments is null, or if *number*
is less than or equal to zero.

Related Functions

OCIErrorGet(), OCINumberExp(), OCINumberLog()

OCINumberLog()

Purpose

Takes the logarithm, to any base, of an Oracle number.

Syntax

```
sword OCINumberLog ( OCIError          *err,
                     CONST OCINumber   *base,
                     CONST OCINumber   *number,
                     OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

base (IN)
Base of the logarithm.

number (IN)
Operand.

result (OUT)
Logarithm result.

Comments

This function returns an error if:

- any of the number arguments is null.

- *number* <= 0

- *base* <= 0

Related Functions

OCIErrorGet(), OCINumberLn()

OCINumberMod()

Purpose

Gets the modulus (remainder) of the division of two Oracle numbers.

Syntax

```
sword OCINumberMod ( OCIError           *err,
                     CONST OCINumber    *number1,
                     CONST OCINumber    *number2,
                     OCINumber          *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1 (IN)
Pointer to the numerator.

number2 (IN)
Pointer to the denominator.

result (OUT)
Remainder of the result.

Comments

This function returns an error if *number1* or *number2* is null, or if there is a
divide-by-zero error.

Related Functions

OCIErrorGet(), OCINumberDiv()

OCINumberMul()

Purpose

Multiplies two Oracle numbers.

Syntax

```
sword OCINumberMul ( OCIError            *err,
                     CONST OCINumber     *number1,
                     CONST OCINumber     *number2,
                     OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1 (IN)
Number to multiply.

number2 (IN)
Number to multiply.

result (OUT)
Multiplication result.

Comments

Multiplies *number1* with *number2* and returns result in *result*.

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberDiv()

OCINumberNeg()

Purpose

Negates an Oracle number.

Syntax

```
sword OCINumberNeg ( OCIError           *err,
                     CONST OCINumber    *number,
                     OCINumber          *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Number to negate.

result (OUT)
Contains negated value of *number*.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberAbs(), OCINumberSign()

OCINumberPower()

Purpose

Raises a given base to a given exponent.

Syntax

```
sword OCINumberPower ( OCIError            *err,
                       CONST OCINumber     *base,
                       CONST OCINumber     *number,
                       OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

base (IN)
Base of the exponentiation.

number (IN)
Exponent to which the base is to be raised.

result (OUT)
Output of exponentiation.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberExp()

OCINumberPrec()

Purpose

Rounds an OCINumber to a specified number of decimal digits.

Syntax

```
sword OCINumberPrec ( OCIError *err,
                      CONST OCINumber *number,
                      eword nDigs,
                      OCINumber *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
The number for which to set precision.

nDigs (IN)
The number of decimal digits desired in the result.

result (OUT)
The result.

Comments

Performs a floating point round with respect to the number of digits.

This function returns an error any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberRound()

OCINumberRound()

Purpose

Rounds an Oracle number to a specified decimal place.

Syntax

```
sword OCINumberRound ( OCIError          *err,
                       CONST OCINumber   *number,
                       sword             decplace,
                       OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Number to round.

decplace (IN)
Number of decimal digits to the right of the decimal point to round to. Negative
values are allowed.

result (OUT)
Output of rounding.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberTrunc()

OCINumberSetPi()

Purpose

Sets an OCINumber to Pi.

Syntax

```
void OCINumberSetPi ( OCIError *err,
                      OCINumber *num );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

num (OUT)
Number set to the value of Pi.

Comments

Initializes the given number to the value of Pi.

Related Functions

OCIErrorGet()

OCINumberSetZero()

Purpose

Initializes an Oracle number to zero.

Syntax

```
void OCINumberSetZero ( OCIError     *err
                        OCINumber    *num );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

num (IN/OUT)
Number to initialize to zero value.

Comments

None.

Related Functions

OCIErrorGet()

OCINumberShift()

Purpose

Multiplies a number by a power of 10 by shifting it a specified number of decimal places.

Syntax

```
sword OCINumberShift ( OCIError        *err,
                       CONST OCINumber  *number,
                       CONST sword      nDig,
                       OCINumber        *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

number (IN)
Oracle Number to be shifted.

nDig (IN)
Number of decimal places to shift.

result (OUT)
Shift result.

Comments

Multiplies number by 10^{nDig} and sets *product* to the result.

This function returns an error if the input number is null.

Related Functions

OCIErrorGet()

OCINumberSign()

Purpose

Gets sign of an Oracle number.

Syntax

```
sword OCINumberSign ( OCIError           *err,
                      CONST OCINumber    *number,
                      sword              *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Number whose sign is returned.

result (OUT)
Possible values:

Value of *number*	Output in *result* parameter
number < 0	-1
number == 0	0
number > 0	1

Comments

This function returns an error if *number* or *result* is null.

Related Functions

OCIErrorGet(), OCINumberAbs()

OCINumberSin()

Purpose

Computes the sine in radians of an Oracle number.

Syntax

```
sword OCINumberSin ( OCIError            *err,
                     CONST OCINumber     *number,
                     OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

number (IN)
Argument of the sine in radians.

result (OUT)
Result of the sine.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberArcSin()

OCINumberSqrt()

Purpose

Computes the square root of an Oracle number.

Syntax

```
sword OCINumberSqrt ( OCIError          *err,
                      CONST OCINumber   *number,
                      OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Input number.

result (OUT)
Output which will contain the square root of the input number.

Comments

This function returns an error if `number` is null or `number` is negative.

Related Functions

OCIErrorGet(), OCINumberPower()

OCINumberSub()

Purpose

Subtract two Oracle numbers.

Syntax

```
sword OCINumberSub ( OCIError          *err,
                     CONST OCINumber   *number1,
                     CONST OCINumber   *number2,
                     OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number1, number2 (IN)
This function subtracts *number2* from *number1*.

result (OUT)
Subtraction result.

Comments

Subtracts *number2* from *number1* and returns result in *result*.

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberAdd()

OCINumberTan()

Purpose

Computes the tangent in radians of an Oracle number.

Syntax

```
sword OCINumberTan ( OCIError            *err,
                     CONST OCINumber     *number,
                     OCINumber           *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Argument of the tangent in radians.

result (OUT)
Result of the tangent.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberArcTan(), OCINumberArcTan2()

OCINumberToInt()

Purpose

Converts an Oracle number type to integer.

Syntax

```
sword OCINumberToInt ( OCIError              *err,
                       CONST OCINumber       *number,
                       uword                 rsl_length,
                       uword                 rsl_flag,
                       dvoid                 *rsl );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Number to convert.

rsl_length (IN)
Size of the desired result.

rsl_flag (IN)
Flag that designates the sign of the output, as follows:

- OCI_NUMBER_UNSIGNED - Unsigned values

- OCI_NUMBER_SIGNED - Signed values

rsl (OUT)
Pointer to space for the result.

Comments

This is a native type conversion function. It converts the given Oracle number into
an integer of the form **xbn**, such as **ub2**, **ub4**, or **sb2**.

This function returns an error if *number* or *rsl* is null, if *number* is too big
(overflow) or too small (underflow), or if an invalid sign flag value is passed in
rsl_flag.

Related Functions

OCIErrorGet(), OCINumberFromInt()

OCINumberToReal()

Purpose

Converts an Oracle number type to Real.

Syntax

```
sword OCINumberToReal ( OCIError          *err,
                        CONST OCINumber   *number,
                        uword             rsl_length,
                        dvoid             *rsl );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Number to convert.

rsl_length (IN)
The size of the desired result, which equals sizeof({ **float** | **double** | **long double**}).

rsl (OUT)
Pointer to space for storing the result.

Comments

This is a native type conversion function. It converts an Oracle number into a machine-native real type. This function only converts numbers up to LDBL_DIG, DBL_DIG, or FLT_DIG digits of precision and removes trailing zeroes. The above constants are defined in float.h.

This function returns an error if *number* or *rsl* is null, or if *rsl_length* = 0.

Related Functions

OCIErrorGet(), OCINumberFromReal()

OCINumberToText()

Purpose

Converts an Oracle number to a character string according to a specified format.

Syntax

```
sword OCINumberToText ( OCIError           *err,
                        CONST OCINumber    *number,
                        CONST OraText      *fmt,
                        ub4                fmt_length,
                        CONST OraText      *nls_params,
                        ub4                nls_p_length,
                        ub4                *buf_size,
                        text               *buf );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

number (IN)
Oracle number to convert.

fmt (IN)
Conversion format.

fmt_length (IN)
Length of the `fmt` parameter.

nls_params (IN)
Global Support format specification. If it is a null string (`(text *)0`), then the
default parameters for the session is used.

nls_p_length (IN)
Length of the `nls_params` parameter.

buf_size (IN)
Size of the buffer.

buf (OUT)
Buffer into which the converted string is placed.

Comments

Refer to the TO_NUMBER conversion function described in the *Oracle8i SQL
Reference* for a description of format and Global Support parameters.

The converted number string is stored in *buf*, up to a maximum of *buf_size* bytes. This function returns an error if:

- *number* or *buf* is null
- buffer is too small
- invalid format or invalid multibyte format is passed
- number to text translation for given format causes an overflow

Related Functions

OCIErrorGet(), OCINumberFromText()

OCINumberTrunc()

Purpose

Truncates an Oracle number at a specified decimal place.

Syntax

```
sword OCINumberTrunc ( OCIError          *err,
                       CONST OCINumber   *number,
                       sword             decplace,
                       OCINumber         *result );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

number (IN)
Input number.

decplace (IN)
Number of decimal digits to the right of the decimal point at which to truncate. Negative values are allowed.

result (OUT)
Output of truncation.

Comments

This function returns an error if any of the number arguments is null.

Related Functions

OCIErrorGet(), OCINumberRound()

OCI Raw Functions

This section describes the OCI Raw functions.

Table 18–5   Raw Functions

Function/Page	Purpose
OCIRawAllocSize() on page 18-146	Get allocated size of raw memory in bytes
OCIRawAssignBytes() on page 18-147	Assign raw bytes to raw
OCIRawAssignRaw() on page 18-148	Assign raw to raw
OCIRawPtr() on page 18-149	Get raw data Pointer
OCIRawResize() on page 18-150	Resize memory of variable-length raw
OCIRawSize() on page 18-151	Get raw size

OCIRawAllocSize()

Purpose

Gets allocated size of raw memory in bytes.

Syntax

```
sword OCIRawAllocSize ( OCIEnv          *env,
                        OCIError        *err,
                        CONST OCIRaw    *raw,
                        ub4             *allocsize );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

raw (IN)
Raw data whose allocated size in bytes is returned. This must be a non-null pointer.

allocsize (OUT)
The allocated size of raw memory in bytes is returned.

Comments

The allocated size is greater than or equal to the actual raw size.

Related Functions

OCIErrorGet(), OCIRawResize(), OCIRawSize()

OCIRawAssignBytes()

Purpose

Assigns raw bytes of type **ub1*** to Oracle **OCIRaw*** datatype.

Syntax

```
sword OCIRawAssignBytes ( OCIEnv          *env,
                          OCIError        *err,
                          CONST ub1       *rhs,
                          ub4             rhs_len,
                          OCIRaw          **lhs );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

rhs (IN)
Right-hand side (source) of the assignment, of datatype **ub1.**

rhs_len (IN)
Length of the *rhs* raw bytes.

lhs (IN/OUT)
Left-hand side (target) of the assignment **OCIRaw** data.

Comments

Assigns *rhs* raw bytes to *lhs* raw datatype. The *lhs* raw may be resized
depending upon the size of the *rhs*. The raw bytes assigned are of type **ub1**.

Related Functions

OCIErrorGet(), OCIRawAssignRaw()

OCIRawAssignRaw()

Purpose

Assign one Oracle raw datatype to another Oracle raw datatype.

Syntax

```
sword OCIRawAssignRaw ( OCIEnv              *env,
                        OCIError            *err,
                        CONST OCIRaw        *rhs,
                        OCIRaw              **lhs );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

rhs (IN)
Right-hand side (source) of the assignment; **OCIRaw** data.

lhs (IN/OUT)
Left-hand side (target) of the assignment; **OCIRaw** data.

Comments

Assigns *rhs* raw to *lhs* raw. The *lhs* raw may be resized depending upon the size
of the *rhs*.

Related Functions

OCIErrorGet(), OCIRawAssignBytes()

OCIRawPtr()

Purpose

Gets the pointer to raw data.

Syntax

```
ub1 *OCIRawPtr ( OCIEnv           *env,
                 CONST OCIRaw     *raw );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

raw (IN)
Pointer to the data of a given raw is returned.

Comments

None.

Related Functions

OCIErrorGet(), OCIRawAssignRaw()

OCIRawResize()

Purpose

Resizes the memory of a given variable-length raw.

Syntax

```
sword OCIRawResize ( OCIEnv           *env,
                     OCIError         *err,
                     ub2              new_size,
                     OCIRaw           **raw );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

new_size (IN)
New size of the raw data in bytes.

raw (IN)
Variable-length raw pointer; the raw is resized to *new_size*.

Comments

This function resizes the memory of the given variable-length raw in the object
cache. The previous contents of the raw are *not* preserved. This function may
allocate the raw in a new memory region in which case the original memory
occupied by the given raw will be freed. If the input raw is null (*raw* == NULL),
then this function will allocate memory for the raw data.

If the *new_size* is 0, then this function frees the memory occupied by *raw* and a
null pointer value is returned.

Related Functions

OCIErrorGet(), OCIRawAllocSize(), OCIRawSize()

OCIRawSize()

Purpose

Returns the size of a given raw in bytes.

Syntax

```
ub4 OCIRawSize ( OCIEnv          *env,
                 CONST OCIRaw    *raw );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

raw (IN/OUT)
Raw whose size is returned.

Comments

None.

Related Functions

OCIErrorGet(), OCIRawAllocSize(), OCIRawSize()

OCI Ref Functions

This section describes the OCI Ref functions.

Table 18–6   Ref Functions

Function/Page	Purpose
`OCIRefAssign()` on page 18-153	Assign one REF to another
`OCIRefClear()` on page 18-154	Clear or nullify a REF
`OCIRefFromHex()` on page 18-155	Convert hexadecimal string to REF
`OCIRefHexSize()` on page 18-157	Return size of hexadecimal representation of REF
`OCIRefIsEqual()` on page 18-158	Compare two REFs for equality
`OCIRefIsNull()` on page 18-159	Test if a REF is null
`OCIRefToHex()` on page 18-160	Convert REF to hexadecimal string

OCIRefAssign()

Purpose

Assigns one REF to another, such that both reference the same object.

Syntax

```
sword OCIRefAssign ( OCIEnv            *env,
                     OCIError          *err,
                     CONST OCIRef      *source,
                     OCIRef            **target );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

source (IN)
REF to copy from.

target (IN/OUT)
REF to copy to.

Comments

Copies *source* REF to *target* REF; both then reference the same object. If the *target* REF pointer is null (i.e. *target == NULL), then OCIRefAssign() will allocate memory for the *target* REF in the OCI object cache prior to the copy.

Related Functions

OCIErrorGet(), OCIRefIsEqual()

OCIRefClear()

Purpose

Clears or nullifies a given REF.

Syntax

```
void OCIRefClear ( OCIEnv        *env,
                   OCIRef        *ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

ref (IN/OUT)
REF to clear.

Comments

A REF is considered to be a null REF if it no longer points to an object. Logically, a null REF is a dangling REF.

Note that a null REF is still a valid SQL value and is not SQL NULL. It can be used as a valid non-null constant REF value for a NOT NULL column or attribute of a row in a table.

If a null pointer value is passed as a REF, then this function is non-operational.

Related Functions

OCIErrorGet(), OCIRefIsNull()

OCIRefFromHex()

Purpose

Converts the given hexadecimal string into a REF.

Syntax

```
sword OCIRefFromHex ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCISvcCtx *svc,
                      CONST OraText   *hex,
                      ub4             length,
                      OCIRef          **ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

svc (IN)
OCI service context handle; if the resulting ref is initialized with this service context.

hex (IN)
Hexadecimal text string, previously output by OCIRefToHex(), to convert into a
REF.

length (IN)
Length of the hexadecimal text string.

ref (IN/OUT)
The REF into which the hexadecimal string is converted. If *ref* is null on input,
then space for the REF is allocated in the object cache, otherwise the memory
occupied by the given REF is re-used.

Comments

This function ensures that the resulting REF is well formed. It does *not* ensure that
the object pointed to by the resulting REF exists or not.

Related Functions

OCIErrorGet(), OCIRefToHex()

OCIRefHexSize()

Purpose

Returns the size of the hex representation of a REF.

Syntax

```
ub4 OCIRefHexSize ( OCIEnv          *env,
                    CONST OCIRef     *ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

ref (IN)
REF whose size in hexadecimal representation in bytes is returned.

Returns

The size of the hexadecimal representation of the REF.

Comments

Returns the size of the buffer in bytes required for the hexadecimal representation
of the ref. A buffer of at least this size must be passed to the ref-to-hex
(OCIRefToHex()) conversion function.

Related Functions

OCIErrorGet(), OCIRefFromHex()

OCIRefIsEqual()

Purpose

Compares two REFs to determine if they are equal.

Syntax

```
boolean OCIRefIsEqual ( OCIEnv          *env,
                        CONST OCIRef    *x,
                        CONST OCIRef    *y );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

x (IN)
REF to compare.

y (IN)
REF to compare.

Returns

TRUE if the two REFs are equal

FALSE if the two REFs are not equal, or x is null, or y is null

Comments

Two REFs are equal if and only if they are both referencing the same object, whether persistent or transient.

> **Note:** Two null REFs are considered *not* equal by this function.

Related Functions

OCIErrorGet(), OCIRefAssign()

OCIRefIsNull()

Purpose

Tests if a REF is null.

Syntax

```
boolean OCIRefIsNull ( OCIEnv        *env,
                       CONST OCIRef  *ref );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

ref (IN)
REF to test for null.

Returns

Returns TRUE if the given REF is null; otherwise, returns FALSE.

Comments

A REF is null if and only if:

- it is supposed to be referencing a persistent object, but the object's identifier is null

- it is supposed to be referencing a transient object, but it is currently not pointing to an object.

> **Note:** A REF is a *dangling REF* if the object that it points to does not exist.

Related Functions

OCIErrorGet(), OCIRefClear()

OCIRefToHex()

Purpose

Converts a REF to a hexadecimal string.

Syntax

```
sword OCIRefToHex ( OCIEnv          *env,
                    OCIError        *err,
                    CONST OCIRef    *ref,
                    OraText         *hex,
                    ub4             *hex_length );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

ref (IN)
REF to be converted into a hexadecimal string; if *ref* is a null REF (i.e.
OCIRefIsNull(*ref*) == TRUE) then zero *hex_length* value is returned.

hex (OUT)
Buffer that is large enough to contain the resulting hexadecimal string; the contents
of the string is opaque to the caller.

hex_length (IN/OUT)
On input specifies the size of the *hex* buffer on output specifies the actual size of
the hexadecimal string being returned in *hex*.

Comments

Converts the given REF into a hexadecimal string, and returns the length of the
string. The resulting string is opaque to the caller.

This function returns an error if the given buffer is not big enough to hold the
resulting string.

Related Functions

OCIErrorGet(), OCIRefFromHex(), OCIRefHexSize(), OCIRefIsNull()

OCI String Functions

This section describes the OCI string functions.

Table 18–7   String Functions

Function/Page	Purpose
`OCIStringAllocSize()` on page 18-162	Get allocated size of string memory in bytes
`OCIStringAssign()` on page 18-163	Assign string to string
`OCIStringAssignText()` on page 18-164	Assign text string to string
`OCIStringGetEncoding()` on page 18-165	Gets the encoding information for a given string.
`OCIStringPtr()` on page 18-166	Get string pointer
`OCIStringResize()` on page 18-167	Resize string memory
`OCIStringSize()` on page 18-168	Get string size

OCIStringAllocSize()

Purpose

Gets allocated size of string memory in codepoints (Unicode) or in bytes.

Syntax

```
sword OCIStringAllocSize ( OCIEnv            *env,
                           OCIError          *err,
                           CONST OCIString   *vs,
                           ub4               *allocsize );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

vs (IN)

String whose allocated size in bytes is returned. *vs* must be a non-null pointer.

allocsize (OUT)

The allocated size of string memory in bytes is returned.

Comments

The allocated size is greater than or equal to the actual string size.

Related Functions

OCIErrorGet(), OCIStringResize(), OCIStringSize()

OCIStringAssign()

Purpose

Assigns one string to another string.

Syntax

```
sword OCIStringAssign ( OCIEnv              *env,
                        OCIError            *err,
                        CONST OCIString     *rhs,
                        OCIString           **lhs );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

rhs (IN)
Right-hand side (source) of the assignment. Can be in UTF-16.

lhs (IN/OUT)
Left-hand side (target) of the assignment. Its buffer is UTF-16 if *rhs* is UTF-16.

Comments

Assigns *rhs* string to *lhs* string. The *lhs* string may be resized depending upon
the size of the *rhs*. The assigned string is null-terminated. The length field will not
include the extra codepoint or byte needed for null-termination.

This function returns an error if the assignment operation runs out of space.

Related Functions

OCIErrorGet(), OCIStringAssignText()

OCIStringAssignText()

Purpose

Assigns the source text string to the target string.

Syntax

```
sword OCIStringAssignText ( OCIEnv          *env,
                            OCIError        *err,
                            CONST OraText   *rhs,
                            ub2             rhs_len,
                            OCIString       **lhs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

rhs (IN)

Right-hand side (source) of the assignment, a text or UTF-16 Unicode string.

rhs_len (IN)

Length of the *rhs* string in either codepoints (if Unicode) or in bytes (if non-Unicode).

lhs (IN/OUT)

Left-hand side (target) of the assignment. Its buffer is Unicode if *rhs* is Unicode.

Comments

Assigns *rhs* string to *lhs* string. The *lhs* string may be resized depending upon the size of the *rhs*. The assigned string is null-terminated. The length field will not include the extra byte or codepoint needed for null-termination.

Related Functions

OCIErrorGet(), OCIStringAssign()

OCIStringGetEncoding()

Purpose

Gets the encoding information for a given string.

Syntax

```
ub4 OCIStringGetEncoding ( OCIEnv              *env,
                           CONST OCIString     *vs );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

vs (IN)
Pointer to the text of the given string.

Comments

Returns:

- OCI_UNICODE - this OCIString object uses UTF-16 Unicode encoding

- OCI_NON_UNICODE - this OCIString object uses non-Unicode encoding

Related Functions

OCIErrorGet()

OCIStringPtr()

Purpose

Gets a pointer to the text of a given string.

Syntax

```
text *OCIStringPtr ( OCIEnv           *env,
                     CONST OCIString  *vs );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

vs (IN)
Pointer to the **OCIString** object whose character string will be returned. If *vs* is in UTF-16, the returned buffer will also be UTF-16. If you want to know the encoding of the returned buffer, check the UTF-16 information in the **OCIString** *vs* itself, since it is not guaranteed that a particular **OCIString** must have the same setting as *env* does. The function to check should be some object OCI function designed to check member fields in objects.

Comments

None.

Related Functions

OCIErrorGet(), OCIStringAssign()

OCIStringResize()

Purpose

Resizes the memory of a given string.

Syntax

```
sword OCIStringResize ( OCIEnv        *env,
                        OCIError      *err,
                        ub4           new_size,
                        OCIString     **str );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

new_size (IN)
New memory size of the string in bytes. *new_size* must include space for the
NULL character as the string terminator.

str (IN/OUT)
Allocated memory for the string which is freed from the OCI object cache.

Comments

This function resizes the memory of the given variable-length string in the object
cache. Contents of the string are *not* preserved. This function may allocate the string
in a new memory region, in which case the original memory occupied by the given
string is freed. If *str* is null, this function allocates memory for the string. If
new_size is 0, this function frees the memory occupied by *str* and a null pointer
value is returned.

Related Functions

OCIErrorGet(), OCIStringAllocSize(), OCIStringSize()

OCIStringSize()

Purpose

Gets the size of the given string *vs*.

Syntax

```
ub4 OCIStringSize ( OCIEnv           *env,
                    CONST OCIString  *vs );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** `OCIEnvCreate()` on page 15-9 and
> `OCIInitialize()` on page 15-14

vs (IN)
String whose size is returned. In number of bytes.

Comments

The returned size does not include an extra byte for null termination.

Related Functions

`OCIErrorGet()`, `OCIStringResize()`

OCI Table Functions

This section describes the OCI Table functions.

Table 18–8   Table Functions

Function/Page	Purpose
OCITableDelete() on page 18-170	Delete element
OCITableExists() on page 18-171	Test whether element exists
OCITableFirst() on page 18-172	Return first index of table
OCITableLast() on page 18-173	Return last index of table
OCITableNext() on page 18-174	Return next available index of table
OCITablePrev() on page 18-176	Return previous available index of table
OCITableSize() on page 18-178	Return current size of table

OCITableDelete()

Purpose

Deletes the element at the specified index.

Syntax

```
sword OCITableDelete ( OCIEnv       *env,
                       OCIError     *err,
                       sb4          index,
                       OCITable     *tbl );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

index (IN)
Index of the element which must be deleted.

tbl (IN)
Table whose element is deleted.

Comments

This function returns an error if the element at the given index has already been
deleted or if the given index is not valid for the given table. It is also an error if any
input parameter is null.

> **Note:**   The position ordinals of the remaining elements of the table
> are not changed by OCITableDelete(). The delete operation
> creates *holes* in the table.

Related Functions

OCIErrorGet(), OCITableExists()

OCITableExists()

Purpose

Tests whether an element exists at the given index.

Syntax

```
sword OCITableExists ( OCIEnv            *env,
                       OCIError          *err,
                       CONST OCITable    *tbl,
                       sb4               index,
                       boolean           *exists );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

tbl (IN)
Table in which the given index is checked.

index (IN)
Index of the element which is checked for existence.

exists (OUT)
Set to TRUE if element at given *index* exists; otherwise, it is set to FALSE.

Comments

This function returns an error if any input parameter is null.

Related Functions

OCIErrorGet(), OCITableDelete()

OCITableFirst()

Purpose

Returns the index of the first existing element in a given table.

Syntax

```
sword OCITableFirst ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCITable  *tbl,
                      sb4             *index );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

tbl (IN)
Table to scan.

index (OUT)
First index of the element which exists in the given table is returned.

Comments

For example, if OCITableDelete() deleted the first 5 elements of a table,
OCITableFirst() returns 6.

> **See Also:** OCITableDelete() for information regarding
> non-data *holes* in tables.

This function returns an error if the table is empty.

Related Functions

OCIErrorGet(), OCITableDelete(), OCITableLast()

OCITableLast()

Purpose

Returns the index of the last existing element of a table.

Syntax

```
sword OCITableLast ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCITable  *tbl,
                     sb4             *index );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

tbl (IN)
Table to scan.

index (OUT)
Index of the last existing element in the table.

Comments

This function returns an error if the table is empty.

Related Functions

OCIErrorGet(), OCITableFirst(), OCITableNext(), OCITablePrev()

OCITableNext()

Purpose

Returns the index of the next existing element of a table.

Syntax

```
sword OCITableNext ( OCIEnv            *env,
                     OCIError          *err,
                     sb4               index,
                     CONST OCITable    *tbl,
                     sb4               *next_index
                     boolean           *exists );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

index (IN)
Index for starting point of scan.

tbl (IN)
Table to scan.

next_index (OUT)
Index of the next existing element after tbl(*index*).

exists (OUT)
FALSE if no next index is available, else TRUE.

Comments

Returns the smallest position j, greater than *index*, such that exists(j) is TRUE.

> **See Also:**   Refer to the description of OCIStringAllocSize(),
> regarding the existence of non-data *holes* in tables.

Related Functions

`OCIErrorGet()`, `OCITablePrev()`

OCITablePrev()

Purpose

Returns the index of the previous existing element of a table.

Syntax

```
sword OCITablePrev ( OCIEnv           *env,
                     OCIError         *err,
                     sb4              index,
                     CONST OCITable   *tbl,
                     sb4              *prev_index
                     boolean          *exists );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** OCIEnvCreate() on page 15-9 and
> OCIInitialize() on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

index (IN)
Index for starting point of scan.

tbl (IN)
Table to scan.

prev_index (OUT)
Index of the previous existing element before tbl(*index*).

exists (OUT)
FALSE if no previous index is available, else TRUE.

Comments

Return the largest position *j*, less than *index*, such that exists(*j*) is TRUE.

> **See Also:** Refer to the description of OCIStringAllocSize(),
> regarding the existence of non-data *holes* in tables.

Related Functions

OCITableNext()

OCITableSize()

Purpose

Returns the size of the given table, not including deleted elements.

Syntax

```
sword OCITableSize ( OCIEnv            *env,
                     OCIError          *err,
                     CONST OCITable    *tbl
                     sb4               *size );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** `OCIEnvCreate()` on page 15-9 and
> `OCIInitialize()` on page 15-14

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

tbl (IN)
Nested table whose number of elements is returned.

size (OUT)
Current number of elements in the nested table. The count does not include deleted
elements.

Comments

The count will be decremented upon deleting elements from the nested table. So
this count does not include any *holes* created by deleting elements. To get the count
not including the deleted elements, use `OCICollSize()`.

For example:

```
OCITableSize(...);
// assume 'size' returned is equal to 5
OCITableDelete(...); // delete one element
OCITableSize(...);
// 'size' returned is equal to 4
```

To get the count plus the count of deleted elements use `OCICollSize()`.
Continuing the previous example:

```
OCICollSize(...)
// 'size' returned is still equal to 5
```

This function returns an error if an error occurs during the loading of the nested table into the object cache, or if any of the input parameters is null.

Related Functions

OCICollSize()

19

OCI Cartridge Functions

The chapter contains the following sections:

- Introduction to External Procedure and Cartridge Services Functions
- Cartridge Services — OCI External Procedures
- Cartridge Services — Memory Services
- Cartridge Services — Maintaining Context
- Cartridge Services — Parameter Manager Interface
- Cartridge Services — File I/O Interface
- Cartridge Services — String Formatting Interface

Introduction to External Procedure and Cartridge Services Functions

This chapter first describes the OCI external procedure functions. These functions enable users of external procedures to raise errors, allocate some memory, and get OCI context information.

> **See Also:** For more information about using these functions in external procedures, see the chapter on external routines in *Oracle9i Application Developer's Guide - Fundamentals*

Then the cartridge services functions are described.

> **See Also:** For more information about using these functions, see *Oracle9i Data Cartridge Developer's Guide*

The Function Syntax

For each function, the following information is listed:

Purpose

A brief description of the action performed by the function.

Syntax

The function declaration.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below:

Mode	Description
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.

Comments

More detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Returns

A list of possible return values for the function.

Related Functions

A list of related function calls. For cartridge services, see all the other functions in the group being documented.

Return Codes

Success and error return codes are defined for certain external procedure interface functions. If a particular interface function returns OCIEXTPROC_SUCCESS or OCIEXTPROC_ERROR, then applications must use these macros to check for return values.

- OCIEXTPROC_SUCCESS - External Procedure Success Return Code

- OCIEXTPROC_ERROR - External Procedure Failure Return Code

With_Context Type

The C callable interface to PL/SQL external procedures requires the *with_context* parameter of the OCI external procedure function to be passed. The type of this structure is **OCIExtProcContext**, which is opaque to the user.

The user can declare the *with_context* parameter in the application as

```
OCIExtProcContext *with_context;
```

Cartridge Services — OCI External Procedures

The OCI external procedure functions for C:

Table 19–1   External Procedures Functions

Function/Page	Purpose
`OCIExtProcAllocCallMemory()` on page 19-5	Allocates memory for the duration of the External Procedure
`OCIExtProcRaiseExcp()` on page 19-6	Raises an Exception to PL/SQL
`OCIExtProcRaiseExcpWithMsg()` on page 19-7	Raises an exception with a message
`OCIExtProcGetEnv()` on page 19-8	Gets the OCI environment, service context, and error handles

OCIExtProcAllocCallMemory()

Purpose

Allocate N bytes of memory for the duration of the External Procedure.

Syntax

```
dvoid * OCIExtProcAllocCallMemory ( OCIExtProcContext    *with_context,
                                    size_t                amount )
```

Parameters

with_context (IN)
The **with_context** pointer that is passed to the C External Procedure.

> **See Also:** "With_Context Type" on page 19-3

amount (IN)
The number of bytes to allocate.

Comments

This call allocates *amount* bytes of memory for the duration of the call of the external procedure.

Any memory allocated by this call is freed by PL/SQL upon return from the external procedure. The application must not use any kind of free() function on memory allocated by OCIExtProcAllocCallMemory(). Use this function to allocate memory for function returns.

A zero return value should be treated as an error

Returns

An untyped (opaque) Pointer to the allocated memory.

Example

```
text *ptr = (text *)OCIExtProcAllocCallMemory(wctx, 1024)
```

Related Functions

OCIErrorGet(), OCIMemoryAlloc().

OCIExtProcRaiseExcp()

Purpose

Raise an Exception to PL/SQL.

Syntax

```
size_t OCIExtProcRaiseExcp ( OCIExtProcContext    *with_context,
                            int                   errnum )
```

Parameters

with_context (IN)
The **with_context** pointer that is passed to the C External Procedure.

> **See Also:** "With_Context Type" on page 19-3

errnum (IN)
Oracle Error number to signal to PL/SQL. *errnum* must be a positive number and in the range 1 to 32767.

Comments

Calling this function signals an exception back to PL/SQL. After a successful return from this function, the external procedure must start its exit handling and return back to PL/SQL. Once an exception is signalled to PL/SQL, IN/OUT and OUT arguments, if any, are not processed at all.

Returns

This function returns OCIEXTPROC_SUCCESS if the call was successful. It returns OCIEXTPROC_ERROR if the call has failed.

Related Functions

OCIExtProcRaiseExcpWithMsg()

OCIExtProcRaiseExcpWithMsg()

Purpose

Raise an exception with a message.

Syntax

```
size_t OCIExtProcRaiseExcpWithMsg ( OCIExtProcContext   *with_context,
                                    int                 errnum,
                                    char                *errmsg,
                                    size_t              msglen )
```

Parameters

with_context (IN)
The **with_context** pointer that is passed to the C External Procedure.

> **See Also:** "With_Context Type" on page 19-3

errnum (IN)
Oracle Error number to signal to PL/SQL. The value of *errnum* must be a positive
number and in the range 1 to 32767

errmsg (IN)
The error message associated with the *errnum*.

len (IN)
The length of the error message. Pass zero if *errmsg* is a null-terminated string.

Comments

Raise an exception to PL/SQL. In addition, substitute the following error message
string within the standard Oracle error message string.

> **See Also:** See the description of OCIExtProcRaiseExcp() for
> more information.

Returns

This function returns OCIEXTPROC_SUCCESS if the call was successful. It returns
OCIEXTPROC_ERROR if the call has failed.

Related Functions

OCIExtProcRaiseExcp()

OCIExtProcGetEnv()

Purpose

Gets the OCI environment, service context, and error handles.

Syntax

```
sword OCIExtProcGetEnv ( OCIExtProcContext    *with_context,
                         OCIEnv               envh,
                         OCISvcCtx            svch,
                         OCIError             errh )
```

Parameters

with_context (IN)
The **with_context** pointer that is passed to the C External Procedure. See "With_Context Type" on page 19-3.

envh (OUT)
The OCI Environment handle.

svch (OUT)
The OCI Service handle.

errh (OUT)
The OCI Error handle.

Comments

The primary purpose of this function is to allow OCI callbacks to use the database in the same transaction. The OCI handles obtained by this function should be used in OCI callbacks to the database. If these handles are obtained through standard OCI calls, then these handles use a new connection to the database and cannot be used for callbacks in the same transaction. In one external procedure you can use either callbacks or a new connection, but not both.

Returns

This function returns OCI_SUCCESS if the call was successful; otherwise, it returns OCI_ERROR.

Related Functions

OCIEnvCreate(), OCIAttrGet(), OCIHandleAlloc()

Cartridge Services — Memory Services

Table 19–2   Memory Services Functions

Function/Page	Purpose
`OCIDurationBegin()` on page 19-10	Starts a user duration.
`OCIDurationEnd()` on page 19-11	Terminates a user duration.
`OCIMemoryAlloc()` on page 19-12	Allocates memory of a given size from a given duration.
`OCIMemoryResize()` on page 19-14	Resizes a memory chunk.
`OCIMemoryFree()` on page 19-15	Frees a memory chunk.

> **See Also:**   For more information about using these functions, see
> *Oracle9i Data Cartridge Developer's Guide*

OCIDurationBegin()

Purpose

Starts a user duration.

Syntax

```
sword OCIDurationBegin ( OCIEnv          *env,
                         OCIError        *err,
                         CONST OCISvcCtx *svc,
                         OCIDuration     parent,
                         OCIDuration     *duration );
```

Parameters

env (IN/OUT)
The OCI environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
`OCIErrorGet()`.

svc (IN)
The OCI service context handle. This should be passed as null for cartridge services.

parent (IN)
The duration number of the parent duration. One of these:

- A user duration that was previously created.

- OCI_DURATION_STATEMENT

- OCI_DURATION_SESSION

duration (OUT)
An identifier unique to the newly created user duration.

Comments

This function starts a user duration. A user can have multiple active user durations
simultaneously. The user durations do not have to be nested. The *duration*
parameter is used to return a number which uniquely identifies the duration
created by this call.

Note that the environment and service context parameters cannot both be null.

Related Functions

OCIDurationEnd()

OCIDurationEnd()

Purpose

Terminates a user duration.

Syntax

```
sword OCIDurationEnd ( OCIEnv          *env,
                       OCIError        *err,
                       CONST OCISvcCtx *svc,
                       OCIDuration     duration,
                       CONST OCISvcCtx *svc );
```

Parameters

env (IN/OUT)
The OCI environment handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
`OCIErrorGet()`.

duration (IN)
A user duration previously created by `OCIDurationBegin()`.

svc (IN)
OCI service context (this should be passed as null for cartridge services, otherwise
non-null)

Comments

This function terminates a user duration.

Note that the environment and service context parameters cannot both be null.

Related Functions

`OCIDurationBegin()`

OCIMemoryAlloc()

Purpose

This call allocates memory of a given size from a given duration.

Syntax

```
sword OCIMemoryAlloc( dvoid       *hndl,
                      OCIError    *err,
                      dvoid       **mem,
                      OCIDuration dur,
                      ub4         size,
                      ub4         flags );
```

Parameters

hndl (IN)
The OCI environment handle.

err (IN)
The error handle.

mem (OUT)
Memory allocated.

dur (IN)
One of the following (a previously created user duration):

OCI_DURATION_CALLOUT

OCI_DURATION_STATEMENT

OCI_DURATION_SESSION

OCI_DURATION_PROCESS

size (IN)
Size of memory to be allocated.

flags (IN)
Set OCI_MEMORY_CLEARED bit to get memory that has been cleared.

Comments

To allocate memory for duration of callout of agent, i.e., external procedure duration, use `OCIExtProcAllocCallMemory()` or `OCIMemoryAlloc()` with *dur* as OCI_DURATION_CALLOUT.

Returns

Error code.

OCIMemoryResize()

Purpose

This call resizes a memory chunk to a new size.

Syntax

```
sword OCIMemoryResize( dvoid      *hndl,
                       OCIError   *err,
                       dvoid      **mem,
                       ub4        newsize,
                       ub4        flags );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN)
The error handle.

mem (IN/OUT)
Pointer to memory allocated previously using OCIMemoryAlloc().

newsize (IN)
Size of memory requested.

flags (IN)
Set OCI_MEMORY_CLEARED bit to get memory that has been cleared

Comments

Memory must have been allocated before this function can be called to resize.

Returns

Error code.

OCIMemoryFree()

Purpose

This call frees a memory chunk.

Syntax

```
sword OCIMemoryFree( dvoid    *hndl,
                     OCIError *err,
                     dvoid    *mem );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN)
The error handle.

mem (IN/OUT)
Pointer to memory allocated previously using OCIMemoryAlloc().

Returns

Error code.

Cartridge Services — Maintaining Context

Table 19–3   Maintaining Context Functions

Function/Page	Purpose
OCIContextSetValue() on page 19-17	Save a value (or address) for a particular duration.
OCIContextGetValue() on page 19-19	Return the value stored in the context.
OCIContextClearValue() on page 19-20	Remove the value stored in the context.
OCIContextGenerateKey() on page 19-21	Returns a unique 4-byte value each time it is called.

> **See Also:**   For more information about using these functions, see
> *Oracle9i Data Cartridge Developer's Guide*

OCIContextSetValue()

Purpose

This call is used to save a value (or address) for a particular duration.

Syntax

```
sword OCIContextSetValue( dvoid      *hndl,
                          OCIError   *err,
                          OCIDuration duration,
                          ub1        *key,
                          ub1        keylen,
                          dvoid      *ctx_value );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN)
The error handle.

duration (IN)
One of the following (a previously created user duration):

OCI_DURATION_STATEMENT

OCI_DURATION_SESSION

key (IN)
Unique key value.

keylen (IN)
Length of the key. Maximum is 64 bits.

ctx_value (IN)
Pointer that will be saved in the context.

Comments

The context value being stored must be allocated out of memory of duration greater than or equal to the duration being passed in. The key being passed in should be unique in this session. Trying to save a context value under the same key and duration again will result in overwriting the old context value with the new one. Typically, a client will allocate a structure, store its address in the context using this

call, and get this address in a separate call using `OCIContextGetValue()`. The (key, value) association can be explicitly removed by calling `OCIContextClearValue()` or else it will go away at the end of the duration.

Returns

- If operation succeeds, return OCI_SUCCESS.
- If operation fails, return OCI_ERROR.

OCIContextGetValue()

Purpose

This call is used to return the value that is stored in the context associated with the given key (by calling `OCIContextSetValue()`).

Syntax

```
sword OCIContextGetValue( dvoid    *hndl,
                          OCIError  *err,
                          ub1       *key,
                          ub1       keylen,
                          dvoid     **ctx_value );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN)
The error handle.

key (IN)
Unique key value.

keylen (IN)
Length of the key. Maximum is 64 bits.

ctx_value (IN)
Pointer to the value stored in the context (null if no value was stored).

Comments

For *ctx_value*: a pointer to a preallocated pointer for the stored context to be returned is required.

Returns

- If operation succeeds, return OCI_SUCCESS.

- If operation fails, return OCI_ERROR.

OCIContextClearValue()

Purpose

This call is used to remove the value that is stored in the context associated with the given key (by calling `OCIContextSetValue()`).

Syntax

```
sword OCIContextClearValue( dvoid   *hndl,
                            OCIError *err,
                            ub1      *key,
                            ub1       keylen );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN)
The error handle.

key (IN)
Unique key value.

keylen (IN)
Length of the key. Maximum is 64 bits.

Comments

An error is returned when a non-existent key is passed.

Returns

- If operation succeeds, returns OCI_SUCCESS.

- If operation fails, returns OCI_ERROR.

OCIContextGenerateKey()

Purpose

This call will return a unique, 4-byte value each time it is called.

Syntax

```
sword OCIContextGenerateKey( dvoid    *hndl,
                             OCIError *err,
                             ub4      *key );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN)
The error handle.

key (IN)
Unique key value.

keylen (IN)
Length of the key. Maximum is 64 bits.

Comments

This value is going to be unique for each session.

Returns

- If operation succeeds, return OCI_SUCCESS.

- If operation fails, return OCI_ERROR.

Cartridge Services — Parameter Manager Interface

Table 19–4   Parameter Manager Interface Functions

Function/Page	Purpose
OCIExtractInit() on page 19-23	Initializes the parameter manager.
OCIExtractTerm() on page 19-24	Releases all dynamically allocated storage.
OCIExtractReset() on page 19-25	Re-initializes memory.
OCIExtractSetNumKeys() on page 19-26	Informs the parameter manager of the number of keys that will be registered.
OCIExtractSetKey() on page 19-27	Registers information about a key with the parameter manager.
OCIExtractFromFile() on page 19-29	The keys and their values in the given file are processed.
OCIExtractFromStr() on page 19-30	The keys and the their values in the given string are processed.
OCIExtractToInt() on page 19-31	Gets the integer value for the specified key.
OCIExtractToBool() on page 19-32	Gets the boolean value for the specified key.
OCIExtractToStr() on page 19-33	Gets the string value for the specified key.
OCIExtractToOCINum() on page 19-35	Gets the number value for the specified key.
OCIExtractToList() on page 19-36	Generates a list of parameters from the parameter structures that are stored in memory.
OCIExtractFromList() on page 19-37	Generates a list of values for the parameter denoted by *index* in the parameter list.

See Also: For more information about using these functions, see
Oracle9i Data Cartridge Developer's Guide

OCIExtractInit()

Purpose

This function initializes the parameter manager.

Syntax

```
sword OCIExtractInit( dvoid    *hndl,
                      OCIError  *err)
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR; diagnostic information can be obtained by calling
`OCIErrorGet()`

Comments

This function must be called before calling any other parameter manager routine
and it must only be called once. The Global Support information is stored inside the
parameter manager context and used in subsequent calls to `OCIExtract*`
routines.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractTerm()

Purpose

This function releases all dynamically allocated storage.

Syntax

```
sword OCIExtractTerm( dvoid    *hndl,
                      OCIError  *err );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling `OCIErrorGet()`.

Comments

This function may perform other internal bookkeeping functions. It must be called when the parameter manager is no longer being used and it must only be called once

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractReset()

Purpose

The memory currently used for parameter storage, key definition storage, and parameter value lists is freed and the structure is re-initialized.

Syntax

```
sword OCIExtractReset( dvoid    *hndl,
                       OCIError  *err );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractSetNumKeys()

Purpose

Informs the parameter manager of the number of keys that will be registered.

Syntax

```
sword OCIExtractSetNumKeys( dvoid   *hndl,
                            CIError *err,
                            uword   numkeys );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Diagnostic information can be obtained by calling
OCIErrorGet().

numkeys (IN)
The number of keys that will be registered with OCIExtractSetKey().

Comments

This routine must be called prior to the first call of OCIExtractSetKey().

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractSetKey()

Purpose

Registers information about a key with the parameter manager.

Syntax

```
sword OCIExtractSetKey( dvoid      *hndl,
                        OCIError    *err,
                        CONST text  *name,
                        ub1         type,
                        ub4         flag,
                        CONST dvoid *defval,
                        CONST sb4   *intrange,
                        CONST text  *CONST *strlist );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

name (IN)
The name of the key.

type (IN)
The type of the key:

OCI_EXTRACT_TYPE_INTEGER,

OCI_EXTRACT_TYPE_OCINUM,

OCI_EXTRACT_TYPE_STRING,

OCI_EXTRACT_TYPE_BOOLEAN.

flag (IN)
Set to OCI_EXTRACT_MULTIPLE if the key can take multiple values or 0 otherwise.

defval (IN)
Set to the default value for the key. It may be null if there is no default. A string default must be a (text*) type, an integer default must be an (sb4*) type, and a boolean default must be a (ub1*) type.

intrange (IN)
Starting and ending values for the allowable range of integer values; may be null if the key is not an integer type or if all integer values are acceptable.

strlist (IN)
List of all acceptable text strings for the key ended with 0 (or null). May be null if the key is not a string type or if all text values are acceptable.

Comments

This routine must be called after calling `OCIExtractNumKeys()` and before calling `OCIExtractFromFile()` or `OCIExtractFromStr()`.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractFromFile()

Purpose

The keys and their values in the given file are processed.

Syntax

```
sword OCIExtractFromFile( dvoid   *hndl,
                          OCIError *err,
                          ub4     flag,
                          text    *filename );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling `OCIErrorGet()`.

flag (IN)
Zero or has one or more of the following bits set:

OCI_EXTRACT_CASE_SENSITIVE,

OCI_EXTRACT_UNIQUE_ABBREVS,

OCI_EXTRACT_APPEND_VALUES.

filename (IN)
A null-terminated filename string.

Comments

`OCIExtractSetNumKeys()` and `OCIExtractSetKey()` routines must be called to define all of the keys before calling this routine.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractFromStr()

Purpose

The keys and their values in the given string are processed.

Syntax

```
sword OCIExtractFromStr( dvoid   *hndl,
                         OCIError *err,
                         ub4      flag,
                         text     *input );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function
returns OCI_ERROR; diagnostic information can be obtained by calling
OCIErrorGet().

flag (IN)
Zero or has one or more of the following bits set:

 OCI_EXTRACT_CASE_SENSITIVE,

OCI_EXTRACT_UNIQUE_ABBREVS,

or OCI_EXTRACT_APPEND_VALUES.

input (IN)
A null-terminated input string.

Comments

OCIExtractSetNumKeys() and OCIExtractSetKey() routines must be called
to define all of the keys before calling this routine.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractToInt()

Purpose

Gets the integer value for the specified key. The `valno`'th value (starting with 0) is returned.

Syntax

```
sword OCIExtractToInt( dvoid    *hndl,
                       OCIError *err,
                       text     *keyname,
                       uword    valno,
                       sb4      *retval );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling `OCIErrorGet()`.

keyname (IN)
Keyname (IN).

valno (IN)
Which value to get for this key.

retval (OUT)
The actual integer value.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_NO_DATA,

OCI_ERROR.

OCI_NO_DATA means that there is no valno'th value for this key.

OCIExtractToBool()

Purpose

Gets the boolean value for the specified key. The valno'th value (starting with 0) is returned.

Syntax

```
sword OCIExtractToBool( dvoid    *hndl,
                        OCIError *err,
                        text     *keyname,
                        uword    valno,
                        ub1      *retval );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

keyname (IN)
Key name.

valno (IN)
Which value to get for this key.

retval (OUT)
The actual boolean value.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_NO_DATA,

OCI_ERROR.

OCI_NO_DATA means that there is no valno'th value for this key.

OCIExtractToStr()

Purpose

Gets the string value for the specified key. The valno'th value (starting with 0) is returned.

Syntax

```
sword OCIExtractToStr( dvoid *hndl,
                       OCIError *err,
                       text *keyname,
                       uword valno,
                       text *retval,
                       uword buflen );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

keyname (IN)
Key name.

valno (IN)
Which value to get for this key.

retval (OUT)
The actual null-terminated string value.

buflen
The length of the buffer for *retval*.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_NO_DATA,

OCI_ERROR.

OCI_NO_DATA means that there is no valno'th value for this key.

OCIExtractToOCINum()

Purpose

Gets the OCINumber value for the specified key. The valno'th value (starting with 0) is returned.

Syntax

```
sword OCIExtractToOCINum( dvoid    *hndl,
                          OCIError *err,
                          text     *keyname,
                          uword    valno,
                          OCINumber *retval );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling `OCIErrorGet()`.

keyname (IN)
Key name.

valno (IN)
Which value to get for this key.

retval (OUT)
The actual OCINumber value.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_NO_DATA, or OCI_ERROR.

OCI_NO_DATA means that there is no valno'th value for this key.

OCIExtractToList()

Purpose

Generates a list of parameters from the parameter structures that are stored in memory. Must be called before OCIExtractValues() is called.

Syntax

```
sword OCIExtractToList( dvoid     *hndl,
                        OCIError  *err,
                        uword      *numkeys );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

numkeys (OUT)
The number of distinct keys stored in memory.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIExtractFromList()

Purpose

Generates a list of values for the parameter denoted by index in the parameter list.

Syntax

```
sword OCIExtractFromList( dvoid      *hndl,
                          OCIError   *err,
                          uword      index,
                          text       **name,
                          ub1        *type,
                          uword      *numvals,
                          dvoid      ***values );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in `err` and this function returns OCI_ERROR; diagnostic information can be obtained by calling `OCIErrorGet()`.

index (IN)
Which parameter to retrieve from the parameter list.

name (OUT)
The name of the key for the current parameter.

type (OUT)
Type of the current parameter:

OCI_EXTRACT_TYPE_STRING,

OCI_EXTRACT_TYPE_INTEGER,

OCI_EXTRACT_TYPE_OCINUM,

OCI_EXTRACT_TYPE_BOOLEAN.

numvals (OUT)
Number of values for this parameter.

values (OUT)
The values for this parameter.

Comments

OCIExtractToList() must be called prior to calling this routine to generate the parameter list from the parameter structures that are stored in memory.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

Cartridge Services — File I/O Interface

Table 19–5   File I/O Interface Functions

Function/Page	Purpose
OCIFileInit() on page 19-40	Initializes the OCIFile package.
OCIFileTerm() on page 19-41	Terminates the OCIFile package.
OCIFileOpen() on page 19-42	Opens a file.
OCIFileClose() on page 19-44	Closes a previously opened file.
OCIFileRead() on page 19-45	Reads from a file into a buffer.
OCIFileWrite() on page 19-47	Writes *buflen* bytes into the file.
OCIFileSeek() on page 19-48	Changes the current position in a file.
OCIFileExists() on page 19-50	Tests to see if the file exists.
OCIFileGetLength() on page 19-51	Gets the length of a file.
OCIFileFlush() on page 19-52	Writes buffered data to a file.

See Also: For more information about using these functions, see
Oracle9i Data Cartridge Developer's Guide

OCIFileObject

The OCIFileObject data structure holds information about the way in which a file
should be opened and the way in which it will be accessed once it has been opened.
When this structure is initialized by OCIFileOpen(), it becomes an identifier
through which operations can be performed on that file. It is a necessary parameter
to every function that operates on open files. This data structure is opaque to
OCIFile clients. It is initialized by OCIFileOpen() and terminated by
OCIFileClose().

OCIFileInit()

Purpose

Initializes the OCIFile package. It must be called before any other OCIFile routine is called.

Syntax

```
sword OCIFileInit( dvoid    *hndl,
                   OCIError *err );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling `OCIErrorGet()`.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileTerm()

Purpose

Terminates the OCIFile package. It must be called after the OCIFile package is no longer being used.

Syntax

```
sword OCIFileTerm( dvoid    *hndl,
                   OCIError *err );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling `OCIErrorGet()`.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileOpen()

Purpose

Opens a file.

Syntax

```
sword OCIFileOpen( dvoid *hndl,
                   OCIError     *err,
                   OCIFileObject **filep,
                   OraText       *filename,
                   OraText       *path,
                   ub4           mode,
                   ub4           create,
                   ub4           type );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function
returns OCI_ERROR; diagnostic information can be obtained by calling
OCIErrorGet().

filep (IN/OUT)
The file identifier.

filename (IN)
The file name as a null-terminated string.

path (IN)
The path of the file as a null-terminated string.

mode (IN)
The mode in which to open the file. Valid modes are

OCI_FILE_READ_ONLY,

OCI_FILE_WRITE_ONLY,

OCI_FILE_READ_WRITE.

create (IN)
Indicates if the file be created if it does not exist — valid values are:

OCI_FILE_TRUNCATE — create a file regardless of whether or not it exists. If the file already exists overwrite the existing file.

OCI_FILE_EXCL — fail if the file exists, else create.

OCI_FILE_CREATE — open the file if it exists, and create it if it does not.

OCI_FILE_APPEND — set the file pointer to the end of the file prior to writing. This flag can be ORed with OCI_FILE_CREATE

type (IN)
File type. Valid values are

OCI_FILE_TEXT,

OCI_FILE_BIN,

OCI_FILE_STDIN,

 OCI_FILE_STDOUT,

OCI_FILE_STDERR.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileClose()

Purpose

Closes a previously opened file.

Syntax

```
sword OCIFileClose( dvoid        *hndl,
                    OCIError     *err,
                    OCIFileObject *filep );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling OCIErrorGet().

filep (IN/OUT)
A pointer to a file identifier to be closed.

Comments

Once this returns, the **OCIFileObject** structure pointed to by *filep* will have been destroyed. Therefore, you should not attempt to access this structure after this returns.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileRead()

Purpose

Reads from a file into a buffer.

Syntax

```
sword OCIFileRead( dvoid         *hndl,
                   OCIError       *err,
                   OCIFileObject *filep,
                   dvoid         *bufp,
                   ub4            bufl,
                   ub4           *bytesread );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in err and this function
returns OCI_ERROR; diagnostic information can be obtained by calling
OCIErrorGet().

filep (IN/OUT)
A file identifier that uniquely references the file.

bufp(IN)
The pointer to a buffer into which the data will be read. The length of the allocated
memory is assumed to be $bufl$.

bufl (IN)
The length of the buffer in bytes.

bytesread (OUT)
The number of bytes read.

Comments

As many bytes as possible will be read into the user buffer. The read will end either
when the user buffer is full, or when it reaches end-of-file.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileWrite()

Purpose

Writes *buflen* bytes into the file.

Syntax

```
sword OCIFileWrite( dvoid         *hndl,
                    OCIError      *err,
                    OCIFileObject *filep,
                    dvoid         *bufp,
                    ub4           buflen,
                    ub4           *byteswritten );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

filep (IN/OUT)
A file identifier that uniquely references the file.

bufp(IN)
The pointer to a buffer from into which the data will be written. The length of the allocated memory is assumed to be *buflen*.

buflen (IN)
The length of the buffer in bytes.

bytesread (OUT)
The number of bytes written.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileSeek()

Purpose

Changes the current position in a file.

Syntax

```
sword OCIFileSeek( dvoid         *hndl,
                   OCIError      *err,
                   OCIFileObject *filep,
                   uword         origin,
                   ubig_ora      offset,
                   sb1           dir );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

filep (IN/OUT)
A file identifier that uniquely references the file.

origin(IN)
The starting point we want to seek from. The starting point may be

 OCI_FILE_SEEK_BEGINNING (beginning),

OCI_FILE_SEEK_CURRENT (current position),

OCI_FILE_SEEK_END (end of file).

offset (IN)
The number of bytes from the origin you want to start reading from.

dir (IN)
The direction to go from the origin.

> **Note:**   The direction can be either OCIFILE_FORWARD or
> OCIFILE_BACKWARD.

Comments

This will allow a seek past the end of the file. Reading from such a position will
cause an end-of-file condition to be reported. Writing to such a position will not
work on all file systems. This is because some systems do not allow files to grow
dynamically. They require that files be preallocated with a fixed size. Note that this
function performs a seek to a byte location.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileExists()

Purpose

Tests to see if the file exists.

Syntax

```
sword OCIFileExists( dvoid   *hndl,
                     OCIError *err,
                     OraText *filename,
                     OraText *path,
                     ub1     *flag );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling
`OCIErrorGet()`.

filename (IN)
The file name as a null-terminated string.

path (IN)
The path of the file as a null-terminated string.

flag (OUT)
Set to TRUE if the file exists or FALSE if it does not.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileGetLength()

Purpose

Gets the length of a file.

Syntax

```
sword OCIFileGetLength( dvoid    *hndl,
                        OCIError *err,
                        OraText  *filename,
                        OraText  *path,
                        ubig_ora *lenp );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in `err` and this function
returns OCI_ERROR; diagnostic information can be obtained by calling
`OCIErrorGet()`.

filename (IN)
The file name as a null-terminated string.

path (IN)
The path of the file as a null-terminated string.

lenp (OUT)
Set to the length of the file in bytes.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFileFlush()

Purpose

Writes buffered data to a file.

Syntax

```
sword OCIFileFlush( dvoid         *h
                    OCIError      *err,
                    OCIFileObject *filep );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function
returns OCI_ERROR; diagnostic information can be obtained by calling
OCIErrorGet().

filep (IN/OUT)
A file identifier that uniquely references the file.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

Cartridge Services — String Formatting Interface

Table 19–6   String Formatting Functions

Function/Page	Purpose
OCIFormatInit() on page 19-54	Initializes the OCIFormat package.
OCIFormatTerm() on page 19-55	Terminates the OCIFormat package.
OCIFormatString() on page 19-56	Writes a text string into the supplied text buffer.

> **See Also:**   For more information about using these functions, see
> *Oracle9i Data Cartridge Developer's Guide*

OCIFormatInit()

Purpose

Initializes the OCIFormat package.

Syntax

```
sword OCIFormatInit( dvoid    *hndl,
                     OCIError *err)
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

Comments

This routine must be called before calling any other OCIFormat routine and it must only be called once.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFormatTerm()

Purpose

Terminates the OCIFormat package.

Syntax

```
sword OCIFormatTerm( dvoid    *hndl,
                     OCIError *err)
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in *err* and this function
returns OCI_ERROR; diagnostic information can be obtained by calling
`OCIErrorGet()`.

Comments

This function must be called after the OCIFormat package is no longer being used
and it must only be called once.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

OCIFormatString()

Purpose

Writes a text string into the supplied text buffer using the argument list submitted to it and in accordance with the format string given.

Syntax

```
sword OCIFormatString( dvoid       *hndl,
                       OCIError     *err,
                       text         *buffer,
                       sbig_ora     bufferLength,
                       sbig_ora    *returnLength,
                       CONST text   *formatString,... );
```

Parameters

hndl (IN)
The OCI environment or user session handle.

err (IN/OUT)
The OCI error handle; if there is an error, it is recorded in err and this function returns OCI_ERROR; diagnostic information can be obtained by calling OCIErrorGet().

buffer (OUT)
The buffer that contains the string.

bufferLength (IN)
The length of the buffer in bytes.

returnLength (OUT)
The number of bytes written to the buffer (excluding the terminating null).

formatString (IN)
The format string which can be any combination of literal text and format specifications. A format specification is delimited by the '%' character and is followed by any number (including none) of optional format modifiers and terminated by a mandatory format code. If the format string ends with '%', i.e. with no format modifiers or format specifier following it, then no action is taken. The format modifiers and format codes available are described in the tables that follow.

...(IN)

Variable number of arguments of the form <OCIFormat type wrapper>(<variable>) where <variable> must be a variable containing the value to be used. No constant values or expressions are allowed as arguments to the OCIFormat type wrappers; The OCIFormat type wrappers that are available are listed below. The argument list must be terminated with OCIFormatEnd.

```
OCIFormatUb1(ub1 variable);

OCIFormatUb2(ub2 variable);

OCIFormatUb4(ub4 variable);

OCIFormatUword(uword variable);

OCIFormatUbig_ora(ubig_ora variable);

OCIFormatSb1(sb1 variable);

OCIFormatSb2(sb2 variable);

OCIFormatSb4(sb4 variable);

OCIFormatSword(sword variable);

OCIFormatSbig_ora(sbig_ora variable);

OCIFormatEb1(eb1 variable);

OCIFormatEb2(eb2 variable);

OCIFormatEb4(eb4 variable);

OCIFormatEword(eword variable);

OCIFormatChar (text variable);

OCIFormatText(CONST text *variable);

OCIFormatDouble(double variable);

OCIFormatDvoid(CONST dvoid *variable);

OCIFormatEnd
```

Comments

The first call to this routine must be preceded by a call to the OCIFormatInit routine that initializes the OCIFormat package for use. When this routine is no longer needed terminate the OCIFormat package by a call to the OCIFormatTerm routine.

Returns

OCI_SUCCESS,

OCI_INVALID_HANDLE,

OCI_ERROR.

Format Modifiers

A format modifier alters or extends the format specification, allowing more specialized output. The format modifiers may be in any order and are all optional.

Flags (in any order)

Flag	Operation
'–'	left-justify the output in the field
'+'	always print a sign ('+' or '-') for numeric types
' '	if a number's sign is not printed then print a space in the sign position
'0'	pad numeric output with zeros not spaces

- If both the '+' and ' ' flags are used in the same format specification then the ' ' flag is ignored.

- If both the '–' and '0' flags are used in the same format specification then the '–' flag is ignored.

Alternate output:
- For the octal format code add a leading zero.
- For the hexadecimal format code add a leading '0x'.
- For floating point format codes the output will always have a radix character.

Field Width
<w> where <w> is a number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument takes up fewer display positions than the field width, it will be padded on the left (or right for left justification) to make up the field width. The padding character is normally a space, but it is a zero if the zero padding flag was specified. The special character '*' may be used in place of <w> and indicates the current argument is to be used for the field width value, the actual field or precision follows as the next sequential argument.

Precision

`.<p>` specifies a period followed by the number `<p>`, specifying the maximum number of display positions to print from a string, or digits after the radix point for a decimal number, or the minimum number of digits to print for an integer type (leading zeroes will be added to make up the difference). The special character '`*`' may be used in place of `<p>` indicating the current argument contains the precision value.

Argument Index

`(<n>)` where `<n>` is an integer index into the argument list with the first argument being 1. If no argument index is specified in a format specification the first argument is selected. The next time no argument index is specified in a format specification the second argument is selected and so on. Format specifications with and without argument indexes can be in any order and are independent of each other in operation.

For example, the format string "`%u %(4)u %u %(2)u %u`" selects the first, fourth, second, second, and third arguments given to `OCIFormatString()`.

Format Codes

A format code specifies how to format an argument that is being written to a string.

Note that these codes can appear in upper case, which will cause all alphabetic characters in the output to appear in upper case except for text strings, which are not converted.

Codes	Operation
'c'	single-byte character in the compiler character set
'd'	signed decimal integer
'e'	exponential (scientific) notation of the form [-]<d><r>[<d>...]e+[<d>]<d><d> where <r> is the radix character for the current language and <d> is any single digit; the default precision is given by the constant OCIFormatDP. the precision may be optionally specified as a format modifier - using a precision of 0 suppresses the radix character; the exponent is always printed in at least 2 digits, and can take up to 3 e.g. 1e+01, 1e+10, and 1e+100
'f'	fixed decimal notation of the form [-]<d>[<d>...]<r>[<d>...] where <r> is the appropriate radix character for the current language and <d> is any single digit; the precision may be optionally specified as a format modifier- using a precision of 0 suppresses the radix character. the default precision is given by the constant OCIFormatDP
'g'	variable floating-point notation; chooses 'e' or 'f', selecting 'f' if the number will fit in the specified precision (default precision if unspecified), and choosing 'e' only if exponential format will allow more significant digits to be printed; does not print a radix character if number has no fractional part
'i'	identical to 'd'
'o'	unsigned octal integer
'p'	platform specific pointer printout

Codes	Operation
's'	prints an argument using the default format code for its type:
	`ociformatub<n>`, `ociformatuword`, `ociformatubig_ora`, `ociformateb<n>`, and `ociformateword`.
	the format code used is 'u'.
	`ociformatsb<n>`, `ociformatsword`, and `ociformatsbig_ora`.
	the format code used is 'd'.
	`ociformatchar`
	the format code used is 'c'.
	`ociformattext`
	prints text until trailing null is found.
	`ociformatdouble`
	the format code used is 'g'.
	`ociformatdvoid`
	the format code used is 'p'.
	`' %' – print a '%'.`
'u'	unsigned decimal integer
'x'	unsigned hexadecimal integer

Example

```
/* This example shows the power of arbitrary argument    */
/* selection in the context of internationalization.  A  */
/* date is formatted in 2 different ways for 2 different  */
/* countries according to the format string yet the      */
/* argument list submitted to OCIFormatString remains    */
/* invariant.                                             */

text     buffer[255];
ub1      day, month, year;
OCIError *err;
dvoid    *hndl;

/* Set the date. */

day   = 10;
month = 3;
year  = 97;

/* Work out the date in United States' style: mm/dd/yy */
OCIFormatString(hndl, err,
                buffer, (sbig_ora)sizeof(buffer),
                (CONST text *)"%(2)02u/%(1)02u/%(3)02u",
                OCIFormatUb1(day),
                OCIFormatUb1(month),
                OCIFormatUb1(year),
                OCIFormatEnd);    /* Buffer is "03/10/97". */

/* Work out the date in New Zealand style: dd/mm/yy */
OCIFormatString(hndl, err,
                buffer, (sbig_ora)sizeof(buffer),
                (CONST text *)"%(1)02u/%(2)02u/%(3)02u",
                OCIFormatUb1(day),
                OCIFormatUb1(month),
                OCIFormatUb1(year),
                OCIFormatEnd);    /* Buffer is "10/03/97". */
```

20

OCI Any Type and Data Functions

This chapter describes the OCI Any Type and Data functions.

The following sections are included in this chapter:

- Introduction to Any Type and Data Interfaces
- OCI Type Interface Functions
- OCI Any Data Interface Functions
- OCI Any Data Set Interface Functions

Introduction to Any Type and Data Interfaces

This chapter describes the OCI datatype mapping and manipulation functions in detail.

> **See Also:** For more information about the functions listed in this chapter, refer to Any Type, AnyData and AnyDataSet Interfaces on page 11-29

The Function Syntax

The entries for each function contain the following information:

Purpose

A brief statement of the purpose of the function.

Syntax

The function declaration.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below:

Mode	Description
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.

Comments

Detailed information about the function if available. This may include restrictions on the use of the function, or other information that might be useful when using the function in an application. An optional section.

All the functions in this chapter are related to each other.

Function Return Values

The OCI Any Type and Data functions typically return one of the following values:

Table 20–1   Function Return Values

Return Value	Meaning
OCI_SUCCESS	The operation succeeded
OCI_ERROR	The operation failed. The specific error can be retrieved by calling OCIErrorGet() on the error handle passed to the function.
OCI_INVALID_HANDLE	The environment or error handle passed to the function is null.

See Also: For more information about return codes and error handling, see the section "Error Handling" on page 2-31

OCI Type Interface Functions

This section describes the Type Interface functions.

Table 20–2   Type Interface Functions

Function/Page	Purpose
OCITypeAddAttr() on page 20-5	Adds an attribute to an object type that was constructed earlier with typecode OCI_TYPECODE_OBJECT.
OCITypeBeginCreate() on page 20-6	Begins the construction process for a transient type. The type will be anonymous (no name).
OCITypeEndCreate() on page 20-8	Finishes construction of a type description. Subsequently, only access will be allowed.
OCITypeSetBuiltin() on page 20-9	Sets built-in type information. This call can be made only if the type has been constructed with a built-in typecode (OCI_TYPECODE_NUMBER, etc.)
OCITypeSetCollection() on page 20-10	Sets collection type information. This call can be made only if the type has been constructed with a collection typecode.

OCITypeAddAttr()

Purpose

Adds an attribute to an object type that was constructed earlier with typecode
OCI_TYPECODE_OBJECT.

Syntax

```
sword OCITypeAddAttr ( OCISvcCtx   *svchp,
                       OCIError    *errhp,
                       OCIType     *type,
                       CONST text  *a_name,
                       ub4         a_length,
                       OCIParam    *attr_info );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type (IN/OUT)
The type description that is being constructed.

a_name (IN)
Optional. The name of the attribute.

a_length (IN)
Optional. The length of attribute name.

attr_info (IN)
Information on the attribute. It is obtained by allocating an **OCIParam** parameter
handle and setting type information in the **OCIParam** using OCIAttrSet() calls.

OCITypeBeginCreate()

Purpose

Begins the construction process for a transient type. The type will be anonymous (no name).

Syntax

```
sword OCITypeBeginCreate ( OCISvcCtx    *svchp,
                           OCIError     *errhp,
                           OCITypeCode  tc,
                           OCIDuration  dur,
                           OCIType      **type );
```

Parameters

svchp (IN)
The OCI Service Context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

tc (IN)
The typecode for the type. The typecode could correspond to an object type or a built-in type.

Currently, the permissible values for User Defined Types are:

- OCI_TYPECODE_OBJECT for an Object Type (structured),

- OCI_TYPECODE_VARRAY for a VARRAY collection type or

- OCI_TYPECODE_TABLE for a nested table collection type.

For Object types, call `OCITypeAddAttr()` to add each of the attribute types. For Collection types, call `OCITypeSetCollection()`. Subsequently, call `OCITypeEndCreate()` to finish the creation process.

The permissible values for built-in type codes are specified in "Typecodes" on page 3-31. Additional information on built-in types (precision, scale for numbers, character set information for VARCHAR2s, etc.) if any, must be set with a subsequent call to `OCITypeSetBuiltin()`. Finally, you must use `OCITypeEndCreate()` to finish the creation process.

dur (IN)
The allocation duration for the type. One of the following:

- A user duration that was previously created. It can be created by using `OCIDurationBegin()`.

- A predefined duration, such as OCI_DURATION_SESSION.

type (OUT)
The **OCIType** (Type Descriptor) that is being constructed.

Comments

To create a persistent named type, use the SQL statement CREATE TYPE. Transient types have no identity. They are pure values.

OCITypeEndCreate()

Purpose

Finishes construction of a type description. Subsequently, only access will be allowed.

Syntax

```
sword OCITypeEndCreate ( OCISvcCtx  *svchp,
                         OCIError   *errhp,
                         OCIType    *type );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type (IN/OUT)
The type description that is being constructed.

OCITypeSetBuiltin()

Purpose

Sets built-in type information. This call can be made only if the type has been constructed with a built-in typecode (OCI_TYPECODE_NUMBER, etc.)

Syntax

```
sword OCITypeSetBuiltin ( OCISvcCtx   *svchp,
                          OCIError    *errhp,
                          OCIType     *type,
                          OCIParam    *builtin_info );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type (IN/OUT)
The type description that is being constructed.

builtin_info (IN)
Provides information on the built-in (precision, scale, character set, etc.) It is obtained by allocating an **OCIParam** parameter handle and setting type information in the **OCIParam** using OCIAttrSet() calls.

OCITypeSetCollection()

Purpose

Sets collection type information. This call can be made only if the type has been constructed with a collection typecode.

Syntax

```
sword OCITypeSetCollection ( OCISvcCtx    *svchp,
                             OCIError     *errhp,
                             OCIType      *type,
                             OCIParam     *collelem_info,
                             ub4          coll_count );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

type (IN/OUT)
The type descriptor that is being constructed.

collelem_info (IN)
collelem_info provides information about the collection element. It is obtained by allocating an **OCIParam** parameter handle and setting type information in the **OCIParam** using OCIAttrSet() calls.

coll_count (IN)
The count of elements in the collection. Pass 0 for a nested table (which is unbounded).

OCI Any Data Interface Functions

This section describes the Any Data Interface functions.

Table 20–3   Any Data Functions

Function/Page	Purpose
OCIAnyDataAccess() on page 20-12	Retrieves the data value of an **OCIAnyData**.
OCIAnyDataAttrGet() on page 20-14	Gets the value of the attribute at the current position in the **OCIAnyData**.
OCIAnyDataAttrSet() on page 20-17	Sets the attribute at the current position with a given value.
OCIAnyDataBeginCreate() on page 20-20	Allocates an **OCIAnyData** for the given duration and initializes it with the type information.
OCIAnyDataCollAddElem() on page 20-22	Adds the next collection element to the collection attribute of the **OCIAnyData** at the current attribute position.
OCIAnyDataCollGetElem() on page 20-24	Accesses sequentially the elements in the collection attribute at the current position in the **OCIAnyData**.
OCIAnyDataConvert() on page 20-26	Constructs an **OCIAnyData** with the given data value which will be of the given type.
OCIAnyDataDestroy() on page 20-28	Frees an **OCIAnyData**.
OCIAnyDataEndCreate() on page 20-29	Marks the end of **OCIAnyData** creation.
OCIAnyDataGetCurrAttrNum() on page 20-30	Returns the current attribute number of the **OCIAnyData**.
OCIAnyDataGetType() on page 20-31	Gets the type corresponding to an **OCIAnyData**.
OCIAnyDataIsNull() on page 20-32	Checks if **OCIAnyData** is null.

OCIAnyDataAccess()

Purpose

Retrieves the data value of an **OCIAnyData**. The data value should be of the type with which the **OCIAnyData** was initialized.This call can be used to access an entire **OCIAnyData** which can be of type OCI_TYPECODE_OBJECT, any of the collection types, or any of the built-in types.

Syntax

```
OCIAnyDataAccess ( OCISvcCtx    *svchp,
                   OCIError     *errhp,
                   OCIAnyData   *sdata,
                   OCITypeCode  tc,
                   OCIType      *inst_type,
                   dvoid        *null_ind,
                   dvoid        *data_value,
                   ub4          *length );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sdata (IN)
Initialized pointer to an **OCIAnyData**.

tc (IN)
Typecode of the data value. This is used for type checking (with the initialization type of the **OCIAnyData**).

inst_type (IN)
The **OCIType** of the data value (if it is not a primitive one). If the *tc* parameter is

- OCI_TYPECODE_OBJECT,

- OCI_TYPECODE_REF,

- OCI_TYPECODE_VARRAY,

- OCI_TYPECODE_TABLE,

then this parameter should be not null. Otherwise, it could be null.

null_ind (OUT)

Indicates if the `data_value` is null. Pass an `(OCIInd *)` for all typecodes except OCI_TYPECODE_OBJECT. The value returned will be OCI_IND_NOTNULL if the value is not null and it will be OCI_IND_NULL for a null value. If the typecode is OCI_TYPECODE_OBJECT, pass a pointer to the indicator struct of the `data_value` as the argument here. See `OCIAnyDataAttrGet()` for details.

data_value (OUT)

The data value (will be of the type with which the **OCIAnyData** was initialized). See `OCIAnyDataAttrGet()` for the appropriate C type corresponding to each allowed typecode and for a description of how memory allocation behavior depends on the value passed for this parameter.

length (OUT)

Currently, this parameter is ignored. In the future, this may be used for certain typecodes where the data representation itself will not give the length implicitly.

OCIAnyDataAttrGet()

Purpose

Gets the value of the attribute at the current position in the **OCIAnyData**. Attribute values can be accessed sequentially.

Syntax

```
sword OCIAnyDataAttrGet ( OCISvcCtx   *svchp,
                          OCIError    *errhp,
                          OCIAnyData  *sdata,
                          OCITypeCode tc,
                          OCIType     *attr_type,
                          dvoid       *null_ind,
                          dvoid       *attr_value,
                          ub4         *length,
                          boolean     is_any );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

sdata (IN/OUT)
Pointer to initialized of type **OCIAnyData**.

tc (IN)
Typecode of the attribute. Type checking happens based on `tc`, `attr_type` and the type information in the **OCIAnyData**.

attr_type (IN)
OPTIONAL

`attr_type` should give the type description of the referenced type (for OCI_TYPECODE_REF) or the type description of the collection type (for OCI_TYPECODE_VARRAY, OCI_TYPECODE_TABLE) or the type description of the object (for OCI_TYPECODE_OBJECT). This parameter is not required for built-in typecodes.

null_ind (OUT)
Indicates if the `attr_value` is null. Pass `(OCIInd *)` in `null_ind` for all typecodes except OCI_TYPECODE_OBJECT.

If the typecode is OCI_TYPECODE_OBJECT, pass a pointer (dvoid **) in *null_ind*.

The indicator returned will be OCI_IND_NOTNULL if the value is not null and it will be OCI_IND_NULL for a null value.

attr_value (IN/OUT)
Value for the attribute

length (IN/OUT)
Currently, this parameter is ignored. Pass 0 here. In the future, this may be used for certain typecodes where the data representation itself will not give the length implicitly.

is_any (IN)
Is attribute to be returned in the form of **OCIAnyData**?

Comments

This call can be used with **OCIAnyData** of typecode OCI_TYPECODE_OBJECT only

- This call gets the value of the attribute at the current position in the **OCIAnyData**.

- *tc* must match the type of the attribute at the current position, otherwise an error is returned.

- *is_any* is applicable only when the typecode of the attribute is one of the following:

    - OCI_TYPECODE_OBJECT,

    - OCI_TYPECODE_VARRAY,

    - OCI_TYPECODE_TABLE.

    If *is_any* is TRUE, then *attr_value* is returned in the form of **OCIAnyData***.

- You must allocate the memory for the attribute before calling the function. You can allocate memory through OCIObjectNew(). In case of built-in types such as NUMBER, VARCHAR, etc, the attribute can be just a pointer to a stack variable. Here is the list of available Oracle datatypes which can be used as object attribute types and the corresponding types of the attribute value that should be passed:

Datatypes	attr_value
VARCHAR2, VARCHAR, CHAR	**OCIString ****
NUMBER, REAL, INT, FLOAT, DECIMAL	**OCINumber ****
DATE	**OCIDate ****
TIMESTAMP	**OCIDateTime ****
TIMESTAMP WITH TIME ZONE	**OCIDateTime ****
TIMESTAMP WITH LOCAL TIME ZONE	**OCIDateTime ****
INTERVAL YEAR TO MONTH	**OCIInterval ****
INTERVAL DAY TO SECOND	**OCIInterval ****
BLOB	**OCILobLocator **** or **OCIBlobLocator ****
CLOB	**OCILobLocator **** or **OCIClobLocator ***
BFILE	**OCILobLocator ****
REF	**OCIRef ****
RAW	**OCIRaw ****
VARRAY	**OCIArray **** (or **OCIAnyData *** if *is_any* is TRUE)
TABLE	**OCITable **** (or **OCIAnyData *** if *is_any* is TRUE)
OBJECT	**dvoid **** (or **OCIAnyData *** if *is_any* is TRUE)

OCIAnyDataAttrSet()

Purpose

Sets the attribute at the current position with a given value.

Syntax

```
sword OCIAnyDataAttrSet ( OCISvcCtx    *svchp,
                          OCIError     *errhp,
                          OCIAnyData   *sdata,
                          OCITypeCode  tc,
                          OCIType      *attr_type,
                          dvoid        *null_ind,
                          dvoid        *attr_value,
                          ub4          length,
                          boolean      is_any );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sdata (IN/OUT)
Initialized **OCIAnyData**.

tc (IN)
Typecode of the attribute. Type checking happens based on *tc*, *attr_type* and
the type information in the **OCIAnyData**.

attr_type (IN)
OPTIONAL

attr_type will give the type description of the referenced type (for
OCI_TYPECODE_REF) and it will give the type description of the collection type
(for OCI_TYPECODE_VARRAY, OCI_TYPECODE_TABLE) and it will give the type
description of the object (for OCI_TYPECODE_OBJECT). This parameter is not
required for built-in typecodes or if OCI_TYPECODE_NONE is specified.

null_ind (IN)
Indicates if the *attr_value* is null. Pass (OCIInd *) for all typecodes except
OCI_TYPECODE_OBJECT. The indicator should be OCI_IND_NOTNULL if the
value is not null and it should be OCI_IND_NULL for a null value.

If the typecode is OCI_TYPECODE_OBJECT, pass a pointer to the indicator struct of the *attr_value* as the argument here.

attr_value (IN)
Value for the attribute

length (IN)
Currently, this parameter is ignored. Pass 0 here. In the future, this may be used for certain typecodes where the data representation itself will not give the length implicitly.

is_any (IN)
Is attribute in the form of **OCIAnyData**?

Comments

OCIAnyDataBeginCreate() creates an **OCIAnyData** with an empty skeleton instance. To fill the attribute values, use OCIAnyDataAttrSet() (for OCI_TYPECODE_OBJECT) or OCIAnyDataCollAttrAddElem() (for the collection typecodes).

Attribute values must be set in order, from the first attribute to the last. The current attribute number is remembered as state maintained inside the **OCIAnyData**. Piece-wise construction of embedded attributes and collection elements are not yet supported.

This call sets the attribute at the current position with *attr_value*. Once piece-wise construction has started for an **OCIAnyData** instance, the OCIAnyDataConstruct() calls can no longer be used.

tc must match the type of the attribute at the current position. Otherwise, an error is returned.

If *is_any* is TRUE, then the attribute must be in the form of OCIAnyData* and it is copied into the enclosing **OCIAnyData** (data) without any conversion.

Here is the list of available datatypes which can be used as object attribute types and the corresponding types of the attribute value that should be passed:

Datatypes	attr_value
VARCHAR2, VARCHAR, CHAR	**OCIString** *
NUMBER, REAL, INT, FLOAT, DECIMAL	**OCINumber** *
DATE	**OCIDate** *

Datatypes	attr_value
TIMESTAMP	**OCIDateTime** *
TIMESTAMP WITH TIME ZONE	**OCIDateTime** *
TIMESTAMP WITH LOCAL TIME ZONE	**OCIDateTime** *
INTERVAL YEAR TO MONTH	**OCIInterval** *
INTERVAL DAY TO SECOND	**OCIInterval** *
BLOB	**OCILobLocator** * or **OCIBlobLocator** *
CLOB	**OCILobLocator** * or **OCIClobLocator** *
BFILE	**OCILobLocator** *
REF	**OCIRef** *
RAW	**OCIRaw** *
VARRAY	**OCIArray** * (or **OCIAnyData** * if *is_any* is TRUE)
TABLE	**OCITable** * (or **OCIAnyData** * if *is_any* is TRUE)
OBJECT	**dvoid** * (or **OCIAnyData** * if *is_any* is TRUE)

OCIAnyDataBeginCreate()

Purpose

Allocates an **OCIAnyData** for the given duration and initializes it with the type information.

Syntax

```
sword OCIAnyDataBeginCreate ( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCITypeCode    tc,
                              OCIType        *type,
                              OCIDuration    dur,
                              OCIAnyData     **sdata );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sdata (IN/OUT)
Initialized **OCIAnyData**.

tc (IN)
Typecode corresponding to the **OCIAnyData**. Can be a built-in typecode or a user-defined type's typecode such as:

- OCI_TYPECODE_OBJECT,

- OCI_TYPECODE_REF,

- OCI_TYPECODE_VARRAY.

type (IN)
The type corresponding to **OCIAnyData.** If the typecode corresponds to a built-in type (OCI_TYPECODE_NUMBER, etc.), this parameter can be null. It should be non-null for user defined types (OCI_TYPECODE_OBJECT, OCI_TYPECODE_REF, collection types, etc.)

dur (IN)
Duration for which **OCIAnyData** is allocated. One of the following:

- A user duration that was previously created. It can be created by using OCIDurationBegin().

■   A predefined duration, such as OCI_DURATION_SESSION.

sdata (OUT)
Initialized **OCIAnyData**. If (*sdata) is not null at the beginning of the call, the memory could be reused instead of reallocating space for the **OCIAnyData**.

Therefore, do not pass an uninitialized pointer here.

Comments

OCIAnyDataBeginCreate() creates an **OCIAnyData** with an empty skeleton instance. To fill in the attribute values, use OCIAnyDataAttrSet() for OCI_TYPECODE_OBJECT, or OCIAnyDataCollAttrAddElem() for the collection typecodes.

Attribute values must be set in order. They must be set from the first attribute to the last one. The current attribute number is remembered as state maintained inside the **OCIAnyData**. Piece-wise construction of embedded attributes and collection elements are not yet supported.

For performance reasons, the **OCIAnyData** will end up pointing to the **OCIType** parameter passed in. It is your responsibility to ensure that the **OCIType** is longer lived (has allocation duration >= the duration of the **OCIAnyData**, if the **OCIType** is a transient one, or has allocation/pin duration >= duration of the **OCIAnyData,** if the **OCIType** is a persistent one).

OCIAnyDataCollAddElem()

Purpose

Adds the next collection element to the collection attribute of the **OCIAnyData** at the current attribute position. If the **OCIAnyData** is of a collection type, then there is no notion of attribute position and this call adds the next collection element.

Syntax

```
sword OCIAnyDataCollAddElem ( OCISvcCtx    *svchp,
                              OCIError     *errhp,
                              OCIAnyData   *sdata,
                              OCITypeCode  collelem_tc,
                              OCIType      *collelem_type,
                              dvoid        *null_ind,
                              dvoid        *elem_value,
                              ub4          length,
                              boolean      is_any,
                              boolean      last_elem );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

sdata (IN/OUT)
Initialized **OCIAnyData**.

collelem_tc (IN)
The typecode of the collection element to be added. Type checking happens based on `collelem_tc`, `collelem_type` and the type information in the **OCIAnyData**.

collelem_type (IN)
OPTIONAL

`collelem_type` will give the type description of the referenced type (for OCI_TYPECODE_REF) and it will give the type description of the collection type (for OCI_TYPECODE_NAMEDCOLLECTION) and it will give the type description of the object (for OCI_TYPECODE_OBJECT).

This parameter is not required for built-in typecodes.

null_ind (IN)
Indicates if the *elem_value* is null. Pass an (OCIInd *) for all typecodes except OCI_TYPECODE_OBJECT. The indicator should be OCI_IND_NOTNULL if the value is not null and it should be OCI_IND_NULL for a null value.

If the typecode is OCI_TYPECODE_OBJECT, pass a pointer to the indicator struct of the *elem_value* as the argument here.

elem_value (IN)
Value for the collection element

length (IN)
Length of the collection element

is_any (IN)
Is the attribute in the form of **OCIAnyData**?

last_elem (IN)
Is the element being added the last in the collection?

Comments

This call can be invoked for an **OCIAnyData** of type OCI_TYPECODE_OBJECT or of any of the collection types. Once piece-wise construction has started for an **OCIAnyData** instance, the OCIAnyDataConstruct() calls can no longer be used.

As in OCIAnyDataAttrSet(), *is_any* is applicable only if the *collelem_tc* is that of typecode OCI_TYPECODE_OBJECT or a collection typecode. If *is_any* is TRUE, the attribute should be in the form of OCIAnyData *.

If the element being added is the last element in the collection, *last_elem* should be set to TRUE.

To add a null element, the null indicator, *null_ind* should be set to OCI_IND_NULL, in which case all other arguments will be ignored. Otherwise, *null_ind* must be set to OCI_IND_NOTNULL.

See OCIAnyDataAttrSet() for the type of attribute to be passed in for all the possible types of the collection elements.

OCIAnyDataCollGetElem()

Purpose

Accesses sequentially the elements in the collection attribute at the current position in the **OCIAnyData**.

Syntax

```
sword OCIAnyDataCollGetElem ( OCISvcCtx    *svchp,
                              OCIError     *errhp,
                              OCIAnyData   *sdata,
                              OCITypeCode  collelem_tc,
                              OCIType      *collelem_type,
                              dvoid        *null_ind,
                              dvoid        *collelem_value,
                              ub4          *length,
                              boolean      is_any );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `err` and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

sdata (IN/OUT)
Initialized **OCIAnyData**.

collelem_tc (IN)
The typecode of the collection element to be retrieved. Type checking happens based on `collelem_tc`, `collelem_type` and the type information in the **OCIAnyData**.

collelem_type (IN)
OPTIONAL

`collelem_type` will give the type description of the referenced type (for OCI_TYPECODE_REF) and it will give the type description of the collection type (for OCI_TYPECODE_NAMEDCOLLECTION) and it will give the type description of the object (for OCI_TYPECODE_OBJECT).

This parameter is not required for built-in typecodes.

null_ind (OUT)

Indicates if the *collelem_value* is null. Pass an (OCIInd *) for all typecodes except OCI_TYPECODE_OBJECT. The indicator should be OCI_IND_NOTNULL if the value is not null and it should be OCI_IND_NULL for a null value.

If the typecode is OCI_TYPECODE_OBJECT, pass a pointer (dvoid **) to the indicator struct of the *collelem_value* as the argument here.

collelem_value (IN/OUT)

Value for the collection element

length (IN/OUT)

Length of the collection element. Currently ignored. Set to 0 on input.

is_any (IN)

Is *attr_value* to be returned in the form of **OCIAnyData**?

Comments

The **OCIAnyData** data can also correspond to a top level collection. If the **OCIAnyData** is of type OCI_TYPECODE_OBJECT, the attribute at the current position must be a collection of appropriate type. Otherwise, an error is returned.

As for OCIAnyDataAttrGet(), the *is_any* parameter is applicable only if *collelem_tc* typecode is that OCI_TYPECODE_OBJECT. If *is_any* is TRUE, the *attr_value* will be in the form of OCIAnyData *.

This call returns OCI_NO_DATA when the end of the collection has been reached. It returns OCI_SUCCESS upon success and OCI_ERROR upon error.

See OCIAnyDataAttrGet()) for the type of attribute to be passed in for all the possible types of the collection elements.

OCIAnyDataConvert()

Purpose

Constructs an **OCIAnyData** with the given data value which will be of the given type. This call can be used to construct an entire **OCIAnyData** which could be of type OCI_TYPECODE_OBJECT, any of the collection types, or any of the built-in types.

Syntax

```
OCIAnyDataConvert ( OCISvcCtx    *svchp,
                    OCIError     *errhp,
                    OCITypeCode  tc,
                    OCIType      *inst_type,
                    OCIDuration  dur,
                    dvoid        *null_ind,
                    dvoid        *data_value,
                    ub4          length,
                    OCIAnyData   **sdata );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

tc (IN)
Typecode of the data value. Can be a built-in typecode or a user-defined type's typecode (such as OCI_TYPECODE_OBJECT, OCI_TYPECODE_REF, OCI_TYPECODE_VARRAY).

If (*sdata*) is not null and it represents a skeleton instance returned during the OCIAnyDataSetAddInstance(), the *tc* as well as the *inst_type* parameters are optional here. This is because the type-information for such a skeleton instance is already known. If the *tc* and *inst_type* parameters are provided here for this situation, they will be used only for type-checking purposes.

inst_type (IN)
Type corresponding to the **OCIAnyData**. If the typecode corresponds to a built-in type (OCI_TYPECODE_NUMBER, etc.), this parameter can be null. It should not be null for user defined types (OCI_TYPECODE_OBJECT, OCI_TYPECODE_REF, or collection types).

dur (IN)
Duration for which the **OCIAnyData** is allocated. One of the following:

- A user duration that was previously created. It can be created by using
  OCIDurationBegin().

- A predefined duration, such as OCI_DURATION_SESSION.

null_ind
Indicates if *data_value* is null. Pass an (OCIInd *) for all typecodes except
OCI_TYPECODE_OBJECT. The indicator will be OCI_IND_NOTNULL if the value
is not null and it will be OCI_IND_NULL for a null value.

If the typecode is OCI_TYPECODE_OBJECT, pass a pointer to the indicator struct of
the *data_value* as the argument here.

data_value (IN)
The data value (should be of the type with which the **OCIAnyData** was initialized).
See OCIAnyDataAttrSet() for the appropriate C type corresponding to each
allowed typecode.

length (IN)
Currently, this parameter is ignored. Pass 0 here. In the future, this may be used for
certain typecodes where the data representation itself will not give the length
implicitly.

sdata (IN/OUT)
Initialized **OCIAnyData**. If (*sdata*) is not null at the beginning of the call, the
memory could bet reused instead of reallocating space for the OCIAnyData.

Therefore, do not pass an un-initialized pointer here.

If (*sdata*) represents a skeleton instance returned during an
OCIAnyDataSetAddInstance() call, the *tc* and *inst_type* parameters will be
used for type-checking purposes if necessary.

Comments

For performance reasons, the **OCIAnyData** pointer will end up pointing to the
passed in **OCIType** parameter. It is your responsibility to ensure that the **OCIType**
is longer lived (has allocation duration >= the duration of the **OCIAnyData**, if the
OCIType is a transient one, or has allocation/pin duration >= duration of the
OCIAnyData, if the **OCIType** is a persistent one).

OCIAnyDataDestroy()

Purpose

Frees an **OCIAnyData**.

Syntax

```
sword OCIAnyDataDestroy ( OCISvcCtx      *svchp,
                          OCIError       *errhp,
                          OCIAnyData     *sdata );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sdata (IN/OUT)
Pointer to an of type **OCIAnyData** to be freed.

OCIAnyDataEndCreate()

Purpose

Marks the end of **OCIAnyData** creation. It should be called after initializing all attributes of its instances with suitable values. This call is valid only if `OCIAnyDataBeginCreate()` had been called earlier for the **OCIAnyData**.

Syntax

```
sword OCIAnyDataEndCreate ( OCISvcCtx    *svchp,
                            OCIError     *errhp,
                            OCIAnyData   *data );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

data (IN/OUT)
Initialized **OCIAnyData**.

OCIAnyDataGetCurrAttrNum()

Purpose

Returns the current attribute number of the **OCIAnyData**. If the **OCIAnyData** is being constructed, it refers to the current attribute that is being set. Else, if the **OCIAnyData** is being accessed, it refers to the attribute that is being accessed.

Syntax

```
sword OCIAnyDataGetCurrAttrNum( OCISvcCtx        *svchp,
                                OCIError         *errhp,
                                OCIAnyData       *sdata,
                                ub4              *attrnum );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sdata (IN)
Initialized **OCIAnyData**.

attrnum (OUT)
The attribute number.

OCIAnyDataGetType()

Purpose

Gets the type corresponding to an **OCIAnyData**. It returns the actual pointer to the type maintained inside an **OCIAnyData**. No copying is done for performance reasons. You are responsible for not using this type once the **OCIAnyData** is freed (or its duration ends).

Syntax

```
sword OCIAnyDataGetType( OCISvcCtx      *svchp,
                         OCIError       *errhp,
                         OCIAnyData     *data,
                         OCITypeCode    *tc,
                         OCIType        **type );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

data (IN)
Initialized **OCIAnyData**.

tc (OUT)
The typecode corresponding to the **OCIAnyData**.

type (OUT)
The type corresponding to the **OCIAnyData**. This will be null if the **OCIAnyData** corresponds to a built-in type.

OCIAnyDataIsNull()

Purpose

Checks if **OCIAnyData** is null.

Syntax

```
sword OCIAnyDataIsNull ( OCISvcCtx        *svchp,
                         OCIError         *errhp,
                         CONST OCIAnyData *sdata,
                         boolean          *isNull) ;
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

sdata (IN)
OCIAnyData to be freed.

isNull (IN/OUT)
TRUE if null, else FALSE.

OCI Any Data Set Interface Functions

This section describes the Any Data Set Interface functions.

Table 20–4   Any Data Set Functions

Function/Page	Purpose
OCIAnyDataSetAddInstance() on page 20-34	Adds a new skeleton instance to the **OCIAnyDataSet** and all the attributes of the instance are set to null.
OCIAnyDataSetBeginCreate() on page 20-36	Allocates an **OCIAnyDataSet** for the given duration and initializes it with the type information.
OCIAnyDataSetDestroy() on page 20-38	Frees the **OCIAnyDataSet**.
OCIAnyDataSetEndCreate() on page 20-39	Marks the end of **OCIAnyDataSet** creation.
OCIAnyDataSetGetCount() on page 20-40	Gets the number of instances in the **OCIAnyDataSet**
OCIAnyDataSetGetInstance() on page 20-41	Returns the **OCIAnyData** corresponding to an instance at the current position and updates the current position.
OCIAnyDataSetGetType() on page 20-42	Gets the type corresponding to an **OCIAnyDataSet**.

OCIAnyDataSetAddInstance()

Purpose

Adds a new skeleton instance to the **OCIAnyDataSet** and all the attributes of the instance are set to null.

Syntax

```
sword OCIAnyDataSetAddInstance ( OCISvcCtx      *svchp,
                                 OCIError       *errhp,
                                 OCIAnyDataSet  *data_set,
                                 OCIAnyData     **data );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

data_set (IN/OUT)
OCIAnyDataSet to which a new instance is added.

data (IN/OUT)
OCIAnyData corresponding to the newly added instance. If (*data) is null, a new **OCIAnyData** will be allocated for the same duration as the **OCIAnyDataSet**. If (*data) is not null, it will be reused. This **OCIAnyData** can be subsequently constructed using the OCIAnyDataConvert() call or it can be constructed piece-wise using the OCIAnyDataAttrSet() or the OCIAnyDataCollAddElem() calls.

Comments

This call returns this skeleton instance through the **OCIAnyData** parameter which can be constructed subsequently by invoking the OCIAnyData API.

> **Note:**   No destruction of the old value is done here. It is your responsibility to destroy the old value pointed to by (`*data`) and set (`*data`) to a null pointer before beginning to make a sequence of these calls. No deep copying (of OCIType information or of the data part) is done in the returned **OCIAnyData**. This **OCIAnyData** cannot be used beyond the allocation duration of the **OCIAnyDataSet** (it is like a reference into the **OCIAnyDataSet**). The returned **OCIAnyData** can be reused on subsequent calls to this function, to sequentially add new data instances to the **OCIAnyDataSet**.

OCIAnyDataSetBeginCreate()

Purpose

Allocates an **OCIAnyDataSet** for the given duration and initializes it with the type information. The **OCIAnyDataSet** can hold multiple instances of the given type.

Syntax

```
sword OCIAnyDataSetBeginCreate ( OCISvcCtx     *svchp,
                                 OCIError      *errhp,
                                 OCITypeCode   typecode,
                                 CONST OCIType *type,
                                 OCIDuration   dur,
                                 OCIAnyDataSet **data_set );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

typecode (IN)
Typecode corresponding to the **OCIAnyDataSet**.

type (IN)
Type corresponding to the **OCIAnyDataSet**. If the typecode corresponds to a built-in type (OCI_TYPECODE_NUMBER, etc.), this parameter can be null. It should be non null for user defined types (OCI_TYPECODE_OBJECT, OCI_TYPECODE_REF, collection types, etc.)

dur (IN)
Duration for which **OCIAnyDataSet** is allocated. One of the following:

- A user duration that was previously created. It can be created by using OCIDurationBegin().

- A predefined duration, such as OCI_DURATION_SESSION.

data_set (OUT)
Initialized **OCIAnyDataSet**.

Comments

For performance reasons, the **OCIAnyDataSet** will end up pointing to the **OCIType** parameter passed in. It is your responsibility to ensure that the **OCIType** is longer

lived (has allocation duration >= the duration of the **OCIAnyData** if the **OCIType** is a transient one, or has allocation/pin duration >= duration of the **OCIAnyData,** if the **OCIType** is a persistent one).

OCIAnyDataSetDestroy()

Purpose

Frees the **OCIAnyDataSet**.

Syntax

```
sword OCIAnyDataSetDestroy ( OCISvcCtx      *svchp,
                             OCIError       *errhp,
                             OCIAnyDataSet  *data_set );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

data_set (IN/OUT)
OCIAnyDataSet to be freed.

OCIAnyDataSetEndCreate()

Purpose

Marks the end of **OCIAnyDataSet** creation. It should be called after constructing all of its instances.

Syntax

```
sword OCIAnyDataSetEndCreate ( OCISvcCtx     *svchp,
                               OCIError      *errhp,
                               OCIAnyDataSet *data_set );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling `OCIErrorGet()`.

data_set (IN/OUT)
Initialized **OCIAnyDataSet**.

OCIAnyDataSetGetCount()

Purpose

Gets the number of instances in the **OCIAnyDataSet**.

Syntax

```
sword OCIAnyDataSetGetCount( OCISvcCtx      *svchp,
                             OCIError       *errhp,
                             OCIAnyDataSet  *data_set,
                             ub4            *count );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

data_set (IN/OUT)
A well-formed **OCIAnyDataSet**.

count (OUT)
Number of instances in **OCIAnyDataSet**.

OCIAnyDataSetGetInstance()

Purpose

Returns the **OCIAnyData** corresponding to an instance at the current position and updates the current position.

Syntax

```
sword OCIAnyDataSetGetInstance ( OCISvcCtx     *svchp,
                                 OCIError      *errhp,
                                 OCIAnyDataSet *data_set,
                                 OCIAnyData    **data );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

data_set (IN/OUT)
A well-formed **OCIAnyDataSet**.

data (IN/OUT)
OCIAnyData corresponding to the instance. If (*data) is null, a new **OCIAnyData** will be allocated for same duration as the **OCIAnyDataSet**. If (*data) is not null, it will be reused.

Comments

Only sequential access to the instances in an **OCIAnyDataSet** is allowed. This call returns the **OCIAnyData** corresponding to an instance at the current position and updates the current position. Subsequently, the OCIAnyData access routines may be used to access the instance.

OCIAnyDataSetGetType()

Purpose

Gets the type corresponding to an **OCIAnyDataSet**.

Syntax

```
sword OCIAnyDataSetGetType ( OCISvcCtx      *svchp,
                             OCIError       *errhp,
                             OCIAnyDataSet  *data_set,
                             OCITypeCode    *tc,
                             OCIType        **type );
```

Parameters

svchp (IN)
The OCI service context.

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function
returns OCI_ERROR. Obtain diagnostic information by calling OCIErrorGet().

data_set (IN)
Initialized **OCIAnyDataSet**.

tc (OUT)
The typecode corresponding to the type of the **OCIAnyDataSet**.

type (OUT)
The type corresponding to the **OCIAnyDataSet**. This will be null if the
OCIAnyData corresponds to a built-in type.

Part IV

Appendixes

This part of the book contains the appendixes:

- Appendix A, "Handle and Descriptor Attributes", lists the attributes of the OCI handles.

- Appendix B, "OCI Demonstration Programs", lists important demonstration programs that provide code examples of OCI features.

- Appendix C, "OCI Function Server Round-trips", provides information about the server roundtrips required by most OCI functions.

A

Handle and Descriptor Attributes

This appendix describes the attributes for OCI handles and descriptors, which can be read with `OCIAttrGet()`, and modified with `OCIAttrSet()`.

- Conventions
- Environment Handle Attributes
- Error Handle Attributes
- Service Context Handle Attributes
- Server Handle Attributes
- User Session Handle Attributes
- Connection Pool Handle Attributes
- Transaction Handle Attributes
- Statement Handle Attributes
- Bind Handle Attributes
- Define Handle Attributes
- Describe Handle Attributes
- Parameter Descriptor Attributes
- LOB Locator Attributes
- Complex Object Attributes
- Advanced Queuing Descriptor Attributes
- Subscription Handle Attributes
- Direct Path Loading Handle Attributes

- Process Handle Attributes

Conventions

For each handle type, the attributes which can be read or changed are listed. Each attribute listing includes the following information:

Mode
The following modes are valid:

READ - the attribute can be read using `OCIAttrGet()`

WRITE - the attribute can be modified using `OCIAttrSet()`

READ/WRITE - the attribute can be read using `OCIAttrGet()`, and it can be modified using `OCIAttrSet()`.

Description
This is a description of the purpose of the attribute.

Attribute Datatype
This is the datatype of the attribute. If necessary, a distinction is made between the datatype for READ and WRITE modes.

Valid Values
In some cases, only certain values are allowed, and they are listed here.

Example
In some cases an example is included.

Environment Handle Attributes

OCI_ATTR_BIND_DN

Mode
READ/WRITE

Description
The login name (DN) to use when connecting to the LDAP server.

Attribute Datatype
 text *

OCI_ATTR_CACHE_ARRAYFLUSH

Mode
READ/WRITE

Description
When this attribute is set to TRUE, during OCICacheFlush() the objects that belong to the same table are flushed together, which can considerably improve performance. This mode should only be used when the order in which the objects are flushed is not important. During this mode it is not guaranteed that the order in which the objects are marked dirty is preserved.

> **See Also:** "Object Cache Parameters" on page 13-5 and "Flushing Changes to Server" on page 13-11

Attribute Datatype
boolean

OCI_ATTR_CACHE_MAX_SIZE

Mode
READ/WRITE

Description
Sets the maximum size (high watermark) for the client-side object cache as a percentage of the optimal size. The default value is 110% of the optimal size. The object cache uses the maximum and optimal values for freeing unused memory in the object cache.

> **See Also:** "Object Cache Parameters" on page 13-5

Attribute Datatype
ub4 *

OCI_ATTR_CACHE_OPT_SIZE

Mode
READ/WRITE

Description
Sets the optimal size for the client-side object cache in bytes. The default value is 8M bytes.

> **See Also:** "Object Cache Parameters" on page 13-5

Attribute Datatype
ub4 *

OCI_ATTR_ENV_CHARSET_ID

Mode
READ

Description
Local (client-side) character set ID. Users can update this setting only after creating the environment handle but before calling any other OCI functions. This restriction ensures the consistency among data and metadata in the same environment handle. In UTF-16 mode, an attempt to get this attribute is invalid.

Attribute Datatype
ub2 *

OCI_ATTR_ENV_NCHARSET_ID

Mode
READ

Description
Local (client-side) national character set ID. Users can update this setting only after creating the environment handle but before calling any other OCI functions. This restriction ensures the consistency among data and metadata in the same environment handle. In UTF-16 mode, an attempt to get this attribute is invalid.

Attribute Datatype
ub2 *

OCI_ATTR_ENV_UTF16

Mode
READ

Description
Encoding method is UTF-16. The value 1 means that the environment handle is created in UTF-16 mode, while 0 means that it is not. This mode can only be set by the call to OCIEnvCreate() and cannot be changed later.

Attribute Datatype
ub1 *

OCI_ATTR_LDAP_AUTH

Mode
READ/WRITE

Description
The authentication mode. The following are the valid values:

0x0:   No authentication; anonymous bind.

0x1:   Simple authentication; username/password authentication.

0x5:   SSL connection with no authentication.

0x6:   SSL: only server authentication required.

0x7:   SSL: both server authentication and client authentication are required.

0x8:   Authentication method will be determined at runtime.

Attribute Datatype
ub2

OCI_ATTR_LDAP_CRED

Mode
READ/WRITE

Description
If the authentication method is "simple authentication" (username/password authentication), then this attribute holds the password to use when connecting to the LDAP server.

Attribute Datatype
text *

OCI_ATTR_LDAP_CTX

Mode
READ/WRITE

Description
The administrative context of the client. This is usually the root of the Oracle RDBMS LDAP schema in the LDAP server.

Attribute Datatype
text *

OCI_ATTR_LDAP_HOST

Mode
READ/WRITE

Description
The name of the host on which the LDAP server runs.

Attribute Datatype
text *

OCI_ATTR_LDAP_PORT

Mode
READ/WRITE

Description
The port on which the LDAP server is listening.

Attribute Datatype
ub2

OCI_ATTR_OBJECT

Mode
READ

Description
Returns TRUE if the environment was initialized in object mode.

Attribute Datatype
boolean *

OCI_ATTR_PINOPTION

Mode
READ/WRITE

Description
This attribute sets the value of OCI_PIN_DEFAULT for the application associated with the environment handle.

For example, if OCI_ATTR_PINOPTION is set to OCI_PIN_RECENT, then if `OCIObjectPin()` is called with the `pin_option` parameter set to OCI_PIN_DEFAULT, then the object is pinned in OCI_PIN_RECENT mode.

Attribute Datatype
OCIPinOpt *

OCI_ATTR_ALLOC_DURATION

Mode
READ/WRITE

Description
This attribute sets the value of OCI_DURATION_DEFAULT for allocation durations for the application associated with the environment handle.

Attribute Datatype
OCIDuration *

OCI_ATTR_PIN_DURATION

Mode
READ/WRITE

Description
This attribute sets the value of OCI_DURATION_DEFAULT for pin durations for the application associated with the environment handle.

Attribute Datatype
OCIDuration *

OCI_ATTR_HEAPALLOC

Mode
READ

Description
The current size of the memory allocated from the environment handle. This may help you track where memory is being used most in an application.

Attribute Datatype
ub4 *

OCI_ATTR_OBJECT_NEWNOTNULL

Mode
READ/WRITE

Description
When this attribute is set to TRUE, newly created objects have non-NULL attributes.

> **See Also:** "Creating Objects" on page 10-33

Attribute Datatype
boolean *

OCI_ATTR_OBJECT_DETECTCHANGE

Mode
READ/WRITE

Description
When this attribute is set to TRUE, applications receive an ORA-08179 error when attempting to flush an object which has been modified in the server by another committed transaction.

> **See Also:** "Implementing Optimistic Locking" on page 13-14

Attribute Datatype
boolean *

OCI_ATTR_SHARED_HEAP_ALLOC

Mode
READ

Description
Returns the size of the memory currently allocated from the shared pool. This attribute works on any environment handle but the process must be initialized in shared mode to return a meaningful value. This attribute is read as follows:

```
ub4 heapsz = 0;
OCIAttrGet((dvoid *)envhp, (ub4)OCI_HTYPE_ENV,
          (dvoid *) &heapsz, (ub4 *) 0,
          (ub4)OCI_ATTR_SHARED_HEAPALLOC, errhp);
```

Attribute Datatype
ub4 *

OCI_ATTR_WALL_LOC

Mode
READ/WRITE

Description
If the authentication method is SSL authentication, this attribute contains the
location of the client wallet.

Attribute Datatype
text *

Error Handle Attributes

OCI_ATTR_DML_ROW_OFFSET

Mode
READ

Description
Returns the offset (into the DML array) at which the error occurred.

Attribute Datatype
ub4 *

Service Context Handle Attributes

OCI_ATTR_ENV

Mode
READ

Description
returns the environment context associated with the service context.

Attribute Datatype
OCIEnv **

OCI_ATTR_IN_V8_MODE

Mode
READ

Description
Allows you to determine whether an application has switched to Oracle release 7
mode (e.g., through an OCISvcCtxToLda() call). A nonzero (true) return value
indicates that the application is currently running in Oracle release 8 mode, a zero
(false) return value indicates that the application is currently running in Oracle
release 7 mode.

Attribute Datatype
ub1 *

Example
The following code sample shows how this parameter might be used:

```
in_v8_mode = 0;
OCIAttrGet ((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (ub1 *)&in_v8_mode,
                (ub4) 0, OCI_ATTR_IN_V8_MODE, errhp);
if (in_v8_mode)
    fprintf (stdout, "In V8 mode\n");
else
    fprintf (stdout, "In V7 mode\n");
```

OCI_ATTR_SERVER

Mode
READ/WRITE

Description

When read, returns the pointer to the server context attribute of the service context.

When changed, sets the server context attribute of the service context.

Attribute Datatype
OCIServer ** (READ) ⁄ **OCIServer** * (WRITE)

OCI_ATTR_SESSION

Mode
READ ⁄ WRITE

Description

When read, returns the pointer to the authentication context attribute of the service context.

When changed, sets the authentication context attribute of the service context.

Attribute Datatype
OCISession ** (READ) ⁄ **OCISession** * (WRITE)

OCI_ATTR_TRANS

Mode
READ ⁄ WRITE

Description

When read, returns the pointer to the transaction context attribute of the service context.

When changed, sets the transaction context attribute of the service context.

Attribute Datatype
OCITrans ** (READ) ⁄ **OCITrans** * (WRITE)

Server Handle Attributes

OCI_ATTR_ENV

Mode
READ

Description
Returns the environment context associated with the server context.

Attribute Datatype
OCIEnv **

OCI_ATTR_EXTERNAL_NAME

Mode
READ/WRITE

Description
The external name is the user-friendly global name stored in
`sys.props$.value$`, where name = 'GLOBAL_DB_NAME'. It is not guaranteed
to be unique unless all databases register their names with a network directory
service.

Database names can be exchanged with the server in case of distributed transaction
coordination. Server database names can only be accessed if the database is open at
the time the `OCISessionBegin()` call is issued.

Attribute Datatype
text ** (READ) / **text** * (WRITE)

OCI_ATTR_FOCBK

Mode
READ/WRITE

Description

> **See Also:** "Application Failover Callbacks" on page 9-35

Attribute Datatype
OCIFocbkStruct *

OCI_ATTR_INTERNAL_NAME

Mode
READ/WRITE

Description
Sets the client database name that will be recorded when performing global transactions. The name can be used by the DBA to track transactions that may be pending in a prepared state due to failures.

Attribute Datatype
text ** (READ) / **text** * (WRITE)

OCI_ATTR_IN_V8_MODE

Mode
READ

Description
Allows you to determine whether an application has switched to Oracle release 7 mode (e.g., through an OCISvcCtxToLda() call). A nonzero (TRUE) return value indicates that the application is currently running in Oracle release 8 mode, a zero (FALSE) return value indicates that the application is currently running in Oracle release 7 mode.

Attribute Datatype
ub1 *

OCI_ATTR_NONBLOCKING_MODE

Mode
READ/WRITE

Description
This attribute determines the blocking mode.

When read, the attribute value returns TRUE if the server context is in non-blocking mode. When set, it toggles the non-blocking mode attribute.

> **See Also:** "Non-Blocking Mode" on page 2-41

Attribute Datatype
ub1

OCI_ATTR_SERVER_GROUP

Mode
READ/WRITE

Description
An alpha-numeric string not exceeding 30 characters specifying the server group.

> **See Also:** "Password and Session Management" on page 8-11

Attribute Datatype
ub4

OCI_ATTR_SERVER_STATUS

Mode
READ

Description
Returns the current status of the server handle. Values are:

- OCI_SERVER_NORMAL - There is an active connection to the server. It means that the last call on the connection went through. There is no guarantee that the next call will go through.

- OCI_SERVER_NOT_CONNECTED - There is no connection to the server.

Attribute Datatype
ub4

Example
The following code sample shows how this parameter might be used:

```
ub4 serverStatus = 0
OCIAttrGet((dvoid *)srvhp, OCI_HTYPE_SERVER,
        (dvoid *)&serverStatus, (ub4 *)0, OCI_ATTR_SERVER_STATUS, errhp);
if (serverStatus == OCI_SERVER_NORMAL)
        printf("Connection is up.\n");
else if (serverStatus == OCI_SERVER_NOT_CONNECTED)
        printf("Connection is down.\n");
```

User Session Handle Attributes

OCI_ATTR_APPCTX_ATTR

Mode
WRITE

Description
Specifies a an attribute name of the externally initialized context.

Attribute Datatype
text *

OCI_ATTR_APPCTX_LIST

Mode
READ

Description
Gets the application context list descriptor for the session.

Attribute Datatype
text *

OCI_ATTR_APPCTX_NAME

Mode
WRITE

Description
Specifies the namespace of the externally initialized context.

Attribute Datatype
text *

OCI_ATTR_APPCTX_SIZE

Mode
WRITE

Description
Initializes the externally initialized context array size with the number of attributes.

Attribute Datatype
text *

OCI_ATTR_APPCTX_VALUE

Mode
WRITE

Description
Specifies a value of the externally initialized context.

Attribute Datatype
text *

OCI_ATTR_CERTIFICATE

Mode
WRITE

Description
Specifies the certificate of the client for use in proxy authentication.

Attribute Datatype
ub1 *

OCI_ATTR_CERTIFICATE_TYPE

Mode
WRITE

Description
Specifies the type for proxy authentication. If not specified, the default type of X.509 is used.

Attribute Datatype
text *

OCI_ATTR_CLIENT_IDENTIFIER

Mode
WRITE

Description
Specifies the user identifier in the session handle. Used for auditing light-weight users.

Attribute Datatype
text *

Example
```
OCIAttrSet(session, OCI_HTYPE_SESSION,(dvoid *)"appuser1",
          (ub4)strlen("appuser1"),OCI_ATTR_APPCTX_IDENTIFIER, error_handle) ;
```

OCI_ATTR_DISTINGUISHED_NAME

Mode
WRITE

Description
Specifies distinguished name of the client for use in proxy authentication.

Attribute Datatype
text *

OCI_ATTR_INITIAL_CLIENT_ROLES

Mode
WRITE

Description
Specifies the role or roles that the client is to initially possess when the application server connects to Oracle on its behalf.

Attribute Datatype
ub4

OCI_ATTR_MIGSESSION

Mode
READ/WRITE

Description
Specifies the session identified for the session handle. Allows you to clone a session from one environment to another, in the same process or between processes. These processes can be on the same machine or different machines. For a session to be cloned, the session must be authenticated as migratable.

> **See Also:** "Password and Session Management" on page 8-11

Attribute Datatype
ub1 *

Example
The following code sample shows how this attribute might be used:

```
OCIAttrSet ((dvoid *) authp, (ub4) OCI_HTYPE_SESSION, (dvoid *) mig_session,
           (ub4) sz, (ub4) OCI_ATTR_MIGSESSION, errhp);
```

OCI_ATTR_PASSWORD

Mode
WRITE

Description
Specifies a password to use for authentication.

Attribute Datatype
text *

OCI_ATTR_PROXY_CREDENTIALS

Mode
WRITE

Description
Specifies that the credentials of the application server are to be used for proxy authentication.

Attribute Datatype
ub4

OCI_ATTR_USERNAME

Mode
WRITE

Description
Specifies a username to use for authentication.

Attribute Datatype
text *

Connection Pool Handle Attributes

OCI_ATTR_CONN_TIMEOUT

Mode
READ/WRITE

Description
Connections idle for more than this time value (in seconds) are terminated, to maintain an optimum number of open connections.This attribute can be set dynamically. If this attribute is not set, the connections are never timed out.

Attribute Datatype
ub4 */ub4

OCI_ATTR_CONN_NOWAIT

Mode
READ/WRITE

Description
This attribute determines if retrial for a connection has to be done when all connections in the pool are found to be busy and the number of connections has already reached the maximum.

If this attribute is set, an error is thrown when all the connections are busy and no more connections can be opened. Otherwise the call waits till it gets a connection.

When read, the attribute value is returned as TRUE if it has been set.

Attribute Datatype
ub1 */ub1

OCI_ATTR_CONN_BUSY_COUNT

Mode
READ

Description
Returns the number of busy connections.

Attribute Datatype
ub4 *

OCI_ATTR_CONN_OPEN_COUNT

Mode
READ

Description
Returns the number of open connections.

Attribute Datatype
ub4 *

OCI_ATTR_CONN_MIN

Mode
READ

Description
Returns the number of minimum connections.

Attribute Datatype
ub4 *

OCI_ATTR_CONN_MAX

Mode
READ

Description
Returns the number of maximum connections.

Attribute Datatype
ub4 *

OCI_ATTR_CONN_INCR

Mode
READ

Description
Returns the connection increment parameter.

Attribute Datatype
ub4 *

Transaction Handle Attributes

OCI_ATTR_TRANS_NAME

Mode
READ∕WRITE

Description
Can be used to establish or read a text string which identifies a transaction. This is an alternative to using the XID to identify the transaction. The text string can be up to 64 bytes long.

Attribute Datatype
text ** (READ) ∕ **text** * (WRITE)

OCI_ATTR_TRANS_TIMEOUT

Mode
READ∕WRITE

Description
Can set or read a timeout value used at prepare time.

Attribute Datatype
ub4 * (READ) ∕ **ub4** (WRITE)

OCI_ATTR_XID

Mode
READ∕WRITE

Description
Can set or read an XID which identifies a transaction.

Attribute Datatype
XID ** (READ) ∕ **XID** * (WRITE)

Statement Handle Attributes

OCI_ATTR_CURRENT_POSITION

Mode
READ

Description
Indicates the current position in the result set. This attribute can only be retrieved. It cannot be set.

Attribute Datatype
ub4 *

OCI_ATTR_ENV

Mode
READ

Description
Returns the environment context associated with the statement.

Attribute Datatype
OCIEnv **

OCI_ATTR_NUM_DML_ERRORS

Mode
READ

Description
Returns the number of errors in the DML operation.

Attribute Datatype
ub4 *

OCI_ATTR_PARAM_COUNT

Mode
READ

Description
This attribute can be used to get the number of columns in the select-list for the statement associated with the statement handle.

Attribute Datatype
ub4 *

Example
The following code sample shows how this attribute might be used:

```
/* Describe of a select-list */
text *selstmt  = "SELECT * FROM EMP";
ub4 parmcnt;
OCIParam *parmdp;

err = OCIStmtPrepare (stmhp, errhp, selstmt,
        (ub4)strlen((char *)selstmt),
          (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
err = OCIStmtExecute (svchp, stmhp, errhp, (ub4)1, (ub4)0,
          (const OCISnapshot*) 0, (OCISnapshot*)0, OCI_DESCRIBE_ONLY);

/* get the number of columns in the select list */
err = OCIAttrGet ((dvoid *)stmhp, (ub4)OCI_HTYPE_STMT, (dvoid *)
        &parmcnt, (ub4 *) 0, (ub4)OCI_ATTR_PARAM_COUNT, errhp);

/* get describe information for each column */
for (i = 0; i < parmcnt; i++) {
    OCIParamGet (dvoid *)stmhp, OCI_HTYPE_STMT, errhp, &parmdp, i);
/* get the attributes for each column */
        }
```

OCI_ATTR_PARSE_ERROR_OFFSET

Mode
READ

Description
Returns the parse error offset for a statement.

Attribute Datatype
ub2 *

OCI_ATTR_PREFETCH_MEMORY

Mode
WRITE

Description
Sets the memory level for top level rows to be prefetched. Rows up to the specified
top level row count are fetched if it occupies no more than the specified memory

usage limit. The default value is 0, which means that memory size is not included in computing the number of rows to prefetch.

Attribute Datatype
ub4 *

OCI_ATTR_PREFETCH_ROWS

Mode
WRITE

Description
Sets the number of top level rows to be prefetched. The default value is 1 row.

Attribute Datatype
ub4 *

OCI_ATTR_ROW_COUNT

Mode
READ

Description
Returns the number of rows processed so far. The default value is 1.

For non-scrollable cursors, OCI_ATTR_ROW_COUNT is the total number of rows fetched into user buffers with the `OCIStmtFetch2()` calls issued since this statement handle was executed. Since they are forward sequential only, this also represents the highest row number seen by the application.

For scrollable cursors, OCI_ATTR_ROW_COUNT will represent the maximum (absolute) row number fetched into the user buffers. Since the application can arbitrarily position the fetches, this need not be the total number of rows fetched into the user's buffers since the (scrollable) statement was executed.

Attribute Datatype
ub4 *

OCI_ATTR_ROWID

Mode
READ

Description
Returns the `ROWID` descriptor allocated with `OCIDescriptorAlloc()`.

See Also: "Positioned Updates and Deletes" on page 2-39 and "DATE" on page 3-15

Attribute Datatype
OCIRowid *

OCI_ATTR_ROWS_FETCHED

Mode
READ

Description
Indicates the number of rows that were successfully fetched into the user's buffers in the last fetch or execute with nonzero iterations. It can be used for both scrollable and non-scrollable statement handles.

Attribute Datatype
ub4 *

Example
```
ub4 rows;
ub4 sizep = sizeof(ub4);
OCIAttrGet((dvoid *) stmhp, (ub4) OCI_HTYPE_STMT,
          (dvoid *)& rows, (ub4 *) &sizep, (ub4)OCI_ATTR_ROWS_FETCHED, errhp));
```

OCI_ATTR_SQLFNCODE

Mode
READ

Description
Returns the function code of the SQL command associated with the statement.

Attribute Datatype
ub2 *

Notes

See Also: The SQL command codes are listed in Table A–1, "SQL Command Codes" on page A-28

Table A–1   SQL Command Codes

Code	SQL Function	Code	SQL Function	Code	SQL Function
01	CREATE TABLE	43	DROP EXTERNAL DATABASE	85	TRUNCATE TABLE
02	SET ROLE	44	CREATE DATABASE	86	TRUNCATE CLUSTER
03	INSERT	45	ALTER DATABASE	87	CREATE BITMAPFILE
04	SELECT	46	CREATE ROLLBACK SEGMENT	88	ALTER VIEW
05	UPDATE	47	ALTER ROLLBACK SEGMENT	89	DROP BITMAPFILE
06	DROP ROLE	48	DROP ROLLBACK SEGMENT	90	SET CONSTRAINTS
07	DROP VIEW	49	CREATE TABLESPACE	91	CREATE FUNCTION
08	DROP TABLE	50	ALTER TABLESPACE	92	ALTER FUNCTION
09	DELETE	51	DROP TABLESPACE	93	DROP FUNCTION
10	CREATE VIEW	52	ALTER SESSION	94	CREATE PACKAGE
11	DROP USER	53	ALTER USER	95	ALTER PACKAGE
12	CREATE ROLE	54	COMMIT (WORK)	96	DROP PACKAGE
13	CREATE SEQUENCE	55	ROLLBACK	97	CREATE PACKAGE BODY
14	ALTER SEQUENCE	56	SAVEPOINT	98	ALTER PACKAGE BODY
15	(NOT USED)	57	CREATE CONTROL FILE	99	DROP PACKAGE BODY
16	DROP SEQUENCE	58	ALTER TRACING	157	CREATE DIRECTORY
17	CREATE SCHEMA	59	CREATE TRIGGER	158	DROP DIRECTORY
18	CREATE CLUSTER	60	ALTER TRIGGER	159	CREATE LIBRARY
19	CREATE USER	61	DROP TRIGGER	160	CREATE JAVA
20	CREATE INDEX	62	ANALYZE TABLE	161	ALTER JAVA
21	DROP INDEX	63	ANALYZE INDEX	162	DROP JAVA
22	DROP CLUSTER	64	ANALYZE CLUSTER	163	CREATE OPERATOR
23	VALIDATE INDEX	65	CREATE PROFILE	164	CREATE INDEXTYPE
24	CREATE PROCEDURE	66	DROP PROFILE	165	DROP INDEXTYPE

Table A–1   SQL Command Codes (Cont.)

Code	SQL Function	Code	SQL Function	Code	SQL Function
25	ALTER PROCEDURE	67	ALTER PROFILE	166	ALTER INDEXTYPE
26	ALTER TABLE	68	DROP PROCEDURE	167	DROP OPERATOR
27	EXPLAIN	69	(NOT USED)	168	ASSOCIATE STATISTICS
28	GRANT	70	ALTER RESOURCE COST	169	DISASSOCIATE STATISTICS
29	REVOKE	71	CREATE SNAPSHOT LOG	170	CALL METHOD
30	CREATE SYNONYM	72	ALTER SNAPSHOT LOG	171	CREATE SUMMARY
31	DROP SYNONYM	73	DROP SNAPSHOT LOG	172	ALTER SUMMARY
32	ALTER SYSTEM SWITCH LOG	74	CREATE SNAPSHOT	73	DROP SUMMARY
33	SET TRANSACTION	75	ALTER SNAPSHOT	174	CREATE DIMENSION
34	PL/SQL EXECUTE	76	DROP SNAPSHOT	175	ALTER DIMENSION
35	LOCK	77	CREATE TYPE	176	DROP DIMENSION
36	NOOP	78	DROP TYPE	177	CREATE CONTEXT
37	RENAME	79	ALTER ROLE	178	DROP CONTEXT
38	COMMENT	80	ALTER TYPE	179	ALTER OUTLINE
39	AUDIT	81	CREATE TYPE BODY	180	CREATE OUTLINE
40	NO AUDIT	82	ALTER TYPE BODY	181	DROP OUTLINE
41	ALTER INDEX	83	DROP TYPE BODY	182	UPDATE INDEXES
42	CREATE EXTERNAL DATABASE	84	DROP LIBRARY	183	ALTER OPERATOR

OCI_ATTR_STATEMENT

Mode
READ

Description
Returns the text of the SQL statement prepared in a statement handle. In UTF-16 mode, the returned statement is in UTF-16 encoding. The length is always in bytes.

Attribute Datatype
text *

OCI_ATTR_STMT_STATE

Mode
READ

Description
Returns the fetch state of that statement. This attribute can be used by the caller to
determine if the session can be used in another service context or if it is still needed
in the current set of data access calls. Basically, if we are in the middle of a
fetch-execute cycle, then we do not want to release the session handle for another
statement execution. Valid values are:

- OCI_STMT_STATE_INITIALIZED

- OCI_STMT_STATE_EXECUTED

- OCI_STMT_STATE_END_OF_FETCH

Attribute Datatype
ub4 *

OCI_ATTR_STMT_TYPE

Mode
READ

Description
The type of statement associated with the handle. Valid values are:

- OCI_STMT_SELECT

- OCI_STMT_UPDATE

- OCI_STMT_DELETE

- OCI_STMT_INSERT

- OCI_STMT_CREATE

- OCI_STMT_DROP

- OCI_STMT_ALTER

- OCI_STMT_BEGIN (PL/SQL statement)

- OCI_STMT_DECLARE (PL/SQL statement)

Attribute Datatype
ub2 *

Bind Handle Attributes

OCI_ATTR_CHAR_COUNT

Mode
WRITE

Description

> **See Also:** "Buffer Expansion During Binding" on page 5-38

Attribute Datatype
ub4 *

OCI_ATTR_CHARSET_FORM

Mode
READ/WRITE

Description
Character set form of the bind handle. The default form is SQLCS_IMPLICIT.
Setting this attribute will cause the bind handle to use the database or national
character set on the client side. Set this attribute to SQLCS_NCHAR for the national
character set or SQLCS_IMPLICIT for the database character set.

Attribute Datatype
ub1 *

OCI_ATTR_CHARSET_ID

Mode
READ/WRITE

Description
Character set ID of the bind handle. If the character set of the input data is UTF-16
(replaces the deprecated OCI_UCS2ID, which is retained for backward
compatibility), the user has to set the character set ID to OCI_UTF16ID. The bind
value buffer is assumed to be a **utext** buffer and length semantics for input length
pointers and return values changes to character semantics (number of utexts).
However the size of the bind value buffer in the preceeding OCIBind call has to be
stated in bytes.

If OCI_ATTR_CHARSET_FORM is set, then OCI_ATTR_CHARSET_ID should be set only afterwards. Setting OCI_ATTR_CHARSET_ID prior to setting OCI_ATTR_CHARSET_FORM will cause unexpected results.

> **See Also:** "Character Conversion Issues in Binding and Defining" on page 5-35

Attribute Datatype
ub2 *

OCI_ATTR_MAXCHAR_SIZE

Mode
WRITE

Description
Sets the number of characters that an application reserves on the server to store the data being bound.

> **See Also:** "Using OCI_ATTR_MAXCHAR_SIZE Attribute" on page 5-37

Attribute Datatype
sb4 *

OCI_ATTR_MAXDATA_SIZE

Mode
READ/WRITE

Description

> **See Also:** "Using OCI_ATTR_MAXDATA_SIZE Attribute" on page 5-36

Attribute Datatype
sb4 *

OCI_ATTR_PDPRC

Mode
WRITE

Description

Specifies packed decimal precision. For SQLT_PDN values, the precision should be equal to **2*(value_sz-1)**. For SQLT_SLS values, the precision should be equal to **(value_sz-1)**.

After a bind or define, this value is initialized to zero. The OCI_ATTR_PDPRC attribute should be set first, followed by OCI_ATTR_PDSCL. If either of these values needs to be changed, a rebind/redefine should be done first, and then the two attributes should be reset in order.

Attribute Datatype
ub2 *

OCI_ATTR_PDSCL

Mode
WRITE

Description

Specifies the scale for packed decimal values.

After a bind or define, this value is initialized to zero. The OCI_ATTR_PDPRC attribute should be set first, followed by OCI_ATTR_PDSCL. If either of these values needs to be changed, a rebind/redefine should be done first, and then the two attributes should be reset in order.

Attribute Datatype
sb2 *

OCI_ATTR_ROWS_RETURNED

Mode
READ

Description

This attribute returns the number of rows that are going to be returned in the current iteration when we are in the OUT callback function for binding a DML statement with RETURNING clause.

Attribute Datatype
ub4 *

Define Handle Attributes

OCI_ATTR_CHAR_COUNT

Mode
WRITE

Description
This attribute is deprecated.

Sets the number of characters in a character type data. This specifies the number of characters desired in the define buffer. The define buffer length as specified in the define call must be greater than number of characters.

Attribute Datatype
ub4 *

OCI_ATTR_CHARSET_FORM

Mode
READ/WRITE

Description
The character set form of the define handle. The default form is SQLCS_IMPLICIT. Setting this attribute will cause the define handle to use the database or national character set on the client side. Set this attribute to SQLCS_NCHAR for the national character set or SQLCS_IMPLICIT for the database character set.

Attribute Datatype
ub1 *

OCI_ATTR_CHARSET_ID

Mode
READ/WRITE

Description
The character set ID of the define handle. If the character set of the output data should be UTF-16, the user has to set the character set IDOTT to OCI_UTF16ID. The define value buffer is assumed to be a utext buffer and length semantics for indicators and return values changes to character semantics (number of utexts). However the size of the define value buffer in the preceeding `OCIDefine` call has to be stated in bytes.

If OCI_ATTR_CHARSET_FORM is set, then OCI_ATTR_CHARSET_ID should be set only afterwards. Setting OCI_ATTR_CHARSET_ID prior to setting OCI_ATTR_CHARSET_FORM will cause unexpected results.

> **See Also:** "Character Conversion Issues in Binding and Defining" on page 5-35

Attribute Datatype
ub2 *

OCI_ATTR_MAXCHAR_SIZE

Mode
WRITE

Description
Specifies the maximum number of characters the client application allows in the define buffer.

> **See Also:** "Using OCI_ATTR_MAXCHAR_SIZE Attribute" on page 5-37

Attribute Datatype
sb4 *

OCI_ATTR_PDPRC

Mode
WRITE

Description
Specifies packed decimal precision. For SQLT_PDN values, the precision should be equal to 2*(value_sz-1). For SQLT_SLS values, the precision should be equal to (value_sz-1).

After a bind or define, this value is initialized to zero. The OCI_ATTR_PDPRC attribute should be set first, followed by OCI_ATTR_PDSCL. If either of these values needs to be changed, a rebind/redefine should be done first, and then the two attributes should be reset in order.

Attribute Datatype
ub2 *

OCI_ATTR_PDSCL

Mode
WRITE

Description
Specifies the scale for packed decimal values.

After a bind or define, this value is initialized to zero. The OCI_ATTR_PDPRC attribute should be set first, followed by OCI_ATTR_PDSCL. If either of these values needs to be changed, a rebind/redefine should be done first, and then the two attributes should be reset in order.

Attribute Datatype
sb2 *

Describe Handle Attributes

OCI_ATTR_PARAM

Mode
READ

Description
Points to the root of the description. Used for subsequent calls to OCIAttrGet()
and OCIParamGet().

Attribute Datatype
ub4 *

OCI_ATTR_PARAM_COUNT

Mode
READ

Description
Returns the number of parameters in the describe handle. When the describe
handle is a description of the select list, this refers to the number of columns in the
select list.

Attribute Datatype
ub4 *

Parameter Descriptor Attributes

> **See Also:** For a detailed list of parameter descriptor attributes,
> refer to Chapter 6, "Describing Schema Metadata"

LOB Locator Attributes

OCI_ATTR_LOBEMPTY

Mode
WRITE

Description
Sets the internal LOB locator to empty. The locator can then be used as a bind variable for an `INSERT` or `UPDATE` statement to initialize the LOB to empty. Once the LOB is empty, `OCILobWrite()` can be called to populate the LOB with data. This attribute is only valid for internal LOBs (i.e., `BLOB`, `CLOB`, `NCLOB`).

Applications should pass address of a **ub4** which has a value of 0; e.g., declare

```
ub4 lobEmpty = 0
```

then pass address `&lobEmpty`.

Attribute Datatype
ub4 *

Complex Object Attributes

See Also:

Complex Object Retrieval Handle Attributes

OCI_ATTR_COMPLEXOBJECT_LEVEL

Mode
WRITE

Description
The depth level for complex object retrieval.

Attribute Datatype
ub4 *

OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE

Mode
WRITE

Description
Whether to fetch collection attributes in an object type out-of-line.

Attribute Datatype
ub1 *

Complex Object Retrieval Descriptor Attributes

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE

Mode
WRITE

Description
A type of REF to follow for complex object retrieval.

Attribute Datatype
dvoid *

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL

Mode
WRITE

Description
Depth level for following REFs of type
OCI_ATTR_COMPLEXOBJECTCOMP_TYPE.

Attribute Datatype
ub4 *

Advanced Queuing Descriptor Attributes

See Also: For more information about Advanced Queueing properties and options, see *Oracle9i Application Developer's Guide - Advanced Queuing*

OCIAQEnqOptions Descriptor Attributes

The following attributes are properties of the **OCIAQEnqOptions** descriptor:

OCI_ATTR_RELATIVE_MSGID

Mode
READ/WRITE

Description
Specifies the message identifier of the message which is referenced in the sequence deviation operation. This value is valid if and only if OCI_ENQ_BEFORE is specified in OCI_ATTR_SEQUENCE_DIVISION. This value is ignored if the sequence deviation is not specified.

Attribute Datatype
OCIRaw *

OCI_ATTR_SEQUENCE_DEVIATION

Mode
READ/WRITE

Description
Specifies whether the message being enqueued should be dequeued before other message(s) already in the queue.

Attribute Datatype
ub4

Valid Values
The only valid values are:

- OCI_ENQ_BEFORE - the message is enqueued ahead of the message specified by OCI_ATTR_RELATIVE_MSGID.

- OCI_ENQ_TOP - the message is enqueued ahead of any other messages.

OCI_ATTR_VISIBILITY

Mode
READ/WRITE

Description
Specifies the transactional behavior of the enqueue request.

Attribute Datatype
ub4

Valid Values
The only valid values are:

- OCI_ENQ_ON_COMMIT - the enqueue is part of the current transaction. The operation is complete when the transaction commits. This is the default case.

- OCI_ENQ_IMMEDIATE - the enqueue is not part of the current transaction. The operation constitutes a transaction of its own.

OCIAQDeqOptions Descriptor Attributes

The following attributes are properties of the **OCIAQDeqOptions** descriptor:

OCI_ATTR_CONSUMER_NAME

Mode
READ/WRITE

Description
Name of the consumer. Only those messages matching the consumer name are accessed. If a queue is not set up for multiple consumers, this field should be set to `null`.

Attribute Datatype
text *

OCI_ATTR_CORRELATION

Mode
READ/WRITE

Description
Specifies the correlation identifier of the message to be dequeued. Special pattern matching characters, such as the percent sign (%) and the underscore (_) can be

used. If more than one message satisfies the pattern, the order of dequeuing is undetermined.

Attribute Datatype
text *

OCI_ATTR_DEQ_MODE

Mode
READ/WRITE

Description
Specifies the locking behavior associated with the dequeue.

Attribute Datatype
ub4

Valid Values
The only valid values are:

- OCI_DEQ_BROWSE - read the message without acquiring any lock on the message. This is equivalent to a SELECT statement.

- OCI_DEQ_LOCKED - read and obtain a write lock on the message. The lock lasts for the duration of the transaction. This is equivalent to a SELECT FOR UPDATE statement.

- OCI_DEQ_REMOVE - read the message and update or delete it. This is the default. The message can be retained in the queue table based on the retention properties.

- OCI_DEQ_NO_DATA - confirm receipt of the message, but do not deliver the actual message content.

OCI_ATTR_DEQ_MSGID

Mode
READ/WRITE

Description
Specifies the message identifier of the message to be dequeued.

Attribute Datatype
OCIRaw *

OCI_ATTR_NAVIGATION

Mode
READ/WRITE

Description
Specifies the position of the message that will be retrieved. First, the position is determined. Second, the search criterion is applied. Finally, the message is retrieved.

Attribute Datatype
ub4

Valid Values
The only valid values are:

- OCI_DEQ_FIRST_MSG - retrieves the first message which is available and matches the search criteria. This will reset the position to the beginning of the queue.

- OCI_DEQ_NEXT_MSG - retrieves the next message which is available and matches the search criteria. If the previous message belongs to a message group, AQ will retrieve the next available message which matches the search criteria and belongs to the message group. This is the default.

- OCI_DEQ_NEXT_TRANSACTION - skips the remainder of the current transaction group (if any) and retrieves the first message of the next transaction group. This option can only be used if message grouping is enabled for the current queue.

OCI_ATTR_VISIBILITY

Mode
READ/WRITE

Description
Specifies whether the new message is dequeued as part of the current transaction.The visibility parameter is ignored when using the BROWSE mode.

Attribute Datatype
ub4

Valid Values
The only valid values are:

- OCI_DEQ_ON_COMMIT - the dequeue will be part of the current transaction. This is the default case.

■ OCI_DEQ_IMMEDIATE - the dequeued message is not part of the current transaction. It constitutes a transaction on its own.

OCI_ATTR_WAIT

Mode
READ/WRITE

Description
Specifies the wait time if there is currently no message available which matches the search criteria. This parameter is ignored if messages in the same group are being dequeued.

Attribute Datatype
ub4

Valid Values
Any **ub4** value is valid, but the following predefined constants are provided:

■ OCI_DEQ_WAIT_FOREVER - wait forever. This is the default.

■ OCI_DEQ_NO_WAIT - do not wait.

OCIAQMsgProperties Descriptor Attributes

The following attributes are properties of the **OCIAQMsgProperties** descriptor:

OCI_ATTR_ATTEMPTS

Mode
READ

Description
Specifies the number of attempts that have been made to dequeue the message. This parameter cannot be set at enqueue time.

Attribute Datatype
sb4

Valid Values
Any sb4 value is valid.

OCI_ATTR_CORRELATION

Mode
READ/WRITE

Description
Specifies the identification supplied by the producer for a message at enqueuing.

Attribute Datatype
text *

Valid Values
Any string up to 128 bytes is valid.

OCI_ATTR_DELAY

Mode
READ/WRITE

Description
Specifies the number of seconds to delay the enqueued message. The delay represents the number of seconds after which a message is available for dequeuing. Dequeuing by msgid overrides the delay specification. A message enqueued with delay set will be in the WAITING state, when the delay expires the messages goes to the READY state. DELAY processing requires the queue monitor to be started. Note that delay is set by the producer who enqueues the message.

Attribute Datatype
sb4

Valid Values
Any sb4 value is valid, but the following predefined constant is available:

- OCI_MSG_NO_DELAY - indicates the message is available for immediate dequeuing.

OCI_ATTR_ENQ_TIME

Mode
READ

Description
Specifies the time the message was enqueued. This value is determined by the system and cannot be set by the user.

Attribute Datatype
OCIDate

OCI_ATTR_EXCEPTION_QUEUE

Mode
READ/WRITE

Description
Specifies the name of the queue to which the message is moved to if it cannot be processed successfully. Messages are moved in two cases: If the number of unsuccessful dequeue attempts has exceeded *max_retries;* or if the message has expired. All messages in the exception queue are in the EXPIRED state.

The default is the exception queue associated with the queue table. If the exception queue specified does not exist at the time of the move the message will be moved to the default exception queue associated with the queue table and a warning will be logged in the alert file. If the default exception queue is used, the parameter will return a null value at dequeue time.

This attribute must refer to a valid queue name.

Attribute Datatype
text *

OCI_ATTR_EXPIRATION

Mode
READ/WRITE

Description
Specifies the expiration of the message. It determines, in seconds, the duration the message is available for dequeuing. This parameter is an offset from the delay. Expiration processing requires the queue monitor to be running.

While waiting for expiration, the message remains in the READY state. If the message is not dequeued before it expires, it will be moved to the exception queue in the EXPIRED state.

Attribute Datatype
sb4

Valid Values
Any sb4 value is valid, but the following predefined constant is available:

- OCI_MSG_NO_EXPIRATION - the message will not expire.

OCI_ATTR_MSG_STATE

Mode
READ

Description
Specifies the state of the message at the time of the dequeue. This parameter cannot be set at enqueue time.

Attribute Datatype
ub4

Valid Values
These are the only values which are returned:

- OCI_MSG_WAITING - the message delay has not yet been reached.

- OCI_MSG_READY - the message is ready to be processed.

- OCI_MSG_PROCESSED - the message has been processed and is retained.

- OCI_MSG_EXPIRED - the message has been moved to the exception queue.

OCI_ATTR_PRIORITY

Mode
READ/WRITE

Description
Specifies the priority of the message. A smaller number indicates higher priority. The priority can be any number, including negative numbers.

The default value is zero.

Attribute Datatype
sb4

OCI_ATTR_RECIPIENT_LIST

Mode
WRITE

Description
This parameter is only valid for queues which allow multiple consumers. The default recipients are the queue subscribers. This parameter is not returned to a consumer at dequeue time.

Attribute Datatype
OCIAQAgent **

OCI_ATTR_SENDER_ID

Mode
READ/WRITE

Description
Identifies the original sender of a message.

Attribute Datatype
OCIAgent *

OCI_ATTR_ORIGINAL_MSGID

Mode
READ/WRITE

Description
The ID of the last queue that generated this message. When a message is propagated from one queue to another, this attribute identifies the ID of the queue from which it was last propagated. When a message has been propagated through multiple queues, this attribute identifies the last queue, not the first queue.

Attribute Datatype
OCIRaw *

OCIAQAgent Descriptor Attributes

The following attributes are properties of the **OCIAQAgent** descriptor:

OCI_ATTR_AGENT_ADDRESS

Mode
READ/WRITE

Description
Protocol-specific address of the recipient. If the protocol is 0 (default), the address is of the form [*schema*.]*queue*[*@dblink*].

Attribute Datatype
text *

Valid Values
Can be any string up to 128 bytes.

OCI_ATTR_AGENT_NAME

Mode
READ/WRITE

Description
Name of a producer or consumer of a message.

Attribute Datatype
text *

Valid Values
Can be any Oracle identifier, up to 30 bytes.

OCI_ATTR_AGENT_PROTOCOL

Mode
READ/WRITE

Description
Protocol to interpret the address and propagate the message. The default (and currently the only supported) value is 0.

Attribute Datatype
ub1

Valid Values
The only valid value is zero, which is also the default.

OCIServerDNs Descriptor Attributes

The following attributes are properties of the **OCIServerDNs** descriptor:

OCI_ATTR_DN_COUNT

Mode
READ

Description
The number of database servers in the descriptor.

Attribute Datatype
ub2

OCI_ATTR_SERVER_DN

Mode
READ/WRITE

Description
For read mode, this attribute returns the list of database server distinguished names that are already inserted into the descriptor.

For write mode, this attribute takes the distinguished name of a database server.

Attribute Datatype
text **/text *

Subscription Handle Attributes

See Also: "Publish-Subscribe Notification" on page 9-46

OCI_ATTR_SERVER_DNS

Mode
READ/WRITE

Description
The distinguished names of the database servers that the client is interested in for the registration.

Attribute Datatype
OCIServerDNs *

OCI_ATTR_SUBSCR_CALLBACK

Mode
READ/WRITE

Description
Subscription callback. If the attribute OCI_ATTR_SUBSCR_RECPTPROTO is set to OCI_SUBSCR_PROTO_OCI or is left not set, then this attribute needs to be set before the subscription handle can be passed into the registration call
`OCISubscriptionRegister()`.

Attribute Datatype
OCISubscriptionNotify *

OCI_ATTR_SUBSCR_CTX

Mode
READ/WRITE

Description
Context that the client wants to get passed to the user callback denoted by OCI_ATTR_SUBSCR_CALLBACK when it gets invoked by the system. If the attribute OCI_ATTR_SUBSCR_RECPTPROTO is set to OCI_SUBSCR_PROTO_OCI or is left not set, then this attribute needs to be set before the subscription handle can be passed into the registration call `OCISubscriptionRegister()`.

Attribute Datatype
dvoid *

OCI_ATTR_SUBSCR_NAME

Mode
READ/WRITE

Description
Subscription name. All subscriptions are identified by a subscription name. A subscription name consists of a sequence of bytes of specified length. The length in bytes of the name needs to be specified as it is not assumed that the name will be null-terminated. This is important because the name could contain multibyte characters.

Clients will be able to set the subscription name attribute of a Subscription handle using an `OCIAttrSet()` call and by specifying a handle type of OCI_HTYPE_SUBSCR and an attribute type of OCI_ATTR_SUBSCR_NAME.

All of the subscription callbacks need a subscription handle with the OCI_ATTR_SUBSCR_NAME and OCI_ATTR_SUBSCR_NAMESPACE attributes set. If the attributes are not set, an error is returned. The subscription name that is set for the subscription handle must be consistent with its namespace.

Attribute Datatype
text *

OCI_ATTR_SUBSCR_NAMESPACE

Mode
READ/WRITE

Description
Namespace in which the subscription handle is used. The valid values for this attribute are OCI_SUBSCR_NAMESPACE_AQ and OCI_SUBSCR_NAMESPACE_ANONYMOUS. The subscription name that is set for the subscription handle must be consistent with its namespace.

Attribute Datatype
ub4 *

OCI_ATTR_SUBSCR_PAYLOAD

Mode
READ/WRITE

Description

Buffer that corresponds to the payload that needs to be sent along with the notification. The length of the buffer can also be specified in the same set attribute call. This attribute needs to be set before a post can be performed on a subscription. For this release, only an untyped (ub1 *) payload is supported.

Attribute Datatype
ub1 *

OCI_ATTR_SUBSCR_RECPT

Mode
READ/WRITE

Description

The name of the recipient of the notification when the attribute OCI_ATTR_SUBSCR_RECPTPROTO is set to OCI_SUBSCR_PROTO_MAIL, OCI_SUBSCR_PROTO_HTTP, or OCI_SUBSCR_PROTO_SERVER.

For OCI_SUBSCR_PROTO_HTTP, OCI_ATTR_SUBSCR_RECPT denotes the HTTP URL (for example, http://www.oracle.com:80) to which notification is sent. The validity of the HTTP URL is never checked by the database.

For OCI_SUBSCR_PROTO_MAIL, OCI_ATTR_SUBSCR_RECPT denotes the e-mail address (for example, xyz@oracle.com) to which the notification is sent. The validity of the e-mail address is never checked by the database system.

For OCI_SUBSCR_PROTO_SERVER, OCI_ATTR_SUBSCR_RECPT denotes the database procedure (for example: schema.procedure) that will be invoked in the event of a notification. The subscriber should have appropriate permissions on the procedure for it to be executed.

> **See Also:** For information about procedure definition, see "Notification Procedure" on page 9-53

Attribute Datatype
text *

OCI_ATTR_SUBSCR_RECPTPRES

Mode
READ/WRITE

Description

The presentation with which the client wants to receive the notification. The valid values for this are OCI_SUBSCR_PRES_DEFAULT and OCI_SUBSCR_PRES_XML.

If not set, this attribute defaults to OCI_SUBSCR_PRES_DEFAULT.

If the event notification is desired in XML presentation then this attribute should be set to OCI_SUBSCR_PRES_XML. Otherwise, it should be left not set or set to OCI_SUBSCR_PRES_DEFAULT.

Attribute Datatype
ub4

OCI_ATTR_SUBSCR_RECPTPROTO

Mode
READ/WRITE

Description

The protocol with which the client wants to receive the notification. The valid values for this are

- OCI_SUBSCR_PROTO_OCI

- OCI_SUBSCR_PROTO_MAIL

- OCI_SUBSCR_PROTO_SERVER

- OCI_SUBSCR_PROTO_HTTP

If an OCI client is interested in receiving the event notification, then this attribute should be set to OCI_SUBSCR_PROTO_OCI.

If you want an e-mail to be sent on event notification, then set this attribute to OCI_SUBSCR_PROTO_MAIL. If you want a PL/SQL procedure to be invoked in the database on event notification, then set this attribute to OCI_SUBSCR_PROTO_SERVER. If you want a HTTP URL to be posted to on event notification, then set this attribute to OCI_SUBSCR_PROTO_HTTP.

If not set, this attribute defaults to OCI_SUBSCR_PROTO_OCI.

For OCI_SUBSCR_PROTO_OCI, the attributes OCI_ATTR_SUBSCR_CALLBACK and OCI_ATTR_SUBSCR_CTX must be set before the subscription handle can be passed into the registration call `OCISubscriptionRegister()`.

For OCI_SUBSCR_PROTO_MAIL, OCI_SUBSCR_PROTO_SERVER, and OCI_SUBSCR_PROTO_HTTP, the attribute OCI_ATTR_SUBSCR_RECPT must be

set before the subscription handle can be passed into the registration call
`OCISubscriptionRegister()`.

Attribute Datatype
ub4

Direct Path Loading Handle Attributes

> **See Also:** For information about direct path loading and allocating the direct path handles, see "Direct Path Loading Overview" on page 12-2 and "Direct Path Loading of Object Types" on page 12-14

Direct Path Context Handle (OCIDirPathCtx) Attributes

OCI_ATTR_BUF_SIZE

Mode
READ/WRITE

Description
Sets the size of the stream transfer buffer. Default value is 64KB.

Attribute Datatype
ub4 */ub4 *

OCI_ATTR_CHARSET_ID

Mode
READ/WRITE

Description
Default character set ID for the character data. Note that the character set ID can be overridden at the column level. If character set ID is not specified at the column level or the table level, then the Global support environment setting is used.

Attribute Datatype
ub2 */ub2 *

OCI_ATTR_DATEFORMAT

Mode
READ/WRITE

Description
Default date format string for SQLT_CHAR to DTYDAT conversions. Note that the date format string can be overridden at the column level. If date format string is not specified at the column level or the table level, then the Global Support environment setting is used.

Attribute Datatype
text **/text *

OCI_ATTR_DIRPATH_MODE

Mode
READ/WRITE

Description
Mode of the direct path context:

- OCI_DIRPATH_LOAD-load operation (default)

- OCI_DIRPATH_CONVERT - convert only operation

Attribute Datatype
ub1 */ub1 *

OCI_ATTR_DIRPATH_NOLOG

Mode
READ/WRITE

Description
The NOLOG attribute of each segment determines whether image redo or invalidation redo is generated:

- 0 - Use the attribute of the segment being loaded.

- 1 - No logging. Overrides DDL statement, if necessary.

Attribute Datatype
ub1 */ub1 *

OCI_ATTR_DIRPATH_OBJ_CONSTR

Mode
READ/WRITE

Description
Indicates the object type of a substitutable object table:

```
text *obj_type; /* stores an object type name */
OCIAttrSet((dvoid *)dpctx,
                        (ub4)OCI_HTYPE_DIRPATH_CTX,
                        (dvoid *) obj_type,
                        (ub4)strlen((const char *) obj_type),
```

```
                                       (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

Attribute Datatype
text **/text *

OCI_ATTR_DIRPATH_PARALLEL

Mode
READ/WRITE

Description
Setting this value to 1 allows multiple load sessions to load the same segment concurrently. The default is 0 (not parallel).

Attribute Datatype
ub1 */ub1 *

OCI_ATTR_LIST_COLUMNS

Mode
READ

Description
Returns the handle to the parameter descriptor for the column list associated with the direct path context. The column list parameter descriptor can be retrieved after the number of columns is set with the OCI_ATTR_NUM_COLS attribute.

> **See Also:** "Accessing Column Parameter Attributes" on page A-66

Attribute Datatype
OCIParam* *

OCI_ATTR_NAME

Mode
READ/WRITE

Description
Name of the table to be loaded into.

Attribute Datatype
text**/text *

OCI_ATTR_NUM_COLS

Mode
READ/WRITE

Description
Number of columns being loaded in the table.

Attribute Datatype
ub2 */ub2 *

OCI_ATTR_NUM_ROWS

Mode
READ/WRITE

Description
Read: The number of rows loaded so far.

Write: The number of rows to be allocated for the direct path and the direct path function column arrays.

Attribute Datatype
ub2 */ub2 *

OCI_ATTR_SCHEMA_NAME

Mode
READ/WRITE

Description
Name of the schema where the table being loaded resides. If not specified, the schema defaults to that of the connected user.

Attribute Datatype
text **/text *

OCI_ATTR_SUB_NAME

Mode
READ/WRITE

Description
Name of the partition, or subpartition, to be loaded. If not specified, the entire table is loaded. The name must be a valid partition or subpartition name which belongs to the table.

Attribute Datatype
text **/text *

Direct Path Function Context Handle (OCIDirPathFuncCtx) Attributes

For further explanations of these attributes:

> **See Also:** "Direct Path Function Context and Attributes" on page 12-32

OCI_ATTR_DIRPATH_EXPR_TYPE

Mode
READ/WRITE

Description
Indicates the type of expression specified in OCI_ATTR_NAME in the function context of a non-scalar column.

Valid values are:

- OCI_DIRPATH_EXPR_OBJ_CONSTR (the object type name of a column object)

- OCI_DIRPATH_EXPR_REF_TBLNAME (table name of a reference object)

- OCI_DIRPATH_EXPR_SQL (a SQL string to derive the column value)

Attribute Datatype
ub1 */ub1 *

OCI_ATTR_LIST_COLUMNS

Mode
READ

Description
Returns the handle to the parameter descriptor for the column list associated with the direct path function context. The column list parameter descriptor can be retrieved after the number of columns (number of attributes or arguments associated with the non-scalar column) is set with the OCI_ATTR_NUM_COLS attribute.

> **See Also:** "Accessing Column Parameter Attributes" on page A-66

Attribute Datatype
OCIParam**

OCI_ATTR_NAME

Mode
READ/WRITE

Description
The object type name if the function context is describing a column object, a SQL function if the function context is describing a SQL string, or a reference table name if the function context is describing a REF column.

Attribute Datatype
text **/text *

OCI_ATTR_NUM_COLS

Mode
READ/WRITE

Description
The number of the object attributes to load if the column is a column object, or the number of arguments to process if the column is a SQL string or a REF column. This parameter must be set before the column list can be retrieved.

Attribute Datatype
ub2 */ub2 *

OCI_ATTR_NUM_ROWS

Mode
READ

Description
The number of rows loaded so far.

Attribute Datatype
ub4 *

Direct Path and Direct Path Function Column Array Handle (OCIDirPathColArray)

Attributes

OCI_ATTR_COL_COUNT

> **Mode**
> READ
>
> **Description**
> Last column of the last row processed.
>
> **Attribute Datatype**
> **ub2** *

OCI_ATTR_NUM_COLS

> **Mode**
> READ
>
> **Description**
> Column dimension of the column array.
>
> **Attribute Datatype**
> **ub2** *

OCI_ATTR_NUM_ROWS

> **Mode**
> READ
>
> **Description**
> Row dimension of the column array.
>
> **Attribute Datatype**
> **ub4** *

OCI_ATTR_ROW_COUNT

> **Mode**
> READ
>
> **Description**
> Number of rows processed.
>
> **Attribute Datatype**
> **ub4** *

Direct Path Stream Handle (OCIDirPathStream) Attributes

OCI_ATTR_BUF_ADDR

Mode
READ

Description
Buffer address of the beginning of the stream data.

Attribute Datatype
ub1 **

OCI_ATTR_BUF_SIZE

Mode
READ

Description
Size of the stream data in bytes.

Attribute Datatype
ub4 *

OCI_ATTR_ROW_COUNT

Mode
READ

Description
Number of rows processed by previous OCIDirPathLoadStream() call.

Attribute Datatype
ub4 *

OCI_ATTR_STREAM_OFFSET

Mode
READ

Description
Offset into the stream buffer of the last processed row.

Attribute Datatype
ub4 *

Direct Path Column Parameter Attributes

The application specifies which columns are to be loaded, and the external format of the data by setting attributes on each column parameter descriptor. The column parameter descriptors are obtained as parameters of the column parameter list by OCIParamGet(). The column parameter list of the table is obtained from the OCI_ATTR_LIST_COLUMNS attribute of the direct path context. If a column is non-scalar, then its column parameter list is obtained from the OCI_ATTR_LIST_COLUMNS attribute of its direct path function context.

Note that all parameters are 1-based.

Accessing Column Parameter Attributes

The following code example illustrates the use of the direct path column parameter attributes for scalar columns. Before the attributes are accessed, you must first set the number of columns to be loaded and get the column parameter list from the OCI_ATTR_LIST_COLUMNS attribute.

> **See Also:** See the data structures defined in the listings in Direct Path Load Example for Scalar Columns on page 12-9

```
...
  /* set number of columns to be loaded */
  OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAttrSet((dvoid *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX,
                     (dvoid *)&tblp->ncol_tbl,
                     (ub4)0, (ub4)OCI_ATTR_NUM_COLS, ctlp->errhp_ctl));

  /* get the column parameter list */
  OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAttrGet((dvoid *)dpctx,
                     OCI_HTYPE_DIRPATH_CTX,
                     (dvoid *)&ctlp->colLstDesc_ctl, (ub4 *)0,
                     OCI_ATTR_LIST_COLUMNS, ctlp->errhp_ctl));
```

Now you can set the parameter attributes.

```
  /* set the attributes of each column by getting a parameter handle on each
   * column, then setting attributes on the parameter handle for the column.
   * Note that positions within a column list descriptor are 1-based. */

  for (i = 0, pos = 1, colp = tblp->col_tbl, fldp = tblp->fld_tbl;
```

```
          i < tblp->ncol_tbl;
          i++, pos++, colp++, fldp++)
  {
    /* get parameter handle on the column */
    OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
             OCIParamGet((CONST dvoid *)ctlp->colLstDesc_ctl,
                          (ub4)OCI_DTYPE_PARAM, ctlp->errhp_ctl,
                          (dvoid **)&colDesc, pos));

    colp->id_col = i;                  /* position in column array */

    /* set external attributes on the column */
    /* column name */
    OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
             OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                          (dvoid *)colp->name_col,
                          (ub4)strlen((const char *)colp->name_col),
                          (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl));

    /* column type */
    OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
             OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                          (dvoid *)&colp->exttyp_col, (ub4)0,
                          (ub4)OCI_ATTR_DATA_TYPE, ctlp->errhp_ctl));

    /* max data size */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
             OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                          (dvoid *)&fldp->maxlen_fld, (ub4)0,
                          (ub4)OCI_ATTR_DATA_SIZE, ctlp->errhp_ctl));

    if (colp->datemask_col)    /* set column (input field) date mask */
    {
      OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
               OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                          (dvoid *)colp->datemask_col,
                          (ub4)strlen((const char *)colp->datemask_col),
                          (ub4)OCI_ATTR_DATEFORMAT, ctlp->errhp_ctl));
    }
    if (colp->prec_col)
    {
      OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
               OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                          (dvoid *)&colp->prec_col, (ub4)0,
                          (ub4)OCI_ATTR_PRECISION, ctlp->errhp_ctl));
```

```
    }
    if (colp->scale_col)
    {
      OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
              OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                        (dvoid *)&colp->scale_col, (ub4)0,
                        (ub4)OCI_ATTR_SCALE, ctlp->errhp_ctl));
    }
    if (colp->csid_col)
    {
      OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
              OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                        (dvoid *)&colp->csid_col, (ub4)0,
                        (ub4)OCI_ATTR_CHARSET_ID, ctlp->errhp_ctl));
    }
    /* free the parameter handle to the column descriptor */
    OCI_CHECK((dvoid *)0, 0, ociret, ctlp,
            OCIDescriptorFree((dvoid *)colDesc, OCI_DTYPE_PARAM));
  }
...
```

OCI_ATTR_CHARSET_ID

Mode
READ/WRITE

Description
Character set ID for character column. If not set, the character set ID defaults to the character set ID set in the direct path context.

Attribute Datatype
ub2 */ub2 *

OCI_ATTR_DATA_SIZE

Mode
READ/WRITE

Description
Maximum size in bytes of the external data for the column. This can affect conversion buffer sizes.

Attribute Datatype
ub2 */ub2 *

OCI_ATTR_DATA_TYPE

Mode
READ/WRITE

Description
Returns or sets the external datatype of the column. Valid datatypes are:

- SQLT_CHR
- SQLT_DAT
- SQLT_INT
- SQLT_UIN
- SQLT_FLT
- SQLT_PDN
- SQLT_BIN
- SQLT_NUM
- SQLT_NTY
- SQLT_REF

Attribute Datatype
ub2 */ub2 *

OCI_ATTR_DATEFORMAT

Mode
READ/WRITE

Description
Date conversion mask for the column. If not set, the date format defaults to the date conversion mask set in the direct path context.

Attribute Datatype
text **/text *

OCI_ATTR_DIRPATH_OID

Mode
READ/WRITE

Description
Indicates that the column to load into is a an object table's object id column.

Attribute Datatype
ub1 */ub1 *

OCI_ATTR_DIRPATH_SID

Mode
READ/WRITE

Description
Indicates that the column to load into is a nested table's setid column.

Attribute Datatype
ub1 */ub1 *

OCI_ATTR_NAME

Mode
READ/WRITE

Description
Returns or sets the name of the column that is being loaded.

Attribute Datatype
text **/text *

OCI_ATTR_PRECISION

Mode
READ/WRITE

Description
Returns or sets the precision.

Attribute Datatype
ub1 */ub1 *

OCI_ATTR_SCALE

Mode
READ/WRITE

Description

Returns or sets the scale (number of digits to the right of the decimal point) for conversions from packed and zoned decimal input data types.

Attribute Datatype
sb1 */sb1 *

Process Handle Attributes

The parameters for the shared system can be set and read using the OCIAttrSet()
and OCIAttrGet() calls. The handle type to be used is the process handle
OCI_HTYPE_PROC.

> **See Also:**

The OCI_ATTR_MEMPOOL_APPNAME, OCI_ATTR_MEMPOOL_HOMENAME,
and OCI_ATTR_MEMPOOL_INSTNAME attributes specify the application, home,
and instance names that can be used together to map the process to the right shared
pool area. If these attributes are not provided, internal default values are used. The
following are valid settings of the attributes for specific behaviors:

- Instance name, application name (unqualified): This allows only executables
  with a specific name to attach to the same shared subsystem. For example, this
  allows an OCI application named *Office* to connect to the same shared
  subsystem regardless of the directory *Office* resides in.

- Instance name, home name: This allows a set of executables in a specific home
  directory to attach to the same instance of the shared subsystem. For example,
  this allows all OCI applications residing in the ORACLE_HOME directory to
  use the same shared subsystem.

- Instance name, home name, application name (unqualified): This allows only a
  specific executable to attach to a shared subsystem. For example, this allows one
  application named *Office* in the ORACLE_HOME directory to attach to a given
  shared subsystem.

OCI_ATTR_MEMPOOL_APPNAME

Mode
READ/WRITE

Description
Executable name or fully-qualified path name of the executable.

Attribute Datatype
text *

OCI_ATTR_MEMPOOL_HOMENAME

Mode
READ/WRITE

Description
Directory name where the executables that use the same shared subsystem instance are located.

Attribute Datatype
text *

OCI_ATTR_MEMPOOL_INSTNAME

Mode
READ/WRITE

Description
Any user-defined name to identify an instance of the shared subsystem.

Attribute Datatype
text *

OCI_ATTR_MEMPOOL_SIZE

Mode
READ/WRITE

Description
Size of the shared pool in bytes. This attribute is set as follows:

```
ub4 plsz = 1000000;
OCIAttrSet((dvoid *)0, (ub4) OCI_HTYPE_PROC,
          (dvoid *)&plsz, (ub4) 0, (ub4) OCI_ATTR_POOL_SIZE, 0)
```

Attribute Datatype
ub4 *

OCI_ATTR_PROC_MODE

Mode
READ

Description
Returns all the currently set process modes. The value read contains the OR'ed value of all the currently set OCI process modes. To determine is a specific mode is set, the value should be OR'ed with that mode. For example:

```
ub4 mode;
boolean is_shared;
```

```
OCIAttrGet((dvoid *)0, (ub4)OCI_HTYPE_PROC,
           (dvoid *) &mode, (ub4 *) 0,
           (ub4)OCI_ATTR_PROC_MODE, 0);

is_shared = (mode & OCI_SHARED);
```

Attribute Datatype
ub4 *

B

OCI Demonstration Programs

Oracle provides code examples illustrating the use of OCI calls. These programs are provided for demonstration purposes, and are not guaranteed to run on all platforms.

The demonstration programs (*demos*) are available with your Oracle installation. The location, names, and availability of the programs may vary on different platforms. On a UNIX workstation, the programs are installed in the `ORACLE_HOME/rdbms/demo` directory.

The `$ORACLE_HOME/rdbms/demo` directory contains not only demos but the file named `Makefile` that must be used as a template on how to build your own OCI applications or external procedures. Development of new `Makefiles` to build an OCI application or an external procedure should consist of the customizing of the `Makefile` provided by adding your own macros to the link line. However, Oracle requires that you keep the macros provided in the demo `Makefile`, as it will result in easier maintenance of your own `Makefiles`.

On a Windows NT machine, the programs are located in the `ORACLE_HOME\Oci\Samples` directory.

When a specific header or SQL file is required by the application, these files are also included. Review the information in the comments at the beginning of the demonstration programs for setups and hints on running the programs.

Table B–1, "OCI Demonstration Programs" lists the important demonstration programs and the OCI features that they illustrate.

Table B–1   OCI Demonstration Programs

Program Name	Features Illustrated
cdemo81.c	Using basic SQL processing with release 8 and later functionality.

Table B–1 OCI Demonstration Programs (Cont.)

Program Name	Features Illustrated
cdemo82.c	Performing basic processing of user-defined objects.
cdemocor.c	Using complex object retrieval (COR) to improve performance.
cdemodr1.c, cdemodr2.c, cdemodr3.c	Using INSERT/UPDATE/DELETE statements with RETURNING clause used with basic datatypes, LOBs and REFs.
cdemodsa.c	Describing information about a table.
cdemodsc.c	Describing information about an object type.
cdemofo.c	Registering and operating application failover callbacks.
cdemolb.c	Creating, accessing, and manipulating LOB objects.
cdemolb2.c	Writing and reading of CLOB/BLOB columns with stream mode and callback functions.
cdemolbs.c	Writing and reading to LOBs with the LOB buffering system.
cdemobj.c	Pinning and navigation of REF object.
cdemorid.c	Using INSERT/UPDATE/DELETE statements and fetches to get multiple rowids in one roundtrip.
cdemoses.c	Using session switching and migration.
cdemothr.c	Using the OCIThread package.
cdemosyev.c	Registering predefined subscriptions and specifying a callback function to be invoked for client notifications (for more information about Advanced Queuing, see *Oracle9i Application Developer's Guide - Advanced Queuing*).
cdemodp.c, cdemdplp.c	Loading data with the direct path load functions.
cdemdpco.c	Loading a column object.
cdemdpno.c	Loading a nested column object.
cdemdpin.c	Loading derived type (inheritance).
cdemdpit.c	Loading an object table with inheritance.
cdemdpro.c	Loading a reference.
cdemdpss.c	Loading SQLl strings.
cdemoucb.c, cdemoucb1.c	Using static and dynamic user callbacks.

Table B–1 OCI Demonstration Programs (Cont.)

Program Name	Features Illustrated

`cdemoupk.c,`
`cdemoup1.c,`
`cdemoup2.c`
	Using dynamic user callbacks with multiple packages.

C

OCI Function Server Round-trips

This appendix provides information about server round-trips incurred during various OCI calls. This information can be useful to programmers when determining the most efficient way to accomplish a particular task in an application.

The appendix contains the following sections:

- Overview of Server Round-trips
- Relational Function Round-trips
- LOB Function Round-trips
- Object and Cache Function Round-trips
- Describe Operation Round-trips
- Datatype Mapping and Manipulation Function Round-trips
- Other Local Functions

Overview of Server Round-trips

This appendix provides information about server round-trips incurred during various OCI calls. This information can be useful when determining the most efficient way to accomplish a particular task in an application.

Relational Function Round-trips

The number of server round-trips required by OCI relational functions are listed in Table C–1:

Table C–1   Server Round-trips for Relational Operations

Function	# of Server Round-trips
OCIBreak()	1
OCIEnvCreate()	0
OCIEnvInit()	0
OCIErrorGet()	0
OCIInitialize()	0
OCILdaToSvcCtx()	0
OCILogoff()	1
OCILogon()	1
OCIPasswordChange()	
OCIReset()	0
OCIServerAttach()	1
OCIServerDetach()	1
OCIServerVersion()	1
OCISessionBegin()	1
OCISessionEnd()	1
OCISvcCtxToLda()	0
OCITerminate()	1
OCIStmtExecute()	1
OCIStmtFetch()	0 or 1

Table C–1   Server Round-trips for Relational Operations (Cont.)

Function	# of Server Round-trips
OCIStmtGetPieceInfo()	1
OCIStmtPrepare()	0
OCIStmtSetPieceInfo()	0
OCITransCommit()	1
OCITransDetach()	1
OCITransForget()	1
OCITransPrepare()	1
OCItransRollback()	1
OCITransStart()	1
OCIUserCallbackGet()	0
OCIUserCallbackregister()	0

LOB Function Round-trips

Table C–2 lists the server round-trips incurred by the OCILob*() calls. Information about the read and write calls is listed after the table.

Table C–2   Server Round-trips for OCILob() Calls*

Function	# of Server Round-trips
OCILobAppend()	1
OCILobAssign()	0
OCILobCharSetForm()	0
OCILobCharSetId()	0
OCILobCopy()	1
OCILobCreateTemporary()	1
OCILobDisableBuffering()	0
OCILobEnableBuffering()	0
OCILobErase()	1
OCILobFileClose()	1

Table C–2   Server Round-trips for OCILob() Calls (Cont.)*

Function	# of Server Round-trips
`OCILobFileCloseAll()`	1
`OCILobFileExists()`	1
`OCILobFileGetName()`	0
`OCILobFileIsOpen()`	1
`OCILobFileOpen()`	1
`OCILobFileSetName()`	0
`OCILobFlushBuffer()`	1 per modified page in the buffer for this LOB
`OCILobFreeTemporary()`	1
`OCILobGetLength()`	1
`OCILobIsEqual()`	0
`OCILobIsTemporary()`	0
`OCILobLoadFromFile()`	1
`OCILobLocatorAssign()`	1 round-trip if the source and/or the destination locator refers to a temporary LOB
`OCILobLocatorIsInit()`	0
`OCILobTrim()`	1
`OCILobOpen()`	1
`OCILobClose()`	1
`OCILobIsOpen()`	1
`OCILobGetChunkSize()`	1

Object and Cache Function Round-trips

Table C–3 lists the number of server round-trips required for the object and cache functions. These values assume the cache is in a *warm* state, meaning that the type descriptor objects required by the application have been loaded.

Table C–3   Server Round -trips for Object and Cache Functions

Function	# of Server Round-trips
`OCIObjectNew()`	0

Table C–3   Server Round -trips for Object and Cache Functions (Cont.)

Function	# of Server Round-trips
OCIObjectPin()	1; 0 if the desired object is already in cache
OCIObjectUnpin()	0
OCIObjectPinCountReset()	0
OCIObjectLock()	1
OCIObjectMarkUpdate()	0
OCIObjectUnmark()	0
OCIObjectUnmarkByRef()	0
OCIObjectFree()	0
OCIObjectMarkDelete()	0
OCIObjectMarkDeleteByRef()	0
OCIObjectFlush()	1
OCIObjectRefresh()	1
OCIObjectCopy()	0
OCIObjectGetTypeRef()	0
OCIObjectGetObjectRef()	0
OCIObjectGetInd()	0
OCIObjectExists()	0
OCIObjectIsLocked()	0
OCIObjectIsDirty()	0
OCIObjectPinTable()	1
OCIObjectArrayPin()	1
OCICacheFlush()	1
OCICacheRefresh()	1
OCICacheUnpin()	0
OCICacheFree()	0
OCICacheUnmark()	0

Describe Operation Round-trips

The number of server round-trips required by `OCIDescribeAny()`, `OCIAttrGet()`, and `OCIParamGet()` are listed in Table C–4:

Table C–4   Server Round-trips for Describe Operations

Function	# of Server Round-trips
`OCIDescribeAny()`	1 round-trip to get the REF of the type descriptor object
`OCIAttrGet()`	2 round-trips to describe a type if the type objects are not in the object cache
	1 round-trip for each collection element, or each type attribute, method, or method argument descriptor. 1 more round-trip if using OCI_ATTR_TYPE_NAME, or OCI_ATTR_SCHEMA_NAME on the collection element, type attribute, or method argument.
	0 if all the type objects to be described are already in the object cache following the first `OCIAttrGet()` call.
`OCIParamGet()`	0

Datatype Mapping and Manipulation Function Round-trips

The number of round-trips for the datatype mapping and manipulation functions are listed in Table C–5. The asterisks in the table indicate that all functions with a particular prefix incur the same number of server round-trips. For example, `OCINumberAdd()`, `OCINumberPower()`, and `OCINumberFromText()` all incur zero server round-trips.

Table C–5   Server Round-trips for Datatype Manipulation Functions

Function	# of Server Round-trips
`OCINumber*()`	0
`OCIDate*()`	0
`OCIString*()`	0
`OCIRaw*()`	0
`OCIRef*()`	0
`OCIColl*()`	0; 1 if the collection is not loaded in the cache
`OCITable*()`	0; 1 if the nested table is not loaded in the cache

Table C–5   Server Round-trips for Datatype Manipulation Functions (Cont.)

Function	# of Server Round-trips
OCIIter*()	0; 1 if the collection is not loaded in the cache

Other Local Functions

The functions listed in Table C–6 are local and do not require a server round-trip:

Table C–6   Locally Processed Functions

Local Function Name	Notes
OCIAttrGet()	
OCIAttrSet()	
OCIBindByName()	
OCIBindByPos()	
OCIBindDynamic()	
OCIBindObject()	
OCIBindArrayOfStruct()	
OCIDefineByPos()	
OCIDefineDynamic()	
OCIDefineArrayOfStruct()	
OCIDefineObject()	
OCIDescriptorAlloc()	
OCIDescriptorFree()	
OCIEnvInit()	
OCIEnvCreate()	
OCIErrorGet()	
OCIHandleAlloc()	
OCIHandleFree()	
OCILdaToSvcCtx()	
OCISvcCtxToLda()	
OCIStmtGetBindInfo()	

Table C–6   Locally Processed Functions (Cont.)

Local Function Name	Notes
OCIStmtPrepare()	
OCIStmtGetBindInfo()	
OCIStmtPrepare()	
OCIStmtFetch()	may be local if retrieving pre-fetched rows

Index

publish-subscribe functions, 16-70

Q

query
   explicit describe, 4-14
query. See SQL query

R

RAW
   external datatype, 3-16
REF
   external datatype, 3-20
REF columns
   direct path loading of, 12-22
REF cursors variables
   binding and defining, 5-43
reference. See REFs
refreshing, 13-11
   objects, 13-11
REFs
   binding, 5-10, 11-37
   cursor variables, binding, 5-17
   defining, 5-22, 11-39
   indicator variables for, 2-35, 2-37
   retrieving from server, 10-11
registering
   user callbacks, 9-24
relational functions
   server roundtrips, C-2, C-7
reserved namespaces, 2-40
reserved words, xlii, 2-39
result set, 4-18
resuming branches, 8-7
return values
   navigational functions, 17-5
RETURNING clause
   binding with, 5-31
   error handling, 5-32
   initializing variables, 5-32
   using with OCI
   with REFs, 5-32
rollback, 2-29
   in object applications, 13-15

roundtrips
   See server roundtrips
ROWID
   external datatype, 3-20
   logical, 3-6
   Universal ROWID, 3-6
   used for positioned updates and deletes, 2-39
ROWID descriptor, 2-18

S

sample programs, B-1
sb1
   definition, 3-35
sb2
   definition, 3-35
sb4
   definition, 3-35
SCHEMA_NAMES OTT parameter, 14-33
   usage, 14-38
schemas
   attributes, 6-19
   describing, 6-2
scrollable cursor
   example, 4-19
scrollable cursors, 4-18
secondary memory
   of object, 13-17
select-list
   describing, 4-12
sequences
   attributes, 6-14
   describing, 6-2
server handle
   attributes, A-14
   description, 2-9
   setting in service context, 2-10
server roundtrips
   cache functions, C-4
   datatype mapping and manipulation
      functions, C-6
   describe operation, C-6
   LOB functions, C-3
   object functions, C-4
   relational functions, C-2, C-7

V

W

X