Oracle® Call Interface

Programmer's Guide,

10*g* Release 2 (10.2)

B14250-02

November 2005

ORACLE®

Oracle Call Interface Programmer's Guide, 10*g* Release 2 (10.2)

B14250-02

Primary Author: Jack Melnick

Contributors: A. Ahluwalia, G. Arora, A. Beguelin, A. Bande, D. Banerjee, S. Banerjee, M. Bastawala, E. Belden, N. Bhatt, James Feng Cao, Yujie Cao, S. Chandrasekar, Thomas H. Chang, D. Chatterjee, D. Chiba, L. Chidambaran, Chi Ching Chui, D. Frumkin, S. Gollapudi, Wenyun. He, Min-Hank Ho, N. Ikeda, Toliver Jue, R. Kasamsetty, S. Kotsovolos, S. Krishnaswamy, S. Lari, Geoff Lee, N. Lewis, Chao Liang, E. Miner, S. Mishra,, K. Mohan, Valarie Moore, J. Narasinghanallur, E. Paapanen, R. Phillips, R. Pingte, T. Pulkita, V. Raja, D. Saha, S. Seshadri, B. Sinha, H. Slattery, Steven Sun, K. Surlaker, B. Thome, P. Tyagi, S. S. Vemuri, R. Vissapragada, Wei Wang, Daniel M. Wong, Mingkang Xu, Jianping Yang, Michael Yau

Contents

2 OCI Programming Basics

3 Datatypes

4 Using SQL Statements in OCI

5 Binding and Defining in OCI

6 Describing Schema Metadata

7 LOB and BFILE Operations

8 Managing Scalable Platforms

9 OCI Programming Advanced Topics

10 OCI Object-Relational Programming

11 Object-Relational Datatypes in OCI

12 Direct Path Loading

13 Object Advanced Topics in OCI

14 Using the Object Type Translator with OCI

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

21 OCI Globalization Support Functions

22 OCI XML DB Functions

A Handle and Descriptor Attributes

Index

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.

Audience

This 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 *Oracle Database SQL Reference* and the *Oracle Database Administrator's Guide*
- For information about basic Oracle concepts, see *Oracle 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 *Oracle Database New Features.*

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. 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. Accessibility 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 more information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

Many of the examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database installation. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Printed documentation is available for sale in the Oracle Store at

http://oraclestore.oracle.com/

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://www.oracle.com/technology/membership/

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

http://www.oracle.com/technology/documentation/

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 more information about cartridge services, and the OCI calls pertaining to development of data cartridges, refer to *Oracle Database Data Cartridge Developer's Guide.*
- For more information about OCI calls pertaining to National Language and Globalization Support, see *Oracle Database Globalization Support Guide.*
- For more information about OCI calls pertaining to Advanced Queuing, see *Oracle Streams Advanced Queuing User's Guide and Reference.*
- For information about using OCI with the XA library, see *Oracle Database Application Developer's Guide - Fundamentals.*
- For more information about using OCI calls to manipulate LOBs, including code examples, see *Oracle Database Application Developer's Guide - Large Objects.*
- For a more detailed explanation of object types, see *Oracle Database Application Developer's Guide - Object-Relational Features.*

Further Sources of Information About Oracle Database

- *Oracle Database Installation Guide for Microsoft Windows (32-Bit)*
- *Oracle Database Release Notes for Microsoft Windows (32-Bit)*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database New Features*
- *Oracle Database Concepts*
- *Oracle Database Reference*
- *Oracle Database Error Messages*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in Oracle Call Interface?

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

- New Features in Oracle Call Interface Release 10.2
- New Features in Oracle Call Interface Release 10.1

New Features in Oracle Call Interface Release 10.2

- Instant Client Light (English) further reduces the footprint of Instant Client.

  See Also: "Instant Client Light (English)" on page 1-22

- Database startup and shutdown can now be done from OCI.

  See Also: "Database Startup and Shutdown" on page 9-72

- You can obtain the client library version at compile-time and runtime.

  See Also: "OCIClientVersion()" on page 16-175

- A new user handle session attribute, OCI_ATTR_CURRENT_SCHEMA, which has the same functionality as the SQL command `ALTER SESSION SET CURRENT_SCHEMA`, is described in "OCI_ATTR_CURRENT_SCHEMA" on page A-15.
- `OCIDirPathFlushRow()` was deprecated. There are small changes in the Direct Path interfaces that are in the following sections:

  See Also:

  - "OCI_DIRPATH_COL_ERROR" on page 12-16
  - "Direct Path Loading of SQL String Columns" on page 12-16
  - "OCIDirPathColArrayToStream()" on page 16-124
  - "OCIDirPathLoadStream()" on page 16-129

- OCI clients can register for Real Application Clusters (RAC) high availability events and decide on actions for each event.

  See Also:

  - "HA Event Notification" on page 9-36
  - "Event Handle Attributes" on page A-65

- Restrictions on selecting remote LOB data defined as character or raw datatypes by means of the data interface are eased.

  See Also: "Creating and Modifying Persistent LOBs" on page 7-2 and the subsequent sections

- Two new functions read and write LOB data for multiple locators in one server round trip.

  See Also:

  - "OCILobArrayRead()" on page 16-26
  - "OCILobArrayWrite()" on page 16-30
  - "Using OCILobArrayRead() and OCILobArrayWrite()" on page 7-9

- For `NCHAR` literal support in OCI, new values of the parameter `mode` (`OCI_NCHAR_LITERAL_REPLACE_ON` and `OCI_NCHAR_LITERAL_REPLACE_OFF`) in the following functions control N' substitution:

  See Also:

  - "OCIEnvCreate()" on page 15-13
  - "OCIEnvNlsCreate()" on page 15-18

- The Notifications Enhancement project improves event notification.

  See Also:

  - "Publish-Subscribe Notification in OCI" on page 9-47
  - "Subscription Handle Attributes" on page A-44 for new attributes `OCI_ATTR_SUBSCR_PORTNO`, `OCI_ATTR_SUBSCR_QOSFLAGS`, `OCI_ATTR_SUBSCR_TIMEOUT`

- Buffered messaging is now supported within the Streams Advanced Queueing capability.

  See Also:

  - "Buffered Messaging" on page 9-44 for concepts
  - "Streams Advanced Queuing and Publish-Subscribe Functions" on page 16-98 for the function reference pages
  - "OCIAQListen2()" on page 16-108 for a new function
  - "Streams Advanced Queuing Descriptor Attributes" on page A-34

- Database change notification enables client applications to receive notifications when the result set of a registered query has changed.

See Also:

- "Database Change Notification" on page 9-61
- "Change Notification Attributes" on page A-48
- "Change Notification Descriptor Attributes" on page A-49

- Asynchronous commit allows the programmer greater control by using new values for the `flags` parameter when making calls to `OCITransCommit()`:

  See Also: "OCITransCommit()" on page 16-159

- An Oracle wallet is a secure software container used to store authentication and signing credentials.

  See Also: "Secure External Password Store" on page 8-9

- Application context enables developers to define, set, and access application attributes.

  See Also: "Client Application Context" on page 8-19

- Proxy access for a single client can be set using various connect strings.

  See Also: "Client Access Through a Proxy" on page 2-15

- `OCIPing()` is used to confirm that the server connection and the server are active. It also can be used to flush all the pending OCI client-side calls to the server.

  See Also: "OCIPing()" on page 16-181

- Windows NT is no longer supported for OCI. The only Microsoft Visual C++ releases supported for the current OCI release are 7.0 and higher.

  See Also: Appendix D, "Getting Started with OCI for Windows"

- Transparent Application Failover (TAF) is enabled for connection pooling.

  See Also: "Transparent Application Failover" on page 9-9

- OCI Scrollable Cursors now works for remote mapped queries. Transparent Application Failover (TAF) works with OCI Scrollable cursors.

  See Also: "Scrollable Cursors in OCI" on page 4-14

New Features in Oracle Call Interface Release 10.1

- Native float and double are supported.

  See Also:

  - "BINARY_FLOAT and BINARY_DOUBLE" on page 3-5
  - "Native Float and Native Double" on page 3-19

- `OCIDescribeAny()` supports rules, rule sets, and evaluation contexts.

See Also: "Rule Attributes" on page 6-16, and so on

- The OCI Instant Client capability simplifies OCI installation and saves disk space for application deployment.

  See Also: "OCI Instant Client" on page 1-16

- Additional information on upgrading to a new release of OCI is available.

  See Also: "Compatibility and Upgrading" on page 1-12

A new discussion describes when to use session pooling or connection pooling.

See Also: "When to Use Connection Pooling, Session Pooling, or Neither" on page 9-17

- Batch array enqueue and dequeue functions and attributes have been added.

  See Also:

  - "OCI and Streams Advanced Queuing" on page 9-39
  - "Streams Advanced Queuing and Publish-Subscribe Functions" on page 16-98
  - "Streams Advanced Queuing Descriptor Attributes" on page A-34

- LOBs can be of greater size than 4 GB. There are several new LOB functions whose names end in "2" that handle LOBs greater and smaller than 4 GB, and replace deprecated LOB functions without "2".

  See Also:

  - "Using LOBs of Size Greater than 4 GB" on page 7-4
  - "LOB Functions" on page 16-19

- Database Globalization Support is now described in this manual.

  See Also:

  - "OCI Globalization Support" on page 2-28
  - "OCI Globalization Support Functions" on page 21-1

- Statement Caching has been enhanced.

  See Also: "Statement Caching in OCI" on page 9-20

- Windows documentation is now included in this guide.

See Also: Appendix D, "Getting Started with OCI for Windows"

- There is OCI support for the unified C API which is used for `XMLType` columns in tables (and XML documents).

  See Also:

  - "OCI Support for XML" on page 13-17
  - Chapter 22, "OCI XML DB Functions"

- There are new or modified functions.

  See Also:

  - "OCICollGetElemArray()" on page 18-10
  - "OCINumberToRealArray()" on page 18-126

- New modes `OCI_BIND_SOFT` and `OCI_DEFINE_SOFT` are documented.

  See Also:

  - "OCIBindByName()" on page 15-61
  - "OCIBindByPos()" on page 15-65
  - "OCIDefineByPos()" on page 15-75

- New attributes for end-to-end application tracing are described.

  See Also:

  - "End-to-End Application Tracing" on page 8-15
  - "User Session Handle Attributes" on page A-12

- New attributes for Direct Path are described.

  See Also:

  - "Direct Path Context Handle (OCIDirPathCtx) Attributes" on page A-51
  - "Direct Path Context Handle (OCIDirPathCtx) Attributes" on page A-51

1

Introduction and Upgrading

This chapter contains these topics:

Overview of OCI

The Oracle Call Interface (OCI) is an application programming interface (API) that lets you create applications that use function calls 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:

- *Oracle C++ Call Interface Programmer's Guide*
- "Related Documents" on page xxxvi

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-tier authentication
- Comprehensive support for application development using Oracle objects
- Access to external databases
- Applications that support an increasing number of users and requests without additional hardware investments

OCI lets you manipulate data and schemas in an Oracle database using C programming language. 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.

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

- Encapsulated or opaque interfaces, whose implementation details are unknown
- 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 application design
- High degree of control over program execution
- Use of familiar third generation language programming techniques and application development tools, such as browsers and debuggers
- Connection pooling, session pooling, and statement caching that enable building of scalable applications
- Support of dynamic SQL
- Availability on the broadest range of operating systems of all the Oracle programmatic interfaces
- Dynamic binding and defining using callbacks
- Description 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 commit requests with executes to reduce round trips
- Optimization of queries using transparent prefetch buffers to reduce round trips
- Thread safety which eliminates the need for mutual exclusive locks (mutexes) on OCI handles

Building an OCI Application

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.

Oracle supports most popular third-party compilers. The details of linking an OCI program vary from system to system. On some operating systems, 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 operating system.

See Also: Appendix D, "Getting Started with OCI for Windows"

Parts of OCI

OCI has the following 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

OCI lets you develop scalable, multithreaded applications in a multitier architecture that combines 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 more 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 preceding 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 attribute values 'John', 'Bonivento', and '30'. Object

types may also contain *methods*—programmatic functions that represent the behavior of that object type.

See Also:

- *Oracle Database Concepts*
- *Oracle Database 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:

- Executing SQL statements that manipulate object data and schema information
- Passing of object references and instances as input variables in SQL statements
- Declaring object references and instances as variables to receive the output of SQL statements
- Fetching object references and instances from a database
- Describing the properties of SQL statements that return object instances and references
- Describing PL/SQL procedures or functions with object parameters or results
- Extension of commit and rollback calls in order to synchronize object and relational functionality

Additional OCI calls are provided to support manipulation of objects after they have been accessed by SQL statements. For a more detailed description of enhancements and new features, refer to "Encapsulated Interfaces" on page 1-8.

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 (DDL)
- 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: *Oracle Database 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
- Require an application to supply data to the database using input (bind) variables

See Also: "Binding Placeholders in OCI" on page 4-4 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 preceding 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 preceding statement, you would need to define an output variable to receive any `name` values returned from the query.

  See Also:

  - "Overview of Binding in OCI" on page 5-1 for more information about input bind variables. See the section "Overview of Defining in OCI" on page 5-12 for information about defining output variables.
  - 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, so that they are 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 empno = :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:

- *Oracle Database PL/SQL User's Guide and Reference* for information about coding PL/SQL blocks.
- "Binding Placeholders in PL/SQL" on page 5-3 for information about working with placeholders in PL/SQL.

Embedded SQL

OCI processes SQL statements as text strings that 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: *Pro*C/C++ Programmer's Guide*

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 of 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 calls 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 types of handles, 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 by means of 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
- Improvement of productivity by eliminating the need for global variables, making error reporting easier, and providing consistency in the way OCI variables are accessed and used
- 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 with 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 by which a single user connects to the database using a login name and password
- A mechanism by 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

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 made 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 round trips, and reduce memory usage
- Elimination of open and closed cursor round trips
- Improved support for multithreaded environments
- Session multiplexing over connections
- Consistent support for a variety of configurations, including standard two-tier client/server configurations, server-to-server transaction coordination, and three-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 a 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 Caching
- 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 controlling data attributes of Oracle types
- Object Type Translator utility, for mapping 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 object instances that 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 object cache provides the following to OCI applications:

- Improved application performance by reducing the number of client/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:

- 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 select-list items 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

OCI Runtime Environment for Objects

OCI provides functions for objects that manages how Oracle objects are used on the client-side. These functions provide 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 Oracle 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 represented in the server as internal Oracle datatypes to their corresponding host language types.
- Type Manipulation functions allow host language applications to manipulate individual attributes of an Oracle schema such as setting and getting their values and flushing their values to the server.

Additionally, the `OCIDescribeAny()` function provides 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 of host language variables, such as structures. The OTT takes as input an `intype` file which contains metadata information about Oracle schema objects. It 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 you to code 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 the use of SQL as the data-definition language of choice. This in turn allows Oracle to support a consistent model of data.
- Facilitates schema evolution of object types: OTT regenerates 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 structures which can either be used with OCI programs or with the Pro*C/C++ precompiler programs

OCI Support for Oracle Streams Advanced Queuing

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

See Also: "OCI and Streams Advanced Queuing" on page 9-39.

XA Library Support

OCI supports the Oracle XA library. The `xa.h` header file is in the same location as all the other OCI header files. For Linux or UNIX, the path is `$ORACLE_HOME/rdbms/public`. Users of the `demo_rdbms.mk` file on Linux or UNIX are not affected because the directory `$ORACLE_HOME/rdms/public` is already in the file.

For Windows, the path is *`ORACLE_BASE`*`\`*`ORACLE_HOME`*`\oci\include`.

See Also:

- "The Oracle XA Library" on page D-3 for more information about Windows and XA applications
- *Oracle Database Application Developer's Guide - Fundamentals*, chapter "Developing Applications with Oracle XA".

Compatibility and Upgrading

The following sections discuss issues concerning compatibility between different versions of OCI client and server, changes in the OCI library routines, and upgrading an application from the release 7.*x* OCI to this release of OCI.

Simplified Upgrading of Existing OCI Release 7 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 Oracle database version 7 server.

Specifically:

- Applications that use the OCI release 7.3 API will work unchanged against this release of the server. They do need to be linked with the current client library.
- 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 same statement execution.

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

- Upgrade to the current OCI client but do not modify the application: If you choose to upgrade from an Oracle release 7 OCI client to the current release OCI client, you need only link the new version of the OCI library and need *not* recompile your application. The re-linked Oracle release 7 OCI applications work unchanged against a current server.
- Upgrade to the current OCI client and modify the 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, re-link them with the new OCI library, and run them against the current release of the server.

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

Statically-Linked and Dynamically-Linked Applications

Here are the rules for re-linking for a new release.

- Statically-linked applications:

  Statically-linked applications need to be re-linked for both major and minor releases, because the linked Oracle client-side library code may be incompatible with the error messages in the upgraded `ORACLE_HOME`. For example, if an error message was updated with additional parameters then it will not be compatible with the statically-linked code.

- Dynamically-linked applications:

Dynamically-linked applications need to be re-linked for major releases only. OCI applications that are dynamically linked have a hard reference to the `libclntsh.so.n`, where `n` is the major release number.

See Also:

- *Oracle Database Upgrade Guide* for information about compatibility and upgrading
- The server versions supported currently are found on Oracle *i*Support in note 207303.1. See the URL `http://metalink.oracle.com/`

Obsolete OCI Routines

Release 8.0 of the OCI introduced an entirely new set of functions which were not available in release 7.3. Oracle continues to support these release 7.3 functions. Many of the earlier 7.*x* calls are available, but Oracle strongly recommends that new applications use the new calls to improve performance and provide increased functionality.

Table 1–1, " Obsolescent OCI Functions" lists the 7.*x* OCI calls with their later equivalents. For more information about the OCI calls, see the function descriptions in 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 Functions

7.*x* OCI Routine	Equivalent or Similar Later OCI Routine
`obindps(), obndra(), obndrn(), obndrv()`	`OCIBindByName(),OCIBindByPos()` (Note: additional bind calls may be necessary for some datatypes)
`obreak()`	`OCIBreak()`
`ocan()`	none
`oclose()`	Note: cursors are not used in release 8.x or higher
`ocof(), ocon()`	`OCIStmtExecute()` with `OCI_COMMIT_ON_SUCCESS` mode
`ocom()`	`OCITransCommit()`
`odefin(), odefinps()`	`OCIDefineByPos()` (Note: additional define calls may be necessary for some datatypes)
`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()`

Table 1–1 (Cont.) Obsolescent OCI Functions

7.*x* OCI Routine	Equivalent or Similar Later OCI Routine
oexec(), oexn()	OCIStmtExecute()
oexfet()	OCIStmtExecute(),OCIStmtFetch() (Note: result set rows can be implicitly prefetched)
ofen(),ofetch()	OCIStmtFetch()
oflng()	none
ogetpi()	OCIStmtGetPieceInfo()
olog()	OCILogon()
ologof()	OCILogoff()
onbclr(), onbset(), onbtst()	Note: nonblocking 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() preceding
oermsg()	OCIErrorGet()
olon()	OCILogon()
orlon()	OCILogon()
oname()	Note: see odescr() preceding
osql3()	Note: see oparse() preceding

OCI Routines Not Supported

Some OCI routines that were available in previous versions of OCI are not supported in later releases. They are listed in Table 1–2, " OCI Functions Not Supported":

Table 1–2 OCI Functions Not Supported

OCI Routine	Equivalent or Similar Later OCI Routine
obind()	OCIBindByName(),OCIBindByPos() (Note: additional bind calls may be necessary for some datatypes)
obindn()	OCIBindByName(),OCIBindByPos() (Note: additional bind calls may be necessary for some datatypes)
odfinn()	OCIDefineByPos() (Note: additional define calls may be necessary for some datatypes)
odsrbn()	Note: see odescr() in Table 1–1

Table 1–2 (Cont.) OCI Functions Not Supported

OCI Routine	Equivalent or Similar Later OCI Routine
`ologon()`	`OCILogon()`
`osql()`	Note: see `oparse()` Table 1–1

Compatibility Between Different Releases of OCI and Servers

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

Existing 7.*x* applications with no new post-release 7.*x* calls have to be re-linked with the new client-side library.

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.

Upgrading OCI

Programmers who wish to incorporate release post-release 7.*x* functionality into existing OCI applications have two options:

- Completely rewrite the application to use only new OCI calls (recommended).
- Incorporate new OCI post-release 7.*x* 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 Post-release 7.*x* 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 an `Lda_Def` to be used with 7.*x* OCI calls.

  See Also: See the description of OCIServerAttach() on page 16-99 and the description of OCISessionBegin() on page 16-99 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 post-release 7.*x* calls.
- To use release 7 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 that share the same server handle, only one can be in Oracle version 7 mode at any one time.

- To begin using post-release 7.*x* 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 OCI 7.*x* and post-release 7.*x* 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 post-release 7.*x* OCI calls.

Caution: You *cannot* open a cursor, parse with OCI 7.*x* calls and then execute the statement with post-release 7.*x* calls.

OCI Instant Client

The Instant Client feature makes it extremely easy to deploy OCI, OCCI, ODBC, and JDBC-OCI based customer applications by eliminating the need for an `ORACLE_HOME`. The storage space requirement of an OCI application running in Instant Client mode is significantly reduced compared to the same application running in a full client side installation. The Instant Client shared libraries only occupy about one-fourth the disk space of a full client installation.

Table 1–3 shows the Oracle client side files required to deploy an OCI application:

Table 1–3 OCI Instant Client Shared Libraries

Linux and UNIX	Description for Linux and UNIX	Windows	Description for Windows
`libclntsh.so.10.1`	Client Code Library	`oci.dll`	Forwarding functions that applications link with
`libociei.so`	OCI Instant Client Data Shared Library	`oraociei10.dll`	Data and code
`libnnz10.so`	Security Library	`orannzsbb10.dll`	Security Library

Oracle Database 10g Release 2 library names are used in the table.

To use the Microsoft ODBC and OLEDB driver, `ociw32.dll` must also be copied from ORACLE_HOME\bin.

Benefits of Instant Client

Why use Instant Client? Here are the reasons:

- Installation involves copying a small number of files.
- The Oracle client-side number of required files and the total disk storage are significantly reduced.
- There is no loss of functionality or performance for applications deployed in Instant Client mode.
- It is simple for independent software vendors to package applications.

OCI Instant Client Installation Process

The Instant Client libraries can also be installed by choosing the Instant Client option from the Oracle Universal Installer. The Instant Client libraries can also be downloaded from the Oracle Technology Network (http://www.oracle.com/technology/index.html) Web site. The installation process is as simple as:

1. Downloading and installing the Instant Client shared libraries to a directory such as instantclient_10_2.
2. Setting the operating system shared library path environment variable to the directory from step 1. For example, on Linux or UNIX, set the LD_LIBRARY_PATH to instantclient_10_2. On Windows, set PATH to locate the instantclient_10_2 directory.

After completing the preceding two steps you are ready to run the OCI application.

The OCI application operates in Instant Client mode when the OCI shared libraries are accessible through the operating system Library Path variable. In this mode, there is no dependency on ORACLE_HOME and none of the other code and data files provided in ORACLE_HOME are needed by OCI (except for the tnsnames.ora file described later).

Instant Client can be installed from the Oracle Universal Installer by selecting the Instant Client option. The installation should be done into an empty directory. As with the OTN install, you must set the LD_LIBRARY_PATH to the instant client directory to operate in instant client mode.

If you have done a complete client installation (by choosing the Admin option) the Instant Client shared libraries are also installed. The locations of the Instant Client shared libraries in a full client installation are:

On Linux or UNIX:

libociei.so library is in $ORACLE_HOME/instantclient

libclntsh.so.10.1 and libnnz10.so are in $ORACLE_HOME/lib

On Windows:

oraociei10.dll library is in ORACLE_HOME\instantclient

oci.dll, ociw32.dll, and orannzsbb10.dll are in ORACLE_HOME\bin

By copying the preceding libraries to a different directory and setting the operating system shared library path to locate this directory you can enable running the OCI application in Instant Client mode.

Note: All the libraries must be copied from the same ORACLE_HOME and must be placed in the same directory.

There should be only one set of Oracle libraries on the operating system Library Path variable. That is, if you have multiple directories containing Instant Client libraries, then only one such directory should be on the operating system Library Path.

Similarly, if an ORACLE_HOME-based installation is done on the same machine, then you should not have ORACLE_HOME/lib and Instant Client directory on the operating system Library Path simultaneously regardless of the order in which they appear on the Library Path. That is, only one of ORACLE_HOME/lib directory (for non-Instant Client operation) or Instant Client directory (for Instant Client operation) should be on the operating system Library Path variable.

To enable other capabilities such as OCCI and JDBC-OCI, a few other files need to be copied over as well. In particular, for the JDBC OCI driver, in addition to the three OCI shared libraries, you must also download OCI JDBC Library (for example libocijdbc10.so on Linux or UNIX and oraocijdbc10.dll on Windows) and ojdbcXY.jar (where XY is the version number, for example, ojdbc14.jar). All libraries must be in the Instant Client directory and ojdbcXY.jar must be able to be loaded from CLASSPATH.

Note: On hybrid platforms, such as Sparc64, if the JDBC OCI driver needs to be operated in the Instant Client mode, the libociei.so library must be copied from the ORACLE_HOME/instantclient32 directory. All other Sparc64 libraries needed for the JDBC OCI Instant Client must be copied from the ORACLE_HOME/lib32 directory.

For OCCI, the OCCI Library (libocci.so.10.1 on Linux or UNIX and oraocci10.dll on Windows) must also be installed in the Instant Client directory.

When to Use Instant Client

Instant Client is a deployment feature and should be used for running production applications. In general, all OCI functionality is available to an application being run in the Instant Client mode, except that the Instant Client mode is for client-side operation only. Therefore, server-side external procedures cannot operate in the Instant Client mode.

For development you can also use the Instant Client SDK.

See Also: "SDK for Instant Client" on page 1-24

Patching Instant Client Shared Libraries on Linux or UNIX

Because Instant Client is a deployment feature, the emphasis has been on reducing the number and size of files (client footprint) required to run an OCI application. Hence all files needed to patch Instant Client shared libraries are not available in an Instant Client deployment. An ORACLE_HOME based full client installation is needed to patch

the Instant Client shared libraries. The opatch utility will take care of patching the Instant Client shared libraries.

After applying the patch in an ORACLE_HOME environment, copy the files listed in Table 1–3, " OCI Instant Client Shared Libraries" to the instant client directory as described in "OCI Instant Client Installation Process" on page 1-17.

Instead of copying individual files, you can generate Instant Client zip files for OCI/OCCI, JDBC, and SQL*Plus as described in "Regeneration of Data Shared Library and Zip Files" on page 1-19. Then, instead of copying individual files as described above, you can instead copy the zip files to the target machine and unzip them as described in"OCI Instant Client Installation Process" on page 1-17.

The opatch utility stores the patching information of the ORACLE_HOME installation in libclntsh.so . This information can be retrieved by the following command:

```
genezi -v
```

Note that if the Instant Client deployment machine does not have the genezi utility, then it must be copied from the ORACLE_HOME/bin directory of the ORACLE_HOME machine.

Note: The opatch utility is not available on Windows.

Regeneration of Data Shared Library and Zip Files

The OCI Instant Client Data Shared Library (libociei.so) can be regenerated by performing the following steps in an Administrator Install of ORACLE_HOME:

```
mkdir -p $ORACLE_HOME/rdbms/install/instantclient/light
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ilibociei
```

A new version of libociei.so based on the current files in the ORACLE_HOME is then placed in the ORACLE_HOME/rdbms/install/instantclient directory.

Note that the location of the regenerated Data Shared Library (libociei.so) is different from the original Data Shared Library (libociei.so) which is located in the ORACLE_HOME/instantclient directory.

The above steps also generate Instant Client zip files for OCI/OCCI, JDBC, and SQL*Plus.

Regeneration of data shared library and zip file is not available on Windows platforms.

Database Connection Strings for OCI Instant Client

The OCI Instant Client can make remote database connections in all the ways that ordinary SQL clients can. However because the Instant Client does not have the ORACLE_HOME environment and directory structure some of the database naming methods will require additional configuration steps.

All Oracle net naming methods that do not require use of ORACLE_HOME or TNS_ADMIN (to locate configuration files such as tnsnames.ora or sqlnet.ora) work in the Instant Client mode. In particular, the connect_identifier in the OCIServerAttach() call can be specified in the following formats:

- A SQL Connect URL string of the form:

  [//]*host*[:*port*][/*service name*]

such as:

```
//dlsun242:5521/bjava21
```

- As an Oracle Net connect descriptor. For example:

```
"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp) (HOST=dlsun242) (PORT=5521))
(CONNECT_DATA=(SERVICE_NAME=bjava21)))"
```

- A Connection Name that is resolved through Directory Naming where the site is configured for LDAP server discovery.

For naming methods such as `tnsnames` and directory naming to work the `TNS_ADMIN` environment variable must be set.

See Also: ■ *Oracle Database Net Services Administrator's Guide* chapter on "Configuring Naming Methods" for more about connect descriptors

If the `TNS_ADMIN` environment variable is not set, and `TNSNAMES` entries such as `inst1`, and so on, are used, then the `ORACLE_HOME` variable must be set, and the configuration files are expected to be in the `$ORACLE_HOME/network/admin` directory.

Note that the `ORACLE_HOME` variable in this case is only used for locating Oracle Net configuration files, and no other component of Client Code Library (OCI, NLS, and so on) uses the value of `ORACLE_HOME`.

If a `NULL` string, "", is used as the connection string in the `OCIServerAttach()` call, then the `TWO_TASK` environment variable can be set to the `connect_identifier`. On Windows platform, the `LOCAL` environment variable is used instead of `TWO_TASK`.

Similarly for OCI command line applications such as SQL*Plus, the `TWO_TASK` (or `LOCAL` on Windows) environment variable can be set to the connect_identifier. Its value can be anything that would have gone to the right of the '@' on a typical connect string.

Examples of Instant Client Connect Identifiers

If you are using SQL*Plus in Instant Client mode, then you can specify the connect identifier in the following ways:

If the `listener.ora` file on the Oracle database server contains the following:

```
LISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=1573))
)

SID_LIST_LISTENER = (SID_LIST=
 (SID_DESC=(SID_NAME=rdbms3)(GLOBAL_DBNAME=rdbms3.server6.us.alchemy.com)
(ORACLE_HOME=/home/dba/rdbms3/oracle))
)
```

The SQL*Plus connect identifier is:

```
"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=1573))(CONNECT_DATA=
(SERVICE_NAME=rdbms3.server6.us.alchemy.com)))"
```

or

```
"//server6:1573/rdbms3.server6.us.alchemy.com"
```

Alternatively, you can set the TWO_TASK environment variable to any of the previous connect identifiers and connect without specifying the connect identifier. For example:

```
setenv TWO_TASK "(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=1573))
(CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.us.alchemy.com)))"
```

or

```
setenv TWO_TASK //server6:1573/rdbms3.server6.us.alchemy.com
```

and invoke SQL*Plus with an empty connect identifier:

```
sqlplus user/password
```

The connect descriptor can also be stored in the tnsnames.ora file. For example, if the tnsnames.ora file contains the following connect descriptor:

```
conn_str =
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=1573))(CONNECT_DATA=
(SERVICE_NAME=rdbms3.server6.us.alchemy.com)))
```

and the tnsnames.ora is located in the /home/webuser/instantclient directory, then you can set the variable TNS_ADMIN (or LOCAL on Windows) as:

```
setenv TNS_ADMIN /home/webuser/instantclient
```

and then use the connect identifier conn_str for invoking SQL*Plus, or for your OCI connection.

> **Note:** TNS_ADMIN specifies the directory where the tnsnames.ora file is located and TNS_ADMIN is not the full path of the tnsnames.ora file.

If the above tnsnames.ora file is located in an ORACLE_HOME-based install in the /network/server6/home/dba/oracle/network/admin directory, then the ORACLE_HOME environment variable can be set as:

```
setenv ORACLE_HOME /network/server6/home/dba/oracle
```

and SQL*Plus can be invoked as previously, with the identifier conn_str.

Finally, if tnsnames.ora can be located by TNS_ADMIN or ORACLE_HOME, then TWO_TASK can be set to:

```
setenv TWO_TASK conn_str
```

and SQL*Plus can be invoked without a connect identifier.

Environment Variables for OCI Instant Client

The ORACLE_HOME environment variable no longer determines the location of NLS, CORE, and error message files. An OCI-only application should not require ORACLE_HOME to be set. However, if it is set, it does not have an impact on OCI's operation. OCI will always obtain its data from the Data Shared Library. If the Data Shared Library is not available, only then is ORACLE_HOME used and a full client installation is assumed. Even though ORACLE_HOME is not required to be set, if it is set, then it must be set to a valid operating system path name that identifies a directory.

If Dynamic User callback libraries are to be loaded, then as this guide specifies, the callback package has to reside in `ORACLE_HOME/lib` (`ORACLE_HOME\bin` on Windows). Therefore, `ORACLE_HOME` should be set in this case.

Environment variables `ORA_NLS10` and `ORA_NLS_PROFILE33` are ignored in the Instant Client mode.

In the Instant Client mode, if the `ORA_TZFILE` variable is not set, then the smaller, default, `timezone.dat` file from the Data Shared Library is used. If the larger `timezlrg.dat` file is to be used from the Data Shared Library, then set the `ORA_TZFILE` environment variable to the name of the file without any absolute or relative path names. That is, on Linux or UNIX:

```
setenv ORA_TZFILE timezlrg.dat
```

On Windows:

```
set ORA_TZFILE=timezlrg.dat
```

If OCI is not operating in the Instant Client mode (because the Data Shared Library is not available), then `ORA_TZFILE` variable, if set, names a complete path name as it does in previous Oracle releases.

If `TNSNAMES` entries are used, then, as mentioned earlier, `TNS_ADMIN` directory must contain the `TNSNAMES` configuration files, and if `TNS_ADMIN` is not set, then the `ORACLE_HOME/network/admin` directory must contain Oracle Net Services configuration files.

Instant Client Light (English)

The Instant Client Light (English) version of Instant Client further reduces the disk space requirements of the client installation by about another 63 MB. For example, the Instant Client Light data shared library, `libociicus.so` on Unix operating systems, occupies only 4 MB, as opposed to 70 MB for `libociei.so`.

This Instant Client Light version is geared toward applications that require English-only error messages and use either US7ASCII, WE8DEC, or one of the Unicode characters. There is no restriction on the `TERRITORY` field of the `NLS_LANG` setting, so the Instant Client Light will operate with any territory setting. Therefore, an application using US7ASCII, WE8DEC, or Unicode can significantly reduce its footprint if it operates in the Instant Client Light environment.

Globalization Settings

Instant Client Light supports the following character sets:

Single-byte

- US7ASCII
- WE8DEC
- WE8MSWIN1252
- WE8ISO8859P1

Unicode

- UTF8
- AL16UTF16
- AL32UTF8

Instant Client Light will return an error if a character set or national character set other than those in the preceding lists is used as the client or database character set. With Instant Client Light, the error messages are only obtained in English. Therefore, in setting NLS_LANG, the valid values are:

American_*territory*.*charset*

where *territory* can be any valid territory that can be specified with NLS_LANG and *charset* is one of the character sets listed above.

Instant Client Light can also operate with the OCI Environment handles created in the OCI_UTF16 mode.

> **See Also:** *Oracle Database Globalization Support Guide* for more information about NLS settings

Operation of Instant Client Light

OCI applications, by default, look for the OCI Data Shared Library, libociei.so (or Oraociei10.dll on Windows) on the LD_LIBRARY_PATH (PATH on Windows) to determine if the application should operate in the Instant Client mode. If this library is not found, then OCI tries to load the Instant Client Light Data Shared Library, libociicus.so (or Oraociicus10.dll on Windows). If the Instant Client Light library is found, then the application operates in the Instant Client Light mode. Otherwise, a full ORACLE_HOME based installation is assumed

Installation of Instant Client Light

Instant Client Light can be installed in one of the following ways:

1. From OTN.

   Go to the Instant Client URL:

   http://www.oracle.com/technology/software/tech/oci/instantclient/

   For Instant Client Light, instead of downloading and expanding the basic.zip package, download and unzip the basiclite.zip package. The instantclient_10_2 directory in which the Instant Client Light libraries are unzipped should be empty before the unzip.

2. From Client Admin Install.

   Instead of copying libociei.so (or Oraociei10.dll on Windows) from the ORACLE_HOME/instantclient directory, copy libociicus.so (or Oraociic10.dll on Windows) from the ORACLE_HOME/instantclient/light subdirectory. That is, the Instant Client directory on the LD_LIBRARY_PATH (PATH on Windows) should contain the Instant Client Light Data Shared Library, libociicus.so (Oraociicus10.dll on Windows), instead of the larger OCI Instant Client Data Shared Library, libociei.so (Oraociei10.dll on Windows).

3. From Oracle Universal Installer.

   If the Instant Client option is selected from the Oracle Universal Installer (OUI), then libociei.so (or Oraociei10.dll on Windows) is installed in the base directory of the installation which is going to be placed on the LD_LIBRARY_PATH (PATH on Widows). This is so that the Instant Client Light is not enabled by default. The Instant Light Client Data Shared Library, libociicus.so (or Oraociicus10.dll on Windows), is installed in the light subdirectory of the base directory. Therefore, to operate in the Instant Client Light mode, the OCI Data

Shared Library, `libociei.so` (or `Oraociei10.dll` on Windows) must be deleted or renamed and the Instant Client Light library must be copied up from the `light` subdirectory to the base directory of the installation.

For example, if the OUI has installed the Instant Client in `my_oraic_10_2` directory on the `LD_LIBRARY_PATH` (`PATH` on Windows), then you need to do the following to operate in the Instant Client Light mode:

```
cd my_oraic_10_2
rm libociei.so
mv light/libociicus.so .
```

Note: All the Instant Client files should always be copied and installed in an empty directory. This is to make sure that no incompatible binaries exist in the installation.

SDK for Instant Client

The SDK can be downloaded from the Instant Client web page:

`http://www.oracle.com/technology/tech/oci/instantclient/instantclient.html`

- The Instant Client SDK package has both C and C++ header files and a Makefile for developing OCI and OCCI applications while in an Instant Client environment. Developed applications can be deployed in any client environment.
- The SDK contains C and C++ demonstration programs.
- On Windows, libraries required to link the OCI or OCCI applications are also included. `Make.bat` is provided to build the demos.
- On Unix or Linux, the Makefile `demo.mk` is provided to build the demos. The `instantclient_10_2` directory must be on the `LD_LIBRARY_PATH` before linking the application. The OCI and OCCI programs require the presence of `libclntsh.so` and `libocci.sh` symbolic links in the `instantclient_10_2` directory. `demo.mk` creates these before the link step. These symbolic links can also be created in a shell:

  ```
  cd instantclient_10_2
  ln -s libclntsh.so.10.1 libclntsh.so
  ln -s libocci.so.10.1 libocci.so
  ```

- The SDK also contains the Object Type Translator (OTT) utility and its classes to generate the application header files.

2

OCI Programming Basics

This chapter introduces you to the basic concepts involved in programming with the OCI.

This chapter contains these topics:

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:

- Header Files - gives the location of header files for OCI client application development.
- OCI Program Structure - covers the basic structure of, and the major steps involved in creating an OCI application.
- 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 in OCI - covers error handling in OCI applications.
- Additional Coding Guidelines - provides useful information for coding an OCI application.
- Using PL/SQL in an OCI Program - discusses important points for 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 *Oracle Database Globalization Support Guide*
- For a discussion of the OCI functions that apply to cartridge services, see the *Oracle Database Data Cartridge Developer's Guide.*

Header Files

With the current release, the OCI/OCCI header files that are required for OCI and OCCI client application development on UNIX platforms reside in the `$ORACLE_HOME/rdbms/public` directory. The `demo_rdbms.mk` file remains in.the `$ORACLE_HOME/rdbms/demo` directory and continues to serve as an example makefile.

Unless you significantly modified the `demo_rdbms.mk` file, you are not affected. This is because the `demo_rdbms.mk` file already includes the `$ORACLE_HOME/rdbms/public` directory. Ensure that your highly customized makefiles have the `$ORACLE_HOME/rdbms/public` directory in the INCLUDE path.

All demonstration programs and header files continue to reside in the `$ORACLE_HOME/rdbms/demo` directory. As with all demonstrations, these files are only installed from the Companion CD. See Appendix B, "OCI Demonstration Programs" for the names of these programs and their purposes.

The OCI/OCCI header files required for development, located in `$ORACLE_HOME/rdbms/public`, are available both with the Oracle Database Server installation, and with the Oracle Database Client Administration and Custom installations.

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.

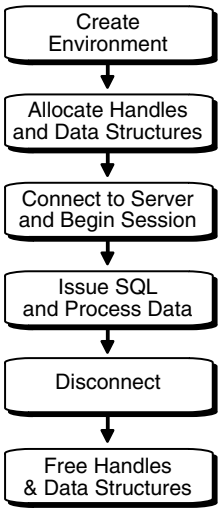
OCI uses the following basic program structure:

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. Execute 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 2-13.

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. If an environment requires any process-level initialization, then it is performed automatically.

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 subsequent chapters

OCI Data Structures

Handles and *descriptors* are opaque data structures which are defined in OCI applications. They can be allocated directly, through specific allocate calls, or they can 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 next section:

> **See Also:** Descriptors are discussed in the section "OCI Descriptors" on page 2-9

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, (for example, an environment or service context handle), or it may store information about OCI functions or data (for example, 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 and set this information.

> **See Also:** "Handle Attributes" on page 2-8

Table 2–1 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 Datatype	Handle Type Constant
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 authentication information handle	`OCIAuthInfo`	`OCI_HTYPE_AUTHINFO`
OCI connection pool handle	`OCICPool`	`OCI_HTYPE_CPOOL`
OCI session pool handle	`OCISPool`	`OCI_HTYPE_SPOOL`
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`

Table 2–1 (Cont.) OCI Handle Types

Description	C Datatype	Handle Type Constant
OCI process handle	`OCIProcess`	`OCI_HTYPE_PROCESS`
OCI administration handle	`OCIAdmin`	`OCI_HTYPE_ADMIN`
OCI HA event handle	`OCIEvent`	N/A

Allocating and Freeing Handles

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

The environment handle is allocated and initialized with a call to `OCIEnvCreate()` or to `OCIEnvNlsCreate()`, one of which is required by all OCI applications.

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

Here are the various types of handles: session handle, direct path context handle, thread handle, COR handle, subscription handle, describe handle, statement handle, service context handle, error handle, server handle, connection pool handle, event handle, and administration handle.

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 all 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.

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"

Environment Handle

The *environment handle* defines a context in which all OCI functions are invoked. Each environment handle contains a memory cache, which enables 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

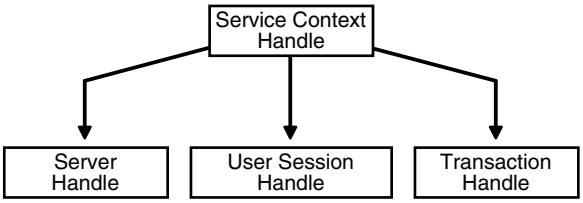
The *error handle* is passed as a parameter to most OCI calls. The error handle maintains information about errors that occur during an OCI operation. If an error occurs in a call, the error handle can be passed to `OCIErrorGet()` to obtain additional information about the error that occurred.

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 2–2, "Components of a Service Context":

Figure 2–2 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 in 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 any fetch state and package instantiation.

Breaking the service context down in this way provides scalability and enables programmers to create sophisticated multitiered applications and transaction processing (TP) monitors for 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 for each 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 and user session handles 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.

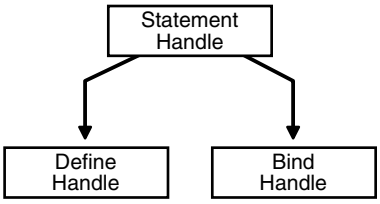
See Also:

- "OCI Support for Transactions" on page 8-1
- For more information about establishing a server connection and user session, see the sections "OCI Environment Initialization" on page 2-14, and "Password and Session Management" on page 8-7

Statement, Bind, and Define Handles

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

Figure 2–3 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.

Describe Handle

The *describe handle* is used by the OCI describe call, `OCIDescribeAny()`. This call obtains information about schema objects in a database (for example, 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 for retrieving objects referenced by another object.

See Also : "Complex Object Retrieval" on page 10-15

Thread Handle

For information about the thread handle, which is used in multithreaded applications:

See Also: "The OCIThread Package" on page 9-3

Subscription Handle

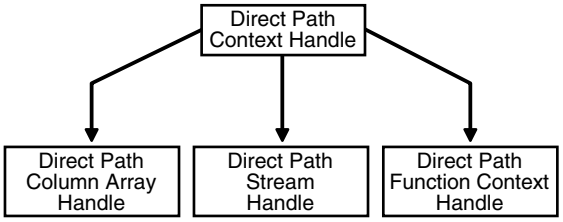
The subscription handle is used by an OCI client application that registers and subscribes to receive notifications of database events or events in the AQ namespace. The subscription handle encapsulates all information related to a registration from a client.

See Also: "Publish-Subscribe Notification in OCI" on page 9-47

Direct Path Handles

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

Figure 2–4 Direct Path Handles



See Also:

- "Direct Path Loading Overview" on page 12-1
- "Direct Path Loading Handle Attributes" on page A-51

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 in OCI" on page 9-7

Handle Attributes

All OCI handles have *attributes* that 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 user name in the session handle by writing to the OCI_ATTR_USERNAME attribute:

```
text username[] = "hr";
err = OCIAttrSet ((dvoid*) mysessp, OCI_HTYPE_SESSION, (dvoid*)username,
      (ub4) strlen((char *)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 user name 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:

- The description of OCIAttrGet() on page 15-48 for an example showing the user name and password handle attributes being set
- Appendix A, "Handle and Descriptor Attributes"

OCI Descriptors

OCI *descriptors* and *locators* are opaque data structures that maintain data-specific information. Table 2–2 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 Datatype	OCI Type Constant
snapshot descriptor	OCISnapshot	OCI_DTYPE_SNAP
result set descriptor	OCIResult	OCI_DTYPE_RSET
LOB datatype locator	OCILobLocator	OCI_DTYPE_LOB
BFILE 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

Table 2–2 (Cont.) Descriptor Types

Description	C Datatype	OCI Type Constant
user callback descriptor	OCIUcb	OCI_DTYPE_UCB
the distinguished names of the database servers in a registration request	OCIServerDNs	OCI_DTYPE_SRVDN
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
advanced queuing agent	OCIAQAgent	OCI_DTYPE_AQAGENT
advanced queuing notification	OCIAQNotify	OCI_DTYPE_AQNFY
advanced queuing listen options	OCIAQListenOpts	OCI_DTYPE_AQLIS_OPTIONS
advanced queuing message properties	OCIAQLisMsgProps	OCI_DTYPE_AQLIS_MSG_PROPERTIES
change notification	none	OCI_DTYPE_CHDES
table change	none	OCI_DTYPE_TABLE_CHDES
row change	none	OCI_DTYPE_ROW_CHDES

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 BFILE (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 which represents the state of a database at a particular point in time.

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 4-6

LOB and BFILE Locators

A large object (LOB) is an Oracle datatype that can hold 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.

Note: Depending on your application, you may or may not want to use LOB locators. You can use the data interface for LOBs, which does not require LOB locators. In this interface, you can bind or define character data for `CLOB` columns or `RAW` data for `BLOB` columns.

See Also:

- "Binding LOB Data" on page 5-8
- "Defining LOB Data" on page 5-15

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

The LOB locator is allocated with a call to `OCIDescriptorAlloc()`, by passing `OCI_DTYPE_LOB` as the `type` parameter for `BLOB`s or `CLOB`s, and `OCI_DTYPE_FILE` for `BFILE`s.

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.

See Also:

- Chapter 7, "LOB and BFILE Operations"
- "Binding LOB Data" on page 5-8
- "Defining LOB Data" on page 5-15

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

> **See Also:** Chapter 6, "Describing Schema Metadata", and "Describing Select-list Items" on page 4-9 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. 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.

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

Date, Datetime, and Interval Descriptors

These descriptors are used by applications which use the date, datetime, or interval datatypes (`OCIDate`, `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 described in Chapter 18, "OCI Datatype Mapping and Manipulation Functions"

Complex Object Descriptor

Application performance when dealing with objects may be increased through the use of *complex object retrieval (COR)*.

> **See Also:** For information about the complex object descriptor and its use, refer to "Complex Object Retrieval" on page 10-15.

Advanced Queuing Descriptors

Oracle AQ provides message queuing as an integrated part of the Oracle server.

> **See Also:**
>
> - "OCI and Streams Advanced Queuing" on page 9-39
> - "Publish-Subscribe Registration Functions in OCI" on page 9-48

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 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 in developing an OCI application is described in 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.

See Also:

- Appendix B, "OCI Demonstration Programs" for an example showing the use of OCI calls for processing SQL statements. See the first sample program.
- The special case of dynamically providing data at run time is described in detail in the section "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-29.
- Special considerations for operations involving arrays of structures are described in the section "Binding and Defining Arrays of Structures in OCI" on page 5-16.
- Refer to the section "Error Handling in OCI" on page 2-20 for an outline of the steps involved in processing a SQL statement within an OCI program.
- For information on using the OCI to write multithreaded applications, refer to "Overview of OCI Multithreaded Development" on page 9-1.
- For more information about types of SQL statements, refer to the section "SQL Statements" on page 1-4.

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

- OCI Environment Initialization
- Processing SQL Statements in OCI
- Commit or Rollback
- Terminating the Application

- Error Handling in OCI

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

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 the database.

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

- "Creating the OCI Environment" on page 2-14
- "Allocating Handles and Descriptors" on page 2-14
- "Application Initialization, Connection, and Session Creation" on page 2-15

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 is executed. The only exception is the setting of a process-level attribute for the OCI shared mode.

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 subscriptions (`mode` = `OCI_EVENTS`).

The mode can be set independently in each environment.

It is necessary to initialize in object mode if the application binds and defines objects, or if it uses 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 uses objects.

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

See Also:

- OCIEnvCreate() on page 15-13 and OCIInitialize() on page 15-22 for more information about the initialization calls.
- "Overview of OCI Multithreaded Development" on page 9-1.
- Chapter 10, "OCI Object-Relational Programming" and the chapters that follow it.
- "Publish-Subscribe Notification in OCI" on page 9-47.

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 types of handles listed in Table 2–1, " OCI Handle Types"with `OCIHandleAlloc()` Depending on the functionality of your application, it needs to allocate some or all of these handles.

Application Initialization, Connection, and Session Creation

An application must call `OCIEnvNlsCreate()` to initialize the OCI environment handle. Existing applications may have used `OCIEnvCreate()`.

Following this step, the application has several options for establishing a server connection and beginning a user session.

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

Single User, Single Connection

This option is the simplified logon method, which can be used if an application maintains only a single user session for each database connection at any time.

When an application calls `OCILogon2()`, the OCI library initializes the service context handle that is passed to it, and creates a connection to the specified server for the user making the request.

The following is an example of what a call to `OCILogon2()` looks like for a single user session with user name `hr`, password `hr`, and database `oracledb`:

```
OCILogon2(envhp, errhp, &svchp, (text *)"hr", (ub4)strlen("hr"), (text *)"hr",
          (ub4)strlen("hr"), (text *)"oracledb", (ub4)strlen("oracledb"),
          OCI_DEFAULT);
```

The parameters to this call include the service context handle (which has been initialized), the user name, the user's password, and the name of the database that are used to establish the connection. With the last parameter, `mode`, set to `OCI_DEFAULT`, this call has the same effect as calling the older `OCILogon()`. Use `OCILogon2()` for any new applications. The server and user session handles are 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; the application cannot switch session or transaction by changing the appropriate attributes of the service context handle by means of an `OCIAttrSet()` call.

An application that initializes its session and authorization using `OCILogon2()` must terminate them using `OCILogoff()`.

Client Access Through a Proxy

Proxy authentication is a process typically employed in an environment with a middle tier such as a firewall, in which the end user authenticates to the middle tier, which then authenticates to the database on the user's behalf—as its proxy. The middle tier logs into the database as a proxy user. A proxy user can switch identities and, once logged into the database, switch to the end user's identity. It can perform operations on the end user's behalf, using the authorization appropriate to that particular end user.

Proxy to database users is supported by means of OCI and the `ALTER USER` statement, whose BNF syntax is:

```
ALTER USER <targetuser> GRANT CONNECT THROUGH <proxy> [AUTHENTICATION REQUIRED];
```

The ALTER USER statement is used once in an application. Connections can be made multiple times afterwards. In OCI, you can either use connect strings or the function OCIAttrSet() with the parameter OCI_ATTR_PROXY_CLIENT.

After a proxy switch is made, the current and connected user is the target user of the proxy. The identity of the original user is not used for any privilege calculations. The original user can be a local or external user.

The following examples show connect strings that can be used in functions such as OCILogon2() (set mode = OCI_DEFAULT), OCILogon(), OCISessionBegin() with OCIAttrSet() (pass the attribute OCI_ATTR_USERNAME of the session handle), and so on:

1. Local user acting on behalf of a local user.

   Dilbert and Joe are two local database users. To enable Dilbert to proxy on behalf of Joe, use the following SQL statement:

   ```
   ALTER USER joe GRANT CONNECT THROUGH dilbert;
   ```

   When user name dilbert is acting on behalf of joe, the connection string is (dilbert has password tiger):

   ```
   dilbert[joe]/tiger@db1
   ```

   The "[" and "]" are actually entered in the connection string.

2. Local user acting on behalf of local user, where user names must be quoted.

   "Dilbert" and "Joe" are two local database users. The names are case sensitive and need to be quoted. To enable "Dilbert" to proxy on behalf of "Joe", use the following statement:

   ```
   ALTER USER "Joe" GRANT CONNECT THROUGH "Dilbert";
   ```

   When "Dilbert" is acting on behalf of "Joe" the connection string is (be sure to also include the " characters):

   ```
   "Dilbert"["Joe"]/tiger@db1
   ```

3. Local user dilbert[mybert] connecting to database.

   There is a user in the database "dilbert[mybert]" and the way this user will connect to the database is (the "[" and "]" are actually entered in the connection string):

   ```
   "dilbert[mybert]"/tiger

   rem the user was already created this way:
   rem CREATE USER "dilbert[mybert]" IDENTIFIED BY tiger;
   ```

4. Local user acting on behalf of local user, where the user name has [].

   dilbert[mybert] and joe[myjoe] are two database users that contain the characters "[" and "]". If dilbert[mybert] wants to act on behalf of joe[myjoe], the connect statement is:

   ```
   "dilbert[mybert]"["joe[myjoe]"]/tiger
   ```

5. You can set the target user name by means of the ALTER USER statement, followed by an OCI program in which OCIAttrSet() sets the attribute OCI_ATTR_PROXY_CLIENT and the proxy dilbert. For example:

```
ALTER USER joe GRANT CONNECT THROUGH dilbert;
```

In your program, use these statements to connect multiple times:

```
OCIAttrSet(session, OCI_HTYPE_SESSION, (dvoid *)"dilbert",
           (ub4)strlen("dilbert"), OCI_ATTR_USERNAME,
           error_handle);
OCIAttrSet(session, OCI_HTYPE_SESSION, (dvoid *)"tiger",
           (ub4)strlen("tiger"), OCI_ATTR_PASSWORD,
           error_handle);
OCIAttrSet(session, OCI_HTYPE_SESSION, (dvoid *)"joe",
           (ub4)strlen("joe"), OCI_ATTR_PROXY_CLIENT,
           error_handle);
```

See Also:

- "OCI_ATTR_PROXY_CLIENT" on page A-17
- *Oracle Database Security Guide* for a discussion of proxy authentication
- "Password and Session Management" on page 8-7
- "OCIAttrSet()" on page 15-50

Compatibility Issues of Client Access Through a Proxy Since this feature was introduced in release 10.2, pre-10.2 clients do not have it. If newer clients use the feature with pre-10.2 databases, the connect will fail and the client will return an error after checking the database release level.

Non-Proxy Multiple Sessions or Connections

This option uses explicit attach and begin-session calls to maintain multiple user sessions and connections on a database connection. Specific calls to attach to the server and begin sessions are:

- `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 execute operations on the server.

A subsequent call to `OCISessionBegin()` using different service context and session context handles logs off the previous user and causes an error. To run two simultaneous non-migratable sessions, a second `OCISessionBegin()` call must be made with the same service context handle and a new session context handle.

These calls set up an operational environment that enables you to execute SQL and PL/SQL statements against a database.

See Also:

- "Connect, Authorize, and Initialize Functions" on page 15-3.
- Chapter 9, "OCI Programming Advanced Topics", for more information about maintaining multiple sessions, transactions, and connections.
- "Client Character Set Control from OCI" on page 2-28 for the use of `OCIEnvNlsCreate()`.

Example of Creating and Initializing an OCI Environment

The following example demonstrates the use of creating and initializing an OCI environment.

- A server context is created and set in the service handle.
- Then a user session handle is created and initialized using a database user name and password.
- For the sake of simplicity, error checking is not included.

```
#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 user name attribute in user session handle */
 (void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
      (dvoid *)"hr", (ub4)strlen("hr"),
      OCI_ATTR_USERNAME, myerrhp);

      /* set password attribute in user session handle */
 (void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
      (dvoid *)"hr", (ub4)strlen("hr"),
      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 in OCI

A chapter of this manual outlines the specific steps involved in processing SQL statements in OCI.

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

Commit or Rollback

An application commits changes to the database by calling `OCITransCommit()`. This call uses a service context as one of its parameters. The transaction is associated with the service context whose changes are committed. This transaction can be explicitly created by the application or implicitly 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 round trip.

To roll back a transaction, use the `OCITransRollback()` call.

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

See Also:

- "Service Context and Associated Handles" on page 2-6, and
- "OCI Support for Transactions" on page 8-1

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 uses 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 allocated.

Error Handling in OCI

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.

To verify that the connection to the server is not terminated by the `OCI_ERROR`, an application can check the value of the attribute `OCI_ATTR_SERVER_STATUS` in the server handle. If the value of the attribute is `OCI_SERVER_NOT_CONNECTED`, then the connection to the server and the user session must be reestablished.

See Also:

- For exceptions, see "Functions Returning Other Values" on page 2-22
- For complete details and an example of usage, see "OCIErrorGet()" on page 16-176
- "Server Handle Attributes" on page A-10

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.

Table 2–3 (Cont.) OCI Return Codes

OCI Return Code	Description
OCI_NEED_DATA	The application must provide runtime data.
OCI_STILL_EXECUTING	The service context was established in nonblocking 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.

Return and Error Codes for Data

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

See Also: "Indicator Variables" on page 2-23 for a discussion of indicator variables.

Table 2–4 Return and Error Codes

State of Data	Return Code	Indicator - not provided	Indicator - provided
not null or truncated	not provided	OCI_SUCCESS error = 0	OCI_SUCCESS error = 0 indicator = 0
not null or truncated	provided	OCI_SUCCESS error = 0 return code = 0	OCI_SUCCESS error = 0 indicator = 0 return code = 0
null data	not provided	OCI_ERROR error = 1405	OCI_SUCCESS error = 0 indicator = -1
null data	provided	OCI_ERROR error = 1405 return code = 1405	OCI_SUCCESS error = 0 indicator = -1 return code = 1405
truncated data	not provided	OCI_ERROR error = 1406	OCI_ERROR error = 1406 indicator = data_len

Table 2–4 (Cont.) Return and Error Codes

State of Data	Return Code	Indicator - not provided	Indicator - provided
truncated data	provided	OCI_SUCCESS_WITH_INFO error = 24345 return code = 1405	OCI_SUCCESS_WITH_INFO error = 24345 indicator = data_len return code = 1406

For truncated data, `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 aware that they return values 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.

Additional Coding Guidelines

This section explains some additional issues when coding OCI applications.

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:** "Connect, Authorize, and Initialize Functions" on page 15-3 for more information about parameter datatypes and parameter passing conventions.

Address Parameters

Address parameters are used to pass the address of the variable to Oracle. You should be careful when developing in C, since it normally passes scalar parameters by value.

Integer Parameters

Binary integer and short binary integer parameters are numbers whose size is 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. Each OCI routine that enables 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.

1. One method is to use a literal NULL in the text of an INSERT or UPDATE statement. For example, the SQL statement

```
INSERT INTO emp1 (ename, empno, deptno)
   VALUES (NULL, 8010, 20)
```

makes the ENAME column NULL.

1. Use indicator variables in the OCI bind call.

   See Also: "Indicator Variables" on page 2-23

2. 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 associates an indicator variable, or an array of indicator variables, with a DML statement, a PL/SQL statement, or a query.

The C language 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 in an OCIStmtFetch() call, or a truncation in an OCIStmtExecute() call, the OCI call returns OCI_SUCCESS_WITH_INFO. The output indicator variable is set.

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:

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:

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

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:

- Documentation for the OTT in Chapter 14, "Using the Object Type Translator with OCI", and section "NULL Indicator Structure" on page 10-22 of this manual for information about null indicator structures
- Descriptions of `OCIBindByName()` and `OCIBindByPos()` in"Bind, Define, and Describe Functions" on page 15-59, and the sections "Information for Named Datatype and REF Binds" on page 11-26, and "Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-27, for more information about setting indicator parameters for named datatypes and REFs

Canceling Calls

On most operating systems, you can cancel long-running or repeated OCI calls, 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) stop of any currently executing OCI

function 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 nonblocking connection after an OCI application stops a function with OCIBreak().

Note: OCIBreak() works on Windows systems, including Windows 2000, and Windows XP.

The status of potentially long-running calls can be monitored through the use of nonblocking calls. Use multithreading for new applications.

See Also:

- "Overview of OCI Multithreaded Development" on page 9-1
- "The OCIThread Package" on page 9-3

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 emp1 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() */
status = OCIDescriptorAlloc ((dvoid *) envhp, (dvoid **) &rowid,
     (ub4) OCI_DTYPE_ROWID, (size_t) 0, (dvoid **) 0);
status = 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 emp1 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 3-5 and "DATE" on page 3-11.

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 Oracle Database SQL Reference and the Oracle Database PL/SQL User's Guide and Reference

Oracle Reserved Namespaces

Table 2–5, " 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.

Table 2–5 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
KO	Kernel Objects

For a complete list of functions within a particular namespace, refer to the document that corresponds to the appropriate Oracle library.

Nonblocking Mode in OCI

Because nonblocking mode increases the number of round trips and the CPU usage, use multithreaded calls for new applications.

See Also:

- "Overview of OCI Multithreaded Development" on page 9-1
- "The OCIThread Package" on page 9-3

The OCI provides the ability to establish a server connection in *blocking mode* or *nonblocking 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 nonblocking 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 nonblocking mode, an application must test the return code of each OCI function to see if it returns `OCI_STILL_EXECUTING`. If it does, the OCI client can continue to process program logic while waiting to retry the OCI call to the server. This mode is particularly useful in Graphical User Interface (GUI) applications, real-time applications, and in distributed environments.

The nonblocking 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 by executing the call again *with the exact same parameters*.

Note: While waiting to retry a nonblocking OCI call, the application *cannot* issue other OCI calls, or an ORA-03124 error will occur; the only exceptions to this rule are the calls `OCIBreak()` and `OCIReset()`.

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`. You must set this attribute only after `OCISessionBegin()` or `OCILogon2()` has been called. Otherwise, an error will be returned.

See Also: "Server Handle Attributes" on page A-10

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

Cancelling a Nonblocking Call

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

Using PL/SQL in an OCI Program

PL/SQL is Oracle's procedural extension to the SQL language. PL/SQL supports tasks that are more complicated than simple queries and SQL data manipulation language (DML) statements. PL/SQL lets you group a number of constructs into a single block and execute it 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 tables, `CURSOR FOR` loops, and exception handling
- Use cursor variables
- Operate on objects in a 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.
- When binding a PL/SQL VARCHAR2 variable in OCI, the maximum size of the bind variable is 32512 bytes, because of the overhead of control structures.

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

OCI Globalization Support

The following sections introduce OCI functions that can be used for globalization purposes, such as deriving locale information, manipulating strings, character set conversion, and OCI messaging. These functions are also described in detail in other chapters of this guide because they have multiple purposes and functionality.

Client Character Set Control from OCI

The function `OCIEnvNlsCreate()` enables you to set character set information in applications, independently from NLS_LANG and NLS_NCHAR settings. One

application can have several environment handles initialized within the same system environment using different client side character set IDs and national character set IDs.

```
OCIEnvNlsCreate(OCIEnv **envhp, ..., csid, ncsid);
```

where `csid` is the value for character set ID, and `ncsid` is the value for national character set ID. Either can be 0 or `OCI_UTF16ID`. If both are 0, this is equivalent to using `OCIEnvCreate()` instead. The other arguments are the same as for the `OCIEnvCreate()` call.

`OCIEnvNlsCreate()` is an enhancement for programmatic control of character sets, because it validates `OCI_UTF16ID`.

When character set IDs are set through the function `OCIEnvNlsCreate()`, they will replace the settings in NLS_LANG and NLS_NCHAR. In addition to all character sets supported by NLSRTL, `OCI_UTF16ID` is also allowed as a character set ID in the `OCIEnvNlsCreate()` function, although this ID is not valid in NLS_LANG or NLS_NCHAR.

Any Oracle character set ID, except AL16UTF16, can be specified through the `OCIEnvNlsCreate()` function to specify the encoding of metadata, SQL `CHAR` data, and SQL `NCHAR` data.

You can retrieve character sets in NLS_LANG and NLS_NCHAR through another function, `OCINlsEnvironmentVariableGet()`.

See Also: "OCIEnvNlsCreate()" on page 15-18

Code Example for Character Set Control in OCI

For a pseudocode fragment that illustrates a sample usage of these calls:

See Also: "Setting Client Character Sets in OCI" on page 5-23

Character Control and OCI Interfaces

`OCINlsGetInfo()` returns information about `OCI_UTF16ID` if this value has been used in `OCIEnvNlsCreate()`.

`OCIAttrGet()` returns the character set ID and national character set ID that were passed into `OCIEnvNlsCreate()`. This is used to get `OCI_ATTR_ENV_CHARSET_ID` and `OCI_ATTR_ENV_NCHARSET_ID`. This includes the value `OCI_UTF16ID`.

If both `charset` and `ncharset` parameters were set to NULL by `OCIEnvNlsCreate()`, the character set IDs in `NLS_LANG` and `NLS_NCHAR` will be returned.

OCIAttrSet() sets character IDs as the defaults if `OCI_ATTR_CHARSET_FORM` is reset through this function. The eligible character set IDs include `OCI_UTF16ID` if `OCIEnvNlsCreate()` has it passed as `charset` or `ncharset`.

`OCIBindByName()` and `OCIBindByPos()` bind variables with default character set in the `OCIEnvNlsCreate()` call, including `OCI_UTF16ID`. The actual length and the returned length are always in bytes if `OCIEnvNlsCreate()` is used.

`OCIDefineByPos()` defines variables with the value of `charset` in `OCIEnvNlsCreate()`, including `OCI_UTF16ID`, as the default. The actual length and returned length are always in bytes if `OCIEnvNlsCreate()` is used. This behavior for bind and define handles is different from that when `OCIEnvCreate()` is used and `OCI_UTF16ID` is the character set ID for the bind and define handles.

Character Length Semantics in OCI

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

There are two kinds of character sets, variable-width and fixed-width, as a single byte character set is just a special case of a 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 of the entire string is needed to determine the number of characters for fixed-width character sets. However, for variable-width ones, complete scanning is needed to determine the number of characters.

Character Set Support in OCI

See "Character Length Semantics Support in Describing" on page 6-18 and "Character Conversion in OCI Binding and Defining" on page 5-22 for a complete discussion.

Other OCI Globalization Support Functions

Many globalization support functions accept either the environment handle or the user session handle. The OCI environment handle is associated with the client NLS environment variables. This environment does not change when `ALTER SESSION` statements are issued to the server. The character set associated with the environment handle is the client character set. The OCI session handle (returned by `OCISessionBegin()`) is associated with the server session environment. The NLS settings change when the session environment is modified with an `ALTER SESSION` statement. The character set associated with the session handle is the database character set.

Note that the OCI session handle does not have NLS settings associated with it until the first transaction begins in the session. `SELECT` statements do not begin a transaction.

For complete details and discussions of the functions that follow:

See Also:

- Chapter 21, "OCI Globalization Support Functions"
- Oracle Database Globalization Support Guide

Getting Locale Information in OCI

An Oracle locale consists of language, territory, and character set definitions. The locale determines conventions such as day and month names, as well as date, time, number, and currency formats. A globalized application follows a user's locale setting and cultural conventions. For example, when the locale is set to German, users expect to see day and month names in German.

See Also:

- "OCI Locale Functions" on page 21-3
- "OCINlsEnvironmentVariableGet()" on page 21-6

You can retrieve the following information with the `OCINlsGetInfo()` function:

- Days of the week (translated)

- Abbreviated days of the week (translated)
- Month names (translated)
- Abbreviated month names (translated)
- Yes/no (translated)
- AM/PM (translated)
- AD/BC (translated)
- Numeric format
- Debit/credit
- Date format
- Currency formats
- Default language
- Default territory
- Default character set
- Default linguistic sort
- Default calendar

Example of Getting Locale Information in OCI

This example code retrieves information and checks for errors.

```
sword MyPrintLinguisticName(envhp, errhp)
OCIEnv   *envhp;
OCIError *errhp;
{
  OraText  infoBuf[OCI_NLS_MAXBUFSZ];
  sword ret;

  ret = OCINlsGetInfo(envhp,                          /* environment handle */
                      errhp,                                /* error handle */
                      infoBuf,                          /* destination buffer */
                      (size_t) OCI_NLS_MAXBUFSZ,              /* buffer size */
                      (ub2) OCI_NLS_LINGUISTIC_NAME);               /* item */

  if (ret != OCI_SUCCESS)
  {
    checkerr(errhp, ret, OCI_HTYPE_ERROR);
    ret = OCI_ERROR;
  }
  else
  {
    printf("NLS linguistic: %s\n", infoBuf);
   }
  return(ret);
}
```

Manipulating Strings in OCI

Multibyte strings and wide character strings are supported for string manipulation:

Multibyte strings are encoded in native Oracle character sets. Functions that operate on multibyte strings take the string as a whole unit with the length of the string

calculated in bytes. Wide character string (wchar) functions provide more flexibility in string manipulation. They support character-based and string-based operations where the length the string calculated in characters.

The wide character datatype, OCIWchar, is Oracle-specific and should not be confused with the wchar_t datatype defined by the ANSI/ISO C standard. The Oracle wide character datatype is always 4 bytes in all operating systems, while the size of wchar_t depends on the implementation and the operating system. The Oracle wide character datatype normalizes multibyte characters so that they have a uniform fixed width for easy processing. This guarantees no data loss for round trip conversion between the Oracle wide character set and the native character set.

String manipulation can be classified into the following categories:

- Conversion of strings between multibyte and wide character
- Character classifications
- Case conversion
- Calculations of display length
- General string manipulation, such as comparison, concatenation, and searching

See Also: "OCI String Manipulation Functions" on page 21-14

Example of Manipulating Strings in OCI

The following example shows a simple case of manipulating strings.

```
size_t MyConvertMultiByteToWideChar(envhp, dstBuf, dstSize, srcStr)
OCIEnv     *envhp;
OCIWchar   *dstBuf;
size_t      dstSize;
OraText    *srcStr;                          /* null terminated source string */
{
  sword  ret;
  size_t dstLen = 0;
  size_t srcLen;

  /* get length of source string */
  srcLen = OCIMultiByteStrlen(envhp, srcStr);

  ret = OCIMultiByteInSizeToWideChar(envhp,            /* environment handle */
                dstBuf,                                /* destination buffer */
                dstSize,                         /* destination buffer size */
                srcStr,                                    /* source string */
                srcLen,                          /* length of source string */
                &dstLen);                  /* pointer to destination length */

  if (ret != OCI_SUCCESS)
  {
    checkerr(envhp, ret, OCI_HTYPE_ENV);
  }
  return(dstLen);
}
```

Example of Classifying Characters in OCI

The OCI character classification functions are described in detail.

See Also: "OCI Character Classification Functions" on page 21-44

The following example shows how to classify characters in OCI.

```
boolean MyIsNumberWideCharString(envhp, srcStr)
OCIEnv   *envhp;
OCIWchar *srcStr;                                  /* wide char source string */
{
  OCIWchar *pstr = srcStr;                         /* define and init pointer */
  boolean status = TRUE;            /* define and initialize status variable */

  /* Check input */
  if (pstr == (OCIWchar*) NULL)
    return(FALSE);


  if (*pstr == (OCIWchar) NULL)
    return(FALSE);

                                              /* check each character for digit */
  do
  {
    if (OCIWideCharIsDigit(envhp, *pstr) != TRUE)
    {
      status = FALSE;
      break;                                    /* non-decimal digit character */
    }
  } while ( *++pstr != (OCIWchar) NULL);

  return(status);
}
```

Converting Character Sets in OCI

Conversion between Oracle character sets and Unicode (16-bit, fixed-width Unicode encoding) is supported. Replacement characters are used if a character has no mapping from Unicode to the Oracle character set. Therefore, conversion back to the original character set is not always possible without data loss.

Character set conversion functions involving Unicode character sets require data bind and define buffers to be aligned at a `ub2` address, or else an error is raised.

> **See Also:** "OCI Character Set Conversion Functions" on page 21-57

Example of Converting Character Sets in OCI

The following example shows a simple conversion into Unicode.

```
/* Example of Converting Character Sets in OCI
-------------------------------------------*/

size_t MyConvertMultiByteToUnicode(envhp, errhp, dstBuf, dstSize, srcStr)
OCIEnv   *envhp;
OCIError *errhp;
ub2 *dstBuf;
size_t dstSize;
OraText *srcStr;
{
  size_t dstLen = 0;
  size_t srcLen = 0;
  OraText tb[OCI_NLS_MAXBUFSZ];   /* NLS info buffer */
  ub2    cid;                     /* OCIEnv character set id */
```

```
  /* get OCIEnv character set */
  checkerr(errhp, OCINlsGetInfo(envhp, errhp, tb, sizeof(tb),
                                OCI_NLS_CHARACTER_SET));
  cid = OCINlsCharSetNameToId(envhp, tb);

  if (cid == OCI_UTF16ID)
  {
    ub2    *srcStrUb2 = (ub2*)srcStr;
    while (*srcStrUb2++) ++srcLen;
    srcLen *= sizeof(ub2);
  }
  else
    srcLen = OCIMultiByteStrlen(envhp, srcStr);

  checkerr(errhp,
    OCINlsCharSetConvert(
      envhp,       /* environment handle */
      errhp,       /* error handle */
      OCI_UTF16ID, /* Unicode character set id */
      dstBuf,      /* destination buffer */
      dstSize,     /* size of destination buffer */
      cid,         /* OCIEnv character set id */
      srcStr,      /* source string */
      srcLen,      /* length of source string */
      &dstLen));   /* pointer to destination length */

  return dstLen/sizeof(ub2);
}
```

OCI Messaging Functions

The user message API provides a simple interface for cartridge developers to retrieve their own messages and Oracle messages.

See Also:

- *Oracle Database Data Cartridge Developer's Guide*
- "OCI Messaging Functions" on page 21-63

Example of Retrieving a Message from a Text Message File

This example creates a message handle, initializes it to retrieve messages from `impus.msg`, retrieves message number 128, and closes the message handle. It assumes that OCI environment handles, OCI session handles, product, facility, and cache size have been initialized properly.

```
OCIMsg msghnd;                                              /* message handle */
         /* initialize a message handle for retrieving messages from impus.msg*/
err = OCIMessageOpen(hndl,errhp, &msghnd, prod,fac,OCI_DURATION_SESSION);
if (err != OCI_SUCCESS)
                                                            /* error handling */
...
                            /* retrieve the message with message number = 128 */
msgptr = OCIMessageGet(msghnd, 128, msgbuf, sizeof(msgbuf));
                          /* do something with the message, such as display it */
...
      /* close the message handle when there are no more messages to retrieve */
OCIMessageClose(hndl, errhp, msghnd);
```

lmsgen Utility

The `lmsgen` utility converts text-based message files (`.msg`) into binary format (`.msb`) so that Oracle messages and OCI messages provided by the user can be returned to OCI functions in the desired language.

BNF Syntax of lmsgen

```
lmsgen text_file product facility [language]
```

where:

- *`text_file`* is a message text file.
- *`product`* is the name of the product.
- *`facility`* is the name of the facility.
- *`language`* is the optional message language corresponding to the language specified in the `NLS_LANG` parameter. The language parameter is required if the message file is not tagged properly with language.

Guidelines for Text Message Files

- Lines that start with "`/`" and "`//`" are treated as internal comments and are ignored.
- To tag the message file with a specific language, include a line similar to the following:

  ```
  #   CHARACTER_SET_NAME= Japanese_Japan.JA16EUC
  ```

- Each message contains 3 fields:

  `message_number`, *`warning_level`*, *`message_text`*

- The message number must be unique within a message file.
- The warning level is not currently used. Set to 0.
- The message text cannot be longer than 76 bytes.

The following is an example of an Oracle message text file:

```
/ Copyright (c) 2001 by the Oracle Corporation.  All rights reserved.
/ This is a test us7ascii message file
# CHARACTER_SET_NAME= american_america.us7ascii
/
00000, 00000, "Export terminated unsuccessfully\n"
00003, 00000, "no storage definition found for segment(%lu, %lu)"
```

Example: Creating a Binary Message File from a Text Message File

The following table contains sample values for the `lmsgen` parameters:

lmsgen Parameter	Value
`product`	`$HOME/myApplication`
`facility`	`imp`
`language`	`AMERICAN`
`text_file`	`impus.msg`

The text message file is found in the following location:

```
$HOME/myApp/mesg/impus.msg
```

One of the lines in the text message file is:

```
00128,2, "Duplicate entry %s found in %s"
```

The `lmsgen` utility converts the text message file (`impus.msg`) into binary format, resulting in a file called `impus.msb`:

```
% lmsgen impus.msg $HOME/myApplication imp AMERICAN
```

The following output results:

```
Generating message file impus.msg -->
/home/scott/myApplication/mesg/impus.msb

NLS Binary Message File Generation Utility: Version 9.2.0.0.0 -Production

Copyright (c) Oracle Corporation 1979, 2001.  All rights reserved.

CORE    9.2.0.0.0       Production
```

3

Datatypes

This chapter provides a reference to Oracle external datatypes used by OCI applications. It also discusses Oracle datatypes and the conversions between internal and external representations that occur when you transfer data between your program and Oracle.

This chapter contains these topics:

> **See Also:** For detailed information about Oracle internal datatypes, see the *Oracle Database 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, but 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 be returned as character data, instead of 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. See the description of the DATE external datatype on page 3-11 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 (for example, indexed table, boolean, record).

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

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

> **See Also:** "External Datatypes" on page 3-6, and "Using External Datatype Codes" on page 3-3

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

  > **See Also:** "Typecodes" on page 3-24, and "Relationship Between SQLT and OCI_TYPECODE Values" on page 3-26

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 3-3.

> **See Also:**
>
> - For detailed information about Oracle internal datatypes, see the *Oracle Database SQL Reference.*
> - For information about describing select-list items in a query, see the section "Describing Select-list Items" on page 4-9.

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

Table 3–1 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

Table 3–1 (Cont.) Internal Oracle Datatypes

Internal Oracle Datatype	Maximum Internal Length	Datatype Code
DATE	7 bytes	12
RAW	2000 bytes	23
LONG RAW	2^31-1 bytes	24
ROWID	10 bytes	69
CHAR, NCHAR	2000 bytes	96
BINARY_FLOAT	4 bytes	100
BINARY_DOUBLE	8 bytes	101
User-defined type (object type, VARRAY, Nested Table)	N/A	108
REF	N/A	111
CLOB, NCLOB	128 terabytes	112
BLOB	128 terabytes	113
BFILE	maximum operating system file size or UB8MAXVAL	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 *Oracle Database SQL Reference.*

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 BFILEs can contain binary data. They are handled differently than other types, so they are not included in this discussion. See Chapter 7, "LOB and BFILE Operations", for more information about these datatypes.

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 graphic image. Character data can be transformed when passed through a gateway between networks. 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. 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)

BINARY_FLOAT and BINARY_DOUBLE

The BINARY_FLOAT and BINARY_DOUBLE datatypes represent single-precision and double-precision floating point values that mostly conform to the IEEE754 standard for Floating Point Arithmetic.

Prior to the addition of these datatypes, all numeric values in an Oracle database were stored in the Oracle NUMBER format. These new binary floating point types will not replace Oracle NUMBER. Rather, they are alternatives to Oracle NUMBER that provide the advantage of using less disk storage.

These internal types are represented by the following codes:

- SQLT_IBFLOAT for BINARY_FLOAT.
- SQLT_IBDOUBLE for BINARY_DOUBLE.

All the following host variables can be bound to BINARY_FLOAT and BINARY_DOUBLE datatypes:

- SQLT_BFLOAT (native float)
- SQLT_BDOUBLE (native double)
- SQLT_INT (integer)

- SQLT_FLT (float)
- SQLT_NUM (Oracle NUMBER)
- SQLT_UIN (unsigned)
- SQLT_VNU (VARNUM)
- SQLT_CHR (VARCHAR2)
- SQLT_VCS (VARCHAR)
- SQLT_STR (NULL-terminated String)
- SQLT_LVC (LONG VARCHAR)
- SQLT_AFC (CHAR)
- SQLT_AVC (CHARZ)
- SQLT_VST (OCIString)

For best performance, you are advised to use external types SQLT_BFLOAT and SQLT_BDOUBLE in conjunction with the BINARY_FLOAT and BINARY_DOUBLE datatypes.

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	CODE	PROGRAM 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
native float	21	float	SQLT_BFLOAT
native double	22	double	SQLT_BDOUBLE
RAW	23	unsigned char[n]	SQLT_BIN
LONG RAW	24	unsigned char[n]	SQLT_LBI
UNSIGNED INT	68	unsigned	SQLT_UIN

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

EXTERNAL DATATYPE	CODE	PROGRAM VARIABLE	OCI DEFINED CONSTANT
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 DATATYPE	108	struct	SQLT_NTY
REF	110	OCIRef	SQLT_REF
Character LOB descriptor	112	OCILobLocator (see note 2)	SQLT_CLOB
Binary LOB descriptor	113	OCILobLocator (see note 2)	SQLT_BLOB
Binary FILE descriptor	114	OCILobLocator	SQLT_FILE
OCI STRING type	155	OCIString	SQLT_VST (see note 1)
OCI DATE type	156	OCIDate *	SQLT_ODT (see note 1)
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
INTERVAL DAY TO SECOND descriptor	190	OCIInterval *	SQLT_INTERVAL_DS
TIMESTAMP WITH LOCAL TIME ZONE descriptor	232	OCIDateTime *	SQLT_TIMESTAMP_LTZ

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).

- For more information on the use of these datatypes, refer to Chapter 11, "Object-Relational Datatypes in OCI".
- 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, "Using the Object Type Translator with OCI".

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 in OCI" 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 zero. 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 2-23

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 in OCI" 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 Also:

- "OCINumber Examples" on page 11-10
- "VARNUM" on page 3-11 for a description of the 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. 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 VARCHAR2 or a NULL-terminated string for the operation.

STRING

The NULL-terminated STRING format behaves like the VARCHAR2 format, 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 VARCHAR2 and CHAR, 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 Database.

Table 3–3 shows several examples of the VARNUM values returned for numbers in a 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.

Do not create tables with LONG columns. Use LOB columns (CLOB, NCLOB, or BLOB) instead. LONG columns are supported only for backward compatibility.

Oracle also recommends that you convert existing LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. Furthermore, LOB functionality is enhanced in every release, but LONG functionality has been static for several releases.

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.

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.

See Also: 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 in OCI" for more information about this datatype.
- For information about `DATETIME` and `INTERVAL` datatypes, refer to "Datetime and Interval Datatype Descriptors" on page 3-18.

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: *Oracle Database 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.

> **See Also:** 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 in OCI" 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, 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:** *Oracle Database 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 Globalization 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 2-23

You can also request output to a character string from an internal NUMBER datatype. Number conversion follows the conventions established by the Globalization 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.

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 Datatypes: Object, VARRAY, Nested Table

Named datatypes 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 datatype* refers to a host language representation of the type. The SQLT_NTY datatype code is used when binding or defining named datatypes.

In a C application, named datatypes 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 datatypes in the OCI, see Chapter 11, "Object-Relational Datatypes in OCI".
- For information about how named datatypes are represented as C structs, refer to Chapter 14, "Using the Object Type Translator with OCI".

REF

This is a reference to a named datatype. 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, see Chapter 13, "Object Advanced Topics in OCI"

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:** "OCI Descriptors" on page 2-9 and "Positioned Updates and Deletes" on page 2-25 for more information about the use of the ROWID descriptor

LOB Descriptor

A LOB (Large Object) stores binary or character data up to 128 terabytes 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* that 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 OCI, the LOB locator maps to a variable of type OCILobLocator.

> **Note:** Depending on your application, you may or may not want to use LOB locators. You can use the data interface for LOBs, which does not require LOB locators. In this interface, you can bind or define character data for CLOB columns or RAW data for BLOB columns.

> **See Also:**
>
> - For more information about descriptors, including the LOB locator, see the section "OCI Descriptors" on page 2-9
> - For more information about LOBs refer to the *Oracle Database SQL Reference* and the *Oracle Database Application Developer's Guide - Large Objects*.
> - "Binding LOB Data" on page 5-8
> - "Defining LOB Data" on page 5-15

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 BFILE Operations"

The datatype codes available for binding or defining LOBs are:

- SQLT_BLOB - a binary LOB datatype.
- SQLT_CLOB - a character LOB datatype.

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 BFILE Functions in OCI" on page 7-8

BFILE

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

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 object and the filename. The maximum size of a BFILE is the smaller of the operating system maximum file size or UB8MAXVAL.

Binary file LOBs do not participate in transactions. Rather, the underlying operating system provides file integrity and durability.

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 BFILEs is:

- SQLT_BFILE - a binary FILE LOB datatype

See Also: For more information about directory aliases, refer to the *Oracle Database Application Developer's Guide - Large Objects*

BLOB

The BLOB datatype stores unstructured binary large objects. BLOBs can be thought of as bit streams with no character set semantics. BLOBs can store up to 128 terabytes 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 variable-width character data. CLOBs can store up to 128 terabytes 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 variable-width character set data. NCLOBs can store up to 128 terabytes 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.

Datetime and Interval Datatype Descriptors

The datetime and interval datatype descriptors are briefly summarized here.

See Also: *Oracle Database SQL Reference*

ANSI DATE

The ANSI DATE is based on the DATE, but contains no time portion. 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 the optional fractional_seconds_precision 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 retrieving the data, Oracle returns it in your 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 optional 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. 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.

Native Float and Native Double

The native float (SQLT_BFLOAT) and native double (SQLT_BDOUBLE) datatypes represent the single-precision and double-precision floating point values. They are represented natively, that is, in the host system's floating point format.

Note that these new external types were added to externally represent the BINARY_FLOAT and BINARY_DOUBLE internal datatypes. Thus, performance for the new internal types will be best when used in conjunction with external types native

float and native double respectively. This draws a clear distinction between the existing representation of floating point values (SQLT_FLT) and these new types.

C Object-Relational Datatype Mappings

OCI supports Oracle-defined C datatypes for mapping user-defined datatypes to C representations (for example, OCINumber, OCIArray). 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 datatypes.

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

Data Conversions

Table 3–5 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 datatypes 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 datatypes 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 Table 3–6, because of the width limitation.

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

Table 3–5 Data Conversions

-	INTERNAL DATATYPES->	-	-	-	-	-	-	-	-
EXTERNAL DATATYPES	**VARCHAR2**	**NUMBER**	**LONG**	**ROWID**	**UROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**
VARCHAR2	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)
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)

Table 3–5 (Cont.) Data Conversions

-	INTERNAL DATATYPES->	-	-	-	-	-	-	-	-
EXTERNAL DATATYPES	VARCHAR2	NUMBER	LONG	ROWID	UROWID	DATE	RAW	LONG RAW	CHAR
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
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)

Legend:

I = Conversion valid for input only

O = Conversion valid for output only

I/O = Conversion valid for input or output

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.

Data Conversions for LOB Datatype Descriptors

These are the data conversions for LOBs:

Table 3–6 Data Conversions for LOBs

EXTERNAL DATATYPES	INTERNAL CLOB	INTERNAL BLOB
VARCHAR	I/O	
CHAR	I/O	
LONG	I/O	

Table 3–6 (Cont.) Data Conversions for LOBs

EXTERNAL DATATYPES	INTERNAL CLOB	INTERNAL BLOB
LONG VARCHAR	I/O	
RAW		I/O
VARRAW		I/O
LONG RAW		I/O
LONG VARRAW		I/O

Data Conversions for Datetime and Interval Datatypes

You can also use one of the character datatypes 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 datatype and datetime/interval datatype 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	INTERVAL 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

Assignment Notes

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

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 upgrading of Oracle DATE to ANSI compliant DATETIME datatypes

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

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

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 Upgrading 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 can 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.

Data Conversion for BINARY_FLOAT and BINARY_DOUBLE in OCI

Table 3–8 shows the supported conversions between internal numerical datatypes and all relevant external types. An (I) implies that the conversion is valid for input only (binds), and (O) implies that the conversion is valid for output only (defines), while an (I/O) implies that the conversion is valid for input as well as output (binds and defines).

Table 3–8 Data Conversion for External Datatypes to Internal Numerical Datatypes

External Types/Internal Types	BINARY_FLOAT	BINARY_DOUBLE
VARCHAR	I/O	I/O
VARCHAR2	I/O	I/O
NUMBER	I/O	I/O

Table 3–8 (Cont.) Data Conversion for External Datatypes to Internal Numerical

External Types/Internal Types	BINARY_FLOAT	BINARY_DOUBLE
INTEGER	I/O	I/O
FLOAT	I/O	I/O
STRING	I/O	I/O
VARNUM	I/O	I/O
LONG	I/O	I/O
UNSIGNED INT	I/O	I/O
LONG VARCHAR	I/O	I/O
CHAR	I/O	I/O
BINARY_FLOAT	I/O	I/O
BINARY_DOUBLE	I/O	I/O

Table 3–9 shows the supported conversions between all relevant internal types and numerical external types. An (I) implies that the conversion is valid for input only (only for binds), and (O) implies that the conversion is valid for output only (only for defines), while an (I/O) implies that the conversion is valid for input as well as output (binds and defines).

Table 3–9 Data Conversions for Internal to External Numerical Datatypes

Internal Types/External Types	Native Float	Native Double
VARCHAR2	I/O	I/O
NUMBER	I/O	I/O
LONG	I	I
CHAR	I/O	I/O
BINARY_FLOAT	I/O	I/O
BINARY_DOUBLE	I/O	I/O

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; for example, querying the OCI_ATTR_TYPECODE attribute of a type returns an OCITypeCode value.

Table 3–10 lists the possible values for an OCITypeCode. There is a value corresponding to each Oracle datatype.

Table 3–10 OCITypeCode Values and Datatypes

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_BFLOAT	BINARY_FLOAT
OCI_TYPECODE_BDOUBLE	BINARY_DOUBLE
OCI_TYPECODE_DECIMAL	decimal
OCI_TYPECODE_OCTET	octet
OCI_TYPECODE_INTEGER	integer
OCI_TYPECODE_SMALLINT	small int
OCI_TYPECODE_RAW	RAW
OCI_TYPECODE_VARCHAR2	variable string ANSI SQL, that is, VARCHAR2
OCI_TYPECODE_VARCHAR	variable string Oracle SQL, that is, VARCHAR
OCI_TYPECODE_CHAR	fixed-length string inside SQL, that is 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)
OCI_TYPECODE_BFILE	binary large object file (BFILE)
OCI_TYPECODE_OBJECT	named object type, or SYS.XMLType
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, enabling you 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.

Table 3–11 illustrates the mappings between SQLT and OCI_TYPECODE types.

Table 3–11 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
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)
BINARY_FLOAT	OCI_TYPECODE_BFLOAT	SQLT_BFLOAT
BINARY_DOUBLE	OCI_TYPECODE_BDOUBLE	SQLT_BDOUBLE
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 or SYS.XMLType	OCI_TYPECODE_OBJECT	SQLT_NTY
SIGNED(8)	OCI_TYPECODE_SIGNED8	SQLT_INT (1)
SIGNED(16)	OCI_TYPECODE_SIGNED16	SQLT_INT (2)

Table 3–11 (Cont.) OCI_TYPECODE to SQLT Mappings

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
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>
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, which is found in the public directory. The exact contents may vary according to the operating system you are using.

Note: The use of the datatypes in oratypes.h is the only supported means of supplying parameters to the OCI.

4

Using SQL Statements in OCI

This chapter discusses the concepts and steps involved in processing SQL statements with the Oracle Call Interface.

This chapter contains these topics:

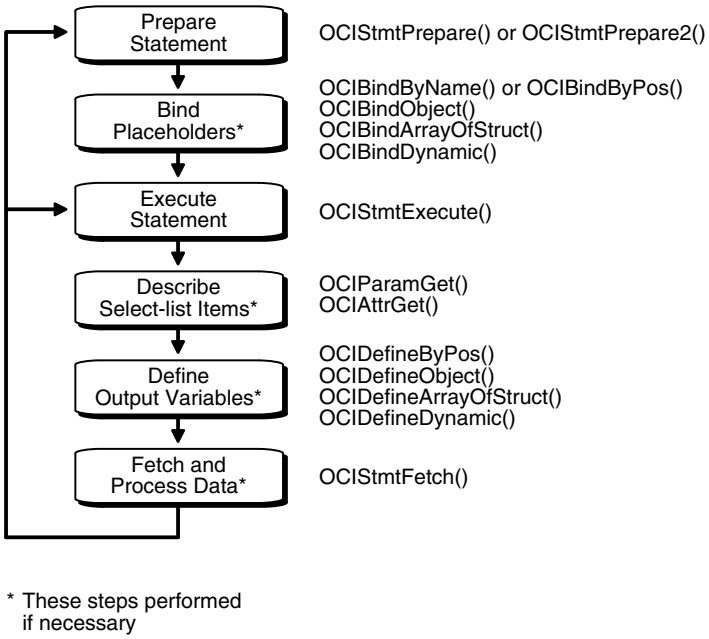
Overview of SQL Statement Processing

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.

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 this processing.

Once you have allocated the necessary handles and connected to a server, follow the steps illustrated in Figure 4–1, "Steps In Processing SQL Statements":

Figure 4–1 Steps In Processing SQL Statements



1. Prepare statement. Define an application request using `OCIStmtPrepare()` or `OCIStmtPrepare2()`.
2. Bind placeholders, if necessary.. For DML statements and queries with input variables, perform one or more bind calls using
   - `OCIBindByPos()`
   - `OCIBindByName()`
   - `OCIBindObject()`
   - `OCIBindDynamic()`
   - `OCIBindArrayOfStruct()`

   to bind the address of each input variable (or PL/SQL output variable) or array to each placeholder in the statement.
3. A statement can also be prepared for execution with `OCIStmtPrepare2()`, an enhanced version of `OCIStmtPrepare()` introduced to support statement caching.
4. Execute. Call `OCIStmtExecute()` to execute the statement. For DDL statements, no further steps are necessary.
5. Describe, if necessary. 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.
6. Define, if necessary. 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.

7. Fetch, if necessary. 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.

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 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.

Additional steps beyond those listed earlier may be required if your application needs to do any of the following:

- initiate and manage multiple transactions
- manage multiple threads of execution
- perform piecewise inserts, updates, or fetches

> **See Also:** "Statement Caching in OCI" on page 9-20

Preparing Statements

SQL and PL/SQL statements are prepared for execution by using the statement prepare call and any necessary bind calls. 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 passes to it a previously allocated statement handle. This is a completely local call, requiring no round trip to the server. No association is made between the statement and a particular server at this point.

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, " OCI_ATTR_STMT_TYPE Values and Statement Types".

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

Table 4–1 (Cont.) OCI_ATTR_STMT_TYPE Values and Statement Types

Attribute Value	Statement Type
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:

- "Using PL/SQL in an OCI Program" on page 2-27
- "OCIStmtPrepare()" on page 16-12

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 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 re-executed frequently on the same server, it is more efficient to prepare a new statement for another service context.

Binding Placeholders in OCI

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. This data can be constant or literal, 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 a 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 a SQL statement or PL/SQL block, you must call an OCI routine that binds the address of a variable in your program to that 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.

Binding is used for both input and output variables in non-query operations. In the following example,

```
empno_out, ename_out, job_out, sal_out, and deptno_out
```

should be bound. These are outbinds (as opposed to regular inbinds).

```
INSERT INTO emp VALUES
         (:empno, :ename, :job, :sal, :deptno)
      RETURNING
         (empno, ename, job, sal, deptno)
      INTO
         (:empno_out, :ename_out, :job_out, :sal_out, :deptno_out)
```

> **See Also:** For detailed information about implementing bind operations, refer to Chapter 5, "Binding and Defining in OCI"

Executing Statements

An OCI application executes prepared statements individually using `OCIStmtExecute()`.

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, the 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: "Defining Output Variables in OCI" on page 4-12 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 as 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 that is allocated with the OCIDescAlloc() function.

See Also: "OCI Descriptors" on page 2-9

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, which is user-developed, evaluates the return code from an OCI application.

Execution Modes of OCIStmtExecute()

You can specify several modes for the OCIStmtExecute() call.

See Also: "OCIStmtExecute()" on page 16-4 for the values of the parameter mode

Batch Error Mode

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 OCI returns an error. Any rows remaining in the array are ignored. The application must then re-execute the remainder of the array, and go through the whole process again if it encounters more errors, which makes additional round trips.

To facilitate processing of array DML operations, 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, OCI attempts to INSERT, UPDATE, or DELETE all rows, and collects information about any errors that occurred. The application can then retrieve error information and re-execute any DML operations which failed during the first call.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.

> **Note:** This feature 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 re-coded to account for the new program logic described in this section.

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_errs, 0, OCI_ATTR_NUM_DML_ERRORS,
              errhp);
   ```

3. The application extracts each error using OCIParamGet(), along with its row information, from the error handle that was passed to the OCIStmtExecute() call. In order to retrieve the information, the application must allocate an additional new error handle for the OCIParamGet() call, populating the new error handle with 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, errhp2;
   for (i=0; i<num_errs; i++)
   {
      OCIParamGet(errhp, OCI_HTYPE_ERROR, errhp2, (dvoid **)&errhndl, i);
      OCIAttrGet(errhndl, OCI_HTYPE_ERROR, &row_offset, 0,
          OCI_ATTR_DML_ROW_OFFSET, errhp2);
      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 re-execute the associated DML statements.

Since it cannot be determined 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 bind buffers used in the array binds done on the first DML operation can be reused.

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'};
OCIError *errhp2;
OCIError *errhndl[MAXROWS];
...
/* Array bind all the positions */
OCIBindByPos (stmtp1,&bindp1[0],errhp,1,(dvoid *)&number[0],
     sizeof(number[0]),SQLT_INT,(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, a different error handler errhp2 is used so that
 * the state of errhp is not changed */
OCIAttrGet (stmtp1, OCI_HTYPE_STMT, &num_errs, 0,
            OCI_ATTR_NUM_DML_ERRORS, errhp2);
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 */
   for (i = 0; i < num_errs; i++) {
      OCIHandleAlloc( (dvoid *)envhp, (dvoid **)&errhndl[i],
      (ub4) OCI_HTYPE_ERROR, 0, (dvoid *) 0);
      OCIParamGet(errhp, OCI_HTYPE_ERROR, errhp2, &errhndl[i], i);
      OCIAttrGet (errhndl[i], OCI_HTYPE_ERROR, &row_off[i], 0,
                  OCI_ATTR_DML_ROW_OFFSET, errhp2);
      /* get server diagnostics */
      OCIErrorGet (..., errhndl[i], ...);
      }
   }
/* make corrections to bind data */
OCIBindByPos (stmtp2,&bindp2[0],errhp,1,(dvoid *)0,sizeof(grade[0]),SQLT_INT,
     (dvoid *)0, (ub2 *)0,(ub2 *)0,0,(ub4 *)0,OCI_DATA_AT_EXEC);
OCIBindByPos (stmtp2,&bindp2[1],errhp,2,(dvoid *)0,sizeof(number[0]),SQLT_DAT,
     (dvoid *)0, (ub2 *)0,(ub2 *)0,0,(ub4 *)0,OCI_DATA_AT_EXEC);
/* register the callback for each bind handle, row_off and position
 * information can be passed to call back function by means of context
 * pointers.
```

```
 */
OCIBindDynamic (bindp2[0],errhp,ctxp1,my_callback,0,0);
OCIBindDynamic (bindp2[1],errhp,ctxp2,my_callback,0,0);
/* execute the UPDATE statement */
OCIStmtExecute (svchp,stmtp2,errhp,num_errs,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, consider a query where the program has no prior information about the columns in the `employees` table:

```
SELECT * FROM employees
```

There are two types of describes available: implicit and explicit.

An *implicit describe* is one that does not require any special calls to retrieve describe information from the server, although special calls *are* necessary to access the information. 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.

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.

You can describe a query explicitly prior to execution. To do this, specify `OCI_DESCRIBE_ONLY` as the mode of `OCIStmtExecute()`, which does not execute the statement, but returns the select-list description. For performance reasons 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 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 datatypes corresponding to a query following query execution. The query was associated with the statement handle by a prior call to `OCIStmtPrepare()`.

```
...
OCIParam     *mypard = (OCIParam *) 0;
ub2          dtype;
text         *col_name;
ub4          counter, col_name_len, char_semantics;
ub2          col_width;
sb4          parm_status;

text *sqlstmt = (text *)"SELECT * FROM employees WHERE employee_id = 100";

checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                    (ub4)strlen((char *)sqlstmt),
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, 0, 0, (OCISnapshot *)0,
(OCISnapshot *)0, OCI_DEFAULT));

/* Request a parameter descriptor for position 1 in the select-list */
counter = 1;
parm_status = OCIParamGet((dvoid *)stmthp, OCI_HTYPE_STMT, errhp,
               (dvoid **)&mypard, (ub4) counter);

/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */

while (parm_status == OCI_SUCCESS) {
   /* Retrieve the datatype 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 */
   col_name_len = 0;
   checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
            (dvoid**) &col_name, (ub4 *) &col_name_len, (ub4) OCI_ATTR_NAME,
            (OCIError *) errhp ));

   /* Retrieve the length semantics for the column */
   char_semantics = 0;
   checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
            (dvoid*) &char_semantics,(ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
            (OCIError *) errhp  ));
```

```
   col_width = 0;
   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  ));

   /* increment counter and get next descriptor, if there is one */
   counter++;
   parm_status = OCIParamGet((dvoid *)stmthp, OCI_HTYPE_STMT, errhp,
          (dvoid **)&mypard, (ub4) counter);
} /* while */
...
```

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:

- "OCIParamGet()" on page 15-56 and.
- "OCIAttrGet()" on page 15-48
- "Parameter Attributes" on page 6-4 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()`; this does not execute the statement, but returns the select-list description.

Note: To maximize performance, it is recommended that applications execute the statement in default mode and use the implicit describe that accompanies the execution.

The following code demonstrates the use of explicit describe in a select-list to return information about columns.

```
...
int i = 0;
ub4 numcols = 0;
ub2 type = 0;
OCIParam *colhd = (OCIParam *) 0;   /* column handle */

text *sqlstmt = (text *)"SELECT * FROM employees WHERE employee_id = 100";

checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                   (ub4)strlen((char *)sqlstmt),
```

```
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

/* 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 */

checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, 0, 0,
        (OCISnapshot *) 0, (OCISnapshot *) 0, OCI_DESCRIBE_ONLY));

/* Get the number of columns in the query */
checkerr(errhp, OCIAttrGet((dvoid *)stmthp, OCI_HTYPE_STMT, (dvoid *)&numcols,
                     (ub4 *)0, OCI_ATTR_PARAM_COUNT, errhp));

/* go through the column list and retrieve the datatype of each column. We
start from pos = 1 */
for (i = 1; i <= numcols; i++)
{
  /* get parameter for column i */
  checkerr(errhp, OCIParamGet((dvoid *)stmthp, OCI_HTYPE_STMT, errhp, (dvoid
**)&colhd, i));

  /* get data-type of column i */
  type = 0;
  checkerr(errhp, OCIAttrGet((dvoid *)colhd, OCI_DTYPE_PARAM,
          (dvoid *)&type, (ub4 *)0, OCI_ATTR_DATA_TYPE, errhp));
}
...
```

Defining Output Variables in OCI

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 that you want to retrieve data from. The define step creates an association that determines where returned results are stored, and in what format.

For example, to process the following statement 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:

```
SELECT name, ssn FROM employees
    WHERE empno = :empnum
```

See Also: Chapter 5, "Binding and Defining in OCI"

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 completed execution. Oracle encourages the use of `OCIStmtFetch2()`, which supports *scrollable cursors*.

See Also: "Scrollable Cursors in OCI" on page 4-14

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 OCI Defining" on page 5-13. Refer to that example for more information.
- For information about defining output variables, see the section "Overview of Defining in OCI" on page 5-12.

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 you define them. If LOB locators are fetched, then the application can perform further operations on these locators through the `OCILobXXX` interfaces.

See Also:

- Chapter 7, "LOB and BFILE Operations", for more information about working with LOB locators in the OCI.
- "Defining LOB Output Variables" on page 5-14 for usage and examples of selecting LOB data without the use of locators.

Setting Prefetch Count

In order to minimize server round trips and optimize the performance, the OCI can prefetch result set rows when executing a query. You can customize this prefetching by setting either the `OCI_ATTR_PREFETCH_ROWS` or `OCI_ATTR_PREFETCH_MEMORY` attribute of the statement handle using the `OCIAttrSet()` function. These attributes are used as follows:

- `OCI_ATTR_PREFETCH_ROWS` sets the number of rows to be prefetched. If it is not set, then the default value is 1. If the `iters` parameter of `OCIStmtExecute()` is 0 and prefetching is enabled, the rows are buffered during calls to `OCIStmtFetch2()`. The prefetch value can be altered after execution and between fetches.
- `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, not the data, is returned by the query.

See Also: "Statement Handle Attributes" on page A-22.

Scrollable Cursors in OCI

A cursor is a current position in a *result set*. Execution of a cursor puts the results of the query into a set of rows called the result set that can be fetched either sequentially or non-sequentially. In the latter case the cursor 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 improves performance by limiting calls to the server.

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. OCI client prefetching works with OCI scrollable cursors. The size of the client prefetch cache can be controlled by the existing OCI attributes OCI_ATTR_PREFETCH_ROWS and OCI_ATTR_PREFETCH_MEMORY.

Note: Do not use scrollable cursors unless you require the functionality, because they use more server resources and can have greater response times than non-scrollable cursors.

The OCIStmtExecute() call has an execution mode for scrollable cursors, OCI_STMT_SCROLLABLE_READONLY. The default for statement handles is non-scrollable, forward sequential access only, where the mode is OCI_FETCH_NEXT. You must set this execution mode each time the statement handle is executed.

The statement handle attribute OCI_ATTR_CURRENT_POSITION can be retrieved using OCIAttrGet() only. This attribute cannot be set by the application; it 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 since this statement handle was executed. Since non-scrollable cursors are forward sequential only, OCI_ATTR_ROW_COUNT 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 does not have to be the total number of rows fetched into the your 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 OCIStmtFetch2() call, instead of the OCIStmtFetch() call, which is retained for backward compatibility. You are encouraged to use OCIStmtFetch2(), for all new applications, even those not using scrollable cursors. This call also works for non-scrollable cursors, but can raise an error if any other orientation besides OCI_DEFAULT or OCI_FETCH_NEXT is passed.

Scrollable cursors are supported for remote mapped queries. Transparent application Failover (TAF) is supported for scrollable cursors.

Note: If you call OCIStmtFetch2() with the nrows parameter set to 0, the cursor is cancelled.

See Also:

- "OCIStmtFetch2()" on page 16-8
- "Setting Prefetch Count" on page 4-13

Increasing Scrollable Cursor Performance

Response time is improved if you use OCI client-side prefetch buffers. After calling OCIStmtExecute() for a scrollable cursor, call OCIStmtFetch2() using OCI_FETCH_LAST to obtain the size of the result set. Then set OCI_ATTR_PREFETCH_ROWS to about 20% of that size, and set OCI_PREFETCH_MEMORY if the result set uses a large amount of memory.

Example of Access on a Scrollable Cursor

Assume that a result set is returned by the SQL query:

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

and that the table EMP has 14 rows. One use of scrollable cursors is:

```
...
/* 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;
/* This checks for any OCI errors before printing the results of the fetch call
   in the define buffers */
 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 in OCI

This chapter contains these topics:

Overview of Binding in OCI

This chapter expands on the basic concepts of binding and defining, 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.

For example, given the `INSERT` statement

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

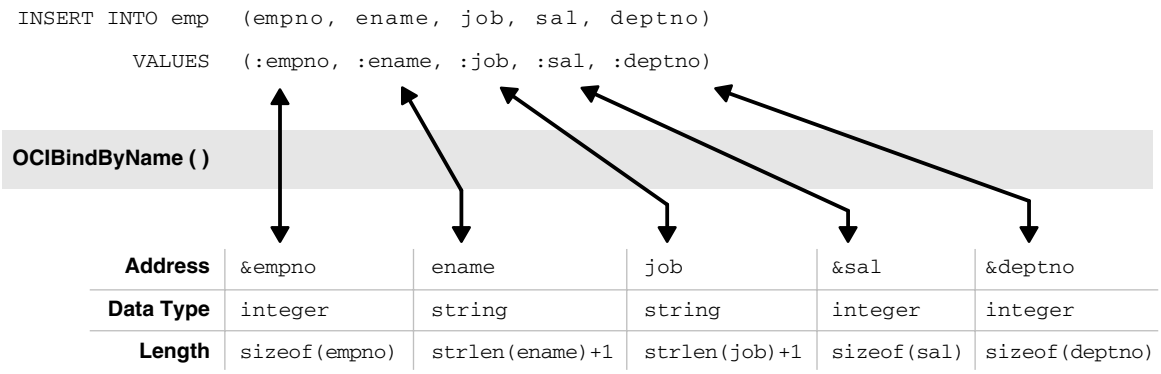
and the following variable declarations

```
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 OCI Binding" on page 5-4.

Figure 5–1 Using OCIBindByName() to Associate Placeholders with Program Variables



If you change only the value of a bind variable, it is not necessary to rebind it in order to execute the statement again. Because the bind is by reference, as long as the address of the variable and handle remain valid, you can re-execute 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 0. 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 datatypes (descriptors or locators) whose sizes are not known, 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.

Using the previous example for a positional bind:

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

The five placeholders are then each 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 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.

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 the database when you are updating or inserting 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 is inserted.

The maximum number of rows allowed in an array DML statement is 4 gigabytes -1.

Binding Placeholders in PL/SQL

You process a PL/SQL block by placing the block in a string variable, binding any variables, and then 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 binds for 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 empno = :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`.

Note: 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: "Information for Named Datatype and REF Binds" on page 11-26 for more information about binding PL/SQL placeholders

Steps Used in OCI Binding

Placeholders are bound in several steps. For a simple scalar or array bind, it is only necessary to specify an association between the placeholder and the data, by using `OCIBindByName()` or `OCIBindByPos()`.

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. 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 placeholder in a SQL statement.

```
...
/* 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, SQLT_STR, (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, SQLT_STR, (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), SQLT_INT,
    (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), SQLT_INT,
    (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), SQLT_INT,
    (dvoid *) 0, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0,OCI_DEFAULT));
```

Note: The `checkerr()` function evaluates the return code from an OCI application. The code for the function is listed in the section "OCI Programming Steps" on page 2-20.

PL/SQL Block in an OCI Program

Perhaps the most common use for PL/SQL blocks in OCI is to call stored procedures or stored functions. Assume that there is a procedure named RAISE_SALARY stored in the database, and you embed a call to that procedure in an anonymous PL/SQL block, and then process the PL/SQL block.

The following program fragment shows how to embed a stored procedure call in an OCI application. The program passes an employee number and a salary increase as inputs to a stored procedure called `raise_salary`:

```
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 variable, `new_salary`, and the program displays this value.

Note that the PL/SQL procedure argument, `new_salary`, although a PL/SQL OUT variable, must be bound, not defined. This is further explained in the section on OCI defines.

```
/* 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( errhp, 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);
}
```

This example demonstrates how to perform a simple scalar bind where only a single bind call is necessary. In some cases, additional bind calls are needed to define attributes for specific bind datatypes or execution modes.

Advanced Bind Operations in OCI

The section "Binding Placeholders in OCI" on page 4-4 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 datatypes and `REF`s.

In some cases, additional bind calls are necessary to define specific attributes for certain bind datatypes or certain execution modes.

The following sections describe these special cases, and the information about binding is summarized in Table 5–1.

Table 5–1 Information Summary for 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 Datatype	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:

- "Named Datatype Binds" on page 11-25 For information on binding named datatypes (objects)
- "Binding REFs" on page 11-25 for information on binding REFs

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 OCI LOB functions.
- Bind the LOB value directly, without using the LOB locator.

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:

```
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 calls 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 */
```

You can also insert an array 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 */
```

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 `BLOB`s, `CLOB`s, and `NCLOB`s. Use `OCI_DTYPE_FILE` when allocating `BFILE`s.

Restrictions on Binding LOB Locators

- Piecewise and callback `INSERT` or `UPDATE` operations are not supported.
- When using a FILE locator as a bind variable for an `INSERT` or `UPDATE` statement, you must first initialize the locator with a directory object and filename, using `OCILobFileSetName()` before issuing the `INSERT` or `UPDATE` statement.

> **See Also:** Chapter 7, "LOB and BFILE Operations" for more information about the OCI LOB functions

Binding LOB Data

Oracle allows nonzero binds for `INSERT`s and `UPDATE`s of any size LOB. So you can bind data into a LOB column using `OCIBindByPos()`, `OCIBindByName()`, and PL/SQL binds.

The bind of more than 4 kilobytes of data to a LOB column uses space from the temporary tablespace. Make sure that your 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 following command to create an extendable temporary tablespace:

```
CREATE TABLESPACE ... AUTOEXTENT ON ... TEMPORARY ...;
```

Restrictions on Binding LOB Data

- 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.
- In an INSERT AS SELECT operation, Oracle does not allow binding of any length data to LOB columns.
- Oracle does not do implicit conversions, 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 LOB Data

Consider the following SQL statements which will be used in the examples that follow:

```
CREATE TABLE foo (a INTEGER );
CREATE TYPE lob_typ AS OBJECT (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 = (text *) "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, (OCISnapshot *) NULL,
                               (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

Example2: Binding LOBs

```
void insert()
{
   /* The following is allowed */
   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, 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, (OCISnapshot *) NULL,
                               (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

Example3: Binding LOBs

```
void update()
{
   /* The following is allowed, no matter how many rows it updates */
   ub1 buffer[8000];
   text *update_sql = (text *)"UPDATE lob_long_tab SET \
                              C1 = :1, C2=:2, L=:3";
   OCIStmtPrepare(stmthp, errhp, update_sql, strlen((char*)update_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, (OCISnapshot *) NULL,
                               (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

Example4: Binding LOBs

```
void update()
```

```
{
   /* The following is allowed, no matter how many rows it updates */
   ub1 buffer[8000];
   text *update_sql = (text *)"UPDATE lob_long_tab SET \
                              C1 = :1, C2=:2, L=:3";
   OCIStmtPrepare(stmthp, errhp, update_sql, strlen((char*)update_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, (OCISnapshot *) NULL,
                              (OCISnapshot *) NULL, 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
    * into 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, (OCISnapshot *) NULL,
                             (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

Example7: Binding LOBs

```
void insert()
{
   /* Insert of data into LOB attributes is allowed */
   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, (OCISnapshot *) NULL,
                             (OCISnapshot *) NULL, 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, (OCISnapshot *) NULL,
                              (OCISnapshot *) NULL, OCI_DEFAULT);
}
```

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 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:** "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-29 for more information about piecewise operations

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 in OCI" on page 5-28

Overview of Defining in OCI

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 for retrieving data. The define step creates an association that determines where returned results are stored, and in what format.

For example, if your program processes the following statement 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:

```
SELECT name, ssn FROM employees
    WHERE empno = :empnum
```

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 returns more than a single row for a query, the output variables you define can be arrays instead of scalar values.

Depending on the application, the define step can take place before or after an execute. If you know the datatypes of select-list items at compile time, the define can take place before the statement is executed. If your application is processing dynamic SQL statements entered by you at runtime or statements that do not have a clearly defined select-list, the application must execute the statement to retrieve describe information. After the describe information is retrieved, the type information for each select-list item is available for use in defining output variables.

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 determines what data conversions must 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:

- Chapter 3, "Datatypes" For more information about datatypes and conversions
- "Describing Select-list Items" on page 4-9 for more information

Steps Used in OCI Defining

A basic define is done with a 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 determines where to put retrieved data. You can make your define calls again to redefine the output variables without having to re-prepare or re-execute the SQL statement.

The following example shows a scalar output variable being defined following an execute and a describe.

```
SELECT department_name FROM departments WHERE department_id = :dept_input

   /* The input placeholder was bound earlier, and the data comes from the
   user input below */

  printf("Enter employee dept: ");
  scanf("%d", &deptno);

 /* 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, stmthp, errhp, 0, 0, (OCISnapshot *) 0,
(OCISnapshot *) 0,
       OCI_DEFAULT))
      && (status != OCI_NO_DATA))
  {
    checkerr(errhp, status);
    return OCI_ERROR;
  }
  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((dvoid *)stmthp, (ub4) OCI_HTYPE_STMT, errhp, (dvoid
**)&parmdp, (ub4) 1));
  checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &deptlen, (ub4 *) &sizelen, (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(stmthp, &defnp, errhp,
             1, (dvoid *) dept, (sb4) deptlen+1,
             SQLT_STR, (dvoid *) 0, (ub2 *) 0,
             (ub2 *) 0, OCI_DEFAULT))
  {
    checkerr(errhp, status);
    return OCI_ERROR;
  }
```

> **See Also:** "Describing Select-list Items" on page 4-9 for an explanation of the describe step

Advanced OCI 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 datatype fetch, `OCIDefineObject()`. For example, to fetch multiple rows with a column of named datatypes, all three calls must be invoked for the column; but to fetch multiple rows of scalar columns, `OCIDefineArrayOfStruct()` and `OCIDefineByPos()` are sufficient.

Oracle also provides pre-defined C datatypes that map object type attributes.

> **See Also:**
>
> - Chapter 11, "Object-Relational Datatypes in OCI"
> - "Advanced Define Operations in OCI" on page 5-14

Advanced Define Operations in OCI

This section covers advanced defined operations, including multi-step defines, and defines of named datatypes and REFs.

In some cases the define step requires additional calls that define the attributes of an array fetch, `OCIDefineArrayOfStruct()`, or a named datatype fetch, `OCIDefineObject()`. For example, to fetch multiple rows with a column of named datatypes, all the three calls must be invoked for the column. To fetch multiple rows of scalar columns only `OCIDefineArrayOfStruct()` and `OCIDefineByPos()` are sufficient.

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 OCI LOB functions.
- Define as a LOB value directly, without using the LOB locator.

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;
```

The following sequence of steps 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 */
```

You can also do an array select using the same SQL `SELECT` statement. In this case, the application would include the following steps:

```
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 `BLOB`s, `CLOB`s, and `NCLOB`s. Use `OCI_DTYPE_FILE` when allocating `BFILE`s.

Defining LOB Data

Oracle allows nonzero defines for `SELECT`s of any size LOB. So you can select up to the maximum allowed size of data from a LOB column using `OCIDefineByPos()`, and PL/SQL defines. Because there can be multiple LOBs in a row, you can select the maximum size of data from each one of those LOBs in the same `SELECT` statement.

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 Before Execution

```
void select_define_before_execute()      /* A function in an OCI program */
{
```

```
  /* The following is allowed */
   ub1 buffer1[8000];
   ub1 buffer2[8000];
   text *select_sql = (text *)"SELECT c1, c2 FROM lob_tab";

   OCIStmtPrepare(stmthp, errhp, select_sql, (ub4)strlen((char*)select_sql),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[0], errhp, 1, (dvoid *)buffer1, 8000,
                 SQLT_LNG, (void *)0, (ub2 *)0, (ub2 *)0, (ub4) OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (dvoid *)buffer2, 8000,
                 SQLT_LNG, (void *)0, (ub2 *)0, (ub2 *)0, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *)0,
                  (OCISnapshot *)0, OCI_DEFAULT);
}
```

Example2: Defining LOBs after Execution

```
void select_execute_before_define()
{
  /* The following is allowed */
   ub1 buffer1[8000];
   ub1 buffer2[8000];
   text *select_sql = (text *)"SELECT c1, c2 FROM lob_tab";

   OCIStmtPrepare(stmthp, errhp, select_sql, (ub4)strlen((char*)select_sql),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
   OCIStmtExecute(svchp, stmthp, errhp, 0, 0, (OCISnapshot *)0,
                  (OCISnapshot *)0, OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[0], errhp, 1, (dvoid *)buffer1, 8000,
                  SQLT_LNG, (void *)0, (ub2 *)0, (ub2 *)0, (ub4) OCI_DEFAULT);
   OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (dvoid *)buffer2, 8000,
                  SQLT_LNG, (void *)0, (ub2 *)0, (ub2 *)0, (ub4) OCI_DEFAULT);
   OCIStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);
}
```

Defining PL/SQL Output Variables

Do not use the define calls to define output variables for select-list items in a SQL SELECT statement inside a PL/SQL block. Use OCI bind calls instead.

> **See Also:** "Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-27 for more information about defining PL/SQL output variables.

Defining for a Piecewise Fetch

A piecewise fetch requires an initial call to `OCIDefineByPos()`. An additional call to `OCIDefineDynamic()` is necessary if the application will use callbacks rather than the standard polling mechanism.

Binding and Defining Arrays of Structures in OCI

Defining arrays of structures requires an initial call to `OCIDefineByPos()`. An additional call to `OCIDefineArrayOfStruct()` is necessary to set up each additional parameter, including the `skip` parameter necessary for arrays of structures operations.

Using arrays of structures can simplify the processing of multi-row, multi-column operations. You 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 columns NAME, AGE, and SALARY. The application can 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.

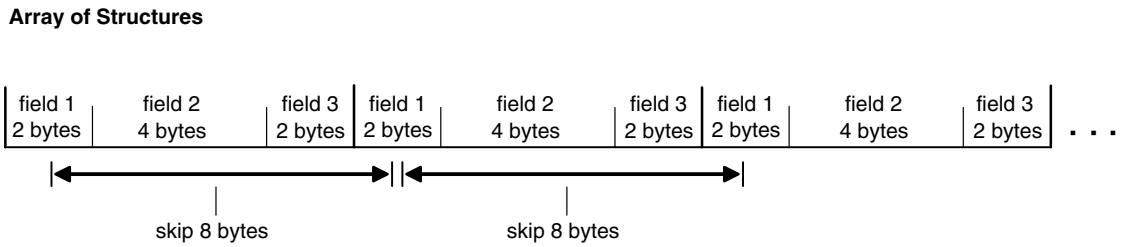
In order to perform a multi-row, multi-column operation using an array of structures, associate each column involved in the operation with a field in a structure. This association, which is part of OCIDefineArrayOfStruct() and OCIBindArrayOfStruct() calls, specifies where data is stored.

Skip Parameters

When you split column data across an array of structures, it is no longer stored contiguously in the database. The single array of structures stores data as though it were composed of several arrays of scalars. For this reason, you must specify a skip parameter for each field you are binding or defining. This skip parameter is the number of bytes that need to be skipped in the array of structures before the same field is encountered again. In general, this will be equivalent to the byte size of one structure.

Figure 5–2 shows 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 5–2 Determining Skip Parameters



On some operating systems it may be necessary to set the skip parameter to sizeof(one_array_element) rather than sizeof(struct), because some compilers insert extra bytes into a structure.

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 operating 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

Arrays of structures are an extension of binding and defining arrays of single variables. When specifying a single-variable 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
- `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 precede a call to `OCIBindArrayOfStruct()`, and a call to `OCIDefineByPos()` must precede a call to `OCIDefineArrayOfStruct()`.

Arrays of Structures and Indicator Variables

The implementation of arrays of structures also supports the use of indicator variables and return codes. You can declare parallel arrays of column-level indicator variables and return codes that correspond to the arrays of information being fetched, inserted, or updated. These arrays can have their own skip parameters, which are specified during `OCIBindArrayOfStruct()` or `OCIDefineArrayOfStruct()` calls.

You can set up arrays of structures of program values and indicator variables in many ways. 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. A one-to-one relationship between the fields in an indicator struct and the number of select-list items is not necessary.

See Also: "Indicator Variables" on page 2-23 for more information about indicator variables.

DML with RETURNING Clause in OCI

OCI supports the use of the `RETURNING` clause with SQL `INSERT`, `UPDATE`, and `DELETE` statements. This section outlines the rules for correctly implementing DML statements with the `RETURNING` clause.

See Also:

- For a complete examples, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs"
- *Oracle Database SQL Reference*. or more information about the use of the `RETURNING` clause with `INSERT`, `UPDATE`, or `DELETE` statements

Using DML with RETURNING Clause

Using the `RETURNING` clause with a DML statement enables you to 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 OCI, values are returned to the application as `OUT` bind variables. In the following examples, the bind variables are indicated by a preceding colon, ":". These examples assume the existence of `table1`, a table that contains columns `col1`, `col2`, and `col3`.

The following statement inserts new values into the database and then retrieves the column values of the affected row from the database, for manipulating inserted rows.

```
INSERT INTO table1 VALUES (:1, :2, :3)
     RETURNING col1, col2, col3
     INTO :out1, :out2, :out3
```

The next example updates the values of all columns where the value of `col1` falls within a given range, and then returns the affected rows which 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 `DELETE` statement deletes the rows where `col1` value falls within a given range, and then returns the data from those rows.

```
DELETE FROM table1 WHERE col1 >= :low AND col2 <= :high
     RETURNING col1, col2, col3
     INTO :out1, :out2, :out3
```

Binding RETURNING...INTO variables

Because both the `UPDATE` and `DELETE` statements can affect multiple rows in the table, and a DML statement can be executed multiple times in a single `OCIExecute()` call, how much data will be returned may not be known at runtime. As a result, the variables corresponding to the `RETURNING...INTO` placeholders must be bound in `OCI_DATA_AT_EXEC` mode. An application must define its own dynamic data handling callbacks rather than using a polling mechanism.

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
```

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.

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.
2. When binding RETURNING clause placeholders, 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 default function which returns NULL values when called.
4. The piecep parameter of OCIBindDynamic() must be set to OCI_ONE_PIECE.

No duplicate binds are allowed in a DML statement with a RETURNING clause, and no duplication between bind variables in the DML section and the RETURNING section of the statement is allowed.

Note: The OCI only supports the callback mechanism for RETURNING clause binds. The polling mechanism is not supported.

OCI 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. If the application has issued a DML statement that is executed 10 times, and an error occurs during the fifth iteration, the server returns the data from iterations 1 through 4. The callback function is still called to receive data for the first four iterations.

DML with RETURNING REF...INTO Clause in OCI

The RETURNING clause can also be used to return a REF to an object which is being inserted into or updated in the database:

```
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
```

The preceding statement updates several attributes of an object in an object table and returns a REF to the object (and a scalar ZIP code) in the RETURNING clause.

Binding the Output Variable

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 preceding 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 OCI 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. During the first callback of an iteration you can allocate space for all rows that are returned for that bind variable. During subsequent callbacks of the same iteration, you merely increment the buffer pointer to the correct memory within the allocated space.

Array Interface for DML RETURNING Statements in OCI

OCI provides additional functionality for single-row DML and array DML operations in which each iteration returns more than one row. To take advantage of this feature, you must specify an OUT buffer in the bind call that is at least as big as the iteration count specified by the `OCIStmtExecute()` call. This is in addition to the bind buffers provided through callbacks.

If any of the iteration returns more than one row, then the application receives an `OCI_SUCCESS_WITH_INFO` return code. In this case, the DML operation is successful. At this point the application may choose to roll back the transaction or ignore the warning.

Character Conversion in OCI Binding and Defining

This section discusses issues involving character conversions between the client and the server.

Choosing Character Set

If a database column containing character data is defined to be an `NCHAR` or `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 the server character set, and also for proper character conversion. During conversion of data between different character sets, the size of the data may increase or decrease by a factor of four. 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` or `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. 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 `csform` attribute (`OCI_ATTR_CHARSET_FORM`) indicates the character set of the client buffer, for binds, and the character set in which to store fetched data for defines. It has two possible values:

- SQLCS_IMPLICIT - default value, indicates database character set ID for the bind or define buffer and the character buffer data are converted to the server database character set
- SQLCS_NCHAR - indicates that the national character set ID for the bind or define buffer and the client buffer data are converted to the server national character set.

If the character set ID attribute, `OCI_ATTR_CHARSET_ID`, is not specified, either the default value of the database or the national character set ID of the client is used,

depending on the value of csform. They are the values specified in the NLS_LANG and NLS_NCHAR environment variables, respectively

---

Note:

- The data is converted and inserted into the database according to the server's database character set ID or national character set ID, regardless of the client-side character set id.
- OCI_ATTR_CHARSET_ID must never be set to 0.
- The define handle attributes OCI_ATTR_CHARSET_FORM and OCI_ATTR_CHARSET_ID do not affect the LOB types. LOB locators fetched from the server retain their original csforms. There is no CLOB/NCLOB conversion as part of define conversion based on these attributes.

See Also: *Oracle Database Reference* for more information about NCHAR data

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. Even though the OCI_ATTR_CHARSET_FORM attribute is set to SQLCS_NCHAR, OCI enables conversion of data to the database character set if the data is inserted into a CHAR column.

Setting Client Character Sets in OCI

You can set the character sets through the OCIEnvNlsCreate() function parameters charset and ncharset. Both of these parameters can be set as OCI_UTF16ID. The charset parameter controls coding of the metadata and CHAR data. ncharset controls coding of NCHAR data. The function OCINlsEnvironmentVariableGet() returns the character set from NLS_LANG and the national character set from NLS_NCHAR.

Here is an example of the use of these functions (OCI provides a typedef called utext to facilitate binding and defining of UTF-16 data):

```
OCIEnv *envhp;
ub2 ncsid = 2; /* we8dec */
ub2 hdlcsid, hdlncsid;
OraText thename[20];
utext *selstmt = L"SELECT ename FROM emp"; /* UTF16 statement */
OCIStmt *stmthp;
OCIDefine *defhp;
OCIError *errhp;
OCIEnvNlsCreate(OCIEnv **envhp, ..., OCI_UTF16ID, ncsid);
...
OCIStmtPrepare(stmthp, ..., selstmt, ...); /* prepare UTF16 statement */
OCIDefineByPos(stmthp, defnp, ..., 1, thename, sizeof(thename), SQLT_CHR,...);
OCINlsEnvironmentVariableGet(&hdlcsid, (size_t)0, OCI_NLS_CHARSET_ID, (ub2)0,
     (size_t*)NULL);
OCIAttrSet(defnp, ..., &hdlcsid, 0, OCI_ATTR_CHARSET_ID, errhp);
          /* change charset ID to NLS_LANG setting*/
...
```

See Also:

- "OCIEnvNlsCreate()" on page 15-18
- "OCINlsEnvironmentVariableGet()" on page 21-6

Using OCI_ATTR_MAXDATA_SIZE Attribute

Update or insert operations are done through variable binding. When binding variables, specify `OCI_ATTR_MAXCHAR_SIZE` and `OCI_ATTR_MAXDATA_SIZE` in the bind handle to indicate character and byte constraints used when inserting data on the server.

These attributes are defined as:

- `OCI_ATTR_MAXCHAR_SIZE` sets the maximum number of characters allowed in the buffer on the server side.
- `OCI_ATTR_MAXDATA_SIZE` sets the maximum number of bytes allowed in the buffer on the server side.

Every bind handle has an `OCI_ATTR_MAXDATA_SIZE` attribute that specifies the number of bytes allocated on the server to accommodate client-side bind data after character set conversions.

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 large enough to accommodate the data after conversion, and the operation will fail.

For `IN/INOUT` binds, when OCI_ATTR_MAXDATA_SIZE attribute is set, the bind buffer must be large enough to hold the number of characters multiplied by the bytes in each character of the character set.

If `OCI_ATTR_MAXCHAR_SIZE` is set to a nonzero value such as 100, then if the character set has 2 bytes in each character, the minimum possible allocated size is 200 bytes.

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)

  There are implicit bind conversions of the data. The recommended value of `OCI_ATTR_MAXDATA_SIZE` is the size of the source buffer multiplied by the worst-case expansion factor between the client and server character sets.

- Scenario 2: CHAR (source data) -> CHAR (destination column) or non-CHAR (source data) -> CHAR (destination column)

  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

`OCI_ATTR_MAXCHAR_SIZE` enables processing 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 reserved on the server to store the bind data.

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.

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 in terms of bytes. The actual length values sent and received by you are also in bytes.

Buffer Expansion During OCI Binding

Do not set OCI_ATTR_MAXDATA_SIZE for OUT binds or for PL/SQL binds. Only set OCI_ATTR_MAXDATA_SIZE for INSERT or UPDATE statements.

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. As long as the actual buffer contains less characters than 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 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 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. It is a good practice to specify OCI_ATTR_MAXDATA_SIZE and OCI_ATTR_MAXCHAR_SIZE to be no more than the actual column width in bytes, or characters.

Buffer Expansion During Inserts

You should avoid unexpected behavior caused by buffer expansion during inserts.

Consider what happens when the database column has character length semantics, and the user tries to insert data using OCIBindByPos() or OCIBindByName() while setting only the OCI_ATTR_MAXCHAR_SIZE to 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. 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 the OCI_ATTR_MAXDATA_SIZE to be 4000, to guarantee that the data will never exceed 4000 bytes.

Constraint Checking During Defining

To select data from 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. There is no OCI_ATTR_MAXDATA_SIZE attribute for define handles since the buffer size in bytes serves as the limit on byte length. 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 avoid data loss through truncation. If the database column is created using character length semantics 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. A maximum number of OCI_ATTR_MAXCHAR_SIZE characters is put in the buffer.

Return Lengths

The following length values are always in bytes regardless 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 character set id; then the lengths are in UTF-16 units.

> **Note:** 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 in OCI

- 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 for each 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 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 for each character in UTF-8.

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 */
sb4 ename_max_chars = EC=20, address_max_chars = ED=60;
 /* EC <= (E' - 1), ED <= (D' - 1) */
sb4 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, &bnd1p, 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 *)dfn1p, (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 or VARCHAR2, or in NCHAR or NVARCHAR variant handles can be set to assume that all data will be passed in UTF-16 (Unicode) encoding. To specify UTF-16, set OCI_ATTR_CHARSET_ID = OCI_UTF16ID.

> **See Also:** "Bind Handle Attributes" on page A-28

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. Operating systems where the encoding scheme of the wchar_t datatype conforms to UTF-16 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 utext datatype as the buffer for input and output data.

The following pseudocode illustrates a bind and define for UTF-16 data:

```
...
OCIStmt  *stmthp1, *stmthp2;
OCIDefine *dfn1p, *dfn2p;
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;
sb4 ename_col_len = 20;
sb4 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 in OCI

The OCI provides the ability to bind and define PL/SQL `REF CURSOR`s 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 employee_id, last_name, job_id, manager_id, \
             salary, department_id \
             FROM employees WHERE job_id=:job ORDER BY employee_id; \
     OPEN :cursor2 FOR SELECT * FROM departments ORDER BY department_id;
  end;";
```

An application allocates a statement handle for binding, by calling `OCIHandleAlloc()`, and then binds the `:cursor1` placeholder to the statement handle, as in the following code, where `:cursor1` is bound to `stm2p`.

```
status = OCIStmtPrepare (stm1p, errhp, (text *) plsql_block,
             strlen((char *)plsql_block), OCI_NTV_SYNTAX, OCI_DEFAULT);
...
status = OCIBindByName (stm1p, (OCIBind **) &bnd1p, 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 last_name, CURSOR(SELECT department_name, location_id \
        FROM  departments)  FROM employees WHERE last_name = 'FORD'";
```

The second position is a nested table, which an OCI application can define as a statement handle as follows:

```
status = OCIStmtPrepare (stm1p, errhp, (text *) nst_tab,
         strlen((char *)nst_tab), OCI_NTV_SYNTAX, OCI_DEFAULT);
...
status = 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 becomes a valid statement handle.

Note: If you have retrieved multiple `REF CURSOR`s, 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`.

OCI does not support PL/SQL `REF CURSOR`s that were executed in scrollable mode.

OCI does not support scrollable `REF CURSOR`s because you cannot scroll back to the rows already fetched by a `REF CURSOR`.

Runtime Data Allocation and Piecewise Operations in OCI

You can use the OCI to perform piecewise inserts, updates, and fetches of data. You can also use the OCI to provide data dynamically in 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 and can be uniform or not.

The piecewise functionality of OCI is particularly useful when performing operations on extremely large blocks of string or binary data, operations involving database columns that store `CLOB`, `BLOB`, `LONG`, `RAW`, or `LONG RAW` data.

The piecewise fetch is complete when the final `OCIStmtFetch()` call returns a value of OCI_SUCCESS.

In both the piecewise fetch and insert, it is important to understand the sequence of calls necessary for the operation to complete successfully. 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.

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`

Another way of using this feature for all datatypes is to provide data dynamically for array inserts or updates. The callbacks should always specify `OCI_ONE_PIECE` for the `piecep` parameter of the callback for datatypes that do not support piecewise operations.

Types of Piecewise Operations

You can perform piecewise operations in two ways:

- Use calls provided in the OCI library to execute piecewise operations under a polling paradigm.
- 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`, it 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`, it 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: Piecewise operations are also valid for SQL and 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 no longer required, they must 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 either 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 that specifies which bind and 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 that are not part of piecewise operations may require additional bind calls, depending on their datatypes.

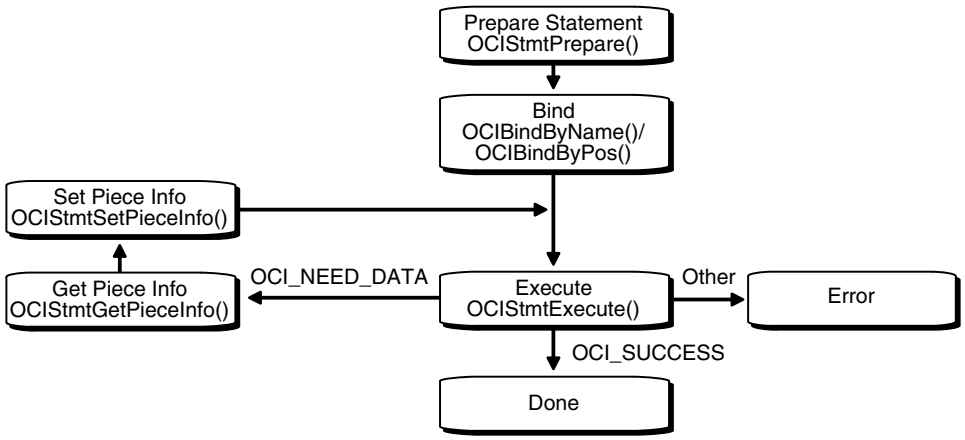
Following the statement preparation and bind, the application performs a series of calls to OCIStmtExecute(), OCIStmtGetPieceInfo() and OCIStmtSetPieceInfo() to complete the piecewise operation. Each call to OCIStmtExecute() returns a value that determines what action should be performed next. In general, 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. You must account for this by indicating the smaller size in the final OCIStmtSetPieceInfo() call.

The procedure is illustrated in Figure 5–3.

Figure 5–3 Performing Piecewise Insert



1. Initialize the OCI environment, allocate the necessary handles, connect to a server, authorize a user, and prepare a statement request.
2. Bind a placeholder using `OCIBindByName()` or `OCIBindByPos()`. 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.
3. Call `OCIStmtExecute()` for the first time. No data is being inserted here, and the `OCI_NEED_DATA` error code is returned to the application. If any other value is returned, it indicates that an error occurred.
4. Call `OCIStmtGetPieceInfo()` to retrieve information about the piece that needs to be inserted. The parameters of `OCIStmtGetPieceInfo()` include a pointer to a value indicating if the required piece is the first piece, `OCI_FIRST_PIECE`, or a subsequent piece, `OCI_NEXT_PIECE`.
5. The application populates a buffer with the piece of data to be inserted and calls `OCIStmtSetPieceInfo()` with these parameters:
   - a pointer to the piece
   - a pointer to the length of the piece
   - a value indicating whether this is the

   a. first piece, `OCI_FIRST_PIECE`

   b. an intermediate piece, `OCI_NEXT_PIECE`

   c. the last piece, `OCI_LAST_PIECE`
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.

Piecewise updates are performed in a similar manner. In a piecewise update operation the insert buffer is populated with data that is being updated and `OCIStmtExecute()` is called to execute the update.

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 previously. 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 may also need to call `OCIDefineDynamic()` to set callback function that will be invoked to get information about your data buffer.

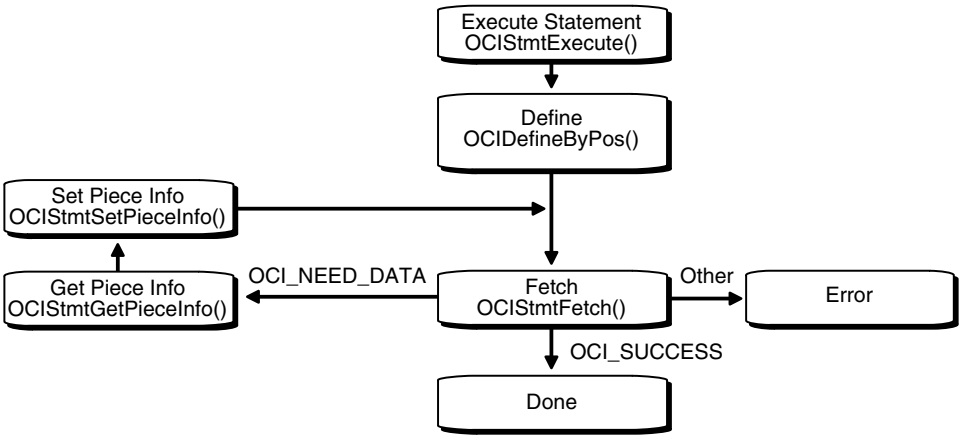
Run-time data is provided in one of the two ways:

- You can define a callback using the `OCIDefineDynamic()`. 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 either 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()`. The `OCIStmtGetPieceInfo()` call provides Information about which define and which piece are involved.

Performing a Piecewise Fetch

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. This process is illustrated in Figure 5–4.

Figure 5–4 Performing Piecewise Fetch



1. Initialize the OCI environment, allocate necessary handles, connect to a database, authorize a user, prepare a statement, and execute the statement.
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.

3. Call `OCIStmtFetch()` for the first time. No data is retrieved, and the `OCI_NEED_DATA` error code is returned to the application. If any other value is returned, an error occurred.
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`.
5. Call `OCIStmtSetPieceInfo()` to specify the fetch buffer.
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.

Piecewise Binds and Defines for LOBs

There are two ways of doing piecewise binds and defines for LOBs:

1. **Using the data interface**

   You can bind or define character data for `CLOB` columns using `SQLT_CHR` (`VARCHAR2`) or `SQLT_LNG` (`LONG`) as the input datatype for the following functions. You can also bind or define raw data for BLOB columns using `SQLT_LBI` (`LONG RAW`), and `SQLT_BIN` (`RAW`) as the input datatype for these functions:

   - OCIDefineByPos()
   - OCIBindByName()
   - OCIBindByPos(}

   See Also:

   - "Binding LOB Data" on page 5-8 for usage and examples for both `INSERT` and `UPDATE statements`
   - "Defining LOB Data" on page 5-15 for usage and examples of `SELECT` statements

   All the piecewise operations described later are supported for `CLOB` and `BLOB` columns in this case.

2. **Using the LOB locator**

   You can bind or define a LOB locator for `CLOB` and `BLOB` columns using `SQLT_CLOB` (`CLOB`) or `SQLT_BLOB` (`BLOB`) as the input datatype for the following functions.

   - OCIDefineByPos()
   - OCIBindByName()
   - OCIBindByPos(}

   You must then call OCILob* functions to read and manipulate the data. `OCILobRead2()` and `OCILobWrite2()` support piecewise and callback modes.

See Also:

- "OCILobRead2()" on page 16-78
- "OCILobWrite2()" on page 16-88
- "LOB Read and Write Callbacks" on page 7-11 for information about streaming using callbacks with `OCILobWrite2()` and `OCILobRead2()`.

6

Describing Schema Metadata

This chapter discusses the use of the OCIDescribeAny() function to obtain information about schema elements.

This chapter contains these topics:

Using OCIDescribeAny()

The `OCIDescribeAny()` function enables you to perform an explicit describe of the following schema objects and their subschema objects:

- tables and views
- synonyms
- procedures
- functions
- packages
- sequences
- collections
- types
- schemas
- databases

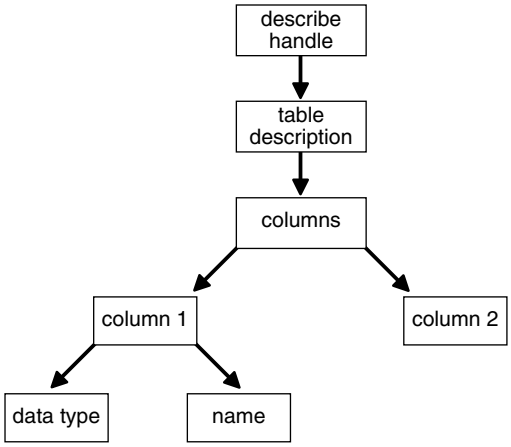
Information about other schema elements (function arguments, columns, type attributes, and type methods) is available through a describe of one of the preceding schema objects or an explicit describe of the subschema object.

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 given the name of the subschema object.

The `OCIDescribeAny()` call requires a describe handle as one of its arguments. The describe handle must be previously allocated with a call to OCIHandleAlloc().

The information returned by `OCIDescribeAny()` is organized hierarchically like a tree, as shown in Figure 6–1:

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 that node, and attributes that are like recursive describe handles and point to subtrees containing further information. If all the attributes are homogenous, as in the case of elements of a column list, they are called *parameters*. The attributes associated with any node are returned by `OCIAttrGet()`, and the parameters are returned by `OCIParamGet()`.

A call to `OCIAttrGet()` on the describe handle for the table returns 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 datatype.

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.

Note: No subsequent `OCIAttrGet()` or `OCIParamGet()` call requires extra round trips, as all the description is cached on the client side by `OCIDescribeAny()`.

Limitations on OCIDescribeAny()

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.

The table name is not returned by `OCIDescribeAny()` or the implicit use of `OCIStmtExecute()`. Sometimes a column is not associated with a table. In most cases, the table is already known.

See Also:

- "Describing Select-list Items" on page 4-9
- "OCIDescribeAny()" on page 15-83

Notes on Types and Attributes

When performing describe operations, you should be aware of the following:

Datatype Codes

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.

> **See Also:** "Typecodes" on page 3-24 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

Describing Types

In order to describe type objects, it is necessary to initialize the OCI process in object mode:

```
/* Initialize the OCI Process */
 if (OCIEnvCreate((OCIEnv **) &envhp, (ub4) OCI_OBJECT, (dvoid *) 0,
                  (dvoid * (*)(dvoid *,size_t)) 0,
                  (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                  (void (*)(dvoid *, dvoid *)) 0, (size_t) 0, (dvoid **) 0))
 {
   printf("FAILED: OCIEnvCreate()\n");
   return OCI_ERROR;
 }
```

> **See Also:** "OCIInitialize()" on page 15-22

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, and have attributes depending on the type of description they contain, or type-specific attributes. This section describes the attributes and handles that belong to different parameters.

Note that OCIDescribeAny() does support more than two name components, for example, schema.type.attr1.attr2.method1. Note that with more than one component, the first component is interpreted as the schema name (unless some other flag is set). There is a flag to specify that the object must be looked up under PUBLIC, that is, describe "a", where "a" can be either in the current schema or a public synonym.

If you do not know what the object type is, specify OCI_PTYPE_UNK. Otherwise an error is returned if the actual object type does not match the specified type.

Table 6–1 lists the attributes of all parameters:

Table 6–1 Attributes of 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	OraText *
OCI_ATTR_OBJ_SCHEMA	Schema name where the object is located	OraText *

Table 6–1 (Cont.) Attributes of All Parameters

Attribute	Description	Attribute Datatype
OCI_ATTR_PTYPE	Type of information described by the parameter. Possible values: 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 - 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 - 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_PTYPE_UNK - unknown schema object	ub1
OCI_ATTR_TIMESTAMP	The timestamp of the object on which the description is based in Oracle date format	ub1 *

The following sections list the attributes and handles specific to different types of parameters.

Table or View Parameters

Parameters for a table or view (type OCI_PTYPE_TABLE or OCI_PTYPE_VIEW) have the following type-specific attributes:

Table 6–2 Attributes of 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*

Table 6–2 (Cont.) Attributes of Tables or Views

Attribute	Description	Attribute Datatype
OCI_ATTR_IS_TEMPORARY	Indicates the table is temporary	ub1
OCI_ATTR_IS_TYPED	Indicates the table is 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_RDBA	Data block address of the segment header	ub4
OCI_ATTR_TABLESPACE	Tablespace the table resides in	word
OCI_ATTR_CLUSTERED	Indicates the table is clustered	ub1
OCI_ATTR_PARTITIONED	Indicates the table is partitioned	ub1
OCI_ATTR_INDEX_ONLY	Indicates the table is 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 of Procedures or Functions

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_ARGUMENTS	Argument list. See "List Attributes" on page 6-14.	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	Indicates the procedure or function has invoker's 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	OraText *
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 of Packages

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_SUBPROGRAMS	Subprogram list. See "List Attributes" on page 6-14.	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	Is the package invoker's 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 of 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 "Datatype Codes" on page 6-3. 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 "Datatype Codes" on page 6-3. 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	Indicates this is an incomplete type	ub1
OCI_ATTR_IS_SYSTEM_TYPE	Indicates this is a system type	ub1
OCI_ATTR_IS_PREDEFINED_TYPE	Indicates this is a predefined type	ub1
OCI_ATTR_IS_TRANSIENT_TYPE	Indicates this is a transient type	ub1
OCI_ATTR_IS_SYSTEM_GENERATED_TYPE	Indicates this is a system-generated type	ub1
OCI_ATTR_HAS_NESTED_TABLE	This type contains a nested table attribute	ub1
OCI_ATTR_HAS_LOB	This type contains a LOB attribute	ub1
OCI_ATTR_HAS_FILE	This type contains a BFILE attribute	ub1
OCI_ATTR_COLLECTION_ELEMENT	Handle to collection element. See "Collection Attributes" on page 6-10.	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 6-14.	dvoid *
OCI_ATTR_NUM_TYPE_METHODS	Number of type methods	ub2
OCI_ATTR_LIST_TYPE_METHODS	List of type methods. See "List Attributes" on page 6-14.	dvoid *

Table 6–7 (Cont.) Attributes of Types

Attribute	Description	Attribute Datatype
OCI_ATTR_MAP_METHOD	Map method of type. See "Type Method Attributes" on page 6-9.	dvoid *
OCI_ATTR_ORDER_METHOD	Order method of type. See "Type Method Attributes" on page 6-9.	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	Indicates the type has invoker's rights	ub1
OCI_ATTR_NAME	A pointer to a string which is the type attribute name	OraText *
OCI_ATTR_SCHEMA_NAME	A string with the schema name where the type has been created	OraText *
OCI_ATTR_IS_FINAL_TYPE	Indicates this is a final type	ub1
OCI_ATTR_IS_INSTANTIABLE_TYPE	Indicates this is an instantiable type	ub1
OCI_ATTR_IS_SUBTYPE	Indicates this is a subtype	ub1
OCI_ATTR_SUPERTYPE_SCHEMA_NAME	Name of the schema that contains the supertype	OraText *
OCI_ATTR_SUPERTYPE_NAME	Name of the supertype	OraText *

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 of 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 "Datatype Codes" on page 6-3.	OCITypeCode
OCI_ATTR_DATA_TYPE	The datatype of the type attribute. See "Datatype Codes" on page 6-3.	ub2
OCI_ATTR_NAME	A pointer to a string which is the type attribute name	OraText *
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 for explicit describe sb2 for implicit describe
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

Table 6–8 (Cont.) Attributes of Type Attributes

Attribute	Description	Attribute Datatype
OCI_ATTR_TYPE_NAME	A string which is the type name. The returned value will contain the type name if the datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, the name of the named datatype's type is returned. If the datatype is SQLT_REF, the type name of the named datatype pointed to by the REF is returned	OraText *
OCI_ATTR_SCHEMA_NAME	A string with the schema name under which the type has been created	OraText *
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 of Type Methods

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Name of method (procedure or function)	OraText *
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 6-3, and "List Attributes" on page 6-14.	dvoid *
OCI_ATTR_IS_CONSTRUCTOR	Indicates method is a constructor	ub1
OCI_ATTR_IS_DESTRUCTOR	Indicates method is a destructor	ub1
OCI_ATTR_IS_OPERATOR	Indicates method is an operator	ub1
OCI_ATTR_IS_SELFISH	Indicates method is selfish	ub1
OCI_ATTR_IS_MAP	Indicates method is a map method	ub1
OCI_ATTR_IS_ORDER	Indicates method is an order method	ub1
OCI_ATTR_IS_RNDS	Indicates "Read No Data State" is set for method	ub1
OCI_ATTR_IS_RNPS	Indicates "Read No Process State" is set for method	ub1

Table 6–9 (Cont.) Attributes of Type Methods

Attribute	Description	Attribute Datatype
OCI_ATTR_IS_WNDS	Indicates "Write No Data State" is set for method	ub1
OCI_ATTR_IS_WNPS	Indicates "Write No Process State" is set for method	ub1
OCI_ATTR_IS_FINAL_METHOD	Indicates this is a final method	ub1
OCI_ATTR_IS_INSTANTIABLE_METHOD	Indicates this is an instantiable method	ub1
OCI_ATTR_IS_OVERRIDING_METHOD	Indicates this is 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 of 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.	ub4
OCI_ATTR_TYPECODE	Typecode. See "Datatype Codes" on page 6-3.	OCITypeCode
OCI_ATTR_DATA_TYPE	The datatype of the type attribute. See "Datatype Codes" on page 6-3.	ub2
OCI_ATTR_NUM_ELEMS	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	OraText *
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 for explicit describe sb2 for implicit describe
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 datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, the name of the named datatype's type is returned. If the datatype is SQLT_REF, the type name of the named datatype pointed to by the REF is returned	OraText *
OCI_ATTR_SCHEMA_NAME	A string with the schema name under which the type has been created	OraText *

Table 6–10 (Cont.) Attributes of Collection Types

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 of Synonyms

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJID	Object id	ub4
OCI_ATTR_SCHEMA_NAME	A string containing the schema name of the synonym translation	OraText *
OCI_ATTR_NAME	A NULL-terminated string containing the object name of the synonym translation	OraText *
OCI_ATTR_LINK	A NULL-terminated string containing the database link name of the synonym translation	OraText *

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 of 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 *
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 NUMBER format)	ub1 *

See Also: "OCINumber Examples" on page 11-10

Column Attributes

Note: For BINARY_FLOAT and BINARY_DOUBLE:

If `OCIDescribeAny()` is used to retrieve the column datatype (`OCI_ATTR_DATA_TYPE`) for `BINARY_FLOAT` or `BINARY_DOUBLE` columns in a table, it returns the internal datatype code.

The SQLT codes corresponding to the internal datatype codes for `BINARY_FLOAT` and `BINARY_DOUBLE` are `SQLT_IBFLOAT` and `SQLT_IBDOUBLE`.

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 of 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 6-18.	`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 6-18.	`ub2`
`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 `NUMBER`s.	`ub4`
`OCI_ATTR_DATA_TYPE`	The datatype of the column. See "Datatype Codes" on page 6-3.	`ub2`
`OCI_ATTR_NAME`	A pointer to a string which is the column name	`OraText *`
`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`
`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 datatype is `SQLT_NTY` or `SQLT_REF`. If the datatype is `SQLT_NTY`, the name of the named datatype's type is returned. If the datatype is `SQLT_REF`, the type name of the named datatype pointed to by the `REF` is returned	`OraText *`

Table 6–13 (Cont.) Attributes of Columns of Tables or Views

Attribute	Description	Attribute Datatype
OCI_ATTR_SCHEMA_NAME	Returns a string with the schema name under which the type has been created	OraText *
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 and Result Attributes

When a parameter is for an argument of a procedure or 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 of Arguments and Results

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Returns a pointer to a string which is the argument name	OraText *
OCI_ATTR_POSITION	The position of the argument in the argument list. Always returns zero.	ub2
OCI_ATTR_TYPECODE	Typecode. See "Datatype Codes" on page 6-3.	OCITypeCode
OCI_ATTR_DATA_TYPE	The datatype of the argument. See "Datatype Codes" on page 6-3.	ub2
OCI_ATTR_DATA_SIZE	The size of the datatype of the argument. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERs.	ub4
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.	b1 for explicit describe sb2 for implicit describe
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 datatype 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 *

Table 6–14 (Cont.) Attributes of Arguments and Results

Attribute	Description	Attribute Datatype
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
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 datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, the name of the named datatype's type is returned. If the datatype is SQLT_REF, the type name of the named datatype pointed to by the REF is returned.	OraText *
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	OraText *
OCI_ATTR_SUB_NAME	For SQLT_NTY or SQLT_REF, returns a string with the type name, in the case of package local types	OraText *
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.	OraText *
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

Table 6–15 (Cont.) List Attributes

List Attribute	Description	Lower Bound
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	OraText *

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	OraText *
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	ub1
OCI_ATTR_MAX_PROC_LEN	Maximum length of a procedure name	ub4
OCI_ATTR_MAX_COLUMN_LEN	Maximum length of a column name	ub4

Table 6–17 (Cont.) Attributes Specific to Databases

Attribute	Description	Attribute Datatype
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 re-execute 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

Rule Attributes

When a parameter is for a rule (type OCI_PTYPE_RULE), it has the attributes listed in Table 6–18:

Table 6–18 Attributes Specific to Rules

Attribute	Description	Attribute Datatype
OCI_ATTR_CONDITION	Rule condition	OraText *
OCI_ATTR_EVAL_CONTEXT_OWNER	Owner name of evaluation context associated with the rule, if any	OraText *
OCI_ATTR_EVAL_CONTEXT_NAME	Object name of evaluation context associated with the rule, if any	OraText *
OCI_ATTR_COMMENT	Comment associated with the rule, if any	OraText *
OCI_ATTR_LIST_ACTION_CONTEXT	List of name value pairs in the action context (type OCI_PTYPE_LIST)	dvoid *

Rule Set Attributes

When a parameter is for a rule set (type OCI_PTYPE_RULE_SET), it has the attributes listed in Table 6–19:

Table 6–19 Attributes Specific to Rule Sets

Attribute	Description	Attribute Datatype
OCI_ATTR_EVAL_CONTEXT_OWNER	Owner name of evaluation context associated with the rule set, if any	OraText *
OCI_ATTR_EVAL_CONTEXT_NAME	Object name of evaluation context associated with the rule set, if any	OraText *
OCI_ATTR_COMMENT	Comment associated with the rule set, if any	OraText *
OCI_ATTR_LIST_RULES	List of rules in the rule set (type OCI_PTYPE_LIST)	dvoid *

Evaluation Context Attributes

When a parameter is for an evaluation context (type OCI_PTYPE_EVALUATION_CONTEXT), it has the attributes listed in Table 6–20:

Table 6–20 Attributes Specific to Evaluation Contexts

Attribute	Description	Attribute Datatype
OCI_ATTR_EVALUATION_FUNCTION	Evaluation function associated with the evaluation context, if any	OraText *
OCI_ATTR_COMMENT	Comment associated with the evaluation context, if any	OraText *
OCI_ATTR_LIST_TABLE_ALIASES	List of table aliases in the evaluation context (type OCI_PTYPE_LIST)	dvoid *
OCI_ATTR_LIST_VARIABLE_TYPES	List of variable types in the evaluation context (type OCI_PTYPE_LIST)	dvoid *

Table Alias Attributes

When a parameter is for a table alias (type OCI_PTYPE_TABLE_ALIAS), it has the attributes listed in Table 6–21:

Table 6–21 Attributes Specific to Table Aliases

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Table alias name	OraText *
OCI_ATTR_TABLE_NAME	Table name associated with the alias	OraText *

Variable Type Attributes

When a parameter is for a variable (type OCI_PTYPE_VARIABLE_TYPE), it has the attributes listed in Table 6–22:

Table 6–22 Attributes Specific to Variable Types

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Variable name	OraText *
OCI_ATTR_TYPE	Variable type	OraText *
OCI_ATTR_VAR_VALUE_FUNCTION	Variable value function associated with the variable, if any	OraText *

Table 6–22 (Cont.) Attributes Specific to Variable Types

Attribute	Description	Attribute Datatype
OCI_ATTR_VAR_METHOD_FUNCTION	Variable method function associated with the variable, if any	OraText *

Name Value Attributes

When a parameter is for a name value pair (type OCI_PTYPE_NAME_VALUE), it has the attributes listed in Table 6–23:

Table 6–23 Attributes Specific to Name Value Pair

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	Name	OraText *
OCI_ATTR_VALUE	Value	OCIAnyData*

Character Length Semantics Support in Describing

Since release Oracle9*i*, query and column information are supported with character length semantics.

The following 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 that gets the byte length.
- Calling OCIAttrGet() with attribute OCI_ATTR_CHAR_SIZE or OCI_ATTR_DATA_SIZE does not return data on stored procedure parameters, because stored procedure parameters are not bounded.
- 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 can 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 the character length, the byte length, and a flag indicating how the database column was created. OCI_ATTR_CHAR_SIZE is the character length of the column or expression. The OCI_ATTR_CHAR_USED flag is 1 in this case, indicating that the column or expression was created with 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 (OCI_ATTR_CHAR_SIZE)*(the client's max bytes) for each 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 (column's byte length)*(the maximum conversion ratio between the client and server's character set), that is, column byte length divided by the server's max bytes for each character multiplied by the client's

max bytes for each 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 the following attributes:

- 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
- 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 can set the OCI_ATTR_MAXCHAR_SIZE in the bind handle to the value returned by OCI_ATTR_CHAR_SIZE in the parameter handle. This will prevent you from violating the size constraint for the column.

> **See Also:** "IN Binds" on page 5-25

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 both cases, 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 Datatypes for a Table

This example illustrates the use of an explicit describe that retrieves the column datatypes for a table.

```
...
int i=0;
text objptr[] = "EMPLOYEES"; /* the name of a table to be described */
ub2          numcols, col_width;
ub1          char_semantics;
ub2  coltyp;
ub4 objp_len = (ub4) strlen((char *)objptr);
OCIParam *parmh = (OCIParam *) 0;         /* parameter handle */
OCIParam *collsthd = (OCIParam *) 0;      /* handle to list of columns */
OCIParam *colhd = (OCIParam *) 0;         /* column handle */
OCIDescribe *dschp = (OCIDescribe *)0;      /* describe handle */

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&dschp,
```

```
        (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0);

/* get the describe handle for the table */
if (OCIDescribeAny(svch, errh, (dvoid *)objptr, objp_len, OCI_OTYPE_NAME, 0,
     OCI_PTYPE_TABLE, dschp))
   return OCI_ERROR;

/* get the parameter handle */
if (OCIAttrGet((dvoid *)dschp, OCI_HTYPE_DESCRIBE, (dvoid *)&parmh, (ub4 *)0,
               OCI_ATTR_PARAM, errh))
    return OCI_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 */
numcols = 0;
if (OCIAttrGet((dvoid *)parmh, OCI_DTYPE_PARAM, (dvoid *)&numcols, (ub4 *)0,
     OCI_ATTR_NUM_COLS, errh))
    return OCI_ERROR;

/* get the handle to the column list of the table */
if (OCIAttrGet((dvoid *)parmh, OCI_DTYPE_PARAM, (dvoid *)&collsthd, (ub4 *)0,
     OCI_ATTR_LIST_COLUMNS, errh)==OCI_NO_DATA)
   return OCI_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((dvoid *)collsthd, OCI_DTYPE_PARAM, errh, (dvoid **)&colhd,
(ub4)i))
        return OCI_ERROR;

    /* for example, get datatype for ith column */
    coltyp = 0;
    if (OCIAttrGet((dvoid *)colhd, OCI_DTYPE_PARAM, (dvoid *)&coltyp, (ub4 *)0,
       OCI_ATTR_DATA_TYPE, errh))
        return OCI_ERROR;

    /* Retrieve the length semantics for the column */
    char_semantics = 0;
    OCIAttrGet((dvoid*) colhd, (ub4) OCI_DTYPE_PARAM,
                (dvoid*) &char_semantics,(ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
                (OCIError *) errh);

    col_width = 0;
    if (char_semantics)
        /* Retrieve the column width in characters */
        OCIAttrGet((dvoid*) colhd, (ub4) OCI_DTYPE_PARAM,
                (dvoid*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
                (OCIError *) errh);
    else
        /* Retrieve the column width in bytes */
        OCIAttrGet((dvoid*) colhd, (ub4) OCI_DTYPE_PARAM,
                (dvoid*) &col_width,(ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
                (OCIError *) errh);
}
```

```
if (dschp)
    OCIHandleFree((dvoid *) dschp, OCI_HTYPE_DESCRIBE);
...
```

Describing the Stored Procedure

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. Consider the following procedure:

```
P1 (arg1 emp.sal%type, arg2 emp%rowtype)
```

Assume that each row in emp table has two columns: `name(VARCHAR2(20))`, and `sal(NUMBER)`. In the argument list for `P1`, there are 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.

```
...
int i = 0, j = 0;
text objptr[] = "add_job_history"; /* the name of a procedure to be described */
ub4 objp_len = (ub4)strlen((char *)objptr);
ub2 numargs = 0, numargs1, pos, level;
text *name, *name1;
ub4 namelen, namelen1;
OCIParam *parmh = (OCIParam *) 0;         /* parameter handle */
OCIParam *arglst = (OCIParam *) 0;          /* list of args */
OCIParam *arg = (OCIParam *) 0;             /* argument handle */
OCIParam *arglst1 = (OCIParam *) 0;          /* list of args */
OCIParam *arg1 = (OCIParam *) 0;             /* argument handle */
OCIDescribe *dschp = (OCIDescribe *)0;      /* describe handle */

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&dschp,
        (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0);

/* get the describe handle for the procedure */
if (OCIDescribeAny(svch, errh, (dvoid *)objptr, objp_len, OCI_OTYPE_NAME, 0,
     OCI_PTYPE_PROC, dschp))
   return OCI_ERROR;

/* get the parameter handle */
if (OCIAttrGet((dvoid *)dschp, OCI_HTYPE_DESCRIBE, (dvoid *)&parmh, (ub4 *)0,
         OCI_ATTR_PARAM, errh))
    return OCI_ERROR;

/* Get the number of arguments and the arg list */
if (OCIAttrGet((dvoid *)parmh, OCI_DTYPE_PARAM, (dvoid *)&arglst,
       (ub4 *)0, OCI_ATTR_LIST_ARGUMENTS, errh))
    return OCI_ERROR;

if (OCIAttrGet((dvoid *)arglst, OCI_DTYPE_PARAM, (dvoid *)&numargs, (ub4 *)0,
    OCI_ATTR_NUM_PARAMS, errh))
    return OCI_ERROR;

/* For a procedure, we begin with i = 1; for a
function, we begin with i = 0. */
```

```
for (i = 1; i <= numargs; i++) {
  OCIParamGet ((dvoid *)arglst, OCI_DTYPE_PARAM, errh, (dvoid **)&arg, (ub4)i);
  namelen = 0;
  OCIAttrGet((dvoid *)arg, OCI_DTYPE_PARAM, (dvoid *)&name, (ub4 *)&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((dvoid *)arg, OCI_DTYPE_PARAM, (dvoid *)&arglst1, (ub4 *)0,
        OCI_ATTR_LIST_ARGUMENTS, errh);

  /* check if the current argument is a record. For arg1 in the procedure
  arglst1 is NULL. */

  if (arglst1) {
    numargs1 = 0;
    OCIAttrGet((dvoid *)arglst1, OCI_DTYPE_PARAM, (dvoid *)&numargs1, (ub4 *)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 the procedure, the number of
    arguments at the level 1 would be 2 */

    for (j = 1; j <= numargs1; j++) {
      OCIParamGet((dvoid *)arglst1, OCI_DTYPE_PARAM, errh, (dvoid **)&arg1,
          (ub4)j);
      namelen1 = 0;
      OCIAttrGet((dvoid *)arg1, OCI_DTYPE_PARAM, (dvoid *)&name1, (ub4
*)&namelen1,
        OCI_ATTR_NAME, errh);
    }
  }
}

if (dschp)
    OCIHandleFree((dvoid *) dschp, OCI_HTYPE_DESCRIBE);
...
```

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.

```
...
int i = 0;
text type_name[] = "inventory_typ";
ub4 type_name_len = (ub4)strlen((char *)type_name);
OCIRef *type_ref = (OCIRef *) 0;
ub2 numattrs = 0, describe_by_name = 1;
ub2 datatype = 0;
OCITypeCode typecode = 0;
OCIDescribe *dschp = (OCIDescribe *) 0;      /* describe handle */
OCIParam *parmh = (OCIParam *) 0;         /* parameter handle */
OCIParam *attrlsthd = (OCIParam *) 0;     /* handle to list of attrs */
OCIParam *attrhd = (OCIParam *) 0;        /* attribute handle */
```

```
/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                   (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
   return OCI_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 OCI_ERROR;
}
else {
  /* get ref to type using OCIAttrGet */

  /* get the describe handle for the type */
  if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF,
        0, OCI_PTYPE_TYPE, dschp))
       return OCI_ERROR;
}

/* get the parameter handle */
if (OCIAttrGet((dvoid *)dschp, OCI_HTYPE_DESCRIBE, (dvoid *)&parmh, (ub4 *)0,
        OCI_ATTR_PARAM, errh))
    return OCI_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((dvoid *)parmh, OCI_DTYPE_PARAM, (dvoid *)&numattrs, (ub4 *)0,
    OCI_ATTR_NUM_TYPE_ATTRS, errh))
    return OCI_ERROR;

/* get the handle to the attribute list of the type */
if (OCIAttrGet((dvoid *)parmh, OCI_DTYPE_PARAM, (dvoid *)&attrlsthd, (ub4 *)0,
      OCI_ATTR_LIST_TYPE_ATTRS, errh))
  return OCI_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((dvoid *)attrlsthd, OCI_DTYPE_PARAM, errh, (dvoid **)&attrhd, i))
      return OCI_ERROR;

/* for example, get datatype and typecode for attribute; note that
OCI_ATTR_DATA_TYPE returns the SQLT code, while OCI_ATTR_TYPECODE returns the
Oracle Type System typecode. */

datatype = 0;
if (OCIAttrGet((dvoid *)attrhd, OCI_DTYPE_PARAM, (dvoid *)&datatype, (ub4 *)0,
               OCI_ATTR_DATA_TYPE,errh))
    return OCI_ERROR;

typecode = 0;
if (OCIAttrGet((dvoid *)attrhd, OCI_DTYPE_PARAM,(dvoid *)&typecode, (ub4 *)0,
               OCI_ATTR_TYPECODE, errh))
    return OCI_ERROR;
```

```
/* if attribute is an object type, recursively describe it */
if (typecode == OCI_TYPECODE_OBJECT)
{
  OCIRef *attr_type_ref;
  OCIDescribe *nested_dschp;

  /* allocate describe handle */
  if (OCIHandleAlloc((dvoid *)envh,(dvoid**)&nested_dschp,
    (ub4)OCI_HTYPE_DESCRIBE,(size_t)0, (dvoid **)0))
    return OCI_ERROR;

  if (OCIAttrGet((dvoid *)attrhd, OCI_DTYPE_PARAM,
          (dvoid *)&attr_type_ref, (ub4 *)0, OCI_ATTR_REF_TDO,errh))
    return OCI_ERROR;

   OCIDescribeAny(svch, errh,(dvoid*)attr_type_ref, 0,
         OCI_OTYPE_REF, 0, OCI_PTYPE_TYPE, nested_dschp);
    /* go on describing the attribute type... */
}
}

if (dschp)
    OCIHandleFree((dvoid *) dschp, OCI_HTYPE_DESCRIBE);
...
```

Retrieving the Collection Element's Datatype of a Named Collection Type

This example illustrates the use of an explicit describe on a named collection type:

```
text type_name[] = "phone_list_typ";
ub4 type_name_len = (ub4) strlen((char *)type_name);
OCIRef *type_ref = (OCIRef *) 0;
ub2 describe_by_name = 1;
ub4 num_elements = 0;
OCITypeCode typecode = 0, collection_typecode = 0, element_typecode = 0;
dvoid *collection_element_parmh = (dvoid *) 0;
OCIDescribe *dschp = (OCIDescribe *) 0;      /* describe handle */
OCIParam *parmh = (OCIParam *) 0;         /* parameter handle */

/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
   return OCI_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 OCI_ERROR;
}
else {
  /* get ref to type using OCIAttrGet */

  /* get the describe handle for the type */
  if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF,
        0, OCI_PTYPE_TYPE, dschp))
       return OCI_ERROR;
}
```

```
/* get the parameter handle */
if (OCIAttrGet((dvoid *)dschp, OCI_HTYPE_DESCRIBE, (dvoid *)&parmh, (ub4 *)0,
        OCI_ATTR_PARAM, errh))
    return OCI_ERROR;

/* get the Oracle Type System type code of the type to determine that this is a
collection type */
typecode = 0;
if (OCIAttrGet((dvoid *)parmh, OCI_DTYPE_PARAM,(dvoid *)&typecode, (ub4 *)0,
      OCI_ATTR_TYPECODE, errh))
    return OCI_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 */
  collection_typecode = 0;
  if (OCIAttrGet((dvoid *)parmh, OCI_DTYPE_PARAM, (dvoid *)&collection_typecode,
 (ub4 *)0,
       OCI_ATTR_COLLECTION_TYPECODE, errh))
    return OCI_ERROR;

  /* get the collection element; you MUST use this to further retrieve information
     about the collection's element */
  if (OCIAttrGet((dvoid *)parmh, OCI_DTYPE_PARAM, &collection_element_parmh,
                 (ub4 *)0,
        OCI_ATTR_COLLECTION_ELEMENT, errh))
    return OCI_ERROR;
  /* get the number of elements if collection is a VARRAY; not valid for nested
     tables */
  if (collection_typecode == OCI_TYPECODE_VARRAY) {
    if (OCIAttrGet((dvoid *)collection_element_parmh, OCI_DTYPE_PARAM,
          (dvoid *)&num_elements, (ub4 *)0, OCI_ATTR_NUM_ELEMS, errh))
      return OCI_ERROR;
  }
  /* now use the collection_element parameter handle to retrieve information about
     the collection element */
  element_typecode = 0;
  if (OCIAttrGet((dvoid *)collection_element_parmh, OCI_DTYPE_PARAM,
        (dvoid *)&element_typecode, (ub4 *)0, OCI_ATTR_TYPECODE, errh))
    return OCI_ERROR;

  /* do the same to describe additional collection element information; this is
  very similar to describing type attributes */
}

if (dschp)
    OCIHandleFree((dvoid *) dschp, OCI_HTYPE_DESCRIBE);
...
```

Describing with Character Length Semantics

The following sample code shows a loop that retrieves the column names and datatypes corresponding to a query following query execution. The query was associated with the statement handle by a prior call to `OCIStmtPrepare()`.

```
...
OCIParam     *mypard = (OCIParam *) 0;
ub2          dtype;
```

```
text         *col_name;
ub4          counter, col_name_len, char_semantics;
ub2          col_width;
sb4          parm_status;

text *sqlstmt = (text *)"SELECT * FROM employees WHERE employee_id = 100";

checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                    (ub4)strlen((char *)sqlstmt),
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, 0, 0, (OCISnapshot *)0,
(OCISnapshot *)0, OCI_DEFAULT));

/* Request a parameter descriptor for position 1 in the select-list */
counter = 1;
parm_status = OCIParamGet((dvoid *)stmthp, OCI_HTYPE_STMT, errhp,
               (dvoid **)&mypard, (ub4) counter);
/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */
while (parm_status == OCI_SUCCESS) {
   /* Retrieve the datatype 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 */
   col_name_len = 0;
   checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
           (dvoid**) &col_name, (ub4 *) &col_name_len, (ub4) OCI_ATTR_NAME,
           (OCIError *) errhp ));
   /* Retrieve the length semantics for the column */
   char_semantics = 0;
   checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
           (dvoid*) &char_semantics,(ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
           (OCIError *) errhp  ));
   col_width = 0;
   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  ));
   /* increment counter and get next descriptor, if there is one */
   counter++;
   parm_status = OCIParamGet((dvoid *)stmthp, OCI_HTYPE_STMT, errhp,
          (dvoid **)&mypard, (ub4) counter);
} /* while */
...
```

7

LOB and BFILE Operations

This chapter contains these topics:

Using OCI Functions for LOBs

The OCI includes a set of functions for performing operations on large objects (LOBs) in a database. Persistent 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. BFILEs 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.

`BFILEs` are read-only. Oracle supports only binary `BFILEs`.

See Also:

- Appendix B, "OCI Demonstration Programs" for code samples showing the use of LOBs.
- `$ORACLE_HOME/rdbms/demo/lobs/oci/` for specific LOB code samples.
- *Oracle Database PL/SQL Packages and Types Reference* for the `DBMS_LOB` package
- *Oracle Database Application Developer's Guide - Large Objects.*

Creating and Modifying Persistent LOBs

LOB instances can be either persistent (stored in the database) or temporary (existing only in the scope of your application). Do not confuse the concept of a persistent LOB with a persistent object.

There are two ways of creating and modifying persistent LOBs:

1. **Using the data interface**

   You can create a LOB by inserting character data into a `CLOB` column or `RAW` data into a BLOB column directly. You can also modify LOBs by using a SQL `UPDATE` statement, binding character data into a `CLOB` column or `RAW` data into a `BLOB` column.

   Insert, update, and select of remote LOBs (over a dblink) is supported as long as neither the remote server or the local server is of a release less than Oracle Database 10g Release 2. The data interface only supports data size up to 2 GB -1, the maximum size of an `sb4`.

   > **See Also:** *Oracle Database Application Developer's Guide - Large Objects* chapter "Data Interface for Persistent LOBs", for more information and examples.

2. **Using the LOB locator**

   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 write to it using one of the OCI LOB functions.

   Note: 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.

   > **See Also:** "Binding LOB Data" on page 5-8 for usage and examples for both `INSERT` and `UPDATE`

For any LOB write command to be successful, a transaction must be open. If you commit a transaction before writing the data, you must lock the row again (by reissuing the `SELECT...FOR UPDATE`, for example), because the commit closes the transaction.

Associating a BFILE in a Table with an Operating System File

The `BFILENAME()` function can be used in an `INSERT` statement to associate an external server-side (operating system) file with a `BFILE` column or attribute in a table. Using `BFILENAME()` in an `UPDATE` statement associates the `BFILE` column or attribute with a different operating system file. `OCILobFileSetName()` can also be used to associate a `BFILE` in a table with an operating system file. `BFILENAME()` is usually used in an `INSERT` or `UPDATE` without bind variables and `OCILobFileSetName()` is used for bind variables.

See Also:

- "OCILobFileSetName()" on page 16-56
- *Oracle Database Application Developer's Guide - Large Objects* for more information about the `BFILENAME()` function

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 when using a **LOB locator**:

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. Re-pin 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: Chapter 10, "OCI Object-Relational Programming" and the chapters that follow it, for more information about objects

There is a second way of writing to a LOB attribute: when using the **data interface**, you can bind or define character data for a `CLOB` attribute or `RAW` data for a BLOB attribute.

See Also:

- "Binding LOB Data" on page 5-8 for usage and examples for both `INSERT` and `UPDATE statements`
- "Defining LOB Data" on page 5-15 for usage and examples of `SELECT` statements

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, such as read or write, on the LOB attribute because transient objects with LOB attributes are not 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 `BFILE` attribute and use the `BFILE` attribute to read data from a file stored in the server's file system. The application can also call `OCIObjectNew()` to create a transient `BFILE`.

Array Interface for LOBs

You can use the OCI array interface with LOBs, just as with any other datatype. There are two ways of using the array interface.

1. **Using the data interface**

   You can bind or define arrays of character data for a `CLOB` column or `RAW` data for a `BLOB` column. You can use array bind and define interfaces to insert and select multiple rows with LOBs in *one round trip* to the server.

   See Also:

   - "Binding LOB Data" on page 5-8 for usage and examples for both `INSERT` and `UPDATE statements`
   - "Defining LOB Data" on page 5-15 for usage and examples of `SELECT` statements

2. **Using the LOB locator**

   When using the LOB locator you must allocate the descriptors:

```
/* First create an array of OCILobLocator pointers: */
OCILobLocator *lobp[10];

for (i=0; i < 10; i++)
{ OCIDescriptorAlloc (...,&lobp[i],...);

/* Then bind the descriptor as follows */
  OCIBindByPos(... &lobp[i], ...);
}
```

Using LOBs of Size Greater than 4 GB

Starting with Oracle Database 10g Release 1 of OCI, functions were introduced to support LOBs of size greater than 4 GB. These new functions can also be used in new applications for LOBs of less than 4 GB.

Oracle Database lets you create tablespaces with blocksizes different from the database block size, and the maximum size of a LOB depends on the size of the tablespace blocks. `CHUNK` is a parameter of LOB storage whose value is controlled by the block size of the tablespace in which the LOB is stored. When you create a LOB column, you can specify a value for `CHUNK`, which is the number of bytes to be allocated for LOB manipulation. The value must be a multiple of the tablespace block size, or Oracle Database rounds up to the next multiple. (If the tablespace block size is the same as the database block size, then `CHUNK` is also a multiple of the database block size.) The default `CHUNK` size is one tablespace block, and the maximum value is 32K.

For example, suppose your database block size is 32K and you create a tablespace with a nonstandard block size of 8K. Further suppose that you create a table with a LOB column and specify a `CHUNK` size of 16K (which is a multiple of the 8K tablespace block size). Then the maximum size of a LOB in this column is (4 gigabytes - 1) * 16K.

In this guide, 4 GB is defined as 4 gigabytes -1, or 4,294,967,295 bytes. The maximum size of a LOB, persistent or temporary, is (4 gigabytes - 1) * (`CHUNK`). The maximum LOB size can range from 8 terabytes to 128 terabytes.

The maximum size of a `BFILE` is the maximum file size allowed in the operating system, or `UB8MAXVAL`, whichever is smaller.

Because the older LOB functions use ub4 as the datatypes of some parameters and ub4 datatype can only hold up to 4 GB. The newer functions use parameters of 8-byte length, oraub8, which is a datatype defined in oratypes.h. The datatypes oraub8 and orasb8 are mapped to appropriate 64-bit native datatypes depending on the compiler and operating system. Macros are used to not define oraub8 and orasb8 if compiling in 32-bit mode with strict ANSI option.

OCILobGetChunkSize() returns the value, in bytes for BLOBs, or in characters for CLOBs, to be used in reading and writing LOBs. For varying-width character sets, the value is the number of Unicode characters that fit. The number of bytes stored in a chunk is actually less than the size of the CHUNK parameter due to internal storage overhead. The function, OCILobGetStorageLimit(), is provided to return the maximum size in bytes of internal LOBs in the current installation.

Note: Oracle Database does not support BFILEs larger than 4 gigabytes in any programmatic environment. An additional file size limit imposed by your operating system also applies to BFILEs.

New Functions for the Increased LOB Sizes

Eight functions with names that end in "2" and use the datatype oraub8 in place of the datatype ub4 were introduced in Oracle Database 10*g* Release 1. Other changes are made in the read and write functions (OCILobRead2(), OCILobWrite2(), and OCILobWriteAppend2()) to solve several problems:

Problem: Before Oracle Database 10*g* Release 1, the parameter amtp assumes either byte or char length for LOBs based on the locator type and character set. It is complicated and users did not have the flexibility to use byte length or char length according to their requirements.

Solution: Read/Write calls should take both byte_amtp and char_amtp as replacement for amtp parameter. char_amtp takes preference for CLOB and NCLOB and byte_amtp is only considered as input if char_amtp is zero. On output for CLOB and NCLOB, both byte_amtp and char_amtp are filled. For BLOB and BFILE, char_amtp parameter is ignored for both input and output.

Problem: For OCILobRead2(), there is no flag to indicate polling mode. There is no easy way for the users to say "I have a 100 byte buffer. Fill it as much as you can". Previously, they had to estimate how many characters to specify for the amount. If they guessed too much, they were forced into polling mode unintentionally. The user code thus can get trapped in the polling mode and subsequent OCI calls are all blocked.

Solution: This call should take piece as an input parameter and if OCI_ONE_PIECE is passed, it should fill the buffer as much as possible and come out even if the amount indicated by byte_amtp or char_amtp is more than the buffer length. The value of bufl is used to specify the maximum amount of bytes to read.

Problem: After calling for a LOB write in polling mode, users do not know how many chars or bytes are actually fetched till the end of the polling.

Solution: Both byte_amtp and char_amtp need to be updated after each call in polling mode.

Problem: While reading or writing data in streaming mode with callback, users have to use the same buffer for each piece of data.

Solution: The callback function needs to have two new parameters to provide a new buffer and the buffer length. Callback functions can set the new buffer parameter to

NULL to follow old behavior: to use the default buffer passed in the first call for all the pieces.

See Also:

- "LOB Functions" on page 16-19
- "OCILobRead2()" on page 16-78
- "OCILobWrite2()" on page 16-88
- "OCILobWriteAppend2()" on page 16-95

Compatibility and Migration

Existing OCI programs can be enhanced to process larger amounts of LOB data (greater than 4GB). Table 7–1 summarizes compatibility issues ("old" refers to releases before Oracle Database 10*g* Release 1):

Table 7–1 LOB Functions Compatibility and Migration

LOB Function	Old Client/New or Old Server	New Client/Old Server	New Client/New Server
OCILobArrayRead()	NA	OK until piece size and offset are < 4GB.	OK
OCILobArrayWrite()	NA	OK until piece size and offset are < 4GB.	OK
OCILobCopy2()	NA	OK until LOB size, piece size (amount) and offset are < 4GB.	OK
OCILobCopy()	OK; limit is 4GB.	OK	OK; limit is 4GB.
OCILobErase2()	NA	OK until piece size and offset are < 4GB.	OK
OCILobErase()	OK; limit is 4GB.	OK	OK; limit is 4GB.
OCILobGetLength2()	NA	OK	OK
OCILobGetLength()	OK; limit is 4GB.	OK	OK; OCI_ERROR if LOB size > 4GB.
OCILobLoadFromFile2()	NA	OK until LOB size, piece size (amount) and offset are < 4GB.	OK
OCILobLoadFromFile()	OK; limit is 4GB.	OK	OK; limit is 4GB.
OCILobRead2()	NA	OK until LOB size, piece size (amount) and offset are < 4GB.	OK

Table 7–1 (Cont.) LOB Functions Compatibility and Migration

LOB Function	Old Client/New or Old Server	New Client/Old Server	New Client/New Server
`OCILobRead()`	OK; limit 4GB. With new server: `OCI_ERROR` will be returned if you try to read any amount >= 4GB from any offset < 4GB. This is because when you read any amount >= 4GB, that will result in overflow of returned value in *amtp, and so will be flagged as an error. Note: 1) If you read up to 4GB-1 from offset, that will not be flagged as an error. 2) When you use streaming mode with polling, no error will be returned if no attempt is made to use piece size > 4GB (you can read data > 4GB in this case).	OK	OK. `OCI_ERROR` will be returned if you try to read any amount >= 4GB from any offset < 4GB. This is because when you read any amount >= 4GB, that will result in overflow of returned value in *amtp, and so will be flagged as an error. Note: 1) If you read up to 4GB-1 from offset, that will not be flagged an as error. 2) When the you use streaming mode with polling, no error will be returned if no attempt is made to use piece size > 4GB.
`OCIlobTrim2()`	NA	OK	OK
`OCIlobTrim()`	OK; limit 4GB.	OK	OK; limit 4GB.
`OCILobWrite2()`	NA	OK until LOB size, piece size (amount) and offset are < 4GB.	OK
`OCILobWrite()`	OK; limit 4GB. With new server: `OCI_ERROR` will be returned if you write any amount >= 4GB (from any offset <4GB) because that will result in overflow of returned value in `*amtp`. Note: Updating a LOB of 10GB from any offset up to 4GB-1 by up to 4GB-1 amount of data will not be flagged as an error.	OK	OK. `OCI_ERROR` will be returned if you write any amount > =4GB (from any offset < 4GB) because that will result in overflow of returned value in *amtp. Note: Updating a LOB of 10GB from any offset up to 4GB-1 by up to 4GB-1 amount of data will not be flagged as an error.
`OCILobWriteAppend2()`	NA	OK until LOB size and piece size are <4GB.	OK

Table 7–1 (Cont.) LOB Functions Compatibility and Migration

LOB Function	Old Client/New or Old Server	New Client/Old Server	New Client/New Server
OCILobWriteAppend()	OK; limit 4GB. With new server: OCI_ERROR will be returned if you append any amount >= 4GB of data because that will result in overflow of returned value in *amtp.	OK	OK; limit 4GB. OCI_ERROR will be returned if you append any amount >= 4GB of data because that will result in overflow of returned value in *amtp.
OCILobGetStorageLimit()	NA	Error	OK

Use the new functions when using the new server and new client. Mixing old and new functions can result in unexpected situations such as data written using OCILobWrite2() being greater than 4GB if the application tries to read it with OCILobRead() and gets only partial data (if a callback function is not used). In most cases the application will get an error message when the size crosses 4GB and older functions are used. However, there will be no issue if you use those older functions for LOBs of size smaller than 4GB.

LOB and BFILE Functions in OCI

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 CLOBs and NCLOBs, 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 of 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 CLOBs or NCLOBs and in bytes for BLOBs or BFILEs.

Exceptions to these general rules are noted in the description of the specific LOB call.

See Also:

- "LOB Functions" on page 16-19
- "Buffer Expansion During OCI Binding" on page 5-25

Improving LOB Read/Write Performance

Here are some hints to improve performance.

Using Data Interface For LOBs

You can bind or define character data for a CLOB column or RAW data for a BLOB column. This requires only one round trip for inserting or selecting a LOB, as opposed to the traditional LOB interface which requires multiple round trips.

See Also:

- "Binding LOB Data" on page 5-8 for usage and examples for both `INSERT` and `UPDATE statements`
- "Defining LOB Data" on page 5-15 for usage and examples of `SELECT` statements

Using OCILobGetChunkSize()

You can use 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 on 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.

Calling `OCILobGetChunkSize()` returns the usable chunk size of the LOB, and an application can batch a series of write operations for the entire chunk, rather than issuing multiple LOB write calls for the same chunk.

To read through the end of a LOB, call `OCILobRead2()` with an amount of 4 gigabytes. This avoids the round trip involved with first calling `OCILobGetLength()`.

Note: For LOBs which store varying width characters, `OCILobGetChunkSize()` returns the number of Unicode characters that fit in a LOB chunk.

Using OCILobWriteAppend2()

OCI provides a shortcut for more efficient writing of data to the end of a LOB. The `OCILobWriteAppend2()` call appends data to the end of a LOB without first requiring a call to `OCILobGetLength()` to determine the starting point for an `OCILobWrite()` operation. `OCILobWriteAppend2()` does both steps.

Using OCILobArrayRead() and OCILobArrayWrite()

Performance improvement is obtained by using these Oracle Database 10g Release 2 functions found in the sections of the LOBs documentation that describe how to read LOB data for multiple locators, using `OCILobArrayRead()`, and how to write LOB data for multiple LOB locators, using `OCILobArrayWrite()`. Code examples are also provided for using them with callback functions and in piecewise mode.

See Also: Oracle Database Application Developer's Guide - Large Objects, sections "LOB Array Read" and "LOB Array Write"

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 in client-side

buffers. This reduces the number of network round trips and LOB versions, thereby improving LOB performance significantly.

See Also:

- *Oracle Database Application Developer's Guide - Large Objects*. For more information on LOB buffering, refer to the chapter on LOBs.
- "LOB Function Round Trips" on page C-3 for a list of the server round trips required for each function

Functions for Opening and Closing LOBs

The OCI provides functions to explicitly open a LOB, `OCILobOpen()`, to close a LOB, `OCILobClose()`, and to test whether a LOB is open, `OCILobIsOpen()`. These functions mark the beginning and end of a series of LOB operations so that specific processing such as updating indexes can be performed when a LOB is closed.

For internal LOBs, the concept of openness is associated with a LOB and not its locator. The locator does not store any information about the state of the LOB. It is possible for more than one locator to point to the same open LOB. However, for `BFILE`s, being open 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.

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 LOBs and `BFILE`s.

An error is returned when you 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 the 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, rebuild your functional and domain indexes on the LOB column.

Restrictions on Opening and Closing LOBs

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 since 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 Oracle

Database Reference. 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

See Also:

- Appendix B, "OCI Demonstration Programs" for examples of the use of the `OCILobOpen()` and `OCILobCLose()` calls in the online demonstration programs
- Table C–2, " Server Round Trips for OCILob Calls"

LOB Read and Write Callbacks

OCI supports read and write callback functions. The following sections describe the use of callbacks in more detail.

The Callback Interface for Streaming

User-defined read and write callback functions for inserting or retrieving data provide an alternative to the polling methods for streaming LOB. These functions are implemented by you and registered with OCI through the `OCILobRead2()`, `OCILobWriteAppend2()`, and `OCILobWrite2()` calls. These callback functions are called by OCI whenever required.

Reading LOBs using Callbacks

The user-defined read callback function is registered through the `OCILobRead2()` function. The callback function should have the following prototype:

```
CallbackFunctionName ( dvoid *ctxp, CONST dvoid *bufp, oraub8 len, ub1 piece,
                       dvoid **changed_bufpp, oraub8 *changed_lenp);
```

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.

If the buffer length provided in the original `OCILobRead2()` 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 whether the information returned in the buffer is the first, next or last piece.

The parameters changed_bufpp and changed_lenp can be used inside the callback function to change the buffer dynamically. changed_bufpp should point to the address of the changed buffer and changed_lenp should point to the length of the changed buffer. changed_bufpp and changed_lenp need not be used inside the callback function if the application does not change the buffer dynamically.

The following code fragment implements read callback functions using OCILobRead2(). Assume that lobl is a valid locator that has been previously selected, svchp is a valid service handle and errhp is a valid error handle In the example. The user-defined function cbk_read_lob() is repeatedly called until all the LOB data has been read.

```
...
oraub8   offset = 1;
oraub8   loblen = 0;
oraub8   byte_amt = 0;
oraub8   char_amt = 0
ub1      bufp[MAXBUFLEN];

sword retval;
byte_amtp = 4294967297;     /* 4 gigabytes plus 1 */

if (retval = OCILobRead2(svchp, errhp, lobl, &byte_amt, &char_amt, offset,
  (dvoid *) bufp, (oraub8) MAXBUFLEN, (dvoid *) 0, OCI_FIRST_PIECE,
  cbk_read_lob, (ub2) 0, (ub1) SQLCS_IMPLICIT))
{
     (void) printf("ERROR: OCILobRead2() LOB.\n");
     report_error();
}
...
sb4 cbk_read_lob(ctxp, bufxp, len, piece, changed_bufpp, changed_lenp)
dvoid       *ctxp;
CONST dvoid *bufxp;
oraub8      len;
ub1         piece;
dvoid       **changed_bufpp;
oraub8      *changed_lenp;
{
    static ub4 piece_count = 0;
    piece_count++;

    switch (piece)
   {
      case OCI_LAST_PIECE:     /*--- 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:   /*--- buffer processing code goes here ---*/
          (void) printf("callback read the %d th piece\n", piece_count);
          /* --Optional code to set changed_bufpp and changed_lenp if the
              buffer needs to be changed dynamically --*/
          break;
      case OCI_NEXT_PIECE:   /*--- buffer processing code goes here ---*/
          (void) printf("callback read the %d th piece\n", piece_count);
         /* --Optional code to set changed_bufpp and changed_lenp if the
             buffer needs to be changed dynamically --*/
          break;
      default:
          (void) printf("callback read error: unkown piece = %d.\n", piece);
           return OCI_ERROR;
```

```
     }
    return OCI_CONTINUE;
}
```

Writing LOBs using Callbacks

Similar to read callbacks, the user-defined write callback function is registered through the `OCILobWrite2()` function. The callback function should have the following prototype:

```
CallbackFunctionName ( dvoid *ctxp, dvoid *bufp, oraub8 *lenp, ub1 *piecep,
                       dvoid **changed_bufpp, oraub8 *changed_lenp);
```

The first parameter, `ctxp`, is the context of the callback that is passed to OCI in the `OCILobWrite2()` 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 `OCILobWrite2()` any data remaining is inserted by the user-defined callback. In the callback provide the data to insert in the storage indicated by `bufp` and also specify the length in `buflp`. You also indicate whether it is the next (`OCI_NEXT_PIECE`) or the last (`OCI_LAST_PIECE`) piece using the `piecep` parameter. 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 parameters `changed_bufpp` and `changed_lenp` can be used inside the callback function to change the buffer dynamically. `changed_bufpp` should point to the address of the changed buffer and `changed_lenp` should point to the length of the changed buffer. `changed_bufpp` and `changed_lenp` need not be used inside the callback function if the application does not change the buffer dynamically.

The following code fragment implements write callback functions using `OCILobWrite2()`. Assume 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. The user-defined function `cbk_write_lob()` is repeatedly called until the `piecep` parameter indicates that the application is providing the last piece.

```
...

ub1      bufp[MAXBUFLEN];
oraub8   byte_amt = MAXBUFLEN * 20;
oraub8   char_amt = 0;
oraub8   offset = 1;
oraub8   nbytes = MAXBUFLEN;

/*-- code to fill bufp with data goes here. nbytes should reflect the size and
 should be less than or equal to MAXBUFLEN --*/
if (retval = OCILobWrite2(svchp, errhp, lobl, &byte_amt, &char_amt, offset,
    (dvoid*)bufp, (ub4)nbytes, OCI_FIRST_PIECE, (dvoid *)0, cbk_write_lob,
    (ub2) 0, (ub1) SQLCS_IMPLICIT))
{
   (void) printf("ERROR: OCILobWrite2().\n");
   report_error();
   return;
}
...
sb4 cbk_write_lob(ctxp, bufxp, lenp,  piecep, changed_bufpp, changed_lenp)
dvoid    *ctxp;
```

```
dvoid    *bufxp;
oraub8  *lenp;
ub1       *piecep;
dvoid  **changed_bufpp;
oraub8  *changed_lenp;
{
    /*-- code to fill bufxp with data goes here. *lenp should reflect the
        size and should be less than or equal to MAXBUFLEN -- */
    /* --Optional code to set changed_bufpp and changed_lenp if the
        buffer needs to be changed dynamically --*/
    if (this is the last data buffer)
        *piecep = OCI_LAST_PIECE;
    else
        *piecep = OCI_NEXT_PIECE;
    return OCI_CONTINUE;
}
```

Temporary LOB Support

The OCI provides functions for creating and freeing temporary LOBs, `OCILobCreateTemporary()` and `OCILobFreeTemporary()`, and a function for determining whether a LOB is temporary, `OCILobIsTemporary()`.

Temporary LOBs are not permanently stored in the database, but 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 persistent 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` - The 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 nullity 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. All temporary LOBs are deleted at the end of the duration. 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, use `OCILobCopy()` to copy the data from the temporary LOB into a permanent one. You can also use the temporary LOB in the `VALUES` clause of an `INSERT` statement, as the source of the assignment in an `UPDATE` statement, or assign it to a persistent LOB attribute and then flush the object. Temporary LOBs can be modified using 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 and their associated attributes 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 `OCIDurationBegin()` and the end of the duration using `OCIDurationEnd()`.

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. An application-specific duration can be as long as a session duration.

At the end of a duration, all temporary LOBs associated with that duration are freed. 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 that 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.

Freeing Temporary LOBs

Any time that your OCI program obtains a LOB locator from SQL or PL/SQL, check that the locator is temporary. If it is, free the locator when your application is finished with it. The locator can be from a define during a select or an out bind. A temporary LOB duration is always upgraded to session when it is shipped to the client side. The application needs to do the following before the locator is overwritten by the locator of the next row:

```
OCILobIsTemporary(env, err, locator, is_temporary);
if(is_temporary)
    OCILobFreeTemporary(svc, err, locator);
```

See Also:

- "OCILobIsTemporary()" on page 16-65
- "OCILobFreeTemporary()" on page 16-58

Take Care When Assigning Pointers

Special care needs to be taken when assigning `OCILobLocator` pointers. Pointer assignments create a shallow copy of the LOB. After the pointer assignment, source and target LOBs point to the same copy of data. This behavior is different from using LOB APIs, such as `OCILobAssign()` or `OCILobLocatorAssign()` to perform assignments.

When the APIs are used, the locators logically point to independent copies of data after assignment.

For temporary LOBs, before pointer assignments, it is your responsibility to make sure any temporary LOB in the target LOB locator is freed by `OCIFreeTemporary()`. When `OCILobLocatorAssign()` is used, the original temporary LOB in the target LOB locator variable, if any, is freed before the assignment happens.

Before an out-bind variable is reused in executing a SQL statement, it is your responsibility to free any temporary LOB in the existing out-bind LOB locator buffer by using the `OCIFreeTemporary()` call.

See Also:

- *Oracle Database Application Developer's Guide - Large Objects*, "Temporary LOB Performance Guidelines" section.
- *Oracle Database Application Developer's Guide - Large Objects* , for a discussion on optimal performance of temporary LOBS.

Temporary LOB Example

The following code example shows how temporary LOBs can 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 print_media 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 print_media WHERE product_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 */
  OCIEnvCreate(&envhp, OCI_DEFAULT, (dvoid *)0, 0, 0, 0,
        (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

This chapter contains these topics:

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 later perform transaction operations. Each call takes a service context handle that must 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: "Transaction Functions" on page 16-158

Levels of Transactional Complexity

The OCI supports several levels of transaction complexity.

- 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: "OCITransCommit()" on page 16-159 for sample code showing the use of simple local transactions. See new flags introduced in Oracle Database 10*g* Release 2: `OCI_TRANS_WRITEBATCH` and `OCI_TRANS_WRITENOWAIT`

Serializable or Read-Only Local Transactions

Applications requiring serializable or read-only transactions require an additional OCI `OCITransStart()` call to start the transaction.

The `OCITransStart()` call must 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.

Transaction Identifiers

Three-tier applications such as transaction processing (TP) monitors create and manage global transactions. They supply a *global transaction identifier* (`XID`), that 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:

Table 8–1 Global Transaction Identifier

Component	Value
gtrid	12
bqual	34
gtrid+bqual=XID	1234

> **See Also:** *Oracle Database Heterogeneous Connectivity Administrator's Guide* for more information about transaction identifiers

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.

Attribute OCI_ATTR_TRANS_NAME

When setting this attribute in a transaction handle, the length of the name can be at most 64 bytes. The formatid of the XID is 0 and the branch qualifier is 0.

When retrieving this attribute from a transaction handle, the returned transaction name is the global transaction identifier. The size is the length of the global transaction identifier.

Transaction Branches

Within a single global transaction, Oracle supports both tightly coupled and loosely coupled relationships between a pair of branches.

- Tightly coupled branches 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 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().

Figure 8–1 illustrates tightly coupled branches within an application. S1 and S2, are sessions, B1 and B2 are branches, and T is a transaction. The XIDs of the two branches share the same gtrid, because they are operating on the same transaction, but they have a different bqual, because they are on separate branches

Figure 8–1 Multiple Tightly Coupled Branches

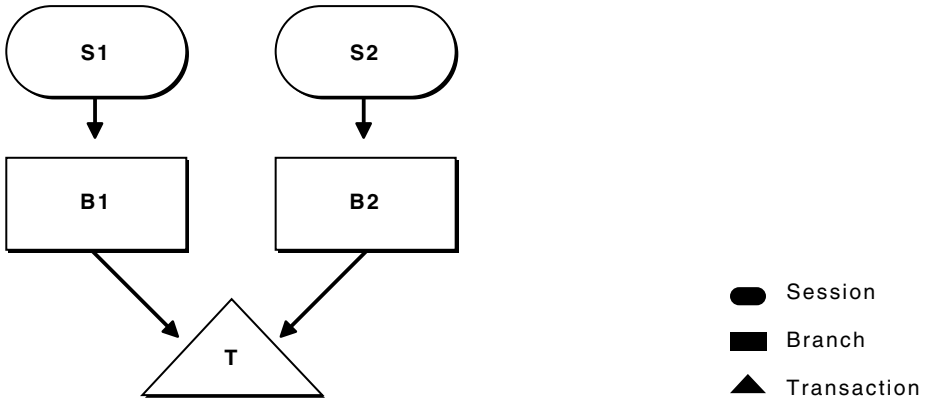
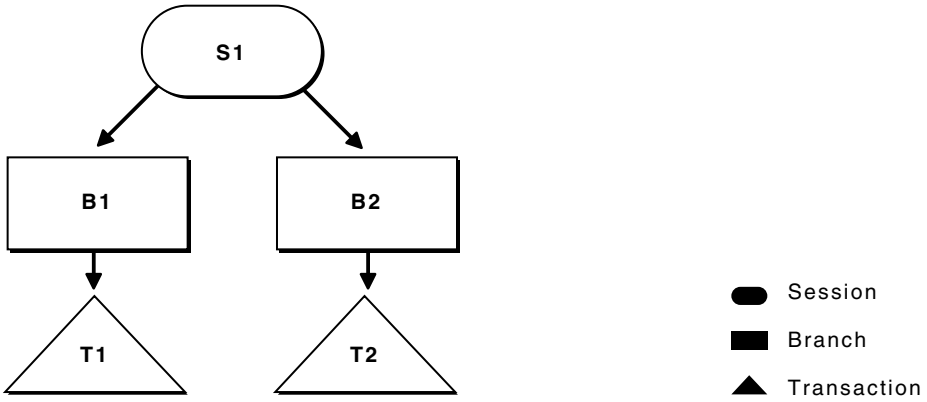


Figure 8–2 illustrates how a single session operates on different branches. The *gtrid* component of the XIDs are different, because they represent separate global transactions

Figure 8–2 Session Operating on Multiple Branches



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:** "OCITransStart()" on page 16-167 for sample code demonstrating this scenario

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 uses 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. 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. These attributes set the client database name recorded when performing global transactions. The name can be used by the database administrator to track transactions that may be pending in a prepared state because of failures.

Caution: An OCI application sets 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 OCITransPrepare() call against each transaction. The transactions return a value indicating whether it is able to commit its current work (OCI_SUCCESS) or not (OCI_ERROR).
2. Commit - If each OCITransPrepare() call returns a value of OCI_SUCCESS, the application can issue a OCITransCommit() call to each transaction. The flags parameter of the commit call must be explicitly set to OCI_TRANS_TWOPHASE for the appropriate behavior, since the default for this call is for a one-phase commit.

Note: The OCITransPrepare() call can also return OCI_SUCCESS_WITH_INFO if a transaction needs to indicate that it is read-only. This means that a commit is neither appropriate nor necessary.

An additional call, OCITransForget(), indicates that a database "forgets" a 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 removes all information about the transaction.

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 becomes inefficient 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, that prepares 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 values passed in the `flags` parameter. The Result column describes the result of the call.

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 maximum 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 to cache connections after committing a transaction.Contrast this with the `OPEN_LINKS` parameter, that controls the number of connections from a session and is not applicable to applications that use global transactions.

Update Successfully, One-Phase Commit

Here is a list of the steps:

Table 8–2 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
3	`OCITransCommit()`	-	-	Commit succeeds

Start a Transaction, Detach, Resume, Prepare, Two-Phase Commit

Here is a list of the steps:

Table 8–3 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.

In step 4, the transaction can be resumed by a different process, as long as it had the same authorization.

Read-Only Update Fails

Here is a list of the steps:

Table 8–4 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

Here is a list of the steps:

Table 8–5 Read-Only Transaction

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

Password and Session Management

The OCI has the ability to authenticate and maintain multiple users.

OCI Authentication Management

The OCISessionBegin() call authenticates a user against the server set in the service context handle. It must be the first call for any given server handle. OCISessionBegin() authenticates the user for access to the Oracle server specified by the server handle and the service context of the call: after OCIServerAttach()

initializes a server handle, OCISessionBegin() must be called to authenticate the user for that server.

When OCISessionBegin() is called for the first time on a 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 be used only with the server handle used with the OCISessionBegin().
- If OCI_MIGRATE mode is specified, then the user authentication can be set with other server handles. However, the user session context may only be used with server handles that resolve to the same database instance. Security checking is performed during session switching.

A migratable session can 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 settings can only be used with a primary user session context.

A migratable session can be switched, or migrated, to a server handle within an environment represented by an environment handle. It can also migrate or be 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 that must not be modified by the caller.

   See Also: "User Session Handle Attributes" on page A-12

2. Transport this session id to another process.
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 fully-authenticated.

To provide credentials for a call to OCISessionBegin(), you must provide a valid user name and password pair for database authentication in the user session handle parameter. 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.

When the user session handle is terminated using OCISessionEnd(), the user name 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.

Or, you can supply 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.

OCI Password Management

The OCIPasswordChange() call enables an 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, and to change the password. 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.

Secure External Password Store

As an alternative for large-scale deployments where applications use password credentials to connect to databases, it is possible to store such credentials in a client-side Oracle wallet. An Oracle wallet is a secure software container that is used to store authentication and signing credentials.

Storing database password credentials in a client-side Oracle wallet eliminates the need to embed user names and passwords in application code, batch jobs, or scripts. This reduces the risk of exposing passwords in the clear in scripts and application code, and simplifies maintenance because you need not change your code each time user names and passwords change. In addition, not having to change application code also makes it easier to enforce password management policies for these user accounts.

When you configure a client to use the external password store, applications can use the following syntax to connect to databases that use password authentication:

```
CONNECT /@database_alias
```

Note that you need not specify database login credentials in this CONNECT statement. Instead your system looks for database login credentials in the client wallet.

> **See Also:** *Oracle Database Security Guide* for information about configuring your client to use secure external password store and for information about managing credentials in it.

OCI Session Management

Transaction servers that actively balance user load by multiplexing user sessions over a few server connections must group these connections into a server group. Oracle uses server groups to identify these connections so that sessions can be managed effectively and securely.

The attribute OCI_ATTR_SERVER_GROUP must be defined to specify the server group name using the OCIAttrSet() call:

```
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 alphanumeric 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 that uses that context. After the session is created successfully and the connection to the server is established, the server group name cannot be changed.

See Also: "Server Handle Attributes" on page A-10

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.

The use of server groups is optional. If no server group is specified, the server is created in a server group called DEFAULT.

The owner of the first non-migratable session created in a nondefault server group becomes the owner of the server group. All subsequent non-migratable sessions for any server in this server group must be created by the owner of the server group.

The server group feature is useful when dedicated servers are used. It has no effect on shared servers. All shared servers effectively belong to the server group DEFAULT.

Middle-Tier Applications in OCI

A middle-tier application receives requests from browser clients and determines database access and whether to generate an HTML page. Applications can have multiple *lightweight* 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-tier architecture is developed around a set of three trust zones.

The first is the client trust zone.Clients connecting to a Web application server are authenticated by the middle tier using any means: password, cryptographic token, or another. This method may be entirely different from the method used to establish the other trust zones.

The second trust zone is the application server. 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 by 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 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 is established and proxy capability verified in one round trip. If the application server is not allowed to act on behalf of the client or if the application server is not allowed to activate the specified roles, the OCISessionBegin() call will fail.

OCI Attributes for Middle-Tier Applications

The following attributes allow you to specify the external name and initial privileges of a client. These credentials are used by applications as alternative means of identifying or authenticating the client.

OCI_CRED_PROXY

Use OCI_CRED_PROXY as the value passed in the credt parameter of OCISessionBegin() when an application server starts a session on behalf of a client, rather than OCI_CRED_RDBMS (database user name 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 in client authentication. You can code the following declarations and OCIAttrSet() call:

```
OCISession *session_handle;
OCISvcCtx  *application_server_session_handle;
OCIError   *error_handle;
...
OCIAttrSet((dvoid *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (dvoid *)application_server_session_handle, (ub4) 0,
           OCI_ATTR_PROXY_CREDENTIALS, 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, instead of the database user name.

To pass the distinguished name of the client, the middle-tier server calls OCIAttrSet(), passing OCI_ATTR_DISTINGUISHED_NAME:

```
/* Declarations */
...
OCIAttrSet((dvoid *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (dvoid *)distinguished_name, (ub4) 0,
           OCI_ATTR_DISTINGUISHED_NAME, error_handle);
```

OCI_ATTR_CERTIFICATE

Certificate-based proxy authentication using OCI_ATTR_CERTIFICATE will not be supported in future Oracle Database releases. Use OCI_ATTR_DISTINGUISHED_NAME or OCI_ATTR_USERNAME attribute instead. This

method of authentication is similar to the use of distinguished name. The entire X.509 certificate is passed by the middle-tier server to the database.

To pass over the entire certificate, the middle tier calls OCIAttrSet(), passing OCI_ATTR_CERTIFICATE:

```
OCIAttrSet((dvoid *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (dvoid *)certificate, ub4 certificate_length,
           OCI_ATTR_CERTIFICATE, error_handle);
```

OCI_ATTR_INITIAL_CLIENT_ROLES

Use the OCI_ATTR_INITIAL_CLIENT_ROLES attribute to specify the roles the client is to possess when the application server connects to the Oracle server. 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((dvoid *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (dvoid *)role_array, (ub4) number_of_strings,
           OCI_ATTR_INITIAL_CLIENT_ROLES, error_handle);
```

OCI_ATTR_CLIENT_IDENTIFIER

Many middle tier applications connect to the database as an application, and rely on the middle-tier to track end user identity. To integrate tracking of these end users in various database components, the database client can set the client identifier (a predefined attribute from the application context namespace USERENV) in the session handle at any time. Use the OCI attribute OCI_ATTR_CLIENT_IDENTIFIER in the call to OCIAttrSet(). On the next request to the server, the information is propagated and stored in the server session.

To support the global application context, the client can set the CLIENT_IDENTIFIER (a predefined attribute from the application context namespace USERENV) in the session handle at any time. Use the OCI attribute OCI_ATTR_CLIENT_IDENTIFIER in the call to OCIAttrSet(). On the next request to the server, the information is propagated and stored in the server session.

```
OCIAttrSet((dvoid *)session_handle, (ub4) OCI_HTYPE_SESSION,
           (dvoid *)"janedoe", (ub4)strlen("janedoe"),
           OCI_ATTR_CLIENT_IDENTIFIER, error_handle);
```

When a client has multiple sessions, execute OCIAttrSet() for each session using the same client identifier. OCIAttrSet() must be executed automatically for each process when the session is reconnected in the event that sessions:

- are migrated across instances
- span instances using dblinks
- are pre-connected using TAF PRECONNECT
- use TAF BASIC

The client identifier is found in V$SESSION as a CLIENT_IDENTIFIER column or through the system context with this SQL statement:

```
SELECT sys_context('userenv', 'client_identifier') FROM dual;
```

> **See Also:** *Oracle Database Security Guide* the chapter on "Preserving User Identity in Multi-tiered Environments"

OCI_ATTR_PASSWORD

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. While it appears that this is the same as a client/server connection, the client does not have to have Oracle software installed on the client's system to be able to perform database operations. To use the password of the client, the application server supplies `OCIAttrSet()` with the authentication data, using the existing attribute `OCI_ATTR_PASSWORD`:

```
OCIAttrSet((dvoid *)session_handle, (ub4) OCI_HTYPE_SESSION, (dvoid *)password,
           (ub4)0, OCI_ATTR_PASSWORD, error_handle);
```

See Also: "User Session Handle Attributes" on page A-12

OCI Middle-Tier Example

Here is a middle-tier example:

```
...
*OCIEnv *environment_handle;
OCIServer *data_server_handle;
OCIError *error_handle;
OCISvcCtx *application_server_service_handle;
OraText *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] = (OraText *) "TELLER";
client_roles[1] = (OraText *) "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 does the
** following. It must 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 applies 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 can switch to
** the second client. That
** is 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
*/
...
```

End-to-End Application Tracing

Use the following attributes to measure server call time, not including server round trips. These attributes can also be set using the PL/SQL package `DBMS_APPLICATION_INFO` which incurs one round trip to the server. Using OCI to set the attributes does not incur a round trip.

OCI_ATTR_COLLECT_CALL_TIME

Set a `boolean` variable to `TRUE` or `FALSE`. Then, after you set this attribute by calling `OCIAttrSet()`, the server measures each call time. All server times between setting the variable to `TRUE` and setting it to `FALSE` are measured.

OCI_ATTR_CALL_TIME

The elapsed time, in milliseconds, of the last server call is returned in a `ub8` variable by calling `OCIAttrGet()` with this attribute. The following code snippet shows how to do this:

```
boolean enable_call_time;
ub8 call_time;
...
enable_call_time = TRUE;
OCIAttrSet(session, OCI_HTYPE_SESSION, (dvoid *)&enable_call_time,
           (ub4)0, OCI_ATTR_COLLECT_CALL_TIME,
           (OCIError *)error_handle);
OCIStmtExecute(...);
OCIAttrGet(session, OCI_HTYPE_SESSION, (dvoid *)&call_time,
           (ub4)0, OCI_ATTR_CALL_TIME,
           (OCIError *)error_handle);
...
```

Attributes for End-to-end Application Tracing

Set these attributes for tracing and debugging applications:

- OCI_ATTR_MODULE - name of the current module in the client application
- OCI_ATTR_ACTION -name of the current action within the current module. Set to NULL if you do not want to specify an action.
- OCI_ATTR_CLIENT_INFO - Client application additional information.

See Also: "User Session Handle Attributes" on page A-12

Externally Initialized Context in OCI

An externally initialized context is an application context where 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(). Issue subsequent 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 set default values and other session attributes through the OCISessionBegin() call, thus reducing server round trips.

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals,* the chapter on Establishing Security
- *Oracle Database SQL Reference,* the CREATE CONTEXT statement and the SYS_CONTEXT function

Externally Initialized Context Attributes in OCI

The client applications 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.

```
OCIAttrSet(session, (ub4) 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. (The parameter ctxl_desc must be of datatype OCIParam *).

```
OCIAttrGet(session, (ub4) OCI_HTYPE_SESSION,
           (dvoid *)&ctxl_desc, (ub4)0, OCI_ATTR_APPCTX_LIST, error_handle);
```

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);
```

Session Handle Attributes Used to Set an Externally Initialized Context

Set the appropriate values of the application context using these 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 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-12

Using OCISessionBegin() with an Externally initialized Context

When you call `OCISessionBegin()`, the context set in the session handle will be pushed to the server. No additional contexts are 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 OraText *username = (OraText *) "HR";
static OraText *password = (OraText *) "HR";

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;

  OCIEnvCreate(&envhp, OCI_DEFAULT, (dvoid *)0, 0, 0, 0,
                           (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, (OraText *)"", 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_EXT, (ub4) OCI_DEFAULT);

}
```

Client Application Context

Application context enables database clients (such as mid-tier applications) to set and send arbitrary session data to the server with each executed statement in only one round trip. The server stores this data in the session context before statement execution, from which it can be used to restrict queries or DML operations. All database features such as views, triggers, VPD policies, or PL/SQL stored procedures can use session data to constrain their operations.

A public writable namespace, `nm`, is created:

```
CREATE CONTEXT nm USING hr.package1;
```

In order to modify the data grouped in that namespace, users need to execute the designated PL/SQL package, `hr.package1`. However, no privilege is needed to query this information in a user session.

The variable length application context data that is stored in the user session is in the form of an attribute and value pair grouped under the context namespace.

For example, if a human resources application wants to store an end-user's responsibility information in the user session, then it could create an `nm` namespace and an attribute called 'responsibility' which can be assigned a value such as 'manager' or 'accountant'. This is referred to as the set operation in this document.

If the application decides to clear the value of the 'responsibility' attribute in the `nm` namespace, then it could set it to `NULL` or an empty string. This is referred to as the clear operation in this document.

If the application wants to clear all information in the `nm` namespace, then it could send the namespace information as a part of the clear-all operation to the server. This is referred to as the clear-all operation in a namespace in this document.

If there is no package-security defined for a namespace, then this namespace is deemed to be a client namespace, and any OCI client will be able to transport data for that namespace to the server. No privilege or package security check is done.

Network transport of application context data is done in a single round trip to the server.

Multiple SET Operations

Use OCIAppCtxSet() to perform a series of set operations on 'responsibility' attribute in the CLIENTCONTEXT namespace. When this information is sent to the server, the latest value will prevail for that particular attribute in a namespace. To change the value of the 'responsibility' attribute in the CLIENTCONTEXT namespace from 'manager' to 'vp', use the code snippet shown next, on the client side. When this information is transported to the server, the server will show the latest value 'vp' for the 'responsibility' attribute in the CLIENTCONTEXT namespace.

```
err = OCIAppCtxSet((void *) sesshndl,(dvoid *)"CLIENTCONTEXT",(ub4) 13,
                   (dvoid *)"responsibility", 14
                   (dvoid *)"manager", 7, errhp, OCI_DEFAULT);
err = OCIAppCtxSet((void *) sesshndl, (dvoid*)"CLIENTCONTEXT", 13,
                   (dvoid *)"responsibility", 14,(dvoid *)"vp",2, errhp,
                   OCI_DEFAULT);
```

You can clear specific attribute information in a client namespace. This can be done by setting the value of an attribute to NULL or an empty string, using the OCIAppCtxSet() function:

```
(void) OCIAppCtxSet((void *) sesshndl, (dvoid *)"CLIENTCONTEXT", 13,
                    (dvoid *)"responsibility", 14, (dvoid *)0, 0,errhp,
                    OCI_DEFAULT);
```

or:

```
(void) OCIAppCtxSet((void *) sesshndl, (dvoid *)"CLIENTCONTEXT", 13
                    (dvoid *)"responsibility", 14, (dvoid *)"", 0,errhp,
                     OCI_DEFAULT);
```

CLEAR-ALL Operations Between SET Operations

You can clear all the context information in a specific client namespace, using the OCIAppCtxClearAll() function and it will also be cleared on the server-side user session, during the next network transport.

If the client application performs a clear-all operation in a namespace after a couple of set operations, then values of all attributes in that namespace that were set prior to this clear-all operation are cleaned up on the client side and the server side. Only the subsequent set operations that were done after the clear-all operation, are reflected on the server side. On the client side:

```
err = OCIAppCtxSet((void *) sesshndl,(dvoid *)"CLIENTCONTEXT", 13,
                   (dvoid *)"responsibility", 14,
                   (dvoid *)"manager", 7,errhp, OCI_DEFAULT);
err = OCIAppCtxClearAll((void *) sesshndl, (dvoid *)"CLIENTCONTEXT", 13, errhp,
                        OCI_DEFAULT);
err = OCIAppCtxSet((void *) sesshndl, (dvoid*)"CLIENTCONTEXT",13
                   (dvoid *)"office",6, (dvoid *)"2op123", 5, errhp, OCI_DEFAULT);
```

The clear-all operation will clear up any information set by earlier operations in the namespace CLIENTCONTEXT: 'responsibility' = 'manager' is cleaned up. The information that was set subsequently will be reflected on the server side.

Network Transport and PL/SQL on Client Namespace

It is possible that an application could send application context information on a OCIStmtExecute() call to the server, and also attempt to change the same context information during that call by executing the DBMS_SESSION package.

In general, on the server side, the transported information is processed first and the main call is processed later. This behavior will apply to the application context network transports as well.

If they are both writing to the same client namespace and attribute set, then the main call's information will overwrite the information set by the fast network transport mechanism. Please note that in case of an error in the network transport call, the main call will not be executed.

Note that an error in the main call will not affect the processing of the network transport call. Once the network transport is processed then there is no way to undo it, based on errors in the main call. When the error is reported to the caller (by an OCI function), it is reported as a generic ORA error. Currently, there is no easy way to distinguish an error in the network transport from an error in the main call. The client should not make any assumptions that an error from main call will undo the round trip network processing and implement appropriate exception-handling mechanisms to prevent any inconsistencies.

See Also:

- "OCIAppCtxClearAll()" on page 15-4
- "OCIAppCtxSet()" on page 15-5

9

OCI Programming Advanced Topics

This chapter contains these topics:

Overview of OCI 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 stacks. Global and static variables are common to all threads, and a mutual exclusivity mechanism is required to manage access to these variables from multiple threads within an application.

Once spawned, threads run asynchronously with respect 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 synchronized mechanism is required to maintain the integrity of data being accessed.

The mechanism to manage data access takes the form of *mutexes* (mutual exclusivity locks), that is implemented to ensure that no conflicts arise between multiple threads accessing shared internal data that are opaque to users. In OCI, mutexes are granted for each environment handle.

The thread safety feature of the Oracle Database server and OCI libraries allows developers to use the OCI in a multithreaded environment. Thread safety ensures code can be reentrant, with multiple threads making OCI calls without side effects.

Note: Thread safety is not available on every operating system. Check your Oracle system-specific documentation for more information.

In a multithreaded UNIX environment, OCI calls are not allowed in a user signal handler, except for `OCIBreak()`.

Advantages of OCI Thread Safety

The implementation of thread safety in OCI has the following 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.

OCI 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, or 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 a database. The applications server, or agent, is very well suited to being a multithreaded application server, with each thread serving a single client application. In an Oracle environment this application server is an OCI or precompiler program.

Implementing Thread Safety

In order to take advantage of thread safety, an application must be running on a thread-safe operating system. The application specifies that it is running in a multithreaded environment by making an `OCIEnvNlsCreate()` call with `OCI_THREADED` as the value of the `mode` parameter.

All subsequent calls to `OCIEnvNlsCreate()` must also be made with `OCI_THREADED`.

Note: Applications running on non-thread-safe operating systems must not pass a value of `OCI_THREADED` to `OCIInitialize()` or `OCIEnvNlsCreate()`.

If an application is single-threaded, whether or not the operating system is thread-safe, the application must pass a value of `OCI_DEFAULT` to `OCIInitialize()` or `OCIEnvNlsCreate()`. Single-threaded applications that run in `OCI_THREADED` mode may incur lower performance.

If a multithreaded application is running on a thread-safe operating system, the OCI library will manage mutexes for the application for each environment handle. An application can override this feature and maintain its own mutex scheme by specifying a value of OCI_NO_MUTEX in the mode parameter of the OCIEnvCreate() call.

The following scenarios are possible, depending on how many connections exist in each environment handle, and how many threads are spawned in each connection.

1. If an application has multiple environment handles, with a single thread in each, mutexes are not required.
2. If an application running in OCI_THREADED mode maintains one or more environment handles, with multiple connections, it has these options:
   - Pass a value of OCI_NO_MUTEX for the mode of OCIEnvNlsCreate(). The application must mutex OCI calls made on the same environment handle. This has the advantage that the mutex scheme can be optimized to 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 for the mode of OCIEnvNlsCreate(). 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 can be blocked while the other finishes processing at the server.

Use one error handle for each thread in an application, since OCI errors can be over-written by other threads.

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.

The OCIThread Package

The OCIThread package provides a number of commonly used threading primitives. It offers a portable interface to threading capabilities native to various operating systems, but does not implement threading on operating systems that do not have native threading capability.

OCIThread does not provide a portable implementation, but it serves as a set of portable covers for native multithreaded facilities. Therefore, operating systems that do not have native support for multithreading will only be able to support a limited implementation of the OCIThread package. As a result, products that rely on all of OCIThread's functionality will not port to all operating systems. Products that must port to all operating systems must use only a subset of OCIThread's functionality.

The OCIThread API consists of three main parts. Each part is described briefly here. The following subsections describe each in greater detail.

- Initialization and Termination. These calls deal with the initialization and termination of OCIThread context, which is required for other OCIThread calls.

OCIThread only requires that the process initialization function, OCIThreadProcessInit(), is called when OCIThread is being used in a multithreaded application. Failing to call OCIThreadProcessInit() in a single-threaded application is not an error.

Separate calls to OCIThreadInit() will all return the same OCIThread context. Each call to OCIThreadInit() must eventually be matched by a call to OCIThreadTerm().

- Passive Threading Primitives.*Passive* threading primitives are used to manipulate mutual exclusion locks (mutex), thread IDs, and thread-specific data keys. These primitives are described as passive because while their specifications allow for the existence of multiple threads, they do not require it. 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 a multiple-thread environment in order to work correctly. They will be able to work in single-threaded environments without branching code.
- Active Threading Primitives. Active threading primitives deal with the creation, termination, and manipulation of threads. These primitives are described as *active* because they can only be used in true multithreaded environments. Their specification explicitly requires multiple threads. If you need to determine at runtime whether or not you are in a multithreaded environment, call OCIThreadIsMulti() before using an OCIThread active primitive.

  In order to write a version of the same application to run on single-threaded operating system, it is necessary to branch your code, whether by branching versions of the source file or by branching at runtime with the OCIThreadIsMulti() call.

  See Also:

  - "Thread Management Functions" on page 16-132
  - cdemothr.c in the demo directory is an example of a multithreading application.

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 initialized before any of its functionality can be used.

The observed behavior of the initialization and termination functions is the same regardless of whether OCIThread is in single-threaded or multithreaded environment. Table 9–1 lists functions for thread initialization and termination.

Table 9–1 Initialization and Termination Multithreading Functions

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.

See Also: "Thread Management Functions" on page 16-132

OCIThread Context

Most calls to `OCIThread` functions use 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.

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 in multithreaded and single-threaded operating systems. Table 9–2 lists functions used to implement passive threading.

Table 9–2 Passive Threading Primitives

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.
`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`.

OCIThreadMutex

The `OCIThreadMutex` datatype is used to represent a mutex. This mutex is used to ensure that:

- only one thread accesses a given set of data at a time, or
- 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 standalone 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 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 datatype `OCIThreadKey` can be thought of as a process-wide variable with a thread-specific value. This means that all threads in a process can use a given key, but 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 other threads. The datatype of the value held by a key is a `dvoid *` generic pointer.

Keys can be created using `OCIThreadKeyInit()`. Key value are 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 them 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 initialization, the task stores data in the thread using `OCIThread` key functions. After initialization, the thread is returned back to the threads pool. Later, the threads pool manager assigns 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. Application developers that use thread pools have to be aware of this.

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 using `OCIThreadKeyInit()`. A key's destructor routine will be called whenever a thread with a non-`NULL` value for the key terminates. The destructor routine returns nothing and takes one parameter, 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; 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` datatype is used to identify a thread. At any given time, no two threads will ever have the same `OCIThreadId`, but `OCIThreadId` values can be recycled; once a thread dies, a new thread may be created that has the same `OCIThreadId` value. 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 is 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.

Active Threading Primitives

The active threading primitives deal with manipulation of actual threads. Because specifications of most of these primitives require multiple threads, they work correctly only in the enabled OCIThread; In the disabled OCIThread, they always return an error. The exception is OCIThreadHandleGet(); it may be called in a single-threaded environment and has no effect.

Active primitives can only be called by code running in a multithreaded environment. You can call OCIThreadIsMulti() to determine whether the environment is multithreaded or single-threaded. Table 9–3 lists functions used to implement active threading.

Table 9–3 Active Threading Primitives

Function	Purpose
OCIThreadHndInit()	Allocates and initializes a thread handle.
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.

OCIThreadHandle

Datatype 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().

Connection Pooling in OCI

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 is 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. The application can create a pool (or a set of pools) in each environment during application 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.

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.

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 can 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 Statefull Sessions

Stateless sessions are serially reusable across mid-tier threads. After a thread is done processing a database request on behalf of a three-tier user, the same database session can be reused for the purpose of a completely different request on behalf of a completely different three-tier user.

Statefull sessions to the database, on the other hand, are not serially reusable across mid-tier threads because they may have some particular state associated with a particular three-tier user. Examples of such state may be: open transactions, fetch state from a statement, PL/SQL package state, and so on. This makes the session not reusable for a different request for the duration that such state persists.

Note: Stateless sessions too may have open transactions, open statement fetch state, and so on. However, such a state persists for a relatively short duration (only during the processing of a particular three-tier request by a mid-tier thread) which allows the session to be serially reused for a different three-tier user (when such state is cleaned up).

Note: Stateless sessions are typically used in conjunction with Statement Caching.

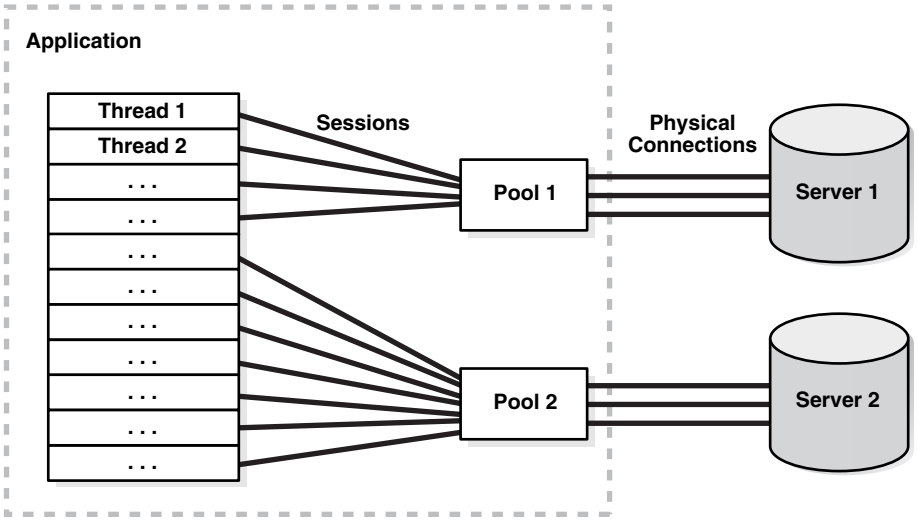
What connection pooling offers is stateless connections and statefull sessions. Users who need to work with stateless sessions, see "Session Pooling in OCI" on page 9-13.

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:

Figure 9–1 OCI Connection Pooling



Transparent Application Failover

Transaction Application Failover (TAF) is enabled for connection pooling. The concepts of TAF apply equally well with connections in the connection pool except that the `BACKUP` and `PRECONNECT` clauses should not be used in the connect string and do not work with connection pooling and TAF.

When a connection in the connection pool fails over, it uses the primary connect string itself to connect. Sessions failover when they use the pool for a database round trip after their instance failure. The listener would be configured to route it to a good instance if available, as is typical with service-based connect strings.

> **See Also:** *Oracle Database Net Services Reference* for definitions of clauses used. See chapter "Local Naming Parameters (tnsnames.ora)", "Local Naming parameters", "Connect Data Section", "fail-over."

OCI Calls for Connection Pooling

The steps in using connection pooling in your application are:

- Allocate the Pool Handle
- Create the Connection Pool
- Logon to the Database
- Deal with SGA Limitations in Connection Pooling
- Logoff from the Database
- Destroy the Connection Pool
- Free the Pool Handle

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.

> **Note:** Shrinkage of the pool only occurs when there is a network round trip. If there are no operations, then the connections stay alive.

All the preceding 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

The application will need to log on to the database for each thread, using one of the following interfaces.

- `OCILogon2()`

  This is the simplest interface. Use this interface when you need a simple Connection Pool connection and do not need to alter any attributes of the session handle. This interface can also be used to make proxy connections to the database.

  Here is an example using `OCILogon2()`:

```
for (i = 0; i < MAXTHREADS; ++i)
{
   OCILogon2(envhp, errhp, &svchp[i], "hr", 2, "hr", 2, poolName,
             poolNameLen, OCI_LOGON2_CPOOL));

}
```

  In order to use this interface to get a proxy connection, set the password parameter to NULL.

- `OCISessionGet()`

  This is the recommended interface. It gives the user the additional option of using external authentication methods, such as certificates, distinguished name, and so on. `OCISessionGet()` is the recommended uniform function call to retrieve a session.

  Here is an example using `OCISessionGet()`:

```
for (i = 0; i < MAXTHREADS; ++i)
{
        OCISessionGet(envhp, errhp, &svchp, authp,
                      (OraText *) poolName,
                      strlen(poolName), NULL, 0, NULL, NULL, NULL,
                      OCI_SESSGET_CPOOL)
 }
```

- `OCIServerAttach()` and `OCISessionBegin()`:

  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 allocate all the handles (connection pool handle, server handles, session handles and service context handles).

  - Create the connection pool.
  - Call `OCIServerAttach()` with mode set to `OCI_CPOOL`.
  - Call `OCISessionBegin()` with mode set to `OCI_DEFAULT`.

The user should not set `OCI_MIGRATE` flag in the call to `OCISessionBegin()`, when the virtual server handle points to a connection pool (`OCIServerAttach()` called with `mode` set to `OCI_CPOOL`). Oracle supports passing this flag, `OCI_MIGRATE`, only for compatibility reasons. Do not use the `OCI_MIGRATE` flag, because the perception that the user gets when using a connection pool is of sessions having their own dedicated (virtual) connections which are transparently multiplexed onto real connections. Credentials can be set to `OCI_CRED_RDBMS` or `OCI_CRED_PROXY`. If the credentials are set to `OCI_CRED_PROXY`, only user name needs to be set on the session

handle. (no explicit primary session needs to be created and OCI_ATTR_MIGSESSION need not be set).

Connection pooling does the multiplexing of a virtual server handle over physical connections transparently, hence eliminating the need for users to do so. The user gets the feeling of a session having a dedicated (virtual) connection. Since the multiplexing is done transparently to the user, Users should not attempt to multiplex sessions over the virtual server handles themselves. The concepts of session migration and session switching, which require explicit multiplexing at the user level, are defunct in the case of connection pooling and should not be used.

In an OCI program, the user should create (OCIServerAttach() with mode set to OCI_CPOOL) a unique virtual server handle for each session that is created using the connection pool. There should be a one-to-one mapping between virtual server handles and sessions.

Deal with SGA Limitations 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:** *Oracle Database Performance Tuning Guide* for more information, see the section on configuring Shared Server

If you are still running into the SGA limitation, you must consider:

- Reducing the session memory consumption by having fewer open statements for each session
- reducing the number of sessions in the back end by pooling sessions on the mid-tier or otherwise
- turning off connection pooling

The application must 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 must consider using shared database links.

> **See Also:** For more information about distributed databases, see the section on shared database links in *Oracle Database Administrator's Guide*

Logoff from the Database

Corresponding to the logon calls, these are the interfaces to use to log off from the database in connection pooling mode.

- OCILogoff():

  If OCILogon2() was used to make the connection, OCILogoff() must be used to log off.

- OCISessionRelease()

If OCISessionGet() was called to make the connection, then OCISessionRelease() must be called to log off.

- OCISessionEnd() and OCIServerDetach()

  If OCIServerAttach() and OCISessionBegin() were called to make the connection and start up the session, then OCISessionEnd() must be called to end the session and OCIServerDetach() must be called to release the connection.

Destroy the 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-18
- "OCIConnectionPoolCreate()" on page 15-7, "OCILogon2()" on page 15-27, and "OCIConnectionPoolDestroy()" on page 15-9

Examples of OCI Connection Pooling

Examples of connection pooling in tested complete programs can be found in cdemocp.c and cdemocpproxy.c in directory demo.

Session Pooling in OCI

Session pooling means that the application will create and maintain a group of stateless sessions to the database. These sessions will be handed over to thin clients as requested. If no sessions are available, a new one may be created. When the client is done with the session, the client will release it to the pool. Thus, the number of sessions in the pool can increase dynamically.

Some of the sessions in the pool may be 'tagged' with certain properties. For instance, a user may request for a default session, set certain attributes on it, then label it or 'tag' it and return in to the pool. That user, or some other user, can require a session with the same attributes, and thus request for a session with the same tag. There may be several sessions in the pool with the same tag. The 'tag' on a session can be changed or reset.

See Also: "Using Tags in Session Pools" on page 9-14

Proxy sessions, too, can be created and maintained through this interface.

The behavior of the application when no free sessions are available and the pool has reached it's maximum size, will depend on certain attributes. A new session may be

created or an error returned, or the thread may just block and wait for a session to become free.

The main benefit of this type of pooling will be performance. Making a connection to the database is a time-consuming activity, especially when the database is remote. Thus, instead of a client spending time connecting to the server, authenticating its credentials, and then receiving a valid session, it can just pick one from the pool.

Functionality of OCI Session Pooling

Session pooling has the following features:

- Create, maintain and manage a pool of stateless sessions transparently.
- Provide an interface for the application to create a pool and specify the minimum, increment and maximum number of sessions in the pool.
- Provide an interface for the user to obtain and release a default or 'tagged' session to the pool. A 'tagged' session is one with certain client-defined properties.
- Allow the application to dynamically change the number of minimum and maximum number of sessions.
- Provide a mechanism to always maintain an optimum number of open sessions, by closing sessions that have been idle for very long, and creating sessions when required.
- Allow for session pooling with authentication.

Homogeneous and Heterogeneous Session Pools

A session pool can be either homogeneous or heterogeneous. Homogeneous session pooling means that sessions in the pool are alike with respect to authentication (have the same user name and password and privileges). Heterogeneous session pooling means that you must provide authentication information because the sessions can have different security attributes and privileges.

Using Tags in Session Pools

The tags provide a way for users to customize sessions in the pool. A client may get a default or untagged session from a pool, set certain attributes on the session (such as NLS settings), and return the session to the pool, labeling it with an appropriate tag in the `OCISessionRelease()` call.

The user, or some other user, may request a session with the same tags in order to have a session withe the same attributes, and can do so by providing the same tag in the `OCISessionGet()` call.

> **See Also:** "OCISessionGet()" on page 15-37 for a further discussion of tagging sessions.

OCI Handles for Session Pooling

Two handle types have been added for session pooling:

OCISPool

This is the session pool handle. It is allocated using `OCIHandleAlloc()`. It needs to be passed to `OCISessionPoolCreate()`, and OCISessionPoolDestroy(). It has the attribute type `OCI_HTYPE_SPOOL`.

An example of the OCIHandleAlloc() call follows:

```
OCISPool *spoolhp;
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &spoolhp, OCI_HTYPE_SPOOL,
                      (size_t) 0, (dvoid **) 0)
);
```

For an environment handle, multiple session pools can be created.

OCIAuthInfo

This is the authentication information handle. It is allocated using OCIHandleAlloc(). It is passed to OCISessionGet(). It supports all the attributes that are supported for user session handle. Please refer to user session handle attributes for more information. The authentication information handle has the attribute type OCI_HTYPE_AUTHINFO.

An example of the OCIHandleAlloc() call follows:

```
OCIAuthInfo *authp;
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &authp, OCI_HTYPE_AUTHINFO,
                      (size_t) 0, (dvoid **) 0)
);
```

See Also:

- "User Session Handle Attributes" on page A-12 for the attributes that belong to the authentication information handle.
- "Session Pool Handle Attributes" on page A-20 for more information about the session pooling attributes.
- "Connect, Authorize, and Initialize Functions" on page 15-3 for complete information about the functions used in session pooling.
- See "OCISessionGet()" on page 15-37 for details of the session handle attributes that can be used with this call.

Using OCI Session Pooling

The steps in writing a simple session pooling application which uses a user name and password are:

- Allocate the session pool handle using OCIHandleAlloc() for an OCISPool handle. Multiple session pools can be created for an environment handle.
- Create the session pool using OCISessionPoolCreate() with mode set to OCI_DEFAULT (for a new session pool). Refer to the function for a discussion of the other modes.
- Loop for each thread. Create the thread with a function that does the following:
- Allocate an authentication information handle of type OCIAuthInfo using OCIHandleAlloc().
- Set the user name and password in the authentication information handle using OCIAttrSet().
- Get a pooled session using OCISessionGet() with mode set to OCI_SESSGET_SPOOL.
- Perform the transaction.

- Commit or rollback the transactions.
- Release the session (logoff) with OCISessionRelease().
- Free the authentication information handle with `OCIHandleFree()`.
- End of the loop for each thread.
- Destroy the session pool using OCISessionPoolDestroy().

OCI Calls for Session Pooling

Here are the usages for OCI calls for session pooling.

Allocate the Pool Handle

Session pooling requires that the pool handle `OCI_HTYPE_SPOOL` be allocated by calling `OCIHandleAlloc()`.

Multiple pools can be created for a given environment handle. For a single session pool, here is an allocation example:

```
OCISPool *poolhp;
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &poolhp, OCI_HTYPE_SPOOL, (size_t) 0,
               (dvoid **) 0));
```

Create the Pool Session

The function `OCISessionPoolCreate()` can be used to create the session pool. Here is an example of how to use this call:

```
OCISessionPoolCreate(envhp, errhp, poolhp, (OraText **)&poolName,
              (ub4 *)&poolNameLen, database,
              (ub4)strlen((const signed char *)database),
              sessMin, sessMax, sessIncr,
              (OraText *)appusername,
              (ub4)strlen((const signed char *)appusername),
              (OraText *)apppassword,
              (ub4)strlen((const signed char *)apppassword),
              OCI_DEFAULT);
```

Logon to the Database

These are the interfaces that can be used to logon to the database in session pooling mode.

- `OCILogon2()`:

  This is the simplest interface. However, it does not give the user the option of using tagging. Here is an example of how `OCILogon2()` can be used to log on to the database in session pooling mode:

  ```
  for (i = 0; i < MAXTHREADS; ++i)
  {
    OCILogon2(envhp, errhp, &svchp[i], "hr", 2, "hr", 2, poolName,
              poolNameLen, OCI_LOGON2_SPOOL));

  }
  ```

- `OCISessionGet()`:

  This is the recommended interface. It gives the user the option of using tagging to label sessions in the pool, and thus make it easier to retrieve specific sessions. An

example of using OCISessionGet() follows. It is taken from cdemosp.c in the demo directory.:

```
OCISessionGet(envhp, errhp, &svchp, authInfop,
              (OraText *)database,strlen(database), tag,
              strlen(tag), &retTag, &retTagLen, &found,
              OCI_SESSGET_SPOOL);
```

Logoff from the Database

Corresponding to the preceding logon calls, these are the interfaces to use to log off from the database in session pooling mode.

- OCILogoff():

  If OCILogon2() was used to make the connection, OCILogoff() must be used to log off.

- OCISessionRelease()

  If OCISessionGet() was called to make the connection, then OCISessionRelease() must be called to log off.

Destroy the Session Pool

OCISessionPoolDestroy() must be called to destroy the session pool. Here is an example of how this call can be made:

```
OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT);
```

Free the Pool Handle

OCIHandleFree() must be called to free the session pool handle. Here is how this call can be made:

```
OCIHandleFree((dvoid *)poolhp, OCI_HTYPE_SPOOL);
```

Note: The application has to ensure either a commit or rollback is done before a session is released to the session pool. OCI does not reset the state of the session.

Example of OCI Session Pooling

Here is an example of session pooling in a tested, complete program:

See Also: cdemosp.c in directory demo

When to Use Connection Pooling, Session Pooling, or Neither

If database sessions are not reusable by mid-tier threads (that is, they are statefull) and the number of back-end server processes may cause scaling problems on the database, use OCI connection pooling.

If database sessions are reusable by mid-tier threads (that is, they are stateless) and the number of back-end server processes may cause scaling problems on the database, use OCI session pooling.

If database sessions are not reusable by mid-tier threads (that is, they are statefull) and the number of back-end server processes will never be large enough to potentially

cause any scaling issue on the database, there is no need to use any pooling mechanism.

> **Note:** Having non-pooled sessions/connections will result in tearing down and recreation of the database session/connection for every mid-tier user request. This can cause severe scaling problems on the database side and excessive latency for the fulfillment of the request. Hence, it is strongly recommended that one of the pooling strategies be adopted for mid-tier applications based on whether the database session is statefull or stateless.

In connection pooling, the pool element is a connection and in session pooling, the pool element is a session.

As with any pool, the pooled resource is locked by the application thread for a certain duration until the thread has done its job on the database and the resource is released. The resource is unavailable to other threads during its period of use. Hence, application developers need to be aware that any kind of pooling works effectively with relatively short tasks. If the application is performing a long running transaction for example, it may deny the pooled resource to other sharers for long periods of time leading to starvation. Hence, pooling should be used in conjunction with short tasks and the size of the pool should be sufficiently large to maintain the desired concurrency of transactions.

Also, note that with:

1. OCI Connection Pool
   a. Connections to the database are pooled. Sessions are created and destroyed by the user. Each call to the database will pick up an appropriate available connection from the pool.
   b. The application is multiplexing several sessions over fewer physical connections to the database. The users can tune the pool configuration to achieve required concurrency.
   c. The life-time of the application sessions is independent of the life-time of the cached pooled connections.
2. OCI Session Pool

   Sessions and connections are pooled by OCI. The application gets sessions from the pool and release sessions back to the pool.

Functions for Session Creation

The choices are:

1. `OCILogon()`

   This is the simplest way to get an OCI Session. The advantage is ease of obtaining an OCI service context. The disadvantage is that you cannot perform any advance OCI operations like session migration, proxy authentication, using a connection pool, or a session pool.

   > **See Also:** "Application Initialization, Connection, and Session Creation" on page 2-15

2. `OCILogon2()`

This includes the functionality of `OCILogon()` to get a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The `mode` parameter value that the function is called with determines its behavior.

The user cannot modify the attributes (except `OCI_ATTR_STMTCACHESIZE`) of the service context returned by OCI.

See Also: "OCILogon2()" on page 15-27

3. `OCISessionBegin()`

   This supports all the various options of an OCI session such as proxy authentication, getting a session from a connection pool or a session pool, external credentials, and migratable sessions. This is the lowest level call where all handles are needed to be explicitly allocated and all attributes set, and `OCIServerAttach()` is to be called prior to this call.

   See Also: "OCISessionBegin()" on page 15-33

4. `OCISessionGet()`

   This is now the recommended method to get a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The `mode` parameter value that the function is called with determines its behavior. This works like `OCILogon2()` but additionally enables you to specify tags for obtaining specific sessions from the pool.

   See Also: "OCISessionGet()" on page 15-37

Choosing Between Different Types of OCI Sessions

The choices are:

- Basic OCI Sessions

  This works by using user name and password over a dedicated OCI server handle. This is the no-pool mechanism. See earlier notes of when to use it.

  If authentication is obtained through external credentials, then user name or password is not required.

- Session Pool Sessions

  These sessions are from the session pool cache. Some sessions may be tagged. These are stateless sessions. Each `OCISessionGet()` and `OCISessionRelease()` call gets and releases a session from the session cache. This saves the server from creating and destroying sessions.

  See the earlier notes on connection pool sessions versus session pooling sessions versus no-pooling sessions.

- Connection Pool Sessions

  These are sessions created using `OCISessionGet()` and `OCISessionBegin()` calls from an OCI Connection Pool. There is no session cache as these are statefull sessions. Each call creates a new session and the user is responsible for terminating these sessions.

The sessions are automatically migratable between the server handles of the connection pool. Each session can have user name and password or be a proxy session. See the earlier notes on connection pool sessions versus session pooling sessions versus no-pooling sessions.

- Sessions Sharing a Server Handle

  You can multiplex several OCI Sessions over a few physical connections. The application does this manually by having the same server handle for these multiple sessions. It is preferred to have the session multiplexing details be left to OCI by using the OCI Connection Pool APIs.

- Proxy Sessions

  This is useful if the password of the client needs to be protected from the middle-tier. Proxy sessions can also be part of OCI Connection Pool and OCI Session Pool.

  See Also: "Middle-Tier Applications in OCI" on page 8-10

- Migratable Sessions

With transaction handles being migratable, there should be no need for applications to use this older feature, in light of OCI Connection Pooling.

See Also: "OCI Session Management" on page 8-9

Statement Caching in OCI

Statement caching refers to the feature that provides and manages a cache of statements for each session. In the server, it means that cursors are ready to be used without the need to parse the statement again. Statement caching can be used with connection pooling and with session pooling, and will improve performance and scalability. It can be used without session pooling as well. The OCI calls that implement statement caching are:

- `OCIStmtPrepare2()`
- `OCIStmtRelease()`

Statement Caching without Session Pooling in OCI

Users perform the usual OCI steps to logon. The call to obtain a session will have a mode that specifies whether statement caching is enabled for the session. Initially the statement cache will be empty. Developers will try to find a statement in the cache using the statement text. If the statement exists the API will return a previously prepared statement handle, otherwise it will return an newly prepared statement handle.

The application developer can perform binds and defines and then simply execute and fetch the statement before returning the statement back to the cache. In the latter case, where the statement handle was not found, the developer will need to set different attributes on the handle in addition to the other steps.

`OCIStmtPrepare2()` will also take a mode which will determine if the developer wants a prepared statement handle or a null statement handle if the statement is not found in the cache.

The pseudo code will look like:

```
OCISessionBegin( userhp, ... OCI_STMT_CACHE)  ;
```

```
OCIAttrset(svchp, userhp, ...);  /* Set the user handle in the service context */
OCIStmtPrepare2(svchp, &stmthp, stmttext, key, ...);
OCIBindByPos(stmthp, ...);
OCIDefineByPos(stmthp, ...);
OCIStmtExecute(svchp, stmthp, ...);
OCIStmtFetch(svchp, ...);
OCIStmtRelease(stmthp, ...);
...
```

Statement Caching with Session Pooling in OCI

The concepts remain the same, except that the statement cache is enabled at the session pool layer rather than at the session layer.

The attribute `OCI_ATTR_SPOOL_STMTCACHESIZE` sets the default statement cache size for each of the sessions in the session pool to this value. It is set on the `OCI_HTYPE_SPOOL` handle. The statement cache size for a particular session in the pool can anytime be overridden by using OCI_ATTR_STMTCACHESIZE on that session. The value of `OCI_ATTR_SPOOL_STMTCACHESIZE` can be changed at any time. This attribute can be used to enable or disable statement caching at the pool level, after creation, just as attribute `OCI_ATTR_STMTCACHESIZE` (on the service context) is used to enable or disable statement caching at the session level. This change will be reflected on individual sessions in the pool, when they are handed to a user. Tagged sessions are an exception to this behavior. This is explained later.

Enabling or disabling of statement caching is allowed on individual pooled sessions similar to non-pooled sessions.

A user can enable statement caching on a session retrieved from a non-statement cached pool in an `OCISessionGet()` or `OCILogon2()` call by specifying `OCI_SESSGET_STMTCACHE` or `OCI_LOGON2_STMTCACHE`, respectively, in the mode argument.

When a user asks for a session from a session pool, the statement cache size for that session will default to that of the pool. This may also mean enabling or disabling statement caching in that session. For example, if a pooled session (session A) has statement caching enabled, and statement caching is turned off in the pool, and a user asks for a session, and session A is returned, then statement caching will be turned off in Session A. As another example, if Session A in a pool does not have statement caching enabled, and statement caching at the pool level is turned on, then before returning session A to a user, statement caching on Session A with size equal to that of the pool is turned on.

This will not hold true if a tagged session is asked for and retrieved. In this case, the size of the statement cache will not be changed. Consequently, it will not be turned on or off. Moreover, if the user specifies mode `OCI_SESSGET_STMTCACHE` in the `OCISessionGet()` call, this will be ignored if the session is tagged. In our earlier example, if Session A was tagged, then it is returned as is to the user.

Rules for Statement Caching in OCI

Here are some rules to follow:

- Use the function `OCIStmtPrepare2()` instead of `OCIStmtPrepare()`. If you are using `OCIStmtPrepare()`, you are strongly urged not to use a statement handle across different service contexts. Doing so will raise an error if the statement has been obtained by `OCIStmtPrepare2()`. Migration of a statement handle to a new service context actually closes the cursor associated with the old session and therefore no sharing is achieved. Client-side sharing is also not

obtained, because OCI will free all buffers associated with the old session when the statement handle is migrated.

- You are urged to keep one service context for each session and use statement handles only for that service context. That will be the preferred and recommended model and usage.
- A call to `OCIStmtPrepare2()`, even if the session does not have a statement cache, will also allocate the statement handle and therefore applications using only `OCIStmtPrepare2()` must not call `OCIHandleAlloc()` for the statement handle.
- A call to the `OCIStmtPrepare2()` must be followed with `OCIStmtRelease()` after the user is done with the statement handle. If statement caching is used, this will release the statement to the cache. If statement caching is not used, the statement will be deallocated. Do not call `OCIHandleFree()` to free the memory.
- If the call to `OCIStmtPrepare2()` is made with the `OCI_PREP2_CACHE_SEARCHONLY` mode and a NULL statement was returned (statement was not found), the subsequent call to `OCIStmtRelease()` is not required and must not be performed.
- Do not call `OCIStmtRelease()` for a statement that was prepared using `OCIStmtPrepare()`.
- The statement cache has a maximum size (number of statements) which can be modified by an attribute on the service context, `OCI_ATTR_STMTCACHESIZE`. The default value is 20. This attribute can also be used to enable or disable statement caching for the session, pooled or non-pooled. If `OCISessionBegin()` is called without mode set as `OCI_STMT_CACHE`, then `OCI_ATTR_STMTCACHESIZE` can be set on the service context to a nonzero attribute to turn on statement caching. If statement caching is not turned on at the session pool level, `OCISessionGet()` will return a non-statement cache-enabled session. `OCI_ATTR_STMTCACHESIZE` can be used to turn the caching on. Similarly the same attribute can be used to turn off statement caching by setting the cache size to zero.
- You can choose to tag a statement at the release time so that the next time you can request a statement of the same tag. The tag will be used to search the cache. An untagged statement (tag is `NULL`) is a special case of a tagged statement. Two statements are considered different if they only differ in their tags, or if one is untagged and the other is not.

See Also: For information about the functions for statement caching,

- "Statement Functions" on page 16-3
- "Service Context Handle Attributes" on page A-8
- "Session Pool Handle Attributes" on page A-20

OCI Statement Caching Code Example

Here is an example of statement caching:

See Also: Please refer to `ocisc.c` in directory `demo` for a working example of statement caching.

User-Defined Callback Functions in OCI

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 Windows, 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 in OCI

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:** "OCIUserCallbackGet()" on page 16-186 and "OCIUserCallbackRegister()" on page 16-188

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: "OCIUserCallbackRegister()" on page 16-188

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: 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 must 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 earlier, 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 or `ociucb.dll` on a Windows 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 operating system. For further information on user-defined callbacks, see your operating system-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) must 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 operating systems. 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: "OCIUserCallbackGet()" on page 16-186

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 the 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(dvoid *metaCtx, dvoid *libCtx, ub4 argfmt, sword argc, dvoid **argv)
{
  return OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv, pkgNEnvCallback);
}
```

The pkgNEnvCallback() function registers the entry, replacement, and exit callbacks as:

```
pkgNEnvCallback(OCIEnv *env, ub4 mode, size_t xtramemsz,
                                dvoid *usrmemp, OCIUcb *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()

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 *Oracle Database Application Developer's Guide - Fundamentals*

Transparent Application Failover Callbacks in OCI

Transparent Application Failover (TAF) callbacks can be used in the event of the failure of one database instance and failover to another instance. Failover means that connections are reestablished using the same connect string or an alternate connect string specified in the application. 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.

In previous versions, TAF with the SELECT failover option would be engaged on the statement that was in use at the time of a failure. For example, if there were 10 statement handles in use by the application, and statement 7 was the failure-time

statement (the statement in use when the failure happened), statements 1-6 and 8-10 would have to be re-executed after statement 7 was failed over using TAF.

Since 10g Release 2 (10.2), this has been improved. Now all statements that an application attempts to use after a failure will attempt failover. That is, an attempt to execute or fetch against other statements will engage TAF recovery just as for the failure-time statement. Notably, this means that subsequent statements may now succeed (whereas in the past they failed), or the application may receive errors corresponding to an attempted TAF recovery (like ORA-25401). However, the TAF callbacks will only be called once (during TAF recovery of the failure-time statement).

Note: TAF is not supported for remote database links. TAF is not supported for DML statements.

See Also: *Oracle Database Net Services Reference* for more detailed information about application failover

Failover Callback Overview

To address the problems described earlier, 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 re-authenticated 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 "Handling OCI_FO_ERROR" on page 9-34 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 9-32 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.
- `OCI_FO_REAUTH` indicates that you have multiple authentication handles and failover has occurred after the original authentication. It indicates that a user handle has been re-authenticated. To find out which, the application checks 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, registration, and unregistration.

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 stopped. 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(srvh, errh)
dvoid *srvh; /* 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.")+1)))
     return(1);
  /* initialize the context. */
  strcpy((char *)failover.fo_ctx, "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);
}
```

Part 3: Failover Callback Unregistration

```
OCIFocbkStruct failover;   /*  failover callback structure */
sword status;

  /* set the failover context to null */
  failover.fo_ctx = NULL;
  /* set the failover callback to null */
  failover.callback_function = NULL;
  /* un-register the callback */
  status = OCIAttrSet(srvhp, (ub4) OCI_HTYPE_SERVER,
                      (dvoid *) &failover, (ub4) 0,
                      (ub4) OCI_ATTR_FOCBK, errhp);
```

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 must 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.

Failover does not work if a LOB column is part of the select list.

Consider the following timeline of events:

Table 9–4 Time and Event

Time	Event
T0	Database fails (failure 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 can be used to implement the failover strategy similar to the scenario described earlier. 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;
}
```

HA Event Notification

Suppose an user employs a Web browser to log in to an application server that accesses a back-end database server. Failure of the database instance can result in a wait that can be up to in minutes in duration before the failure is known to the user. The ability to quickly detect failures of server instances, communicate this to the client, close connections, and clean up idle connections in connection pools is provided by HA event notification.

For High Availability clients connected to a Real Application Clusters (RAC) database, HA event notification can be used to provide a best-effort programmatic signal to the client in the event of a database failure. Client applications can register a callback on the environment handle to signal interest in this information. When a significant failure event occurs (which applies to a connection made by this client), the callback is invoked, with information concerning the event (the event payload), and a list of connections (server handles) that were disconnected as a result of the failure.

For example, consider a client application that has two connections to instance A and two connections to instance B of the same database. If instance A goes down, a notification of the event will be sent to the client, which will then disconnect the two connections to instance B, and invoke the registered callback. Note that if another instance, C, of the same database, goes down, the client will not be notified (since it will not affect any of the client's connections).

The HA event notification mechanism improves the response time of the application in the presence of failure. In the past, a failure would result in the connection being broken only after the TCP time out expired, which could take minutes. With HA event notification, standalone, connection pool, and session pool connections are automatically broken and cleaned up by OCI and the application callback is invoked within seconds of the failure event. If any of these server handles are TAF-enabled, failover will also automatically be engaged by OCI.

Applications must connect to a RAC instance to enable HA event notification. Furthermore, these applications must:

- Initialize the OCI Environment in `OCI_EVENTS` mode.
- Connect to a service that has notifications enabled (use the `DBMS_SERVICE.MODIFY_SERVICE` procedure to set `AQ_HA_NOTIFICATIONS` to `TRUE`).
- Link with a thread library.

Then these applications can register a callback that is invoked whenever an HA event occurs.

OCIEvent Handle

The `OCIEvent` handle encapsulates the attributes from the event payload. OCI implicitly allocates this handle prior to calling the event callback, which can obtain the read-only attributes of the event by calling `OCIAttrGet()`. Memory associated with these attributes is only valid for the duration of the event callback.

See Also: "Event Handle Attributes" on page A-65

OCI Failover for Connection and Session Pools

A connection pool in an instance of Real Application Clusters (RAC) consists of a pool of connections connected to different instances of the RAC. Upon receiving the node failure notification, all the connections connected to that particular instance should be cleaned up. For the connections that are in use, OCI has to close the connections: transparent application failover (TAF) occurs immediately and those connections will be reestablished. The connections that are idle and in the free list of the pool have to be purged, so that a bad connection is never returned back to the user from the pool.

To accommodate custom connection pools, OCI will provide a callback function that can be registered on the environment handle. If registered, this callback is invoked when an HA event occurs. Sessions pools are treated the same way as connection pools. Note that server handles from OCI connection pools or session pools will not be passed to the callback. Hence in some cases, the callback could be called with an empty list of connections.

OCI Failover for Independent Connections

No special handling is required for independent connections; all such connections that are connected to failed instances are immediately disconnected. For idle connections, TAF will be engaged to reestablish the connection when the connection is used on a subsequent OCI call. Connections that are in use at the time of the failure event will be broken out immediately, so that TAF can begin. Note that this applies for the "in-use" connections of connection and session pools also.

Event Callback

The event callback, of type `OCIEventCallback`, has the following signature:

```
void evtcallback_fn (dvoid     *evtctx,
                     OCIEvent  *eventhp );
```

where `evtctx` is the client context and `OCIEvent` is an event handle which is opaque to the OCI library. The other input argument is `eventhp`, the event handle: that is, the attributes associated with an event.

If registered, this function will be called once for each event. In the case of RAC HA events, this callback will be invoked after the affected connections have been disconnected. The following environment handle attributes are used to register an event callback and context, respectively:

- `OCI_ATTR_EVTCBK` is of datatype `OCIEventCallback *`. It is read-only.
- `OCI_ATTR_EVTCTX` is of datatype `dvoid *`. It is also read-only.

```
text *myctx = "dummy context"; /* dummy context passed to callback fn */
...
/* OCI_ATTR_EVTCBK and OCI_ATTR_EVTCTX are read-only. */
OCIAttrSet(envhp, (ub4) OCI_HTYPE_ENV, (dvoid *) evtcallback_fn,
           (ub4) 0, (ub4) OCI_ATTR_EVTCBK, errhp);
OCIAttrSet(envhp, (ub4) OCI_HTYPE_ENV, (dvoid *) myctx,
           (ub4) 0, (ub4) OCI_ATTR_EVTCTX, errhp);
...
```

Within the OCI Event callback, the list of affected server handles is encapsulated in the `OCIEvent` handle. For RAC HA DOWN events, client applications can iterate over a list of server handles that are affected by the event by using `OCIAttrGet()` with attribute types `OCI_ATTR_HA_SVRFIRST` and `OCI_ATTR_HA_SVRNEXT`:

```
OCIAttrGet(eventhp, OCI_HTYPE_EVENT, (dvoid *)&srvhp, (ub4 *)0,
           OCI_ATTR_HA_SRVFIRST, errhp);
/* or, */
OCIAttrGet(eventhp, OCI_HTYPE_EVENT, (dvoid *)&srvhp, (ub4 *)0,
           OCI_ATTR_HA_SRVNEXT, errhp);
```

When called with attribute OCI_ATTR_HA_SRVFIRST, this function will retrieve the first server handle in the list of server handles affected. When called with attribute OCI_ATTR_HA_SRVNEXT, this function will retrieve the next server handle in the list. This function will return OCI_NO_DATA and srvhp will be a NULL pointer, when there are no more server handles to return.

srvhp is an output pointer to a server handle whose connection has been closed as a result of an HA event. errhp is an error handle to populate. The application returns an OCI_NO_DATA error when there are no more affected server handles to retrieve.

When retrieving the list of server handles that have been affected by an HA event, be aware that the connection has already been closed and many server handle attributes are no longer valid. Instead, use the user memory segment of the server handle to store any per-connection attributes required by the event notification callback. This memory remains valid until the server handle is freed.

Custom Pooling: Tagged Server Handles

The following features apply to custom pools:

1. You can tag a server handle with its parent connection object if it is created on behalf of a custom pool. Use the "user memory" parameters of OCIHandleAlloc() to request that the server handle be allocated with a user memory segment. A pointer to the "user memory" segment is returned by OCIHandleAlloc().
2. When an HA event occurs, and an affected server handle has been retrieved, there is a means to retrieve the server handle's tag information so appropriate cleanup can be performed. The attribute OCI_ATTR_USER_MEMORY is used to retrieve a pointer to a handle's user memory segment. OCI_ATTR_USER_MEMORY is valid for all user-allocated handles. If the handle was allocated with extra memory, this attribute will return a pointer to the user memory. A NULL pointer will be returned for those handles not allocated with extra memory. This attribute is read-only and is of datatype dvoid*.

> **Note:** You are free to define the precise contents of the server handle's user memory segment to facilitate cleanup activities from within the HA event callback (or for other purposes if needed) because OCI does not write or read from this memory in any way. The user memory segment is freed along with the OCIHandleFree() call on the server handle.

Event Notification Example

```
sword retval;
OCIServer *srvhp;
struct myctx {
   dvoid *parentConn_myctx;
   uword numval_myctx;
};
typedef struct myctx myctx;
myctx  *myctxp;
```

```
/* Allocate a server handle with user memory - pre 10.2 functionality */
if (retval = OCIHandleAlloc(envhp, (dvoid **)&srvhp, OCI_HTYPE_SERVER,
                            (size_t)sizeof(myctx), (dvoid **)&myctxp)
/* handle error */
myctxp->parentConn_myctx = <parent connection reference>;

/* In an event callback function, retrieve the pointer to the user memory */
evtcallback_fn(dvoid *evtctx, OCIEvent *eventhp)
{
  myctx *ctxp = (myctx *)evtctx;
  OCIServer *srvhp;
  OCIError *errhp;
  sb4       retcode;
  retcode = OCIAttrGet(eventhp, OCI_HTYPE_SERVER, &srvhp, (ub4 *)0,
                       OCI_ATTR_HA_SRVFIRST, errhp);
  while (!retcode) /* OCIAttrGet will return OCI_NO_DATA if no more srvhp */
  {
     OCIAttrGet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *)&ctxp,
                (ub4)0, (ub4)OCI_ATTR_USER_MEMORY, errhp);
           /* Remove the server handle from the parent connection object */
     retcode = OCIAttrGet(eventhp, OCI_HTYPE_SERVER, &srvhp, (ub4 *)0,
                          OCI_ATTR_HA_SRVNEXT, errhp);
...
  }
...
}
```

Determining Transparent Application Failover (TAF) Capabilities

You can have the application adjust its behavior if a connection is or is not TAF-enabled. Use `OCIAttrGet()` as follows to find out if a server handle is TAF-enabled:

```
boolean taf_capable;
...
OCIAttrGet(srvhp, (ub4) OCI_HTYPE_SERVER, (dvoid *) &taf_capable,
           (ub4) sizeof(taf_capable), (ub4)OCI_ATTR_TAF_ENABLED, errhp);
...
```

where `taf_capable` is a boolean variable, which this call sets to `TRUE` if the server handle is TAF-enabled, and `FALSE` if not; `srvhp` is an input target server handle; `OCI_ATTR_TAF_ENABLED` is a an attribute which is a pointer to a boolean variable and is read-only; `errhp` is an input error handle.

OCI and Streams Advanced Queuing

The OCI provides an interface to Streams Advanced Queuing (Streams AQ) feature. Streams AQ provides message queuing as an integrated part of the Oracle server. Streams AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database.* By providing an integrated solution Streams 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 Streams Advanced Queuing, you must be using the Enterprise Edition

---

See Also:

- *Oracle Streams Advanced Queuing User's Guide and Reference*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Integration Server Overview*
- For example code demonstrating the use of the OCI with AQ, refer to the description of OCIAQEnq() on page 16-103

OCI Streams Advanced Queuing Functions

The OCI library includes several functions related to Streams Advanced Queuing:

- `OCIAQEnq()`
- `OCIAQDeq()`
- `OCIAQListen()`
- `OCIAQListen2()`
- `OCIAQEnqArray()`
- `OCIAQDeqArray()`

You can enqueue an array of messages to a single queue. The messages all share the same enqueue options, but each message in the array can have different message properties. You can also dequeue an array of messages from a single queue. For transaction group queues, you can dequeue all messages for a single transaction group using one call.

See Also: "Streams Advanced Queuing and Publish-Subscribe Functions" on page 16-98

OCI Streams Advanced Queuing Descriptors

The following descriptors are used by OCI Streams 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 or read.

See Also: "Streams Advanced Queuing Descriptor Attributes" on page A-34

Streams Advanced Queuing in OCI Versus PL/SQL

The following tables compare functions, parameters, and options for OCI Streams AQ functions and descriptors, and PL/SQL AQ functions in the DBMS_AQ package.

Table 9–5 AQ Functions

PL/SQL Function	OCI Function
DBMS_AQ.ENQUEUE	`OCIAQEnq()`
DBMS_AQ.DEQUEUE	`OCIAQDeq()`
DBMS_AQ.LISTEN	`OCIAQListen(), OCIAQListen2()`
DBMS_AQ.ENQUEUE_ARRAY	`OCIAQEnqArray()`
DBMS_AQ.DEQUEUE_ARRAY	`OCIAQDeqArray()`

The following table compares the parameters for the enqueue functions:

Table 9–6 Enqueue Parameters

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.`

The following table compares the parameters for the dequeue functions:

Table 9–7 Dequeue Parameters

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, dequeue_options, message_properties, payload_tdo, payload, payload_ind, and flags.`

The following table compares parameters for the listen functions:

Table 9–8 Listen Parameters

DBMS_AQ.LISTEN Parameter	OCIAQListen2() Parameter
`agent_list`	`agent_list`
`wait`	`wait`
`agent`	`agent`

Table 9–8 (Cont.) Listen Parameters

DBMS_AQ.LISTEN Parameter	OCIAQListen2() Parameter
listen_delivery_mode	lopts
-	Note: `OCIAQListen2()` also requires the following additional parameters: `svchp, errhp, agent_list, num_agents, agent, lmops, and flags`.

The following table compares parameters for the array enqueue functions:

Table 9–9 Array Enqueue Parameters

DBMS_AQ.ENQUEUE_ARRAY Parameter	OCIAQEnqArray() Parameter
`queue_name`	`queue_name`
`enqueue_options`	`enqopt`
`array_size`	`iters`
`message_properties_array`	`msgprop`
`payload_array`	`payload`
`msgid_array`	`msgid`
-	Note: `OCIAQEnqArray()` also requires the following additional parameters: `svch, errh, payload_tdo, payload_ind, ctxp, enqcbfp, and flags`.

The following table compares parameters for the array dequeue functions:

Table 9–10 Array Dequeue Parameters

DBMS_AQ.DEQUEUE_ARRAY Parameter	OCIAQDeqArray() Parameter
`queue_name`	`queue_name`
`dequeue_options`	`deqopt`
`array_size`	`iters`
`message_properties_array`	`msgprop`
`payload_array`	`payload`
`msgid_array`	`msgid`
-	Note: `OCIAQDeqArray()` also requires the following additional parameters: `svch, errh, msgprop, payload_tdo, payload_ind, ctxp, deqcbfp, and flags`.

The following table compares parameters for the agent attributes:

Table 9–11 Agent Parameters

PL/SQL Agent Parameter	OCIAQAgent Attribute
`name`	`OCI_ATTR_AGENT_NAME`

Table 9–11 (Cont.) Agent Parameters

PL/SQL Agent Parameter	OCIAQAgent Attribute
address	OCI_ATTR_AGENT_ADDRESS
protocol	OCI_ATTR_AGENT_PROTOCOL

The following table compares parameters for the message properties:

Table 9–12 Message Properties

PL/SQL Message Property	OCIAQMsgProperties Attribute
priority	OCI_ATTR_PRIORITY
delay	OCI_ATTR_DELAY
expiration	OCI_ATTR_EXPIRATION
correlation	OCI_ATTR_CORRELATION
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
transaction_group	OCI_ATTR_TRANSACTION_NO
original_msgid	OCI_ATTR_ORIGINAL_MSGID
delivery_mode	OCI_ATTR_MSG_DELIVERY_MODE

The following table compares enqueue option attributes:

Table 9–13 Enqueue Option Attributes

PL/SQL Enqueue Option	OCIAQEnqOptions Attribute
visibility	OCI_ATTR_VISIBILITY
relative_msgid	OCI_ATTR_RELATIVE_MSGID
sequence_deviation	OCI_ATTR_SEQUENCE_DEVIATION (deprecated)

The following table compares dequeue option attributes:

Table 9–14 Dequeue Option Attributes

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

Table 9–14 (Cont.) Dequeue Option Attributes

PL/SQL Dequeue Option	OCIAQDeqOptions Attribute
correlation	OCI_ATTR_CORRELATION
delivery_mode	OCI_ATTR_MSG_DELIVERY_MODE

Note: OCIAQEnq() returns the error ORA-25219 while specifying the enqueue option OCI_ATTR_SEQUENCE along with OCI_ATTR_RELATIVE_MSGID. This happens when enqueuing two messages. For the second message, enqueue options OCI_ATTR_SEQUENCE and OCI_ATTR_RELATIVE_MSGID is set to dequeue this message before the first one. It does not return an error if you do not specify the 'sequence' but, of course, the message is not dequeued before the relative message.

OCIAQEnq() does not return an error if OCI_ATTR_SEQUENCE attribute is not set, but the message is not dequeued before the message with relative message Id.

Buffered Messaging

Buffered messaging is a nonpersistent messaging capability within Streams AQ, that was first available in Oracle Database 10*g* Release 2. Buffered messages reside in shared memory, and can be lost in the event of instance failure. Unlike persistent messages, no redo is written to disk. Buffered message enqueue and dequeue is much faster than persistent message operations. Since shared memory is limited, buffered messages may have to be spilled to disk. Flow control may be enabled to prevent applications from flooding the shared memory when the message consumers are slow or have stopped for some reason. The following functions are used for buffered messaging:

See Also:

- "OCIAQEnq()" on page 16-103
- "OCIAQEnqArray()" on page 16-105
- "OCIAQDeq()" on page 16-99
- "OCIAQDeqArray()" on page 16-101
- "OCIAQListen2()" on page 16-108

Enqueue Buffered Messaging Example

```
...
OCIAQMsgProperties  *msgprop;
OCIAQEnqueueOptions *enqopt;
message              msg;    /* message is an object type */
null_message         nmsg;   /* message indicator */
...
/* Allocate descriptors */
  OCIDescriptorAlloc(envhp, (dvoid **)&enqopt, OCI_DTYPE_AQENQ_OPTIONS, 0,
                     (dvoid **)0));

 OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,OCI_DTYPE_AQMSG_PROPERTIES, 0,
                    (dvoid **)0));
```

```
/* Set delivery mode to buffered */
 dlvm = OCI_MSG_BUFFERED;
 OCIAttrSet(enqopt,  OCI_DTYPE_AQENQ_OPTIONS, (dvoid *)&dlvm, sizeof(ub2),
            OCI_ATTR_MSG_DELIVERY_MODE, errhp);
/* Set visibility to Immediate (visibility must always be immediate for buffered
   messages) */
vis = OCI_ENQ_ON_COMMIT;

OCIAttrSet(enqopt, OCI_DTYPE_AQENQ_OPTIONS,(dvoid *)&vis, sizeof(ub4),
           OCI_ATTR_VISIBILITY, errhp)

/* Message was an object type created earlier, msg_tdo is its Type
   Descriptor Object */
OCIAQEnq(svchp, errhp, "Test_Queue", enqopt, msgprop, msg_tdo, (dvoid **)&mesg,
         (dvoid **)&nmesg, (OCIRaw **)0, 0));
...
```

Array Enqueue Buffered Messaging Example

```
...
#define NMESGS 10
OCIAQMsgProperties  *msgprop[NMESGS];
OCIAQEnqueueOptions *enqopt;
message              msg[NMESGS];    /* message is an object type  */
null_message         nmsg[NMESGS];   /* message indicator */
...
iters = NMESGS;

OCIDescriptorAlloc(envhp, (dvoid **)&enqopt, OCI_DTYPE_AQENQ_OPTIONS, 0,
                   (dvoid **)0));
/* visibility must be on_commit */
vis = OCI_ENQ_ON_COMMIT;
OCIAttrSet(enqopt, OCI_DTYPE_AQENQ_OPTIONS,(dvoid *)&vis, sizeof(ub4),
           OCI_ATTR_VISIBILITY, errhp)
/* delivery mode is buffered */
dlvm = OCI_MSG_BUFFERED;
OCIAttrSet(enqopt,  OCI_DTYPE_AQENQ_OPTIONS, (dvoid *)&dlvm, sizeof(ub2),
           OCI_ATTR_MSG_DELIVERY_MODE, errhp);
for (i = 0; i < NMESGS; i++)
{
OCIDescriptorAlloc(envhp, (dvoid **)&msgprop[i] ,OCI_DTYPE_AQMSG_PROPERTIES, 0,
                   (dvoid **)0));
/* initialize the message */
...
}

OCIAQEnqArray(svchp, errhp, "Test_Queue", enqopt,  &iters, msgprop, msg_tdo,
              (dvoid **)&msg,(dvoid **)&nmsg, (OCIRaw **)0, (dvoid *)0,
              (OCICallbackAQEnq)0, (ub4)0);
...
```

Dequeue Buffered Messaging Example

```
...
OCIAQMsgProperties  *msgprop;
OCIAQDequeueOptions *deqopt;
...
OCIDescriptorAlloc(envhp, (dvoid **)&mprop, OCI_DTYPE_AQMSG_PROPERTIES, 0,
                   (dvoid **)0));
OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
                   (dvoid **)0);
```

```
/* Set visibility to Immediate (visibility must always be immediate for buffered
   message operations) */
vis = OCI_ENQ_ON_COMMIT;
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,(dvoid *)&vis, sizeof(ub4),
           OCI_ATTR_VISIBILITY, errhp)
/* delivery mode is buffered */
dlvm  = OCI_MSG_BUFFERED;
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&dlvm,  sizeof(ub2),
           OCI_ATTR_MSG_DELIVERY_MODE, errhp);
/* set the consumer for which to dequeue the message (this needs to be specified
   regardless of the type of message being dequeued.
*/
consumer = "FIRST_SUBSCRIBER";
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)consumer,
           (ub4)strlen((char*)consumer), OCI_ATTR_CONSUMER_NAME, errhp);
/* Dequeue the message but don't return the payload (to simplify the code
   snippet)
*/
OCIAQDeq(svchp, errhp,  "test_queue", deqopt, msgprop, msg_tdo, (dvoid **)0,
         (dvoid **)0, (OCIRaw**)0, 0);
...
```

Array Dequeue Buffered Messaging Example

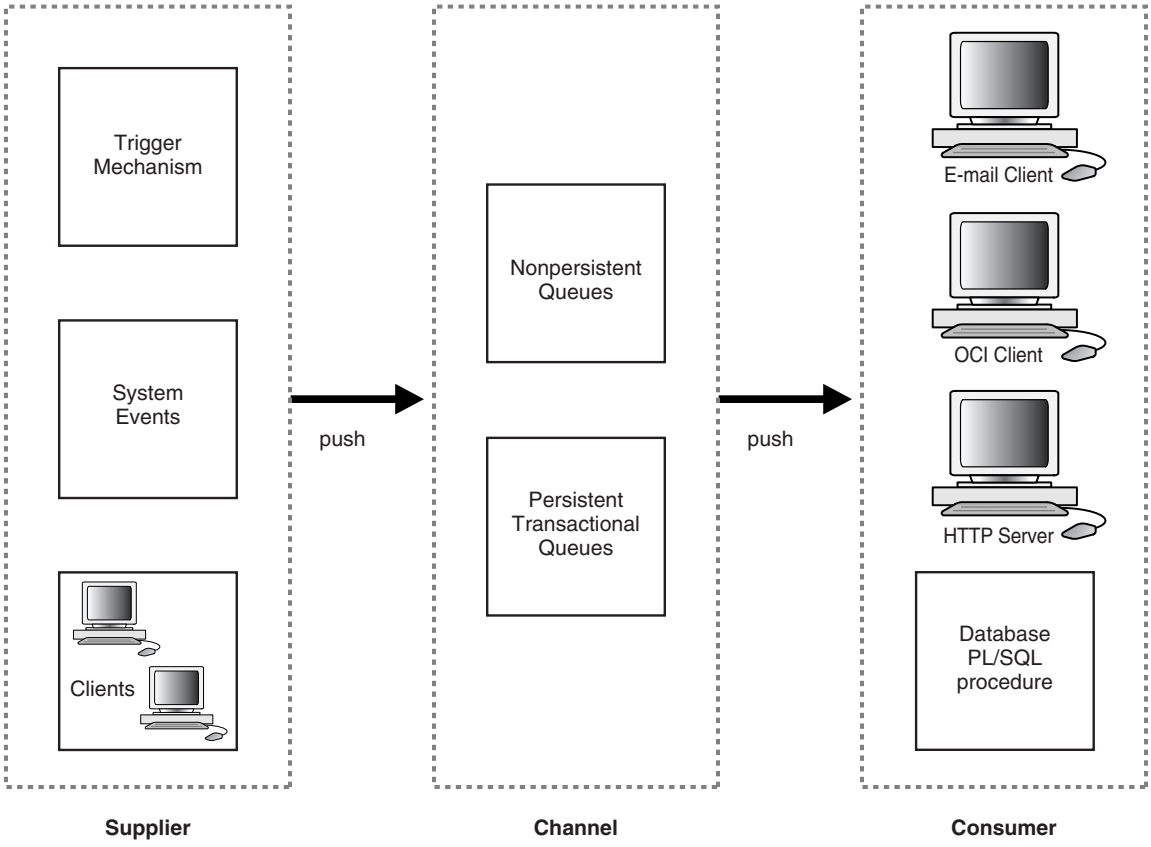
```
#define NMESGS 10
OCIAQMsgProperties  *msgprop[NMESGS];
OCIAQDequeueOptions *deqopt;
...

OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
                   (dvoid **)0);
/* Set visibility to Immediate (visibility must always be immediate for buffered
   message operations) */
vis = OCI_ENQ_ON_COMMIT;
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,(dvoid *)&vis, sizeof(ub4),
           OCI_ATTR_VISIBILITY, errhp)
/* delivery mode is buffered */
dlvm  = OCI_MSG_BUFFERED;
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&dlvm, sizeof(ub2),
           OCI_ATTR_MSG_DELIVERY_MODE, errhp);
/* set the consumer for which to dequeue the message (this needs to be specified
   regardless of the type of message being dequeued.
*/
consumer = "FIRST_SUBSCRIBER";
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)consumer,
           (ub4)strlen((char*)consumer), OCI_ATTR_CONSUMER_NAME, errhp);
for (i = 0; i < NMESGS; i++)
{
OCIDescriptorAlloc(envhp, (dvoid **)&msgprop[i] ,OCI_DTYPE_AQMSG_PROPERTIES, 0,
                   (dvoid **)0));
}
/* don't return message payload (to simplify the code snippet */
OCIAQDeqArray(svchp, errhp, "test_queue", (OCIAQDeqOptions *)deqopt,  &iters, 0,
              mesg_tdo,  (dvoid **)0, (dvoid **)0, 0, 0, 0, 0));
...
```

Publish-Subscribe Notification in OCI

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 preceding 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 Streams Advanced Queuing, see "OCI and Streams Advanced Queuing" on page 9-39
- For information on creating queues and about Streams AQ, including concepts, features, and examples, refer to *Oracle Streams Advanced Queuing User's Guide and Reference*
- For information on creating triggers, refer to the chapter on Commands in the *Oracle Database SQL Reference*.

Publish-Subscribe Registration Functions in OCI

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. The initialization parameter `COMPATIBLE` must be set to 8.1 or higher.

See Also:

- Detailed descriptions of the functions noted can be found in"Streams Advanced Queuing and Publish-Subscribe Functions" on page 16-98
- For examples of the use of these functions in an application, see "Publish-Subscribe Direct Registration Example" on page 9-53

Note: The publish-subscribe feature is only available on multithreaded operating systems.

---

1. Call `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. Call `OCIHandleAlloc()` with handle type `OCI_HTYPE_SUBSCRIPTION` to allocate a subscription handle.
3. 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 with
- OCI_ATTR_SUBSCR_RECPTPRES - presentation to receive notification with
- OCI_ATTR_SUBSCR_QOSFLAGS - QOS (quality of service) levels
- OCI_ATTR_SUBSCR_TIMEOUT - Registration timeout interval in seconds. The default is 0 if a timeout is not set.

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: "Subscription Handle Attributes" on page A-44

4. The values of QOS, timeout interval, namespace, and port are set:

   See Also: "Setting QOS, Timeout Interval, Namespace, and Port Number" on page 9-51

5. 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 in OCI" on page 9-52

6. 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

7. Call 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 Database 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 is authorized to register for events in the database, grant enterprise role ENTERPRISE_AQ_USER_ROLE.

Using OCI to Open 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 (user name and password)
   - `OCI_ATTR_WALL_LOC`: for SSL authentication, the location of the client wallet
   - `OCI_ATTR_LDAP_AUTH`: the authentication method code

     See Also: "Environment Handle Attributes" on page A-2 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_QOSFLAGS` - QOS (quality of service) levels
   - `OCI_ATTR_SUBSCR_TIMEOUT` - Registration timeout interval in seconds. The default is 0 if a timeout is not set.
   - `OCI_ATTR_SUBSCR_SERVER_DN`: the descriptor handles you populated in step 5
7. The values of QOS, timeout interval, namespace, and port are set:

   See Also: "Setting QOS, Timeout Interval, Namespace, and Port Number" on page 9-51

8. 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.

Setting QOS, Timeout Interval, Namespace, and Port Number

You can set QOSFLAGS to the following QOS levels using `OCIAttrSet()`:

- `OCI_SUBSCR_QOS_RELIABLE` - Reliable notification persists across instance and database restarts. Reliability is of the server only and is only for persistent queues or buffered messages. This option describes the persistence of the notifications. Registrations are persistent by default.
- `OCI_SUBSCR_QOS_PURGE_ON_NTFN` - Once received, purge registration on first notification. (Subscription is unregistered.)

```
/* Set QOS levels */
ub4 qosflags = OCI_SUBSCR_QOS_RELIABLE | OCI_SUBSCR_QOS_PURGE_ON_NTFN;

/* Set flags in subscription handle */
(void)OCIAttrSet((dvoid *)subscrhp, (ub4)OCI_HTYPE_SUBSCRIPTION,
                 (dvoid *)&qosflags, (ub4)0, (ub4)OCI_ATTR_SUBSCR_QOSFLAGS, errhp);

/* Set auto-expiration after 30 seconds */
ub4 timeout = 30;
(void)OCIAttrSet((dvoid *)subscrhp, (ub4)OCI_HTYPE_SUBSCRIPTION,
                   (dvoid *)&timeout, (ub4)0, (ub4)OCI_ATTR_SUBSCR_TIMEOUT, errhp);
```

The registration is purged when the timeout is exceeded, and a notification is sent to the client, so that the client can invoke its callback and take any necessary action. In the case of client failure before the timeout, the registration is purged.

You can set the port number on the environment handle, which is important if the client is on a system behind a firewall that is able to receive notifications only on certain ports. Clients can specify the port for the listener thread before the first registration, using an attribute in the environment handle. The thread is started the first time `OCISubscriptionRegister()` is called. If available, this specified port number will be used. An error is returned if the client tries to start another thread on a different port using a different environment handle.

```
ub4 port = 1581;
(void)OCIAttrSet((dvoid *)envhp, (ub4)OCI_HTYPE_ENV, (dvoid *)&port, (ub4)0,
                 (ub4)OCI_ATTR_SUBSCR_PORTNO, errhp);
```

If instead, the port is determined automatically, you can get the port number at which the client thread is listening for notification by obtaining the attribute from the environment handle.

```
(void)OCIAttrGet((dvoid *)subhp, (ub4)OCI_HTYPE_ENV, (dvoid *)&port,
                 (ub4)0, (ub4)OCI_ATTR_SUBSCR_PORTNO, errhp);
```

OCI Functions Used to Manage Publish-Subscribe Notification

The following functions are used to manage publish-subscribe notification.

Table 9–15 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 in OCI

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: "Subscription Handle Attributes" on page A-44

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: "OCI and Streams Advanced Queuing" on page 9-39

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.

See Also:

- See "Subscription Handle Attributes" on page A-44.
- "Oracle Streams AQ PL/SQL Callback" in *Oracle Database PL/SQL Packages and Types Reference* for the PL/SQL procedure specification.

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. The initialization parameter _SYSTEM_TRIG_ENABLED must be set to TRUE (the default) to enable triggers for system events.

```
-------------------------------------------------------
----create queue table for persistent multiple consumers
-------------------------------------------------------
connect pubsub/pubsub;
---- 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;
/
-------------------------------------------------------
---- Create a persistent queue for publishing messages
-------------------------------------------------------
---- 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;
/
-------------------------------------------------------
---- Start the queue
-------------------------------------------------------
begin
  DBMS_AQADM.START_QUEUE('pubsub.logon');
end;
/
-------------------------------------------------------
---- define new_enqueue for convenience
-------------------------------------------------------
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;
/
-------------------------------------------------------
---- add subscriber with rule based on current user name,
---- using correlation_id
-------------------------------------------------------
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 = ''ix'' ');
end;
/
-------------------------------------------------------
---- create a trigger on logon on database
-------------------------------------------------------
---- 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;
/

-------------------------------------------------------
---- create a PL/SQL callback for notification of logon
---- of user 'ix' on database
-------------------------------------------------------
----
create or replace procedure plsqlnotifySnoop(
  context raw, reginfo sys.aq$_reg_info, descr sys.aq$_descriptor,
  payload raw, payloadl number)
as
begin
 dbms_output.put_line('Notification : User ix 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.

```
...
static ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;

static OCISubscription *subscrhpSnoop = (OCISubscription *)0;
static OCISubscription *subscrhpSnoopMail = (OCISubscription *)0;
static OCISubscription *subscrhpSnoopServer = (OCISubscription *)0;

/* callback function for notification of logon of user 'ix' on database */

static ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
    dvoid *ctx;
    OCISubscription *subscrhp;
    dvoid *pay;
    ub4 payl;
    dvoid *desc;
    ub4 mode;
{
    printf("Notification : User ix Logged on\n");
  (void)OCIHandleFree((dvoid *)subscrhpSnoop,
            (ub4) OCI_HTYPE_SUBSCRIPTION);
    return 1;
```

```
}

static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  ub4 buflen;
  sb4 errcode;

  if (status == OCI_SUCCESS) return;

  switch (status)
  {
  case OCI_SUCCESS_WITH_INFO:
    printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    printf("Error - OCI_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("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    printf("Error - OCI_STILL_EXECUTING\n");
    break;
  case OCI_CONTINUE:
    printf("Error - OCI_CONTINUE\n");
    break;
  default:
    printf("Error - %d\n", status);
    break;
  }
}

static 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);

    /* 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);

    printf("Begining Registration for subscription %s\n", subscriptionName);
    checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,
        OCI_DEFAULT));
   printf("done\n");

}


int main( argc, argv)
int    argc;
char * argv[];
{
    OCISession *authp = (OCISession *) 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*/
    (char*) "PUBSUB.LOGON:SNOOP", /* 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 */
     (char*) "PUBSUB.LOGON:SNOOP",              /* subscription name */
                                                /* <queue_name>:<agent_name> */
        (dvoid*)0, /* callback function */
        OCI_SUBSCR_PROTO_MAIL, /* receive with protocol */
        (char*)  "longying.zhao@oracle.com", /* recipient address */
        OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

/* Register for server to server notification */
    initSubscriptionHn(    &subscrhpSnoopServer, /* subscription handle */
       (char*)  "PUBSUB.LOGON:SNOOP",            /* subscription name */
                                                 /* <queue_name>:<agent_name> */
        (dvoid*)0, /* callback function */
        OCI_SUBSCR_PROTO_SERVER, /* receive with protocol */
         (char*) "pubsub.plsqlnotifySnoop", /* recipient address */
        OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

    checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) OCI_DEFAULT));

/************************************************
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);
}
```

If user `IX` 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);
}
...
```

Database Change Notification

Database Change Notification is a feature that enables client applications to register queries with the database and receive notifications in response to DML or DDL changes on the objects associated with the queries. The notifications are published by the database when the DML or DDL transaction commits.

During registration, the application specifies a notification handler and associates a set of interesting queries with the notification handler. A notification handler can be either a server side PL/SQL procedure or a client side C callback. Registrations are created on all objects referenced during the execution of the queries. The notification handler is invoked when a transaction subsequently changes any of the registered objects and commits.

> **See Also:** *Oracle Database Application Developer's Guide - Fundamentals*, chapter 13, "Developing Applications with Database Change Notification"

One use of this feature is in middle-tier applications that need to have cached data and keep the cache as recent as possible with respect to the back-end database.

The contents of the notification includes the following information:

- Names of the modified objects
- Operation type (`INSERT`, `UPDATE`, `DELETE`, `ALTER TABLE`, `DROP TABLE`)
- ROWIDs of the changed rows and the associated operation (`INSERT`, `UPDATE`, `DELETE`)
- Global database events (`STARTUP`, `SHUTDOWN`)

Registering for Database Change Notification

The calling session must have the `CHANGE NOTIFICATION` system privilege and `SELECT` privileges on all objects that it attempts to register. A registration is a persistent entity that is recorded in the database, and is visible to all instances of Real Application Clusters (RAC). Transactions that modify registered objects in any instance of the RAC generate notification.

The objects referenced are identified during the execution of the queries. If the query refers to a non-materialized view, then the underlying tables on top of which the view is based are registered. Queries based on materialized views are not supported.

The registration interface employs a callback to respond to changes in underlying objects of a query and uses a namespace extension to AQ, `DBCHANGE`.

The steps in writing the registration are:

- The environment must be created in OCI_EVENTS and OCI_OBJECT mode.
- The subscription handle attribute OCI_ATTR_SUBSCR_NAMESPACE must be set to namespace OCI_SUBSCR_NAMESPACE_DBCHANGE.
- The subscription handle attribute OCI_ATTR_SUBSCR_CALLBACK is used to store the OCI callback associated with the query handle. The callback has the following prototype:

```
notification_callback (dvoid *ctx, OCISubscription *subscrhp, dvoid *payload,
                       ub4 paylen, dvoid *desc, ub4 mode)
```

  The parameters are described in "Notification Callback in OCI" on page 9-52.

- You can optionally associate a client-specific context using OCI_ATTR_SUBSCR_CTX.
- OCI_ATTR_SUBSCR_TIMEOUT can be set to specify a ub4 timeout interval in seconds. If not set, there is no timeout.
- If OCI_SUBSCR_QOS_PURGE_ON_NTFN is set, the registration is purged on the first notification.
- If OCI_SUBSCR_QOS_RELIABLE is set, notifications are persistent. Surviving instances of a RAC can be used to send and retrieve change notification messages, even after a node failure because invalidations associated with this registration are queued persistently into the database. If FALSE, then invalidations are enqueued into a fast in-memory queue. Note that this option describes the persistence of notifications and not the persistence of registrations. Registrations are automatically persistent by default.
- Call OCISubscriptionRegister() to create a new registration in the DBCHANGE namespace.
- Multiple query statements can be associated with the subscription handle by setting the attribute OCI_ATTR_CHNF_REGHANDLE of the statement handle, OCI_HTYPE_STMT. The registration is completed when the query is executed.
- Optionally, to unregister a subscription, the client can call the OCISubscriptionUnRegister() function with the subscription handle as a parameter.

A binding of a statement handle to a subscription handle is only valid for the first execution of a query. If the application needs to use the same OCIStatement handle for subsequent executions, it must repopulate the registration handle attribute of the statement handle. A binding of a subscription handle to a statement handle is only permitted when the statement is a query (determined at execute time). If a DML statement is executed as part of the execution, then an exception is issued.

Subscription Handle Attributes for Change Notification

The subscription handle attributes for change notification are described next. The attributes can be divided into generic (which are common to all subscriptions) and namespace- specific attributes particular to change notification. The IN attributes on the statement handle can only be modified before the registration is created.

OCI_ATTR_SUBSCR_NAMESPACE (generic) - This must be set to OCI_SUBSCR_NAMESPACE_DBCHANGE for subscription handles.

OCI_ATTR_SUBSCR_CALLBACK (generic) - Use to store the callback associated with the subscription handle. The callback is executed when a notification is received.

When a new change notification message becomes available, the callback is invoked in the listener thread with desc pointing to a descriptor of type OCI_DTYPE_CHDES which contains detailed information about the invalidation.

For setting OCI_ATTR_SUBSCR_QOSFLAGS, a generic flag, a parallel example is presented here:

> **See Also:** "Setting QOS, Timeout Interval, Namespace, and Port Number" on page 9-51

OCI_ATTR_CHNF_TABLENAMES (datatype is (OCIColl *)) attributes provided to retrieve the list of table names that were registered. These attributes are available from the subscription handle, after the query is executed.

OCI_ATTR_CHNF_ROWIDS is a boolean attribute (default FALSE). If set to TRUE, then the change notification message includes row level details such as operation type and ROWID.

OCI_ATTR_CHNF_OPERATIONS - This is a ub4 flag that can be used to selectively filter notifications based on operation type. Flags stored are:

- OCI_OPCODE_ALL - All operations
- OCI_OPCODE_INSERT - Insert operations on the table
- OCI_OPCODE_UPDATE - Update operations on the table
- OCI_OPCODE_DELETE - Delete operations on the table

OCI_ATTR_CHNF_CHANGELAG - This is a ub4 value that can be used by the client to specify the number of transactions by which the client is willing to lag behind. This option can be used by the client as a throttling mechanism for change notification messages. When this option is chosen, ROWID level granularity of information will not be available in the notifications even if OCI_ATTR_CHNF_ROWID was set to TRUE

Once the OCISubscriptionRegister() call is invoked, none of the above attributes can be subsequently modified on the registration already created. Any attempt to modify those attributes is not reflected on the registration already created, but it does take effect on newly created registrations that use the same registration handle.

Change Notification Descriptor

The change notification descriptor is passed into the desc parameter of the notification callback specified by the application. The following attributes are specific to change notification. The OCI type constant of the change notification descriptor is OCI_DTYPE_CHDES.

OCI_ATTR_CHDES_DBNAME (OCIText *) - Name of the database (source of the change notification)

OCI_ATTR_CHDES_NFYTYPE - Flags describing the notification type:

- OCI_EVENT_NONE - No further information about the change notification
- OCI_EVENT_STARTUP - Startup
- OCI_EVENT_SHUTDOWN - Instance shutdown

- OCI_EVENT_SHUTDOWN_ANY - Any instance shutdown - Real Application Clusters (RAC)
- OCI_EVENT_DEREG - Unregistered or timed out
- OCI_EVENT_OBJCHANGE - Object change notification

OCI_ATTR_CHDES_TABLE_CHANGES - A collection type describing operations on tables of datatype (OCIColl *). Each element of the collection is a table change descriptor of type OCI_DTYPE_TABLE_CHDES which has the following attributes:

- OCI_ATTR_CHDES_TABLE_NAME (OCIText *) - Schema annotated table name.
- OCI_ATTR_CHDES_TABLE_OPFLAGS (ub4) - Flag field describing the operations on the table. Each of the following flag fields is in a separate bit position in the attribute:
  - OCI_OPCODE_ALLROWS - The table is completely invalidated.
  - OCI_OPCODE_INSERT- Insert operations on the table.
  - OCI_OPCODE_UPDATE - Update operations on the table.
  - OCI_OPCODE_DELETE - Delete operations on the table.
  - OCI_OPCODE_ALTER - Table altered (schema change) - this includes DDL statements and internal operations that cause row migration.
  - OCI_OPCODE_DROP - Table dropped.
- OCI_ATTR_CHDES_TABLE_ROW_CHANGES - This is an embedded collection describing the changes to the rows within the table. Each element of the collection is a row change descriptor of type OCI_DTYPE_ROW_CHDES which has the following attributes:
  - OCI_ATTR_CHDES_ROW_ROWID (OraText *) - String representation of a ROWID.
  - OCI_ATTR_CHDES_ROW_OPFLAGS - Reflects the operation type: INSERT, UPDATE, DELETE, or OTHER.

Database Change Notification Example

The following is a simple OCI program. See the comments.

```
/*************************************************************************
 The following is an example of an OCI program that creates a registration
 involving the HR.EMPLOYEES and HR.DEPARTMENTS tables and waits to
 receive a Notification. After compilation the program can be executed
 at the command prompt as
 ./regquery

  Sample output after registration:
  --------------------------------
  Allocated handles
  Created Registration
  Registered query SELECT MANAGER_ID from EMPLOYEES where EMPLOYEE_ID=206
  Registered query SELECT department_id  from DEPARTMENTS where DEPARTMENT_NAME =
   'Payroll'
  Waiting for Notification to arrive.

  Now we can update the EMPLOYEES table and receive the notification. From
  sqlplus perform the following
```

```
  UPDATE EMPLOYEES SET SALARY=SALARY WHERE EMPLOYEE_ID = 200;
  COMMIT;

  Program output from notification handler:
  ----------------------------------------
  Received Notification
  EMPLOYEE table modified
  Number of rows modified is 1
  HR.EMPLOYEES table has been modified in row AAAKCeAABAAAKC1AAC
  Executing stmt select * from HR.EMPLOYEES where rowid='AAAKCeAABAAAKC1AAC'
  First column of modified row is 200

*******************************************************************************/

#ifndef S_ORACLE
# include<s.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>

#define MAXSTRLENGTH 1024

static int notifications_processed =0;
static void checker();



int main(int argc, char **argv)
{
  NotificationDriver();
}


void checker(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;
  int retval = 1;

  switch (status)
  {
  case OCI_SUCCESS:
    retval = 0;
    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;
  }
}

void my_notification_callback(ctx, subscrhp, payload, payl, descriptor, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *payload;
ub4 *payl;
dvoid *descriptor;
ub4 mode;
{
  dvoid *change_descriptor =  descriptor;
  ub4   notify_type;
  OCIEnv *envhp;
  OCIError *errhp;
  OCIServer *srvhp;
  OCISvcCtx *svchp;
  OCISession *usrhp;
  OCIStmt *stmthp;
  dvoid *tmp;

  dvoid *elemind = (dvoid *)0;
  OCIColl *table_changes = (OCIColl  *)0 ;
                 /* Collection of pointers to table chg descriptors */
  dvoid **table_descp;          /* Pointer to Table Change Descriptor */
  dvoid *table_desc;              /*  Table Change Descriptor */
  ub4 num_rows = 0;
  ub4 table_op;
  ub4 num_tables = 0;
  ub2 i, j;
  boolean exist;
  text *table_name;

  printf("Received Notification\n");



  /* Initialize environment and allocate Error Handle.
     Note that the environment has to be initialized in object mode
     since we might be operating on collections.
  */
  OCIEnvCreate( (OCIEnv **) &envhp, OCI_OBJECT, (dvoid *)0,
                (dvoid * (*)(dvoid *, size_t)) 0,
                (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                (void (*)(dvoid *, dvoid *)) 0,
                (size_t) 0, (dvoid **) 0 );
```

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                (size_t) 0, (dvoid **) 0);

/* Get the Notification Type */
checker(errhp,
        OCIAttrGet( change_descriptor, OCI_DTYPE_CHDES, &notify_type, NULL,
              OCI_ATTR_CHDES_NFYTYPE, errhp));
if (notify_type == OCI_EVENT_SHUTDOWN)
  printf("Shutdown Notification\n");
else if (notify_type == OCI_EVENT_DEREG)
  printf("Registration Removed\n");

if (notify_type != OCI_EVENT_OBJCHANGE)
{
  OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
  OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR);
  return;
}

/* The below code is only executed if the notification is of type
   OCI_EVENT_OBJCHANGE
 */
/* server contexts */
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp,
                          OCI_HTYPE_SERVER,
                 (size_t) 0, (dvoid **) 0);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp,
                           OCI_HTYPE_SVCCTX,
                            (size_t) 0, (dvoid **) 0);

/* Allocate a statement handle */
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
               (ub4) OCI_HTYPE_STMT, 52, (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);

checker(errhp, OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0,
        (ub4) OCI_DEFAULT));

 /* allocate a SESSION  handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                  (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
           (dvoid *)((text *)"HR"),
           (ub4)strlen((char *)"HR"),  OCI_ATTR_USERNAME, errhp);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
           (dvoid *)((text *)"HR"), (ub4)strlen((char *)"HR"),
           OCI_ATTR_PASSWORD, errhp);

checker(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);

/* Obtain the collection of table change descriptors */
```

```
 checker(errhp,OCIAttrGet(change_descriptor, OCI_DTYPE_CHDES, &table_changes,
                          NULL, OCI_ATTR_CHDES_TABLE_CHANGES, errhp));
/* Obtain the size of the collection (i.e number of tables modified) */
 if (table_changes)
   checker(errhp,OCICollSize(envhp, errhp, (CONST OCIColl *) table_changes,
           &num_tables));
  else
    num_tables =0;

 /* For each element of the collection, extract the table name of the modified
    table */
 for (i=0; i < num_tables; i++)
 {
   OCIColl *row_changes = (OCIColl  *)0;
   /* Collection of pointers to row chg. Descriptors */
   dvoid **row_descp;            /*  Pointer to Row Change Descriptor */
   dvoid   *row_desc;               /*   Row Change Descriptor */
   text *row_id;
   ub4 rowid_size;
   text *ocistmt;
   OCIDefine *defnp1 = (OCIDefine *)0;
   char *outstr;

   checker(errhp,OCICollGetElem(envhp, errhp, (OCIColl *) table_changes, i,
                                &exist, &table_descp, &elemind));

   table_desc = *table_descp;
   checker(errhp,OCIAttrGet(table_desc, OCI_DTYPE_TABLE_CHDES, &table_name,
                            NULL,
                            OCI_ATTR_CHDES_TABLE_NAME, errhp));
   if (strcmp(table_name, "HR.EMPLOYEES") == 0)
     printf("EMPLOYEE table modified \n");
   else if (strcmp(table_name, "HR.DEPARTMENTS") == 0)
     printf("DEPARTMENTS table modified \n");

   checker(errhp,OCIAttrGet (table_desc, OCI_DTYPE_TABLE_CHDES,
                             (dvoid *)&table_op, NULL,
                              OCI_ATTR_CHDES_TABLE_OPFLAGS, errhp));

   /* If the ROWID granularity of info not available, move-on. Rowids
      can be rolled up into a full table notification if too many rows
      were updated on a single table or insufficient shared memory on
      the server side to hold rowids
    */
   if (table_op & OCI_OPCODE_ALLROWS)
   {
      printf("Full Table Invalidation\n");
      continue;
    }

    /* Obtain the collection of  ROW CHANGE descriptors */
   checker(errhp,OCIAttrGet (table_desc, OCI_DTYPE_TABLE_CHDES, &row_changes,
                             NULL, OCI_ATTR_CHDES_TABLE_ROW_CHANGES, errhp));

   if (row_changes)
     checker(errhp,OCICollSize(envhp, errhp, row_changes, &num_rows));
   else
     num_rows =0;

   printf ("Number of rows modified is %d\n", num_rows);
```

```
    fflush(stdout);
    for (j=0; j<num_rows; j++)
    {
      OCICollGetElem(envhp, errhp, (OCIColl *) row_changes,
                     j, &exist, &row_descp, &elemind);
      row_desc = *row_descp;

      OCIAttrGet (row_desc, OCI_DTYPE_ROW_CHDES, (dvoid *)&row_id,
                  &rowid_size, OCI_ATTR_CHDES_ROW_ROWID, errhp);
      printf ("%s table has been modified in row %s \n", table_name, row_id);
      fflush(stdout);
      ocistmt = (text *)malloc(MAXSTRLENGTH*sizeof(char));

      /* QUERY FROM DATABASE TO VIEW CONTENTS OF CHANGED ROW */
      sprintf (ocistmt, "select * from %s where rowid='%s'", table_name, row_id);
      printf("Executing stmt %s\n", ocistmt);

      /* prepare query statement*/
      checker(errhp,OCIStmtPrepare(stmthp, errhp, ocistmt,
                                   (ub4)strlen((char *)ocistmt),
                                   (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));
      outstr = (char *)malloc(MAXSTRLENGTH*sizeof(char));

      checker(errhp,OCIDefineByPos(stmthp, &defnp1, errhp, 1, (dvoid *)outstr,
                                   MAXSTRLENGTH * sizeof(char),
                                   SQLT_STR, (dvoid *)0, (ub2 *)0, (ub2 *)0,
                                   OCI_DEFAULT));

      /* execute the statement */
      checker(errhp,OCIStmtExecute(svchp, stmthp, errhp, (ub4)1, (ub4) 0,
                                   (CONST OCISnapshot *) NULL,
                                   (OCISnapshot *) NULL, OCI_DEFAULT));

      printf("First column of modified row is %s\n", outstr);
    }  /* Loop for j in 1..numrows */

  } /* Loop for I in 1..numtables */

  /* End session and detach from server */
  checker(errhp, OCISessionEnd(svchp, errhp, usrhp, OCI_DEFAULT));
  checker(errhp, OCIServerDetach(srvhp, errhp, OCI_DEFAULT));
   if (stmthp)
    OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT);
  if (errhp)
    OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR);
  if (srvhp)
    OCIHandleFree((dvoid *)srvhp, OCI_HTYPE_SERVER);
  if (svchp)
    OCIHandleFree((dvoid *)svchp, OCI_HTYPE_SVCCTX);
  if (usrhp)
    OCIHandleFree((dvoid *)usrhp, OCI_HTYPE_SESSION);
  if (envhp)
    OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);

  notifications_processed++;
}  /* End function my_notification_callback */


/* The following routine creates registrations and waits for notifications. */
void NotificationDriver()
```

```
{
  OCISvcCtx *svchp;
  OCIError *errhp;
  OCISession *usrhp;
  OCIStmt *stmthp;
  OCIEnv *envhp;
  OCIServer *srvhp;
  text *username;
  OCISubscription *subscrhp;
  ub4 namespace = OCI_SUBSCR_NAMESPACE_DBCHANGE;
  ub4 timeout = 1800;
  text dname[MAXSTRLENGTH];
  OCIDefine *defnp1 = (OCIDefine *)0;
  OCIDefine *defnp2 = (OCIDefine *)0;
  OCIDefine *defnp3 = (OCIDefine *)0;
  OCIDefine *defnp4 = (OCIDefine *)0;
  OCIDefine *defnp5 = (OCIDefine *)0;
  int mgr_id =0;
  int dept_id =0;
  dvoid *tmp;
  boolean rowids_needed = TRUE;

  char query_text1[] = "SELECT MANAGER_ID from EMPLOYEES where EMPLOYEE_ID=206";
  char query_text2[] =
  "SELECT department_id  from DEPARTMENTS where DEPARTMENT_NAME = 'Payroll'";
  /* DEPARTMENTS  is also a cached object */

  /* Initialize the environment. The environment has to be initialized
     with OCI_EVENTS and OCI_OBJECTS to create a change notification
     registration and receive notifications.
  */
  OCIEnvCreate( (OCIEnv **) &envhp, OCI_EVENTS|OCI_OBJECT, (dvoid *)0,
                    (dvoid * (*)(dvoid *, size_t)) 0,
                    (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                    (void (*)(dvoid *, dvoid *)) 0,
                    (size_t) 0, (dvoid **) 0 );

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                         (size_t) 0, (dvoid **) 0);
   /* server contexts */
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                        (size_t) 0,
                        (dvoid **) 0);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                         (size_t) 0, (dvoid **) 0);
   checker(errhp,OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0,
           (ub4) OCI_DEFAULT));
   OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                  52, (dvoid **)0);
  /* 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 context handle */
  OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                                (size_t) 0, (dvoid **) 0);

  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
             (dvoid *)((text *)"HR"), (ub4)strlen((char *)"HR"),
              OCI_ATTR_USERNAME, errhp);
```

```
 OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
            (dvoid *)((text *)"HR"), (ub4)strlen((char *)"HR"),
             OCI_ATTR_PASSWORD, errhp);
   checker(errhp,OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
           OCI_DEFAULT));
    /* Allocate a statement handle */
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                                (ub4) OCI_HTYPE_STMT, 52, (dvoid **) &tmp);

  OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp, (ub4)0,
                       OCI_ATTR_SESSION, errhp);

  /* allocate subscription handle */
  OCIHandleAlloc ((dvoid *) envhp, (dvoid **) &subscrhp, OCI_HTYPE_SUBSCRIPTION,
                  (size_t) 0,
                   (dvoid **) 0);
  printf("Allocated handles\n");

  /* set the namespace to DBCHANGE */
  OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,  (dvoid *) &namespace,
              sizeof(ub4),
              OCI_ATTR_SUBSCR_NAMESPACE, errhp);
  /* Associate a notification callback */
  OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
              (void *)my_notification_callback,  0,
              OCI_ATTR_SUBSCR_CALLBACK, errhp);

  /* Allow extraction of rowid information */
  checker(errhp, OCIAttrSet (subscrhp, OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *)&rowids_needed, sizeof(ub4),
                  OCI_ATTR_CHNF_ROWIDS, errhp));

  /* Can optionally provide a client specific context using
     OCI_ATTR_SUBSCR_CTX */

  /* Set a timeout value of half an hour */
  OCIAttrSet(subscrhp, OCI_HTYPE_SUBSCRIPTION,
             (dvoid *)&timeout, 0, OCI_ATTR_SUBSCR_TIMEOUT, errhp);

  /* Create a new registration in the  DBCHANGE namespace */
  checker(errhp,OCISubscriptionRegister(svchp, &subscrhp, 1, errhp, OCI_DEFAULT));

  printf("Created Registration\n");
/* Multiple queries can now be associated with the subscription */

  /* Prepare the statement */
  checker(errhp,OCIStmtPrepare (stmthp, errhp, query_text1,
                                strlen(query_text1), OCI_V7_SYNTAX, OCI_DEFAULT));
  checker(errhp,OCIDefineByPos(stmthp, &defnp1, errhp, 1, (dvoid *)&mgr_id,
                               sizeof(mgr_id), SQLT_INT, (dvoid *)0,
                               (ub2 *)0, (ub2 *)0, OCI_DEFAULT));
  /* Associate the statement with the subscription handle */
  checker(errhp,OCIAttrSet (stmthp, OCI_HTYPE_STMT, subscrhp, 0,
                            OCI_ATTR_CHNF_REGHANDLE, errhp));

  /* Execute the statement The execution of the statement  performs the object
 registration */
  checker(errhp,OCIStmtExecute (svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                 (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL ,
```

```
                OCI_DEFAULT));
  printf("Registered query %s\n", query_text1);

  /* Use the same registration for the departments table */
  checker(errhp,OCIStmtPrepare (stmthp, errhp, query_text2,
               strlen(query_text2), OCI_V7_SYNTAX, OCI_DEFAULT));

  checker(errhp,OCIDefineByPos(stmthp, &defnp3,
              errhp, 1, (dvoid *)&dept_id, sizeof(dept_id),
               SQLT_INT, (dvoid *)0, (ub2 *)0, (ub2 *)0, OCI_DEFAULT));
  checker(errhp,OCIAttrSet (stmthp, OCI_HTYPE_STMT, subscrhp, 0,
                    OCI_ATTR_CHNF_REGHANDLE, errhp));
  checker(errhp,OCIStmtExecute (svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL ,
                OCI_DEFAULT));
  printf("Registered query %s\n", query_text2);

  printf("Waiting for Notifications to arrive\n");
  /* Wait for notifications to arrive */
  while (notifications_processed != 1);

  /* Unregister the subscription */
  checker(errhp,
           OCISubscriptionUnRegister(svchp,subscrhp, errhp, OCI_DEFAULT));

  /* End the session and detach from the server */
  checker(errhp, OCISessionEnd(svchp, errhp, usrhp, (ub4) 0));
  checker(errhp, OCIServerDetach(srvhp, errhp, (ub4) OCI_DEFAULT));

  /* Free all the handles */
  OCIHandleFree((dvoid *)subscrhp, OCI_HTYPE_SUBSCRIPTION);
  OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT);
  OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER);
  OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX);
  OCIHandleFree((dvoid *) usrhp, (ub4) OCI_HTYPE_SESSION);
  OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR);
  OCIHandleFree((dvoid *) envhp, (ub4) OCI_HTYPE_ENV);
}
```

Database Startup and Shutdown

The OCI functions, `OCIDBStartup()` and `OCIDBShutdown()`, provide the minimal interface needed to start up and shut down an Oracle database. Before calling `OCIDBStartup()`, the C program must connect to the server and start a `SYSDBA` or `SYSOPER` session in the preliminary authentication mode. This mode is the only one permitted when the instance is not up and it is used only to bring up the instance. A call to `OCIDBStartup()` will start up one server instance without mounting or opening the database. To mount and open the database, end the preliminary authentication session and start a regular `SYSDBA` or `SYSOPER` session to execute the appropriate `ALTER DATABASE` statements.

An active `SYSDBA` or `SYSOPER` session is needed to shut down the database. For all modes other than `OCI_DBSHUTDOWN_ABORT`, make two calls to `OCIDBShutdown()`: one to initiate shutdown by prohibiting further connections to the database, followed by the appropriate `ALTER DATABASE` commands to dismount and close it; and the other call to finish shutdown by bringing the instance down. In special circumstances, to shut down the database as fast as possible, just call `OCIDBShutdown()` in the `OCI_DBSHUTDOWN_ABORT` mode, which is equivalent to `SHUTDOWN ABORT` in SQL*Plus.

Both of these functions require a dedicated connection to the server. ORA-106 will be signaled if an attempt is made to start up or shut down the database when connected to a shared server through a dispatcher.

The OCIAdmin administration handle datatype is used to make the interface extensible. OCIAdmin is associated with the handle type OCI_HTYPE_ADMIN. Passing a value for the OCIAdmin parameter, admhp, is optional for OCIDBStartup() and is not needed by OCIDBShutdown().

See Also:

- "OCIDBStartup()" on page 15-12
- "OCIDBShutdown()" on page 15-10
- "Administration Handle Attributes" on page A-18
- *Oracle Database Administrator's Guide*

Examples of Startup and Shutdown in OCI

To do a startup, you must be connected to the database as SYSOPER or SYSDBA in OCI_PRELIM_AUTH mode. You cannot be connected to a shared server through a dispatcher. To use a client-side parameter file (pfile), the attribute OCI_ATTR_ADMIN_PFILE must be set in the administration handle using OCIAttrSet(); otherwise, a server-side parameter file (spfile) will be used. In the latter case, pass (OCIAdmin *)0. A call to OCIDBStartup() starts up one instance on the server.

The following sample code uses a client-side parameter file (pfile) that is set in the administration handle:

```
...

/*  Example 0 - Startup:  */
OCIAdmin *admhp;
text *mount_stmt = (text *)"ALTER DATABASE MOUNT";
text *open_stmt = (text *)"ALTER DATABASE OPEN";
text *pfile = (text *)"/ade/viewname/oracle/work/t_init1.ora";

/* Start the authentication session */
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp,
         OCI_CRED_RDBMS, OCI_SYSDBA|OCI_PRELIM_AUTH));

/* Allocate admin handle for OCIDBStartup */
checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **) &admhp,
         (ub4) OCI_HTYPE_ADMIN, (size_t) 0, (dvoid **) 0));

/* Set attribute pfile in the admin handle
(do not do this if you want to use the spfile) */
checkerr (errhp, OCIAttrSet( (dvoid *) admhp, (ub4) OCI_HTYPE_ADMIN,
          (dvoid *) pfile, (ub4) strlen(pfile),
          (ub4) OCI_ATTR_ADMIN_PFILE, (OCIError *) errhp));

/* Startup in NOMOUNT mode */
  checkerr(errhp, OCIDBStartup(svchp, errhp, admhp, OCI_DEFAULT, 0));
   checkerr(errhp, OCIHandleFree((dvoid *) admhp, (ub4) OCI_HTYPE_ADMIN));

/* End the authentication session */
OCISessionEnd(svchp, errhp, usrhp, (ub4)OCI_DEFAULT);

/* Start the sysdba session */
```

```
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
         OCI_SYSDBA));

/* Mount the database */
checkerr(errhp, OCIStmtPrepare2(svchp, &stmthp, errhp, mount_stmt, (ub4)
         strlen((char*) mount_stmt),
         (CONST OraText *) 0, (ub4) 0, (ub4) OCI_NTV_SYNTAX, (ub4)
         OCI_DEFAULT));
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4)0,
         (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT));
checkerr(errhp, OCIStmtRelease(stmthp, errhp, (OraText *)0, 0, OCI_DEFAULT));

/* Open the database */
checkerr(errhp, OCIStmtPrepare2(svchp, &stmthp, errhp, open_stmt, (ub4)
         strlen((char*) open_stmt),
         (CONST OraText *)0, (ub4)0, (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4)0,
         (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT));
checkerr(errhp, OCIStmtRelease(stmthp, errhp, (OraText *)0, 0, OCI_DEFAULT));

/* End the sysdba session */
OCISessionEnd(svchp, errhp, usrhp, (ub4)OCI_DEFAULT);
...
```

To do a shutdown, you must be connected to the database as SYSOPER, or SYSDBA. You cannot be connected to a shared server through a dispatcher. When shutting down in any mode other than OCI_DBSHUTDOWN_ABORT, the following procedure should be followed:

1. Call OCIDBShutdown() in OCI_DEFAULT, OCI_DBSHUTDOWN_TRANSACTIONAL, OCI_DBSHUTDOWN_TRANSACTIONAL_LOCAL, or OCI_DBSHUTDOWN_IMMEDIATE mode to prohibit further connects.
2. Use the necessary ALTER DATABASE commands to close and dismount the database.
3. Call OCIDBShutdown() in OCI_DBSHUTDOWN_FINAL mode to shut down the instance.

```
/*  Example 1 - Orderly shutdown:  */
...
text *close_stmt = (text *)"ALTER DATABASE CLOSE NORMAL";
text *dismount_stmt = (text *)"ALTER DATABASE DISMOUNT";

/* Start the sysdba session */
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
         OCI_SYSDBA));

/* Shutdown in the default mode (transactional, transactional-local,
  immediate would be fine too) */
checkerr(errhp, OCIDBShutdown(svchp, errhp, (OCIAdmin *)0, OCI_DEFAULT));

/* Close the database */
checkerr(errhp, OCIStmtPrepare2(svchp, &stmthp, errhp, close_stmt, (ub4)
         strlen((char*) close_stmt),
         (CONST OraText *)0, (ub4)0, (ub4) OCI_NTV_SYNTAX,
         (ub4) OCI_DEFAULT));
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4)0,
        (OCISnapshot *) NULL,
        (OCISnapshot *) NULL, OCI_DEFAULT));
checkerr(errhp, OCIStmtRelease(stmthp, errhp, (OraText *)0, 0, OCI_DEFAULT));
```

```
/* Dismount the database */
checkerr(errhp, OCIStmtPrepare2(svchp, &stmthp, errhp, dismount_stmt,
         (ub4) strlen((char*) dismount_stmt), (CONST OraText *)0, (ub4)0,
         (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4)0,
         (OCISnapshot *) NULL,
         (OCISnapshot *) NULL, OCI_DEFAULT));
checkerr(errhp, OCIStmtRelease(stmthp, errhp, (OraText *)0, 0, OCI_DEFAULT));

/* Final shutdown */
checkerr(errhp, OCIDBShutdown(svchp, errhp, (OCIAdmin *)0,
         OCI_DBSHUTDOWN_FINAL));

/* End the sysdba session */
checkerr(errhp, OCISessionEnd(svchp, errhp, usrhp, (ub4)OCI_DEFAULT));
...
```

The next shutdown example uses `OCI_DBSHUTDOWN_ABORT` mode:

```
/*  Example 2 - Shutdown using abort:  */
...
/* Start the sysdba session */
...
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
         OCI_SYSDBA));

/* Shutdown in the abort mode */
checkerr(errhp, OCIDBShutdown(svchp, errhp, (OCIAdmin *)0,
         OCI_SHUTDOWN_ABORT));

/* End the sysdba session */
checkerr(errhp, OCISessionEnd(svchp, errhp, usrhp, (ub4)OCI_DEFAULT));
...
```

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.

This chapter contains these topics:

OCI Object Overview

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 metadata.

> **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 in OCI"

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, "Using the Object Type Translator with OCI"

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*. You can also select a non-referenceable object. Fetching the entire object in this way is described in "Fetching Embedded Objects" on page 10-11.

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 10-5, and Chapter 14, "Using the Object Type Translator with OCI"

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 *Oracle Database 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 a temporary instance whose life does not exceed that of the application, and which cannot be stored or flushed to the server. The application can 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: "Creating Objects" on page 10-24 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.

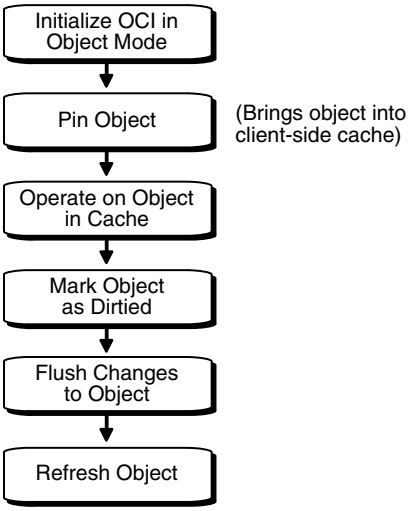
Note: It is possible to `SELECT` a referenceable object into the object cache, rather than pinning it, in which case you fetch the object *by value* instead of fetching its `REF`.

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 10-2 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 13-1.

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.

These struct declarations are automatically written to a header 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:** These functions are mentioned later in this chapter, and are discussed in more detail in Chapter 11, "Object-Relational Datatypes in OCI".
>
> - "NULL Indicator Structure" on page 10-22
> - For more information about OTT, see Chapter 14, "Using the Object Type Translator with OCI".
> - For more information on the use of the `NULL` indicator struct, see the section "NULL Indicator Structure" on page 10-22.

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 Advanced Topics in OCI".

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 2-13. 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 for each 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 10-11 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 an 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 10-5:

```
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 2-13. For specific information about binding and defining `REF` variables, refer to the sections "Advanced Bind Operations in OCI" on page 5-6 and "Advanced Define Operations in OCI" on page 5-14.
- 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 10-15.

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 13-5 for more information.
- `OCI_DURATION_TRANS` is the pin duration. See "Object Duration" on page 13-11 for more information.
- `OCI_LOCK_X` is the lock option. See "Locking Objects For Update" on page 13-10 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 13-14.

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 10-10 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 (for example, `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 in OCI".

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 8-1
- For information about transient and persistent objects, see the section "Creating Objects" on page 10-24
- For information about seeing and checking object meta-attributes, such as *dirty*, see the section "Object Meta-Attributes" on page 10-12

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 10-7. 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 10-5:

  ```
  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 datatypes.
- 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 10-9, 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 "OCI Programming Steps" on page 2-13.

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

Table 10–1 shows the meta-attributes for *standalone* persistent objects.

Table 10–1 Meta-Attributes of Persistent Objects

Meta-Attributes	Meaning
existent	does the object exist?
nullity	null information of the instance

Table 10–1 (Cont.) Meta-Attributes of Persistent Objects

Meta-Attributes	Meaning
locked	has the object been locked?
dirty	has the object been marked as *dirtied*?
pinned	is the object pinned?
allocation duration	see "Object Duration" on page 13-11
pin duration	see "Object Duration" on page 13-11

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 follow.

See Also: "OCIObjectGetProperty()" on page 17-23.

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

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 13-11.

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 an object view 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.

Just as a view is a virtual table, an object view is a virtual object table. Each row in the view is an object: you can call its methods, access its attributes using the dot notation, and create a REF that points to it.

Additional Attribute Functions

The OCI also provides functions which allow an application to set or check some of these attributes directly or indirectly, as shown in the following table:

Table 10–2 Set and Check Functions

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:

Table 10–3 Transient Meta-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 13-11
pin duration	see "Object Duration" on page 13-11

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 round trips 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 later, 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 preceding, 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 REFs (transitive closure) to a certain depth. To do so, set the level parameter to the depth desired. For the preceding 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 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 preceding 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 2-4 and "OCI Descriptors" on page 2-9.

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 `REF`s, 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"), (text *) 0,
              (ub4) 0, OCI_DURATION_SESSION,
              OCI_TYPEGET_HEADER, &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_DURATION_SESSION, OCI_LOCK_NONE, (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, (dvoid **)&liref, (dvoid **)0, &eoc);
while (!eoc)        /* not end of collection */
{
/* pin line item */
   OCIObjectPin(envhp, errhp, *liref, (dvoid *)0, OCI_PIN_RECENT,
       OCI_DURATION_SESSION,
       OCI_LOCK_NONE, (dvoid **)&li));
  display_line_item(li);

/* get next line item */
OCIIterNext(envhp, errhp, itr, (dvoid **)&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 round trips. 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 (for example, SELECT DEREF(ref) from tbl;). Consider situations where the same REF (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 round trips 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 13-7 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 10-10.
- See the section "Freeing an Object Copy" on page 13-7 for more information about objects being aged out of the cache.

NULL Indicator Structure

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)
   printf ("instance is atomically NULL\n");
else
if( my_person_ind -> fname == OCI_IND_NULL)
   printf ("fname attribute is NULL\n");
```

In the preceding 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 if it is `NULL`. 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 17-33 to retrieve the NULL indicator structure of an object.

If you update an attribute in its C structure, you must also set the NULL indicator for that attribute:

```
obj->attr1 = string1;
OCIObjectGetInd(envhp, errhp, obj, &ind);
ind->attr1 = OCI_IND_NOTNULL;
```

> **See Also:** For more information about OTT-generated NULL indicator structures, refer to Chapter 14, "Using the Object Type Translator with OCI".

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 (for example, REF, LOB, string, raw, number, and date) and collections (for example, 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 10–4 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).

Table 10–4 (Cont.) Attribute Values for New Objects

Attribute Type	Default Value
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.
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, NVARCHAR2	OCIString with 0 length and first char set to NULL. The default value is the same as that of a NULL string attribute.
CHAR, NCHAR	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, NCLOB	empty CLOB.
BLOB	empty BLOB.
BFILE	The user must initialize the BFILE to a valid value by setting the directory object and filename.

Freeing and Copying Objects

Use OCIObjectFree() to free memory allocated by OCIObjectNew(). An object instance can have attributes that are pointers to additional memory (secondary memory chunks).

See Also: "Memory Layout of an Instance" on page 13-13

Freeing an object deallocates all the memory allocated for the object, including the associated NULL indicator structure and any secondary memory chunks. You must neither explicitly free the secondary memory chunks nor reassign the pointers. Doing so can result in memory leaks as well as memory corruption. 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: Chapter 17, "OCI Navigational and Type Functions"

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 (Oracle Internet Directories). If `OCIObjectNew()` receives a handle to an object view or a table with a user-defined Oracle Internet Directory, 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 or 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 Oracle Internet Directory 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 `HR` schema:

```
void object_view_new ()
{
dvoid    *table;
OCIRef   *pkref;
dvoid    *object;
OCIType  *emptdo;
...
/* Set up the service context, error handle and so on.. */
...
/* Pin the object view */
OCIObjectPinTable(envp,errorp,svctx, "HR", strlen("HR"), "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 *)emptdo, 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 "Error Handling in OCI" on page 2-20.

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:** *Oracle Database Application Developer's Guide - Object-Relational Features* for a more complete discussion

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 VARCHAR2(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 14-13

This subtype `Student_t`, can have its own subtype, such as `PartTimeStudent_t`:

```
CREATE TYPE PartTimeStudent_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 NOT FINAL;
```

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 of Types" and Table 6–9, " Attributes of 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-sizable) 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 in OCI" for details of the bind and define processes

OCIObjectGetTypeRef()

This function will return the REF of 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()

This function copies the contents of the source instance to the target instance. The source and target instances must be of the same type. It is not possible to copy between a supertype and a subtype.

Similarly, the `tdo` argument must describe the same object type as the source and target objects, and must not refer to a subtype or supertype of the source and target objects.

> **See Also:** "OCIObjectCopy()" on page 17-29

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 Support for Type Inheritance" on page 14-13 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:** *Oracle Database 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.

> **See Also:** OCITypeArrayByName() and OCITypeByName(). To access type information use these functions, as well as `OCIDescribeAny()`.

> **See Also:** "Type Evolution and the Object Cache" on page 13-17 for a discussion of the impact of type evolution on the object cache

11

Object-Relational Datatypes in OCI

This chapter contains these topics:

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, and described in Chapter 13, "Object Advanced Topics in OCI".

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.

The OCI client must allocate a descriptor before bind or define is done. `OCIStmtExecute()` and `OCIStmtFetch2()` will not allocate the memory for the descriptors if they are not allocated by `OCIDescriptorAlloc()`.

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 10-7.

> **See Also:** For detailed information about object types, attributes, and collection datatypes, refer to *Oracle Database Concepts*.

Note: Operations on object types such as `OCIDate`, and so on, allow the address of the result to be the same as that of one of the operands.

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.

NCHAR and NVARCHAR2 can be used as attributes in objects and map to `OCIString *` in C.

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, "Using the Object Type Translator with OCI".

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 `OCINumber`s 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 preceding addition of `OCINumber`s 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:

Table 11–1 Function Prefix examples

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.

Unless otherwise specified, the term *date* in these function calls 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 Example

The following code provides examples of how to manipulate an attribute of type `OCIDate` using OCI calls. For this example, assume that `OCIEnv` and `OCIError` have been initialized as described in Chapter 2, "OCI Programming Basics". See Chapter 13, "Object Advanced Topics in OCI" for information about pinning.

```
#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);

/* Pin tim person object in the object cache. */
/*  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_buf);
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: FRIDAY   , OCTOBER   05, 1990
The last day of the month is: WEDNESDAY, OCTOBER   31, 1990
The next Wednesday is: WEDNESDAY, OCTOBER   10, 1990
```

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:

Table 11–2 Binding and Defining Datetime and Interval Datatypes

OCI Datatype	Type of Data	External Typecode for Binding/Defining
`OCIDateTime`	ANSI DATE	SQLT_DATE
`OCIDateTime`	TIMESTAMP	SQLT_TIMESTAMP

Table 11–2 (Cont.) Binding and Defining Datetime and Interval Datatypes

OCI Datatype	Type of Data	External Typecode for Binding/Defining
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 18-22.

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.

Table 11–3 Datetime Functions

Function	Purpose
OCIDateTimeAssign() on page 18-41	Performs datetime assignment
OCIDateTimeCheck() on page 18-42	Checks if the given date is valid
OCIDateTimeCompare() on page 18-44	Compares two datetime values
OCIDateTimeConstruct() on page 18-45	Constructs a datetime descriptor
OCIDateTimeConvert() on page 18-47	Converts one datetime type to another
OCIDateTimeFromArray() on page 18-48	Converts an array of size OCI_DT_ARRAYLEN to an OCIDateTime descriptor
OCIDateTimeFromText() on page 18-49	Converts the given string to Oracle datetime type in the OCIDateTime descriptor, according to the specified format
OCIDateTimeGetDate() on page 18-51	Gets the date (year, month, day) portion of a datetime value
OCIDateTimeGetTime() on page 18-52	Gets the time (hour, minute, second, fractional second) out of a datetime value

Table 11–3 (Cont.) Datetime Functions

Function	Purpose
OCIDateTimeGetTimeZoneName() on page 18-53	Gets the time zone name portion of a datetime value
OCIDateTimeGetTimeZoneOffset() on page 18-54	Gets the time zone (hour, minute) portion of a datetime value
OCIDateTimeIntervalAdd() on page 18-55	Adds an interval to a datetime to produce a resulting datetime
OCIDateTimeIntervalSub() on page 18-56	Subtracts an interval from a datetime and stores the result in a datetime
OCIDateTimeSubtract() on page 18-57	Takes two datetimes as input and stores their difference in an interval
OCIDateTimeSysTimeStamp() on page 18-58	Gets the system current date and time as a timestamp with time zone
OCIDateTimeToArray() on page 18-59	Converts a OCIDateTime descriptor to an array
OCIDateTimeToText() on page 18-60	Converts the given date to a string according to the specified format
OCIDateZoneToZone() on page 18-62	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 18-22.

Table 11–4 Interval Functions

Function	Purpose
OCIIterCreate() on page 18-16	Adds two intervals to produce a resulting interval
OCIIntervalAssign() on page 18-64	Copies one interval to another
OCIIntervalCheck() on page 18-65	Checks the validity of an interval
OCIIntervalCompare() on page 18-67	Compares two intervals
OCIIntervalDivide() on page 18-68	Divides an interval by an Oracle Number to produce an interval
OCIIntervalFromNumber() on page 18-69	Converts an Oracle Number to an interval
OCIIntervalFromText() on page 18-70	Given an interval string, produces the interval represented by the string
OCIIntervalGetDaySecond() on page 18-73	Gets values of day and second from an interval
OCIDateTimeGetTime() on page 18-52	Gets year and month from an interval
OCIIntervalMultiply() on page 18-75	Multiplies an interval by an Oracle Number to produce an interval
OCIIntervalSetDaySecond() on page 18-76	Sets day and second in an interval
OCIIntervalSetYearMonth() on page 18-77	Sets year and month in an interval
OCIIntervalSubtract() on page 18-78	Subtracts two intervals and stores the result in an interval
OCIIntervalToNumber() on page 18-79	Converts an interval to an Oracle Number
OCIIntervalToText() on page 18-80	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.

Unless otherwise specified, the term *number* in these functions 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".

OCINumber Examples

The following two snippets show how to manipulate an attribute of type `OCINumber`. The second snippet shows how to convert values in `OCINumber` format returned from `OCIDescribeAny()` calls to unsigned integers.

```
/* Example 1  */
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;
text   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 */
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's */
OCINumberAssign(err, stevesal, scottsal);

/* initialize jason's salary to be 20% more than steve's */
dnum = 1.2;
status = OCINumberFromReal(err, &dnum, sizeof(dnum), &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, sizeof(dnum), &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, (text *)"C0999G9999D99", 13,
         (text *)"NLS_NUMERIC_CHARACTERS='.'' NLS_ISO_CURRENCY='Germany'",
         54, &buflen, (text *)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, (text *)"48'000.00", 9, (text
*)"99G999D99", 9,
    (text *)"NLS_NUMERIC_CHARACTERS='.''", 27, debsal);
if (status != OCI_SUCCESS)  /* handle error from OCINumberFromText */;
/* jason's salary is now 48000.00 */
```

Here is an example of how to convert a numeric type returned from an `OCIDescribeAny()` call in `OCINumber` format, such as `OCI_ATTR_MAX` or `OCI_ATTR_MIN`, to an unsigned C integer:

```
/* Example 2 */
ub4  max_seq_val  = 0;
ub1 *max_valp     = NULL;
ub4  max_val_size;
OCINumber max_val;
    OCINumberSetZero(_errhp, &max_val);
    OCIParam* parmdp = 0;
    status = OCIAttrGet ((dvoid*)_dschp, (ub4)OCI_HTYPE_DESCRIBE, &parmdp, 0,
                         (ub4)OCI_ATTR_PARAM, _errhp);
if (isError (status, _errhp))
{
return 0;
}
status = OCIAttrGet ((dvoid*)parmdp, (ub4)OCI_DTYPE_PARAM, &max_valp,
                     &max_val_size, (ub4)OCI_ATTR_MAX, _errhp);
//create an OCINumber object from the ORACLE NUMBER FORMAT
max_val.OCINumberPart[0] = max_val_size; //set the length byte
memcpy(&max_val.OCINumberPart[1], max_valp, max_val_size); //copy the actual bytes
//now convert max_val to an unsigned C integer, max_seq_val
status = OCINumberToInt(_errhp, &max_val, sizeof(max_seq_val),
                        OCI_NUMBER_UNSIGNED, &max_seq_val);
```

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.

Table 11–5 String Functions

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((char *)c_string, "hello world");
/* Assign a text string to an OCIString */
status = OCIStringAssignText(envhp, errhp, c_string,
      (ub4)strlen((char *)c_string),&vstring1);
/* Memory for vstring1 is allocated as part of string assignment */

status = OCIStringAssignText(envhp, errhp, (text *)"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.

Table 11–6 Raw Functions

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
`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 = (OCIRaw *) 0;
ub1 *raw1_pointer;

/* Set up the RAW */
/* assume 'data_block' has been initialized */
status = OCIRawAssignBytes(envhp, errhp, data_block, data_block_len,
&raw1);

/* 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 11-16

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 18-4

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: "Nested Table Manipulation Functions" on page 11-16

Note: Indexes passed to collection functions are zero-based

Collection Data Manipulation Functions

The following generic functions manipulate collection data:

Table 11–7 Collection Functions

Function	Purpose
`OCICollAppend()`	append an element
`OCICollAssign()`	assign one collection to another
`OCICollAssignElem()`	assign element at given index
`OCICollGetElem()`	get pointer to an element given its index
`OCICollGetElemArray()`	get array of elements from a collection
`OCICollIsLocator()`	Indicates whether a collection is locator-based or not
`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()`.

Table 11–8 Collection Scanning Functions

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 previously, 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.

Table 11–9 Nested Table Functions

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: *Oracle Database 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(errhp, OCIIterCreate(envhp, errhp, outer_coll, &itr1));
for(eoc = FALSE;!OCIIterNext(envhp, errhp, itr1, (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, &itr1));
...
```

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.

Table 11–10 REF Manipulation Functions

Function	Purpose
OCIRefAssign()	assign one REF to another
OCIRefClear()	clear or nullify a REF
OCIRefFromHex()	convert hexadecimal string to a REF

Table 11–10 (Cont.) REF Manipulation Functions

Function	Purpose
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

The OCI datatypes and type descriptors are discussed in this section.

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.

Table 11–11 Descriptor Objects

Information Type	OCI Datatype
Type	OCIType

Table 11–11 (Cont.) Descriptor Objects

Information Type		OCI Datatype
Type Attributes Method Parameters	Collection Elements Method Results	OCITypeElem
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.

AnyType, AnyData and AnyDataSet Interfaces

These capabilities allow you to model self-descriptive data. You can store heterogeneous datatypes 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 10-3, and Oracle Database 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 */
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, and so on. 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

  OCI_TYPECODE_TIMESTAMP, OCI_TYPECODE_TIMESTAMP_TZ,

  OCI_TYPECODE_TIMESTAMP_LTZ.

  OCI_TYPECODE_INTERVAL_YM, OCI_TYPECODE_INTERVAL_DS.

- 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 *)"HR.EMPLOYEES",
               (ub4)strlen("HR.EMPLOYEES"),
               (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 */
```

NCHAR Typecodes for OCIAnyData Functions

The function `OCIAnyDataTypeCodeToSqlt()` converts the `OCITypeCode` for an AnyData value to the SQLT code that corresponds to the representation of the value as returned by the `OCIAnyData` API.

The following typecodes are used in the OCIAnyData functions only:

- `OCI_TYPECODE_NCHAR`
- `OCI_TYPECODE_NVARCHAR2`
- `OCI_TYPECODE_NCLOB`

In calls to other functions, such as `OCIDescribeAny()`, these typecodes are not returned and you must use the charset form to determine if the data is NCHAR (if charset form is SQLCS_NCHAR).

`OCIAnyDataTypeCodeToSqlt()` converts `OCI_TYPECODE_CHAR` as well as `OCI_TYPECODE_VARCHAR2` to the output values SQLT_VST (which corresponds to the `OCIString` mapping) with a charset form of SQLCS_IMPLICIT. `OCI_TYPECODE_NVARCHAR2` will also return SQLT_VST (`OCIString` mapping is used by OCIAnyData API) with a charset form of SQLCS_NCHAR.

> **See Also:** For more information see "OCIAnyDataTypeCodeToSqlt()" on page 20-30

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 datatype. The TDO, of datatype OCIType is created and stored in the database when a named datatype 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 datatype bind.

When binding a named datatype, 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 datatype 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 datatypes may require the use of three bind calls in some circumstances. For example, to bind a static array of named datatypes to a PL/SQL table, three calls must be invoked: OCIBindByName(), OCIBindArrayOfStruct(), and OCIBindObject().

See Also:

- For information about using these datatypes to fetch an embedded object from the database, refer to the section "Fetching Embedded Objects" on page 10-11.
- For additional important information, see the section "Information for Named Datatype and REF Binds" on page 11-26
- For more information about descriptor objects, see "Descriptor Objects" on page 11-19.

Binding REFs

As with named datatypes, 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 10-7.
- For additional important information, see the section "Information for Named Datatype and REF Binds" on page 11-26.

Information for Named Datatype and REF Binds

This section presents some additional important information to keep in mind when working with named datatype 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 datatypes or REFs, for array inserts or fetches, the user needs to pass in an array of pointers to buffers (preallocated 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 "Named Datatypes: Object, VARRAY, Nested Table" on page 3-15.

Defining Named Datatypes

This section provides information on defining named datatypes (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 datatype 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 datatype. 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.

> **See Also:** "Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-27 for more important information about defining named datatypes

Defining REF Output Variables

As with named datatypes, 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 datatypes, 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.

> **See Also:** "Information for Named Datatype and REF Defines, and PL/SQL OUT Binds" on page 11-27 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 datatype 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 5-3 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 nullnes.
- 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 preallocated memory must come from the cache (the result of an OCIObjectNew() call).

> **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 preallocate 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 preallocate 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 datatypes or `REFs`, the user needs to pass in an array of pointers to buffers (preallocated 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. "Binding Placeholders in OCI" on page 4-4 discussed the basics of OCI bind operations, while "Defining Output Variables in OCI" on page 4-12 discusses the basics of OCI define operations. Information specific to binding and defining named datatypes and `REFs` is found in Chapter 5, "Binding and Defining in OCI".

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 datatypes 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 11–12 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
BLOB	OCILobLocator *	SQLT_BLOB
CLOB, NCLOB	CILobLocator *	SQLTY_LOB
VARCHAR2, NVARCHAR2	OCIString *	SQLT_VST (see note following)
RAW	OCIRaw *	SQLT_LVB (see note following)
CHAR, NCHAR	OCIString *	SQLT_VST
Object	struct *	Named Datatype (SQLT_NTY)
REF	OCIRef *	REF (SQLT_REF)
VARRAY	OCIArray *	Named Datatype (SQLT_NTY)
Nested Table	OCITable *	Named Datatype (SQLT_NTY)
DATETIME	OCIDateTime *	See "Datetime and Interval (OCIDateTime, OCIInterval)" on page 11-6.
INTERVAL	OCIInterval *	See "Datetime and Interval (OCIDateTime, OCIInterval)" on page 11-6.

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.

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, "Using the Object Type Translator with OCI"

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 in OCI" on page 5-14. 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 preceding 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.

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 precedingexamples, 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.

SQLT_NTY 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;
{
  OCIType *addr_tdo = (OCIType *)0 ;
  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(envhp, errhp, svchp, (const text *)
                SCHEMA, (ub4) strlen((char *)SCHEMA),
                (const text *)"ADDRESS_VALUE",
                (ub4) strlen((char *)"ADDRESS_VALUE"),
                (text *)0, 0, OCI_DURATION_SESSION,
                OCI_TYPEGET_HEADER, &addr_tdo));

  if(!addr_tdo)
  {
    printf("Null tdo returned\n");
    return;
  }

  checkerr(errhp, OCIBindObject(bnd2p, errhp, addr_tdo, (dvoid **) &addr,
        (ub4 *) 0, (dvoid **) &naddr, (ub4 *) 0));
```

SQLT_NTY 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;
{
```

```
  OCIType *addr_tdo = (OCIType *)0;
  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(envhp, errhp, svchp, (const text *)
               SCHEMA, (ub4) strlen((char *)SCHEMA),
               (const text *) "ADDRESS_VALUE",
               (ub4) strlen((char *)"ADDRESS_VALUE"),
               (text *)0, 0, OCI_DURATION_SESSION,
               OCI_TYPEGET_HEADER, &addr_tdo));

  if(!addr_tdo)
  {
    printf("NULL tdo returned\n");
    return;
  }


  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.

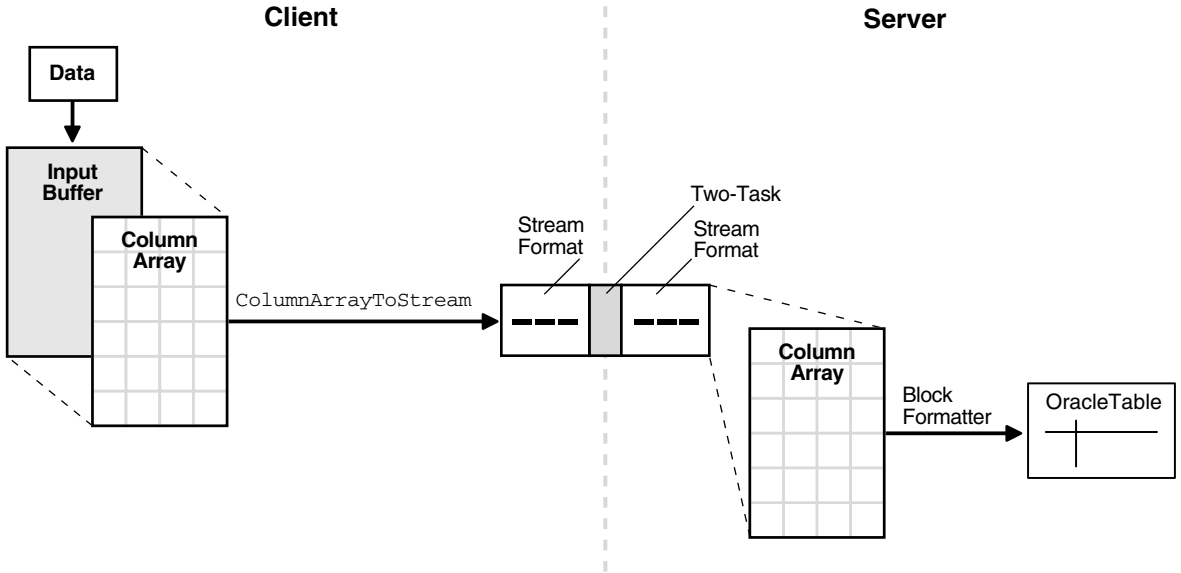
This chapter contains these topics:

- Direct Path Loading Overview
- Direct Path Loading of Object Types
- Direct Path Loading in Pieces
- Direct Path Context Handles and Attributes for Object Types

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 datatypes 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_ATTR_DIRPATH_PARALLEL` option, and if a partition or sub-partition load is being done as opposed to an entire table load.

> **See Also:** "Direct Path Context Handle (OCIDirPathCtx) Attributes" on page A-51

- For a table load, if the `OCI_ATTR_DIRPATH_PARALLEL` 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_ATTR_DIRPATH_PARALLEL` 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
- SQLT_DATE
- SQLT_TIMESTAMP
- SQLT_TIMESTAMP_TZ
- SQLT_TIMESTAMP_LTZ
- SQLT_INTERVAL_YM
- SQLT_INTERVAL_DS

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 "Accessing Column Parameter Attributes" on page 12-3. 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-51 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(). Note that the parent handle of a direct path context is always the environment handle. A direct path context is freed with OCIHandleFree(). Include the header files in the first two lines in all direct path programs:

```
...
#include <cdemodp0.h>
#include <cdemodp.h>

OCIEnv *envp;
OCIDirPathCtx *dpctx;
sword error;
error = OCIHandleAlloc((dvoid *)envp, (dvoid **)&dpctx,
                    OCI_HTYPE_DIRPATH_CTX, (size_t)0,(dvoid **)0);
...
error = OCIHandleFree(dpctx, OCI_HTYPE_DIRPATH_CTX);
```

OCI Direct Path Function Context

> **See Also:** For more about the datatypes supported, see *Oracle Database 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 */
sword error;

error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpfnctx,
                      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 function 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 function 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 "Direct Path Loading Functions" on page 16-117

Operations on the direct path context are performed by the functions in Table 12–1.

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 partially loaded row from server. This function is deprecated.
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 function OCIDirPathStreamReset() which 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.
- `LONG`s 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

Here are some code examples 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;                /* yet 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 */
  OCIDirPathColArray *dpobjca_ctl;          /* dp column array handle for obj*/
  OCIDirPathColArray *dpnestedobjca_ctl;  /* dp col array hndl for nested obj*/
  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 */
  ub4                *otor_ctl;                 /* Offset to Recnum mapping */
  ub1                *inbuf_ctl;                /* buffer for input records */
  struct pctx         pctx_ctl;                    /* partial field context */
  boolean             loadobjcol_ctl;            /* load to obj col(s)? T/F */
};
```

The header file `cdemodp.h`, which is 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 */
  struct obj * obj_col;          /* description of object, if applicable */
#define COL_OID 0x1                                        /* col is an OID */
  ub4   flag_col;
};
```

```
/* 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 obj
{
  text               *name_obj;                              /* type  name*/
  ub2                 ncol_obj;             /* number of columns in col_obj*/
  struct col         *col_obj;                          /* column attributes*/
  struct fld         *fld_obj;                           /* field descriptor*/
  ub4                 rowoff_obj;  /* current row offset in the column array*/
  ub4                 nrows_obj;             /* number of rows in col array*/
  OCIDirPathFuncCtx  *ctx_obj;       /* Function context for this obj column*/
  OCIDirPathColArray *ca_obj;           /* column array  for this obj column*/
  ub4                 flag_obj;                               /* type of obj */
#define OBJ_OBJ  0x1                                           /* obj col */
#define OBJ_OPQ  0x2                                  /* opaque/sql str col */
#define OBJ_REF  0x4                                           /* ref col */
};

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 */
  text         *objconstr_tbl;   /* obj constr/type if loading a derived obj */
};

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 */
  /* See cdemodp.c for the definition of OCI_CHECK */
  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.

```
ub4            rowoff;                          /* column array row offset */
ub4            clen;                                     /* column length */
ub1            cflg;                                  /* column state flag */
ub1           *cval;                             /* column character value */

OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
            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.

```
ub4            rowcnt;                    /* number of rows in column array */
ub4            startoff;          /* starting row offset into column array  */

/* convert array to stream, filter out 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.

```
/* load the stream */
ociret = OCIDirPathLoadStream(ctlp->dpctx_ctl, ctlp->dpstr_ctl,
            ctlp->errhp_ctl);
```

Finish the Direct Path Load

```
/* free up server data structures for the load */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          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);
```

Using a Date Cache in Direct Path Loading of Dates in OCI

The *date cache* feature provides improved performance when loading Oracle date and timestamp values that require datatype conversions in order to be stored in the table.

This feature is specifically targeted to loads where the same input date values are loaded over and over again. Date conversions are very expensive and can account for a large percentage of the total load time, especially if there are multiple date columns loaded. This feature can significantly improve performance by reducing the actual number of date conversions done when many duplicate date values occur in the input data. However, date cache will only improve performance when many duplicate input date values are loaded into date columns (the word *date* in this chapter applies to all the date and timestamp datatypes).

When you explicitly specify the date cache size, the date cache feature will not be disabled, by default. To override this behavior, set `OCI_ATTR_DIRPATH_DCACHE_DISABLE` to 1. Otherwise, the cache will continue to be searched to avoid date conversions. However any misses will be converted the hard way.

Query the attributes `OCI_ATTR_DIRPATH_DCACHE_NUM`, `OCI_ATTR_DIRPATH_DCACHE_MISSES`, `OCI_ATTR_DIRPATH_DCACHE_HITS` and then tune the cache size for future loads.

You can lower the cache size when there are no misses and the number of elements in the cache is less than the cache size. The cache size can be increased if there are many cache misses and relatively few hits. Note that increasing the cache size too much can cause other problems, like paging or exhausting memory. If increasing the cache size does not improve performance, the feature should not be used.

The date cache feature can be explicitly and totally disabled by setting the date cache size to 0.

The following OCI direct path context attributes support this functionality:

OCI_ATTR_DIRPATH_DCACHE_SIZE

This attribute, when not equal to 0, sets the date cache size (in elements) for a table. For example, if the date cache size is set to 200, then at most 200 unique date or timestamp values can be stored in the cache. The date cache size cannot be changed once `OCIDirPathPrepare()` has been called. The default value is 0, meaning a date cache will not be created for a table. A date cache will be created for a table only if one or more date or timestamp values are loaded that require datatype conversions and the attribute value is nonzero.

OCI_ATTR_DIRPATH_DCACHE_NUM

This attribute is used to query the current number of entries in a date cache.

OCI_ATTR_DIRPATH_DCACHE_MISSES

This attribute is used to query the current number of date cache misses. If this number is high, consider tuning the application with a larger date cache size. If increasing the date cache size doesn't cause this number to decrease significantly, the date cache should probably not be used. Date cache misses are expensive, due to hashing and look up times.

OCI_ATTR_DIRPATH_DCACHE_HITS

This attribute is used to query the number of date cache hits. This number should be relatively large in order to see any benefit of using the date cache support.

OCI_ATTR_DIRPATH_DCACHE_DISABLE

Setting this attribute to 1 indicates that the date cache should be disabled if the size is exceeded. Note that this attribute cannot be changed or set after `OCIDirPathPrepare()` has been called.

The default (= 0) is to not disable a cache on overflow. When not disabled, the cache is searched to avoid conversions, but overflow input date value entries will not be added to the date cache, and will be converted using expensive date conversion functions. Again, excessive date cache misses can cause the application to run slower than not using the date cache at all.

This attribute can also be queried to see if a date cache has been disabled due to overflow.

See Also: "Direct Path Context Handle (OCIDirPathCtx) Attributes" on page A-51

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 `SETID`s 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:*
  1. Describe the parent table and its columns as usual, except:
  2. *When describing the nested-table column, this is the column that stores the SETIDs. Its external datatype 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.*
     * *Set its OCI_ATTR_NAME using a dummy name (for example "setid") because the API does not expect you to know its system name.*
     * *Set the column attribute with OCI_ATTR_DIRPATH_SID to indicate that this is a SETID column:*

```
ub1 flg = 1;
sword error;

error = OCIAttrSet((dvoid *)colDesc,
                   OCI_DTYPE_PARAM,
                   (dvoid *)&flg, (ub4)0,
```

```
                                        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:

```
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:

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
text *obj_type;   /* column object's object type */
sword error;

error = OCIAttrSet((dvoid *)dpfnctx,
                   OCI_HTYPE_DIRPATH_FN_CTX,
                   (dvoid *)obj_type, (ub4)strlen((const char *)obj_type),
                   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:

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
ub1 expr_type = OCI_DIRPATH_EXPR_OBJ_CONSTR;
sword error;

error = OCIAttrSet((dvoid *)dpfnctx,
```

```
                    OCI_HTYPE_DIRPATH_FN_CTX,
                    (dvoid *)&expr_type, (ub4)0,
                    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:

1. Get the column descriptor for the object attribute with `OCI_DTYPE_PARAM`.
2. Set the attribute's column name with `OCI_ATTR_NAME`.
3. Set the external column type (the type of the data that will be passed to the direct path API) with `OCI_ATTR_DATA_TYPE`.
4. Set any other external column attributes (maximum data size, precision, scale, and so on.)
5. *If this attribute column is a column object, then do steps 3-10 for its object attributes.*
6. 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 */
sword error;

error = OCIHandleAlloc((dvoid *)dpfnctx, (dvoid **)&dpfnca,
                       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 a 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()`.

OCI_DIRPATH_COL_ERROR

This value is passed to `OCIDirPathColArrayEntry()` to indicate that the current column array row should be ignored. A typical use of this value is to back out all previous conversions for a row when an error occurs providing more data for a partial column (`OCI_NEED_DATA` was returned from the previous `OCIDirPathColArrayToStream()` call). Any previously converted data placed in the output stream buffer for the current row is removed. Conversion then continues with the next row in the column array. The purged row is counted in the converted row count.

When `OCI_DIRPATH_COL_ERROR` is specified, the current row is ignored, as well as are any corresponding rows in any child column arrays referenced, starting from the top level column array row. Any `NULL` child column array references are ignored when moving all referenced child column arrays to their next row.

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.

The bind variable namespace is limited to a column's SQL string. The same bind variable name can be used for multiple columns, but any arguments with the same name only apply to the SQL string of that column.

If a SQL string of a column contains multiple references to a bind variable and multiple arguments are specified for that name, all of the values must be the same, otherwise the results are undefined. Only one argument is actually required for this case, as all references to the same bind variable name in a particular SQL expression will be bound to that single argument.

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.*

```
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.*

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
text *sql_str;   /* column's SQL string expression */
sword error;

error = OCIAttrSet((dvoid *)dpfnctx,
                   OCI_HTYPE_DIRPATH_FN_CTX,
                   (dvoid *)sql_str, (ub4)strlen((const char *)sql_str),
                   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.*

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
ub1 expr_type = OCI_DIRPATH_EXPR_SQL;
sword error;

error = OCIAttrSet((dvoid *)dpfnctx,
                   OCI_HTYPE_DIRPATH_FN_CTX,
                   (dvoid *)&expr_type, (ub4)0,
                   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:*
   a. Get the column descriptor for the object attribute with `OCI_DTYPE_PARAM`.
   b. *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.*

c. Set the attribute's column name with `OCI_ATTR_NAME`.

d. *There is a naming convention for SQL string arguments.*

e. *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".*

f. *If an argument is used multiple times in an SQL string, declaring it once and counting it as one argument only is correct.*

g. Set the external column type (the type of the data that will be passed to the direct path API) with `OCI_ATTR_DATA_TYPE`.

h. Set any other external column attributes (maximum data size, precision, scale, and so on).

i. 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 */
sword error;

error = OCIHandleAlloc((dvoid *)dpfnctx, (dvoid **)&dpfnca,
                       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.

See Also: "OCI_DIRPATH_COL_ERROR" on page 12-16

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.*

```
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.*

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
text *ref_tbl;     /* column's reference table */
sword error;

error = OCIAttrSet((dvoid *)dpfnctx,
                   OCI_HTYPE_DIRPATH_FN_CTX,
                   (dvoid *)ref_tbl, (ub4)strlen((const char *)ref_tbl),
                   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.*

   - *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.)

- *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 earlier.*

   - *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.

   - *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 later), 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 datatype.*

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 datatype.

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, and so on).

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 */
sword error;

error = OCIHandleAlloc((dvoid *)dpfnctx, (dvoid **)&dpfnca,
                       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()`.

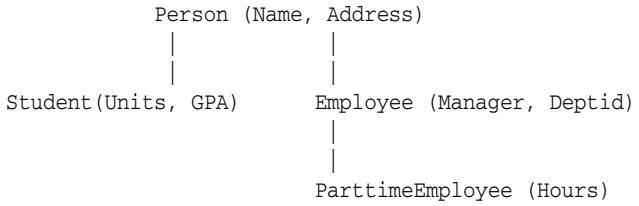
See Also: "OCI_DIRPATH_COL_ERROR" on page 12-16

NOT FINAL Object and REF Columns

An example of an inheritance hierarchy is shown in the following section.

Inheritance Hierarchy Diagram

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:



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 a `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 (Oracle Internet Directory):

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.

```
OCIDirPathColArray *dpca;  /* direct path column array */
sword error;

error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpca,
                       OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                       (size_t)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 derived 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 */
sword error;

error = OCIAttrSet((dvoid *)dpctx,
                   OCI_HTYPE_DIRPATH_CTX,
                   (dvoid *) obj_type,
                   (ub4)strlen((const char *) obj_type),
                   OCI_ATTR_DIRPATH_OBJ_CONSTR, ctlp->errhp_ctl);
```

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` flagand 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.

See Also: "OCI_DIRPATH_COL_ERROR" on page 12-16

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 tht 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.

See Also: "OCI_DIRPATH_COL_ERROR" on page 12-16

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

There is one.

OCI_ATTR_DIRPATH_OBJ_CONSTR

Indicates the object type to load into a `NOT FINAL` object table.

```
ttext *obj_type;            /* the object type to load into this NOT FINAL */
                           /* object table */
sword error;

error = OCIAttrSet((dvoid *)dpctx,
                   OCI_HTYPE_DIRPATH_CTX,
                   (dvoid *) obj_type,
                   (ub4)strlen((const char *) obj_type),
                   OCI_ATTR_DIRPATH_OBJ_CONSTR, ctlp->errhp_ctl);
```

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-51

OCI_ATTR_DIRPATH_OBJ_CONSTR

Indicates the object type to load into a substitutable object table.

```
text *obj_type; /* stores an object type name */
sword error;

error = OCIAttrSet((dvoid *)dpctx,
                   OCI_HTYPE_DIRPATH_CTX,
                   (dvoid *) obj_type,
                   (ub4)strlen((const char *) obj_type),
                   OCI_ATTR_DIRPATH_OBJ_CONSTR, ctlp->errhp_ctl);
```

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 preceding rules:

```
OCIDirPathFuncCtx  *dpfnctx; /* function context for this non-scalar column */
ub1 expr_type; /* expression type */
sword error;

if (...) /* (column type is an object) */
expr_type = OCI_DIRPATH_EXPR_OBJ_CONSTR;
...
if (...) /* (column_type is a REF && function context name exists) */
expr_type = OCI_DIRPATH_EXPR_REF_TBLNAME;
...
if (...) /* (column_type is a SQL string) */
expr_type = OCI_DIRPATH_EXPR_SQL;
...
error = OCIAttrSet((dvoid *)(dpfnctx),
                   OCI_HTYPE_DIRPATH_FN_CTX,
                   (dvoid *)&expr_type, (ub4)0,
                   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():

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
sword error;

error = OCIAttrSet((dvoid *)colDesc,
                   OCI_DTYPE_PARAM,
                   (dvoid *)(dpfnctx), (ub4)0,
                   OCI_ATTR_DIRPATH_FN_CTX, ctlp->errhp_ctl);
```

Attributes for column parameter context handles follow.

> **See Also:** "Direct Path Column Parameter Attributes" on page A-59

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 later), 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 datatype.

– 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 datatype.

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;
sword error;

error = OCIAttrSet((dvoid *)colDesc,
                   OCI_DTYPE_PARAM,
                   (dvoid *)&flg, (ub4)0,
```

```
                    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;
sword error;

error = OCIAttrSet((dvoid *)colDesc,
                   OCI_DTYPE_PARAM,
                   (dvoid *)&flg, (ub4)0,
                   OCI_ATTR_DIRPATH_OID, ctlp->errhp_ctl);
```

Direct Path Function Column Array Handle for Non-scalar Columns

See Also: "Direct Path Function Column Array Handle (OCIDirPathColArray) Attributes" on page A-57

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 */
sword error;

error = OCIHandleAlloc((dvoid *)dpfnctx, (dvoid **)&dpfnca,
                       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 Advanced Topics in OCI

This chapter introduces OCI's facility for working with objects in an Oracle database server. It also discusses OCI's object navigational function calls, type evolution, and support for XML.

This chapter contains these topics:

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. The object cache is logically partitioned by the connection.

Dereferencing a REF many times or dereferencing several equivalent REFs in the same connection 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 terminates 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 terminated transaction.

See Also: "Pinning an Object Copy" on page 13-5

The object cache is created when the OCI environment is initialized using `OCIEnvCreate()` with `mode` set to `OCI_OBJECT`.

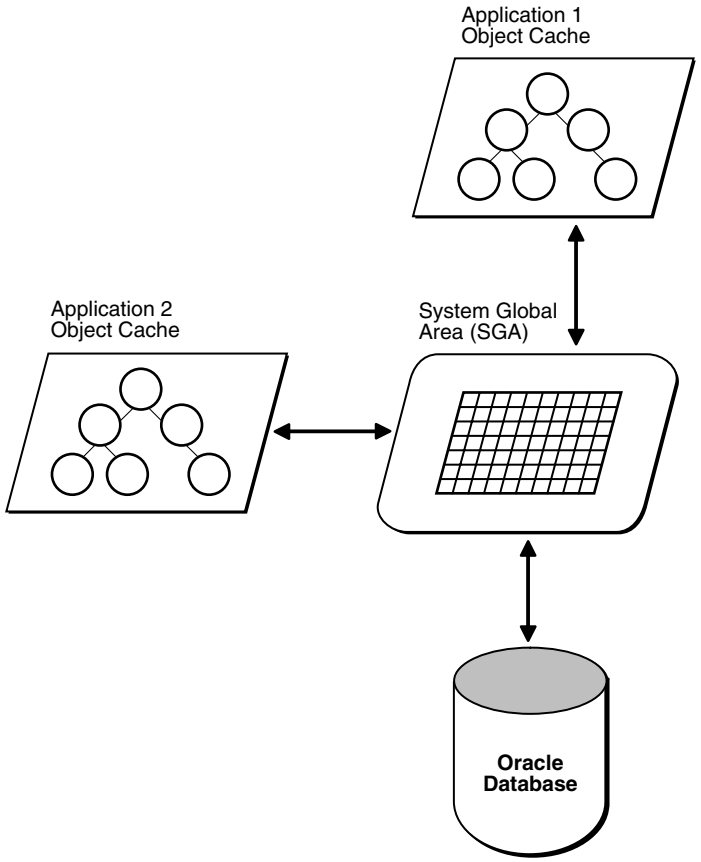
The object cache maintains a fast look-up table for mapping `REFs` to objects. When an application de-references a `REF` and the corresponding object is not yet cached in the object cache, the object cache automatically sends a request to the server to fetch the object from the database and load it into 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
```

You can set the value of OCI_ATTR_CACHE_MAX_SIZE at 10% (the default) of the OCI_ATTR_CACHE_OPT_SIZE. 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-2 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 13-15.

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 (for example, string text pointers, collections, and so on) may become invalid after a refresh.

For example, if the object is of type `person` with two attributes: `salary (number)`, and `name (varchar2(20))`. The type is:

```
struct Person {
OCINumber salary;
OCIString *name;
}
```

If the client has a pointer `scott_p` to `Person` instance, and calls `OCIObjectRefresh()` on that instance, the pointer `scott_p` is still the same after refresh, but the pointers to second-level memory, such as `scott_p->name` can be different.

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. `OCIObjectPin()` takes 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.

Note that the object cache looks for the object copy only within the logical partition of the cache for the specified connection. If there is no copy in the partition, the latest copy of the object is loaded from the server.

- 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 (that is, 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.

Read-committed and serialized transactions refer to different isolation levels that a database can support. There are other isolation levels also, such as read-uncommitted, repeatable read, and so on. Each isolation level permits more or less interference among concurrent transactions. Typically, when an isolation level permits more interference, simultaneous transactions will have higher concurrency. In a read-committed transaction, when a query is executed multiple times, it can produce inconsistent sets of data because it allows changes made by other committed transactions to be seen. This will not happen in serializable transaction.

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 (for example, read committed versus serializable).

Note: For `OCIObjectArrayPin()` the pin option has no effect, because objects are always retrieved from the database. If a REF is to an object in the cache, `OCIObjectArrayPin()` will fail with:

```
Ora-22881: dangling REF.
```

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: "Object Cache Parameters" on page 13-4 For more information.

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 17-7.

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 "Environment Handle Attributes" on page A-2

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 (that is, 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 (that is, 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 13-13 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 through 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 preceding 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 terminates). 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

Table 13–2 (Cont.) Example of Allocation and Pin Durations

Time	Application Action	Function	Object 1	Object 2	Object 3	Object 4
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 Also:

- 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 10-24 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 (for example, `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 preceding 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);
```

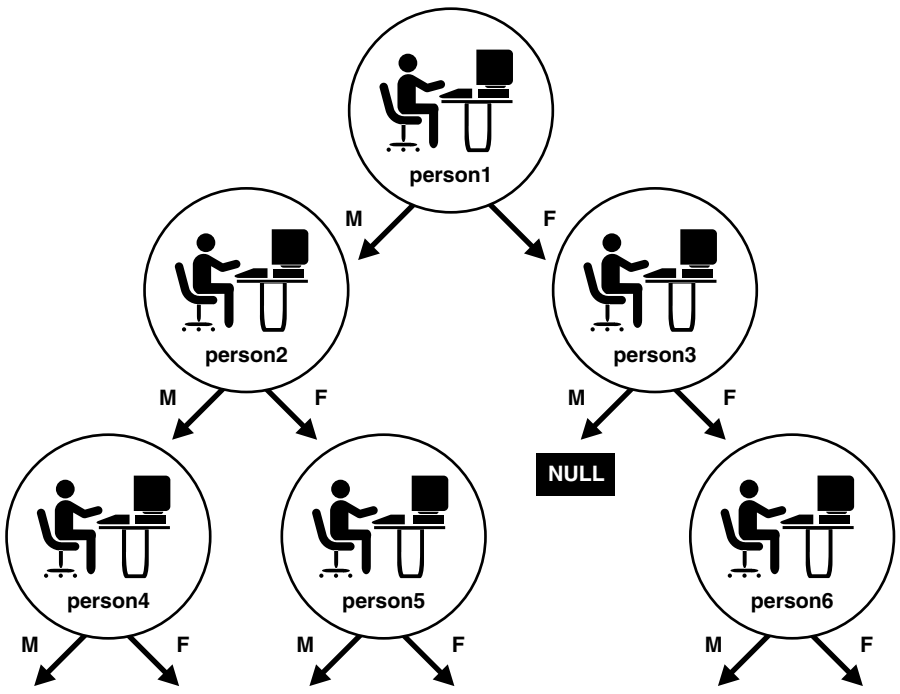
An object table of `person_t` objects is created with the following statement:

```
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 preceding 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:

Table 13–3 Pin, Free, and Unpin Functions

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

Table 13–3 (Cont.) Pin, Free, and Unpin Functions

Function	Purpose
`OCIObjectPinTable()`	Pin a table object with a given duration
`OCIObjectUnpin()`	Unpin an object

Flush and Refresh Functions

The following functions are available to flush modified objects to the server:

Table 13–4 Flush and Rfresh Functions

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:

Table 13–5 Mark and Unmark Functions

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:

Table 13–6 Object Meta-Attributes Functions

Function	Purpose
`OCIObjectExists()`	Get existence status of an instance
`OCIObjectFlushStatus()`	Get the flush status of an instance
`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:

Table 13–7 Other Object Functions

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 stop the application or refresh the TDO and re-pin the object. Continuing with the application may cause the application to fail because even if the image and the TDO are at the same version, there is no guarantee that the object structure (that is, 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 10-30

OCI Support for XML

Oracle XML DB provides support for storing and manipulating XML instances by using the `XMLType` datatype. You can access these XML instances by means of OCI, in conjunction with the C DOM API for XML.

An application program must initialize the usual OCI handles such as the server handle or the statement handle, and it must then initialize the XML context. The program can either operate on XML instances in the backend or create new instances in the client side. The initialized XML context can be used with all the C DOM functions.

XML data stored in Oracle XML DB can be accessed on the client side by means of the C DOM structure `xmldocnode`.You can use this structure for binding, defining, and operating on XML values in OCI statements.

See Also: For information about the XML support in C, see

- Chapter 22, "OCI XML DB Functions"
- *Oracle XML DB Developer's Guide*, the chapter on "C API for XML" and the OCI code examples in the appendix "Oracle-Supplied XML Schemas and Examples"
- *Oracle XML Developer's Kit Programmer's Guide,* chapter on "XML Parser for C"
- *Oracle XML Reference* the DOM C API HTML

XML Context

An XML context is a required parameter in all the C DOM API functions. This opaque context encapsulates information pertaining to data encoding, error message language, and so on. The contents of this context are different for XDK and for Oracle XML DB applications.

For Oracle XML DB, there are two OCI functions provided to initialize and free an XML context:

```
xmlctx *OCIXmlDbInitXmlCtx (OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
       ocixmldbparam *params, ub4 num_params);

void OCIXmlDbFreeXmlCtx (xmlctx *xctx);
```

XML Data on the Server

XML data on the server can be operated on by means of OCI statement calls. You can bind and define `XMLType` values using `xmldocnode`, as with other object instances. OCI statements are used to select XML data from the server. This data can be used in the C DOM functions directly. Similarly, the values can be bound back to SQL statements directly.

Using OCI XML DB Functions

To initialize and terminate the XML context, use the functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` respectively. The header file `ocixmldb.h` is used with the unified C API.

The next fragment of a tested example shows how to perform operations with the C API:

```
#ifndef S_ORACLE
#include <s.h>
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>
```

```
typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
} test_ctx;

...
void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;
 ...
  /* Initialize envhp, svchp, errhp, dur, stmthp */
  ...

  /* Get an xml context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &ctx->dur;
  xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

/* Do unified C API operations next */
...

/* Free the statement handle using OCIHandleFree() */
...
/* Free the allocations associated with the context */
OCIXmlDbFreeXmlCtx(xctx);
/* Free envhp, svchp, errhp, stmthp */
...
}
```

14

Using the Object Type Translator with OCI

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 applications.

This chapter contains these topics:

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 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. 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 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 user name 'scott' and password 'tiger'.

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 emptype
```

Sample `demoout.typ` file:

```
CASE = LOWER
TYPE SCOTT.EMPTYPE AS emptype
  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 emptype`). 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 emptype 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 emptype
{
    OCIString * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate   hiredate;
    OCINumber salary;
```

```
};
typedef struct emptype emptype;

struct emptype_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,
           "HR", 2,
           "EMPTYPE", 7,
           "$8.0", 4);
   return status;
}
```

Parameters in the intype file control the way generated structs are named. In this example, the struct name `emptype` matches the database type name `emptype`. 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 (for example, `OCIString`, `OCIInd`) are special datatypes.

> **See Also:** For more information about these types, see the section "OTT Datatype Mappings" on page 14-8

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 *Oracle Database 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:** "The OTT Command Line" on page 14-5

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 operating system-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:** "The Intype File" on page 14-6

The OTT Command Line

On most operating systems, 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 operating system-specific documentation to see how to invoke OTT.

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, see "OTT Reference" on page 14-19

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 user name '`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 14-5, 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 14-5, 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 operating systems 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 operating systems 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: "CASE" on page 14-23

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, for example, 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 14-25

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 datatypes. 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 14-12

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 14-10, 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), NCHAR(N)	OCIString *
CLOB, NCLOB	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
BINARY_FLOAT	float
BINARY_DOUBLE	double
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 *
VARCHAR(N)	OCIString *
VARCHAR2(N), NVARCHAR2(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 14-24, 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 preceding 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:** ,"NULL Indicator Structure" on page 10-22 for more information about atomic nullity.

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_ind;
   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 preceding 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 preceding 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.

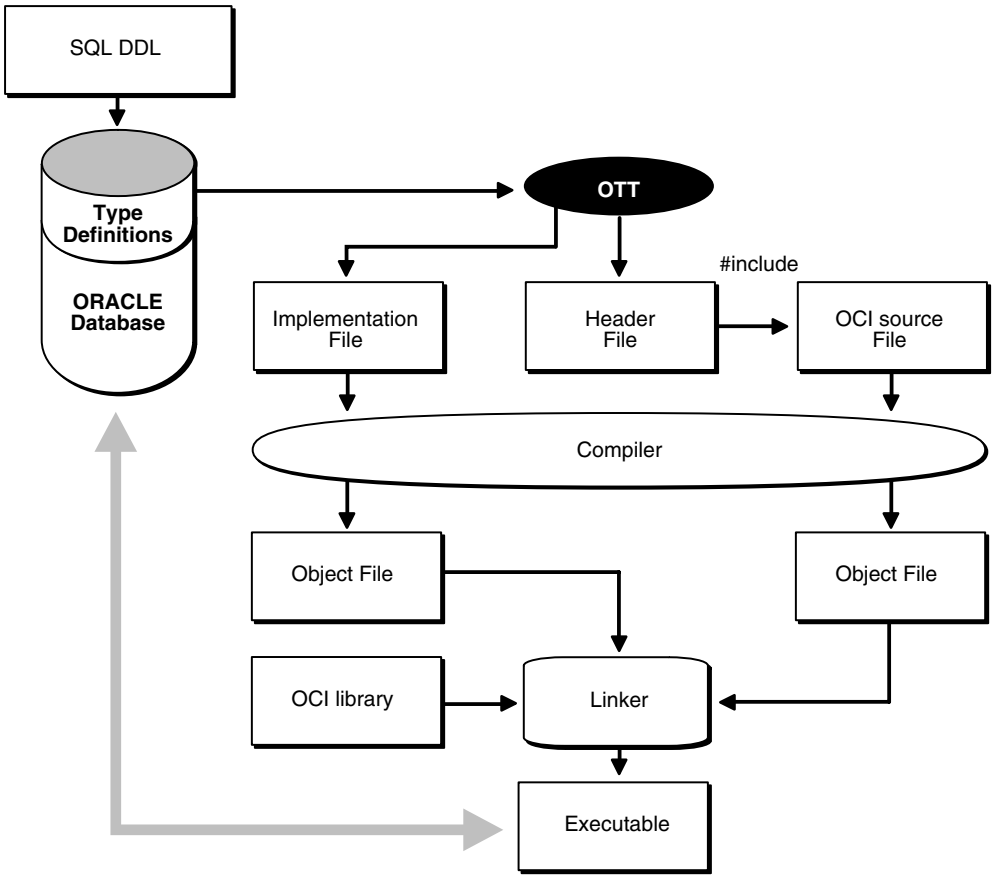
Once the header file has been included, the OCI application can access and manipulate object data in the host language format.

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 earlier are entered as shown. These include '.', '@', and so on.

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]

[URL=url]
```

> **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 user name, 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 14-25

If the file name on the command line or in the intype file does not include an extension, a operating system-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 operating system-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 operating system-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 operating system-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 operating system-dependent location. All remaining parameter specifications must appear on the command line, or in the intype file.

config=*filename*

Note: A CONFIG 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 operating system-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 14-28

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=TRUE` is specified, then types needed by other types but not mentioned in the `intype` file are generated.

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.

URL

OTT uses JDBC (Java Database Connectivity), the Java interface for connecting to the database. The default value of parameter URL is:

```
URL=jdbc:oracle:oci8:@
```

The OCI8 driver is for client-side use with an Oracle installation. To specify the Thin driver (the Java driver for client-side use without an Oracle installation):

```
URL=jdbc:oracle:thin:@host:port:sid
```

where `host` is the name of the host on which the database is running, `port` is the port number, and `sid` is the Oracle SID.

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 operating system-dependent location. This file must exist, but can be empty. Parameters in a configuration file must appear one in each 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 operating system-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 14-15

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 next.

- 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 the operation of OTT, 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 14-30

- 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 pxl
  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 (for example, a UNIX absolute path name 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 operating system-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 `#includes`, 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.

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.

> **See Also:** For code examples, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".

This chapter contains these topics:

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 in OCI" on page 2-20

Conventions for OCI Functions

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 next.

Table 15–1 Mode of a Parameter

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.

Returns

This optional section describes the possible values that can be returned. It can be found either before or after the Comments section.

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 15–2 Connect, Authorize, and Initialize Functions

Function	Purpose
OCIAppCtxClearAll() on page 15-4	To clear all attribute-value information in a namespace of an application context.
OCIAppCtxSet() on page 15-5	To set an attribute and its associated value in a namespace of an application context.
OCIConnectionPoolCreate() on page 15-7	Initializes the connection pool.
OCIConnectionPoolDestroy() on page 15-9	Destroys the connection pool.
OCIDBShutdown() on page 15-10	Shuts down the Oracle Database
OCIDBStartup() on page 15-12	Starts an Oracle Database instance
OCIEnvCreate() on page 15-13	Creates and initializes an OCI environment.
OCIEnvInit() on page 15-16	Initialize an environment handle.
OCIEnvNlsCreate() on page 15-18	Creates and initializes an environment for OCI functions to work under. Allows you to set character set id and national character set id at environment creation time.
OCIInitialize() on page 15-22	Initialize OCI process environment.
OCILogon() on page 15-25	Simplified single-session logon.
OCILogon2() on page 15-27	This function is used to create a logon session in various modes.
OCIServerAttach() on page 15-30	Attach to a server; initialize server context handle.
OCIServerDetach() on page 15-32	Detach from a server; uninitialize server context handle.
OCISessionBegin() on page 15-33	Authenticate a user.
OCISessionEnd() on page 15-36	Terminate a user session.
OCISessionGet() on page 15-37	Get a session from a session pool.
OCISessionPoolCreate() on page 15-40	Initializes a session pool.
OCISessionPoolDestroy() on page 15-43	Destroys a session pool.
OCISessionRelease() on page 15-44	Releases a session.
OCITerminate() on page 15-46	Detaches from a shared memory subsystem.

OCIAppCtxClearAll()

Purpose

To clear all attribute-value information in a namespace of an application context.

Syntax

```
sword OCIAppCtxClearAll ( void      *sesshndl,
                          dvoid     *nsptr,
                          ub4       nsptrlen,
                          OCIError  *errhp,
                          ub4       mode ;
```

Parameters

sesshndl (IN/OUT)
Pointer to a session handle.

nsptr (IN)
Pointer to namespace string (currently only `CLIENTCONTEXT`).

nsptrlen (IN)
Length of the namespace string.

mode (IN)
Mode (`OCI_DEFAULT` is the default).

errhp (OUT)
An error handle which can be passed to `OCIErrorGet()`.

Returns

Returns error number.

Comments

This will clean up the context information on the server side during the next call to the server. This namespace information is cleared from the session handle once the information has been sent to the server and must be set up again if needed.

Related Functions

OCIAppCtxSet()

OCIAppCtxSet()

Purpose

To set an attribute and its associated value in a namespace of an application context.

Syntax

```
sword OCIAppCtxSet ( void      *sesshndl,
                     dvoid     *nsptr,
                     ub4       nsptrlen,
                     dvoid     *attrptr,
                     ub4       attrptrlen,
                     dvoid     *valueptr,
                     ub4       valueptrlen,
                     OCIError  *errhp,
                     ub4       mode );
```

Parameters

sesshndl (IN/OUT)
Pointer to a session handle.

nsptr (IN)
Pointer to namespace string (currently only `CLIENTCONTEXT`).

nsptrlen (IN)
Length of the namespace string.

attrptr (IN)
Pointer to attribute string.

attrptrlen (IN)
The length of the string pointed to by `attrptr`.

valueptr (IN)
Pointer to value string.

valueptrlen (IN)
The length of the string pointed to by `valueptr`.

mode (IN)
Mode (`OCI_DEFAULT` is the default.

errhp (OUT)
An error handle which can be passed to `OCIErrorGet()`.

Returns

Returns error number

Comments

The information set on the session handle is sent to the server during the next call to the server.

This information is cleared from the session handle once the information has been sent to the server and must be set up again if needed.

Related Functions

OCIAppCtxClearAll()

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 higher.

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 higher.

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 higher.

poolUsername (IN)
Connection pooling requires an implicit primary session and this attribute provides a user name for that session.

poolUserLen (IN)
The length of `poolUsername`.

poolPassword (IN)
The password for the user name `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-18

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()

OCIDBShutdown()

Purpose

Shuts down an Oracle Database instance.

Syntax

```
sword OCIDBShutdown ( OCISvcCtx     *svchp,
                      OCIError      *errhp,
                      OCIAdmin      *admhp,
                      ub4           mode);
```

Parameters

svchp (IN)
A handle to a service context. There must be a valid server handle and a valid user handle set in `svchp`.

errhp (IN/OUT)
An error handle which can be passed to `OCIErrorGet()` for diagnostic information in the event of an error.

admhp (IN) - Optional
An instance administration handle. Currently not used; pass `(OCIAdmin *)0`.

mode (IN)
OCI_DEFAULT - Further connects are prohibited. Waits for users to disconnect from the database.

OCI_DBSHUTDOWN_TRANSACTIONAL - Further connects are prohibited and no new transactions are allowed. Waits for active transactions to complete.

OCI_DBSHUTDOWN_TRANSACTIONAL_LOCAL - Further connects are prohibited and no new transactions are allowed. Waits only for local transactions to complete.

OCI_DBSHUTDOWN_IMMEDIATE - Does not wait for current calls to complete or users to disconnect from the database. All uncommitted transactions are terminated and rolled back.

OCI_DBSHUTDOWN_FINAL - Shuts down the database. Should be used only in the second call to `OCIDBShutdown()` after the database is closed and dismounted.

OCI_DBSHUTDOWN_ABORT - Does not wait for current calls to complete or users to disconnect from the database. All uncommitted transactions are terminated and are not rolled back. This is the fastest possible way to shut down the database, but the next database startup may require instance recovery. Therefore, this option should be used only in unusual circumstances: if a background process terminates abnormally.

Comments

To do a shut down, you must be connected to the database as `SYSOPER`, or `SYSDBA`. You cannot be connected to a shared server via a dispatcher. When shutting down in any mode other than `OCI_DBSHUTDOWN_ABORT`, the following procedure should be followed:

1. Call `OCIDBShutdown()` in `OCI_DEFAULT`, `OCI_DBSHUTDOWN_TRANSACTIONAL`, `OCI_DBSHUTDOWN_TRANSACTIONAL_LOCAL`, or `OCI_DBSHUTDOWN_IMMEDIATE` mode to prohibit further connects.

2. Issue the necessary `ALTER DATABASE` commands to close and dismount the database.
3. Call `OCIDBShutdown()` in `OCI_DBSHUTDOWN_FINAL` mode to shut down the instance.

See Also: "Database Startup and Shutdown" on page 9-72

Related Functions

OCIAttrSet(), OCIDBStartup()

OCIDBStartup()

Purpose

Starts an Oracle Database instance.

Syntax

```
sword OCIDBStartup ( OCISvcCtx     *svchp,
                     OCIError      *errhp,
                     OCIAdmin      *admhp,
                     ub4           mode,
                     ub4           flags);
```

Parameters

svchp (IN)
A handle to a service context. There must be a valid server handle and user handle set in `svchp`

errhp (IN/OUT)
An error handle which can be passed to `OCIErrorGet()` for diagnostic information in the event of an error.

admhp (IN) - Optional
An instance administration handle. Use to pass additional arguments to the startup call, or pass `(OCIAdmin *)0` if you do not set `OCI_ATTR_ADMIN_PFILE`.

mode (IN)
`OCI_DEFAULT` - This is the only supported mode. It starts up the instance, but does not mount or open the database. Same as `STARTUP NOMOUNT`.

flags (IN)
`OCI_DBSTARTUPFLAG_FORCE` - Shuts down a running instance (if there is any) using `ABORT` before starting a new one. This mode should be used only in unusual circumstances.

`OCI_DBSTARTUPFLAG_RESTRICT` - Allows database access only to users with both the `CREATE SESSION` and `RESTRICTED SESSION` privileges (normally, the DBA).

Comments

You must be connected to the database as `SYSOPER` or `SYSDBA` in `OCI_PRELIM_AUTH` mode. You cannot be connected to a shared server via a dispatcher (that is, when you restart a running instance with `OCI_DBSTARTUPFLAG_FORCE`). To use a client-side parameter file (`pfile`), `OCI_ATTR_ADMIN_PFILE` must be set in the administration handle; otherwise, a server-side parameter file (`spfile`) will be used. A call to `OCIDBStartup()` starts up one instance on the server.

See Also: "Database Startup and Shutdown" on page 9-72

Related Functions

OCIAttrSet(), OCIDBShutdown(), OCIServerAttach(), OCISessionBegin()

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_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.
- `OCI_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.
- `OCI_NEW_LENGTH_SEMANTICS` - byte-length semantics is used consistently for all handles, regardless of character sets.
- `OCI_NCHAR_LITERAL_REPLACE_ON` - turns on N' substitution.
- `OCI_NCHAR_LITERAL_REPLACE_OFF` - turns off N' substitution. If neither this mode or `OCI_NCHAR_LITERAL_REPLACE_ON` is used, the substitution is determined by the environment variable `ORA_NCHAR_LITERAL_REPLACE`, which can be set to `TRUE` or `FALSE`. When it is set to `TRUE`, the replacement will be turned on, otherwise it will be turned off, which is the default setting in OCI.

See Also: "Dynamic Callback Registrations" on page 9-26

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 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 N' substitution is turned on, `OCIStmtPrepare()` or `OCIStmtPrepare2()` will perform the N' substitution on the SQL text and store the resulting SQL text in the statement handle. Thus, if the application uses `OCI_ATTR_STATEMENT` to retrieve the SQL text from the OCI statement handle, the modified SQL text, instead of the original SQL text, will be returned.

To turn on N' substitution in `ksh` shell:

```
export ORA_NCHAR_LITERAL_REPLACE=TRUE
```

To turn on N' substitution in `csh` shell:

```
setenv ORA_NCHAR_LITERAL_REPLACE TRUE
```

If a remote database is of a release before 10.2, N' substitution is not done.

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 2-13

Example

```
OCIEnv *envhp;
...
/* Create a thread-safe OCI environment with N' substitution turned on.  */
if(OCIEnvCreate((OCIEnv **)&envhp,
         (ub4)OCI_THREADED | OCI_NCHAR_LITERAL_REPLACE_ON,
         (dvoid *)0, (dvoid * (*)(dvoid *, size_t))0,
         (dvoid * (*)(dvoid *, dvoid *, size_t))0,
         (void (*)(dvoid *, dvoid *))0,
         (size_t)0, (dvoid **)0))
{
  printf("Failed: OCIEnvCreate()\n");
  return 1;
}
...
```

Related Functions

OCIHandleAlloc(),OCIHandleFree(),OCIEnvInit(), OCIEnvNlsCreate(),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_ENV_NO_MUTEX`
- `OCI_ENV_NO_UCB`

In `OCI_DEFAULT` mode, the OCI library always mutexes handles. In `OCI_ENV_NO_MUTEX` modes, there is no mutexing in this environment.

In `OCI_ENV_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 9-26

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 2-13.

Related Functions

OCIHandleAlloc(),OCIHandleFree(),OCIEnvCreate(),OCITerminate()

OCIEnvNlsCreate()

Purpose

Creates and initializes an environment handle for OCI functions to work under. It is an enhanced version of the `OCIEnvCreate()` function.

Syntax

```
sword OCIEnvNlsCreate   ( OCIEnv        **envhpp,
                          ub4           mode,
                          dvoid         *ctxp,
                          dvoid         *(*malocfp)
                                             (dvoid *ctxp,
                                              size_t size),
                          dvoid         *(*ralocfp)
                                             (dvoid *ctxp,
                                              dvoid *memptr,
                                              size_t newsize),
                          void          (*mfreefp)
                                             (dvoid *ctxp,
                                              dvoid *memptr))
                          size_t        xtramemsz,
                          dvoid         **usrmempp
                          ub2           charset,
                          ub2           ncharset );
```

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_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.
- `OCI_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.
- `OCI_NCHAR_LITERAL_REPLACE_ON` - turns on N' substitution.
- `OCI_NCHAR_LITERAL_REPLACE_OFF` - turns off N' substitution. If neither this mode or `OCI_NCHAR_LITERAL_REPLACE_ON` is used, the substitution is determined by the environment variable `ORA_NCHAR_LITERAL_REPLACE`, which can be set to `TRUE` or `FALSE`. When it is set to `TRUE`, the replacement will be turned on, otherwise it will be turned off, the default setting in OCI.

See Also: "Dynamic Callback Registrations" on page 9-26

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.

charset (IN)
The client-side character set for the current environment handle. If it is 0, the `NLS_LANG` setting is used. `OCI_UTF16ID` is a valid setting; it is used by the metadata and the `CHAR` data.

ncharset (IN)
The client-side national character set for the current environment handle. If it is 0, `NLS_NCHAR` setting is used. `OCI_UTF16ID` is a valid setting; it is used by the `NCHAR` data.

Returns

OCI_SUCCESS - environment handle has been successfully created.

OCI_ERROR - an error occurred.

Comments

This call creates an environment for all the OCI calls using the modes specified by the user.

After using OCIEnvNlsCreate() to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes. This applies to the following calls:

- OCIBindByName()
- OCIBindByPos()
- OCIBindDynamic()
- OCIDefineByPos()
- OCIDefineDynamic()

This function enables you to set charset and ncharset ids at environment creation time. It is an enhanced version of the OCIEnvCreate() function.

This function sets nonzero charset and ncharset as client side database and national character sets, replacing the ones specified by NLS_LANG and NLS_NCHAR. When charset and ncharset are 0, it behaves exactly the same as OCIEnvCreate(). Specifically, charset controls the encoding for metadata and data with implicit form attribute and ncharset controls the encoding for data with SQLCS_NCHAR form attribute.

Although OCI_UTF16ID can be set by OCIEnvNlsCreate(), it cannot be set in NLS_LANG or NLS_NCHAR. To access the character set ids in NLS_LANG and NLS_NCHAR, use OCINlsEnvironmentVariableGet().

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 N' substitution is turned on, OCIStmtPrepare() or OCIStmtPrepare2() will perform the N' substitution on the SQL text and store the resulting SQL text in the statement handle. Thus, if the application uses OCI_ATTR_STATEMENT to retrieve the SQL text from the OCI statement handle, the modified SQL text, instead of the original SQL text, will be returned.

To turn on N' substitution in ksh shell:

```
export ORA_NCHAR_LITERAL_REPLACE=TRUE
```

To turn on N' substitution in csh shell:

```
setenv ORA_NCHAR_LITERAL_REPLACE TRUE
```

If a remote database is of a release before 10.2, N' substitution is not done.

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() calls.

See Also:

- For more information about the `xtramemsz` parameter and user memory allocation, refer to "User Memory Allocation" on page 2-13.
- For a code example illustrating setting N' substitution in a related function, see "OCIEnvCreate()" on page 15-13

Related Functions

OCIHandleAlloc(), OCIHandleFree(), OCITerminate(), OCINlsEnvironmentVariableGet()

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_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 (that is, 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.

See Also:

- For information about using the OCI to write multithreaded applications, refer to "Overview of OCI Multithreaded Development" on page 9-1.
- 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 release a session that was retrieved using `OCILogon2()` or `OCILogon()`.

Syntax

```
sword OCILogoff ( OCISvcCtx      *svchp
                  OCIError       *errhp );
```

Parameters

svchp (IN)
The service context handle which was used in the call to OCILogon() or OCILogon2().

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

Comments

This function is used to release a session that was retrieved using `OCILogon2()` or `OCILogon()`. If `OCILogon()` was used, then this function terminates the connection and session. If `OCILogon2()` was used, then the exact behavior of this call is determined by the `mode` in which the corresponding `OCILogon2()` function was called. In the default case, it will close the session/connection. For connection pooling, it closes the session and returns the connection to the pool. For session pooling, it returns the session/connection pair to the pool.

> **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 2-15.

Related Functions

OCILogon(),OCILogon2()

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 user name. Must be in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

uname_len (IN)
The length of `user name`, in number of bytes, regardless of the encoding.

password (IN)
The user's password. Must be in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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 the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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 2-15.

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

Get a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The `mode` that the function is called with determines its behavior.

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. For connection pooling and session pooling, this must be the one that the respective pool was created in.

errhp (IN/OUT)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error.

svchp (IN/OUT)
Address of an OCI service context pointer. This will be filled with a server and session handle.

In the default case, a new session and server handle will be allocated, the connection and session will be started, and the service context will be populated with these handles.

For connection pooling, a new session handle will be allocated, and the session will be started over a virtual connection from the connection pool.

For session pooling, the service context will be populated with an existing session/server handle pair from the session pool.

Note that the user must not change any attributes of the server and user/session handles associated with the service context pointer. Doing so will result in an error being returned by the `OCIAttrSet()` call.

The only attribute of the service context that can be altered is `OCI_ATTR_STMTCACHESIZE`.

username (IN)
The user name used to authenticate the session. Must be in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

uname_len (IN)
The length of `username`, in number of bytes, regardless of the encoding.

password (IN)
The user's password. For connection pooling, if this parameter is NULL then 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 the encoding specified by the charset parameter of a previous call to OCIEnvNlsCreate().

passwd_len (IN)
The length of password, in number of bytes, regardless of the encoding.

dbname (IN)
For the default case, this indicates the connect string to use to connect to the Oracle database server.

For connection pooling, it indicates the connection pool to retrieve the virtual connection from, in order to start up the session. This value is returned by the OCIConnectionPoolCreate() call.

For session pooling, it indicates the pool to get the session from. It is returned by the OCISessionPoolCreate() call.

Must be in the encoding specified by the charset parameter of a previous call to OCIEnvNlsCreate().

dbname_len (IN)
The length of dbname. For session pooling and connection pooling, this value is returned by the OCISessionPoolCreate() or OCIConnectionPoolCreate() call respectively.

mode (IN)
The values accepted are

- OCI_DEFAULT
- OCI_LOGON2_CPOOL
- OCI_LOGON2_SPOOL
- OCI_LOGON2_STMTCACHE
- OCI_LOGON2_PROXY

For the default (non-pooling case), the following modes are valid:

OCI_DEFAULT - Equivalent to calling OCILogon().

OCI_LOGON2_STMTCACHE - Enable statement caching.

For connection pooling, the following modes are valid:

OCI_LOGON2_CPOOL or OCI_CPOOL - This must be set in order to use connection pooling.

OCI_LOGON2_STMTCACHE - Enable statement caching.

In order to use proxy authentication for connection pooling, the password must be set to NULL. The user will then be given a session that is authenticated by the user name provided in the OCILogon2() call, through the proxy credentials supplied in the OCIConnectionPoolCreate() call.

For session pooling, the following modes are valid:

OCI_LOGON2_SPOOL - This must be set in order to use session pooling.

OCI_LOGON2_STMTCACHE - Enable statement caching.

`OCI_LOGON2_PROXY` - Use proxy authentication.The user is given a session that is authenticated by the user name provided in the `OCILogon2()` call, through the proxy credentials supplied in the `OCISessionPoolCreate()` call.

Comments

None.

Related Functions

OCILogon(),OCILogoff(),OCISessionGet(),OCISessionRelease()

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 the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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.

Example

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 **) 0);
OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
     OCI_HTYPE_SVCCTX, 0, (dvoid **) 0);
/* 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 a user session context, which is initialized by this call.

Specifies the type of credentials to use for establishing the user session. Valid values for `credt` are:

- `OCI_CRED_RDBMS` - authenticate using a database user name 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 user name 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_MIGRATE` should not be used when the session uses connection pool underneath. The session migration and multiplexing happens transparently to the user.
- `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 user name 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 user name 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.

Do not set OCI_MIGRATE flag in the call to OCISessionBegin(), when the virtual server handle points to a connection pool(OCIServerAttach() called with mode set to OCI_CPOOL). Oracle supports passing this flag only for compatibility reasons. Do not use the OCI_MIGRATE flag, as the perception that the user gets when using a connection pool is of sessions having their own dedicated (virtual) connections which are transparently multiplexed onto real connections.

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 user name 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 user name 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 user name 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 *)"hr",
     (ub4)strlen("hr"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"hr",
     (ub4)strlen("hr"), 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()

OCISessionGet()

Purpose

Get a session. This session may be a new one with a new underlying connection, or one that is started over a virtual connection from an existing connection pool, or one from an existing session pool. The `mode` that the function is called with determines its behavior.

Syntax

```
sword OCISessionGet ( OCIenv            *envhp,
                      OCIError          *errhp,
                      OCISvcCtx         **svchp,
                      OCIAuthInfo       *authInfop,
                      OraText           *dbName,
                      ub4               dbName_len,
                      CONST OraText     *tagInfo,
                      ub4               tagInfo_len,
                      OraText           **retTagInfo,
                      ub4               *retTagInfo_len,
                      boolean           *found,
                      ub4               mode );
```

Parameters

envhp (IN/OUT)
OCI environment handle. For connection pooling and session pooling, this should be the one that the respective pool was created in.

errhp (IN/OUT)
OCI error handle.

svchp (OUT)
Address of an OCI service context pointer. This will be filled with a server and session handle.

In the default case, a new session and server handle will be allocated, the connection and session will be started, and the service context will be populated with these handles.

For connection pooling, a new session handle will be allocated, and the session will be started over a virtual connection from the connection pool.

For session pooling, the service context will be populated with an existing session and server handle pair from the session pool.

Do not change any attributes of the server and user and session handles associated with the service context pointer. Doing so will result in an error being returned by the `OCIAttrSet()` call.

The only attribute of the service context that can be altered is `OCI_ATTR_STMTCACHESIZE`.

authInfop (IN)
Authentication Information handle to be used while getting the session.

In the default and connection pooling cases, this handle can take all the attributes of the session handle.

For session pooling, the authentication information handle is considered only if the session pool mode is not set to OCI_SPC_HOMOGENEOUS. In this case, this handle can have the following attributes set:

OCI_ATTR_USERNAME

OCI_ATTR_PASSWORD

OCI_ATTR_INITIAL_CLIENT_ROLES

Please refer to user handle attributes for more information.

See Also: "User Session Handle Attributes" on page A-12

dbName (IN)
For the default case, this indicates the connect string to use to connect to the Oracle database server.

For connection pooling, it indicates the connection pool to retrieve the virtual connection from, in order to start up the session. This value is returned by the OCIConnectionPoolCreate() call.

For session pooling, it indicates the pool to get the session from. It is returned by the OCISessionPoolCreate() call.

dbname_len (IN)
The length of dbName. For session pooling and connection pooling, this value is returned by the call to OCISessionPoolCreate() or OCIConnectionPoolCreate(), respectively.

tagInfo (IN)
This parameter is only used for session pooling.

This indicates the type of session that the user wants. If the user wants a default session, the user must set this to NULL. Please refer to the Comments for a detailed usage of this parameter.

tagInfo_len (IN)
The length in bytes, of tagInfo. Used for session pooling only.

retTagInfo (OUT)
This parameter is only used for session pooling. This indicates the type of session that is returned to the user. Please refer to the Comments for a detailed usage of this parameter.

retTagInfo_len (OUT)
The length in bytes, of retTagInfo. Used for session pooling only.

found (OUT)
This parameter is only used for session pooling. If the type of session that the user requested was returned (that is, the value of tagInfo and retTagInfo is the same), then found is set to TRUE, else, found is set to FALSE.

mode (IN)
The valid modes are

- OCI_DEFAULT
- OCI_SESSGET_CPOOL
- OCI_SESSGET_SPOOL

- OCI_SESSGET_CREDPROXY
- OCI_SESSGET_CREDEXT
- OCI_SESSGET_SPOOL_MATCHANY
- OCI_SESSGET_STMTCACHE.

In the default (non-pooling) case, the following modes are valid:

OCI_SESSGET_STMTCACHE - This will enable statement caching in the session.

OCI_SESSGET_CREDEXT - This will return a session authenticated with external credentials.

For connection pooling, the following modes are valid:

OCI_SESSGET_CPOOL - This must be set in order to use connection pooling.

OCI_SESSGET_STMTCACHE - This will enable statement caching in the session.

OCI_SESSGET_CREDPROXY - This will return a proxy session. The user is given a session that is authenticated by the user name provided in the OCISessionGet() call, through the proxy credentials supplied in the OCIConnectionPoolCreate() call.

OCI_SESSGET_CREDEXT - This will return a session authenticated with external credentials.

For session pooling, the following modes are valid:

OCI_SESSGET_SPOOL - This must be set in order to use session pooling.

OCI_SESSGET_CREDPROXY - In this case, the user is given a session that is authenticated by the user name provided in the OCISessionGet() call, through the proxy credentials supplied in the OCISessionPoolCreate() call.

OCI_SESSGET_SPOOL_MATCHANY - This refers to the tagging behavior. If this mode is set, then a session which has a different tag than what was asked for, may be returned. Please refer to the Comments section.

Comments

The tags provide a way for users to customize sessions in the pool. A client can get a default or untagged session from a pool, set certain attributes on the session (such as Globalization settings), and return the session to the pool, labeling it with an appropriate tag in the OCISessionRelease() call.

The user, or some other user, can request for a session with the same attributes, and can do so by providing the same tag in the OCISessionGet() call.

If a user asks for a session with tag 'A', and a matching session is not available, an appropriately authenticated untagged session (session with a NULL tag) will be returned, if such a session is free. If even an untagged session is not free *and* OCI_SESSGET_SPOOL_MATCHANY has been specified, then an appropriately authenticated session with a different tag will be returned. If OCI_SESSGET_SPOOL_MATCHANY is not set, then a session with a different tag is never returned.

Related Functions

OCISessionRelease(), OCISessionPoolCreate(), OCISessionPoolDestroy()

OCISessionPoolCreate()

Purpose

Initializes a session pool. It starts up `sessMin` number of sessions and connections to the database. Before making this call, make a call to `OCIHandleAlloc()` to allocate memory for the session pool handle.

Syntax

```
sword OCISessionPoolCreate ( OCIEnv           *envhp,
                             OCIError         *errhp,
                             OCISPool         *spoolhp,
                             OraText          **poolName,
                             ub4              *poolNameLen,
                             CONST OraText    *connStr,
                             ub4              connStrLen,
                             ub4              sessMin,
                             ub4              sessMax,
                             ub4              sessIncr,
                             OraText          *userid,
                             ub4              useridLen,
                             OraText          *password,
                             ub4              passwordLen,
                             ub4              mode );
```

Parameters

envhp (IN)
A pointer to the environment handle in which the session pool needs to be created.

errhp (IN/OUT)
An error handle which can be passed to `OCIErrorGet()`.

spoolhp (IN/OUT)
A pointer to the session pool handle that is initialized.

poolName (OUT)
The name of the session pool returned. It is unique across all session pools in an environment. This value must be passed to the `OCISessionGet()` call.

poolNameLen (OUT)
Length of `poolName` in bytes.

connStr (IN)
The TNS alias of the database to connect to.

connStrLen (IN)
The length of `connStr` in bytes.

sessMin (IN)
Specifies the minimum number of sessions in the session pool.

This number of sessions are started by `OCISessionPoolCreate()`. After this, sessions are opened only when necessary.

This value is used when `mode` is set to `OCI_SPC_HOMOGENEOUS`. In all other cases it is ignored.

sessMax (IN)
Specifies the maximum number of sessions that can be opened in the session pool. Once this value is reached, no more sessions are opened. The valid values are 1 and higher.

sessIncr (IN)
Allows applications to set the next increment for sessions to be started if the current number of sessions are less than `sessMax`. The valid values are 0 and higher.

`sessMin + sessIncr` cannot be more than `sessMax`.

userid (IN)
Specifies the userid with which to start up the sessions.

> **See Also:** "Authentication Note." on page 15-41

useridLen (IN)
Length of the userid in bytes.

password (IN)
The password for the corresponding userid.

passwordLen (IN)
The length of the password in bytes.

mode (IN)
The modes supported are

- `OCI_DEFAULT` - for a new session pool creation.
- `OCI_SPC_REINITIALIZE` - After creating a session pool, if you wish to change the pool attributes dynamically (change the `sessMin`, `sessMax`, and `sessIncr` parameters), call `OCISessionPoolCreate()` with `mode` set to `OCI_SPC_REINITIALIZE`. When `mode` is set to `OCI_SPC_REINITIALIZE`, then `connStr`, `userid`, and `password` will be ignored.

`OCI_SPC_STMTCACHE` - an OCI statement cache will be created for the session pool. If the pool is not created with OCI statement caching turned on, server-side statement caching will automatically be used. Please note that in general, client- side statement caching will give better performance.

> **See Also:** "Statement Caching in OCI" on page 9-20

- `OCI_SPC_HOMOGENEOUS` - all sessions in the pool will be authenticated with the user name and password passed to `OCISessionPoolCreate()`. The authentication handle (parameter `authInfo`) passed into `OCISessionGet()` is ignored in this case. Moreover, the `sessMin` and the `SessIncr` values are considered only in this case. No proxy session can be created in this mode.

Comments

Authentication Note.
Please note that a session pool may contain two types of connections to the database: direct connections and proxy connections. To make a proxy connection, a user must have Connect through Proxy privilege.

See Also: For more information on proxy connections, see

- "Client Access Through a Proxy" on page 2-15
- *Oracle Database SQL Reference*
- *Oracle Database Concepts*

When the session pool is created, the `userid` and `password` may or may not be specified. If these values are `NULL`, no proxy connections can exist in this pool. If `mode` is set to `OCI_SPC_HOMOGENEOUS`, no proxy connection can exist.

A `userid` and `password` pair may also be specified through the authentication handle in the `OCISessionGet()` call. If this call is made with `mode` set to `OCI_SESSGET_CREDPROXY`, then the user is given a session that is authenticated by the `userid` provided in the `OCISessionGet()` call, through the proxy credentials supplied in the `OCISessionPoolCreate()` call. In this case, the password in the `OCISessionGet()` call is ignored.

If `OCISessionGet()` is called with `mode` *not* set to `OCI_SESSGET_CREDPROXY`, then the user gets a direct session which is authenticated by the credentials provided in the `OCISessionGet()` call. If none have been provided in this call, the user gets a session authenticated by the credentials in the `OCISessionPoolCreate()` call.

Related Functions

OCISessionRelease(), OCISessionGet(), OCISessionPoolDestroy()

OCISessionPoolDestroy()

Purpose

Destroys a session pool.

Syntax

```
sword OCISessionPoolDestroy ( OCISPool     *spoolhp,
                              OCIError     *errhp,
                              ub4          mode );
```

spoolhp (IN/OUT)
The session pool handle for the session pool to be destroyed.

errhp (IN/OUT)
An error handle which can be passed to `OCIErrorGet()`.

mode (IN)
Currently, `OCISessionPoolDestroy()` will support modes `OCI_DEFAULT` and `OCI_SPD_FORCE`.

If this call is made with `mode` set to `OCI_SPD_FORCE`, and there are active sessions in the pool, the sessions will be closed and the pool will be destroyed. However, if this mode is not set, and there are busy sessions in the pool, an error will be returned.

Related Functions

OCISessionPoolCreate(), OCISessionRelease(), OCISessionGet()

OCISessionRelease()

Purpose

This function is used to release a session that was retrieved using `OCISessionGet()`. The exact behavior of this call is determined by the `mode` in which the corresponding `OCISessionGet()` function was called. In the default case, it will close the session/connection. For connection pooling, it closes the session and returns the connection to the pool. For session pooling, it returns the session/connection pair to the pool.

Syntax

```
sword OCISessionRelease ( OCISvcCtx       *svchp,
                          OCIError        *errhp,
                          OraText         *tag,
                          ub4             tag_len,
                          ub4             mode );
```

Parameters

svchp (IN)
The service context that was populated during the corresponding `OCISessionGet()` call.

In the default case, the session and connection associated with this handle will be closed.

In the connection pooling case, the session will be closed and the connection released to the pool.

For session pooling, the session/connection pair associated with this service context will be released to the pool.

errhp (IN/OUT)
The OCI error handle.

tag (IN)
This parameter is only used for session pooling.

This parameter will be ignored unless mode `OCI_SESSRLS_RETAG` is specified. In this case, the session is labelled with this tag and returned to the pool. If this is `NULL`, then the session is not tagged.

tag_len (IN)
This parameter is only used for session pooling.

Length of the tag. This is ignored unless mode `OCI_SESSRLS_RETAG` is set.

mode (IN)
The supported modes are

- `OCI_DEFAULT`
- `OCI_SESSRLS_DROPSESS`
- `OCI_SESSRLS_RETAG`

For the default case and for connection pooling, only `OCI_DEFAULT` can be used.

OCI_SESSRLS_DROPSESS and OCI_SESSRLS_RETAG are only used for session pooling.

When OCI_SESSRLS_DROPSESS is specified, the session will be removed from the session pool.

If and only if OCI_SESSRLS_RETAG is set, will the tag on the session be altered. If this mode is not set, the tag and tag_len parameters will be ignored.

Comments

In this call the user be careful to pass in the correct tag. If a default session is requested and the user sets certain properties on this session (probably through an ALTER SESSION command), then the user must label this session appropriately by tagging it as such.

If on the other hand, the user requested a tagged session and got one, and has changed the properties on the session, then the user must pass in a different tag if appropriate.

For the correct working of the session pool layer the application developer must be very careful to pass in the correct tag to the OCISessionGet() and OCISessionRelease() calls.

Related Functions

OCISessionGet(), OCISessionPoolCreate(), OCISessionPoolDestroy(),OCILogon2()

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 terminate.

Related Functions

OCIInitialize()

Handle and Descriptor Functions

This section describes the OCI handle and descriptor functions.

Table 15–3 Handle and Descriptor Functions

Function	Purpose
OCIAttrGet() on page 15-48	Get the attributes of a handle
OCIAttrSet() on page 15-50	Set an attribute of a handle or descriptor
OCIDescriptorAlloc() on page 15-51	Allocate and initialize a descriptor or LOB locator
OCIDescriptorFree() on page 15-53	Free a previously allocated descriptor
OCIHandleAlloc() on page 15-54	Allocate and initialize a handle
OCIHandleFree() on page 15-55	Free a previously allocated handle
OCIParamGet() on page 15-56	Get a parameter descriptor
OCIParamSet() on page 15-58	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, and so on. 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. Will be in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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:** 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() to get the Unicode information in an environment or statement handle.

> **See Also:** "Examples Using OCIDescribeAny()" on page 6-19 and "Describing Select-list Items" on page 4-9

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 the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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.

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 2-13

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
- OCI_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 handles are described in this table:

> **See Also:** Table 2–1, " OCI Handle Types"

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 2-13

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 handles are described in this table:

See Also: Table 2–1, " OCI Handle Types"

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.

When a statement handle is freed, the cursor associated with the statement handle will be closed, but the actual cursor closing may be deferred to the next round trip to the server. So if the application needs to close the cursor immediately, you can make a server round trip call, such as `OCIServerVersion()` or `OCIPing()`, after the `OCIHandleFree()` call.

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: 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 10-15.

Related Functions

OCIParamGet()

Bind, Define, and Describe Functions

This section describes the bind, define, and describe functions.

Table 15–4 Bind, Define, and Describe Functions

Function	Purpose
OCIBindArrayOfStruct() on page 15-60	Set skip parameters for static array bind
OCIBindByName() on page 15-61	Bind by name
OCIBindByPos() on page 15-65	Bind by position
OCIBindDynamic() on page 15-69	Sets additional attributes after bind with `OCI_DATA_AT_EXEC` mode
OCIBindObject() on page 15-72	Set additional attributes for bind of named datatype
OCIDefineArrayOfStruct() on page 15-74	Set additional attributes for static array define
OCIDefineByPos() on page 15-75	Define an output variable association
OCIDefineDynamic() on page 15-79	Sets additional attributes for define in `OCI_DYNAMIC_FETCH` mode
OCIDefineObject() on page 15-81	Set additional attributes for define of named datatype
OCIDescribeAny() on page 15-83	Describe existing schema objects
OCIStmtGetBindInfo() on page 15-86	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, "Binding and Defining Arrays of Structures in OCI" on page 5-16.

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 in the encoding specified by the charset parameter of a previous call to OCIEnvNlsCreate(), unless users call OCIAttrSet() to manually reset the character set for the bind handle.

See Also: "Bind Handle Attributes" on page A-28.

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 datatype of the value(s) being bound. Named datatypes (SQLT_NTY) and REFs (SQLT_REF) are valid only if the application has been initialized in object mode. For named datatypes, 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 datatypes 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 2-23

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` - The default mode. The statement handle `stmtp` uses whatever is specified by its parent environment handle.
- `OCI_BIND_SOFT` - Soft bind mode. This mode increases the performance of the call. If this is the first bind or some input value like `dty` or `value_sz` is changed from the previous bind, this mode is ignored. An error is returned if the statement is not executed. Unexpected behavior results if the bind handle passed is not valid.
- `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, "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-29.

  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.

> **Note:** After using `OCIEnvNlsCreate()` to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes.

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, "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-29

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 datatypes 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 64 Kbytes are required, use `OCIBindDynamic()`.
- If a named datatype 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 a LOB, the buffer pointer must be a pointer to a LOB locator of type `OCILobLocator`. Give the address of the pointer.

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: Bind Handle Attributes on page A-28.

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 `REF`s, whose size is unknown to client applications, use the size of the structure you are passing: for example, `sizeof(OCILobLocator *)`.

For a PL/SQL block, a `value_sz` greater than the width of a `CHAR` column will cause an error, because of how PL/SQL processes the `CHAR` datatype.

dty (IN)
The datatype of the value(s) being bound. Named datatypes (SQLT_NTY) and `REF`s (SQLT_REF) are valid only if the application has been initialized in object mode. For named datatypes, or `REF`s, 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 datatypes, this is a pointer to `sb2` or an array of `sb2` values. 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()`. `indp` is ignored for dynamic binds. If `valuep` is an `OUT` parameter, then you must set `indp` to point to `OCI_IND_NULL`.

See Also: "Indicator Variables" on page 2-23

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. If `valuep` is an `OUT` parameter, then you must set `alenp` to point to 0.

Note: If `alenp` is greater than `value_sz`, data will be skipped.

---

rcodep (OUT)
Pointer to an 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_BIND_SOFT - Soft bind mode. This mode increases the performance of the call. If this is the first bind or some input value like dty or value_sz is changed from the previous bind, this mode is ignored. An error is returned if the statement is not executed. Unexpected behavior results if the bind handle passed is not valid.
- 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 in OCI" on page 5-29.

  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.

Note: After using OCIEnvNlsCreate() to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes.

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 in OCI" on page 5-29.

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 datatypes 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 datatype 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

™p 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 set `*bufpp` to `lobp`, not `*lobp`.

For `REF`s, pass the address of the ref; that is, 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: "Bind Handle Attributes" on page A-28.

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; for example, `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 datatypes.

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 "Bind Handle Attributes" on page A-28.

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 datatypes.

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 terminate processing immediately.

> **See Also:** For more information about the OCI_DATA_AT_EXEC mode, see the section "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-29.

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.

Note: After using OCIEnvNlsCreate() to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes.

Related Functions

OCIBindByName(), OCIBindByPos()

OCIBindObject()

Purpose

This function sets up additional attributes which are required for a named datatype (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 `REF`s in SQL, but required for `REF`s 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 Datatype 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 Datatype 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 datatype 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 Datatypes and `REF`s 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 pointers to these newly allocated indicator structures. 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 sizes 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 datatype 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 datatype 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 5-17.

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.

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.

For a LOB, the buffer pointer must be a pointer to a LOB locator of type `OCILobLocator`. Give the address of the pointer.

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:** "Bind Handle Attributes" on page A-28

dty (IN)
The datatype. Named datatype (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 datatypes, pointer to sb2 or an array of sb2s. Ignored for SQLT_NTY defines. For SQLT_NTY defines, a pointer to a named datatype indicator structure or an array of named datatype indicator structures is associated by a subsequent OCIDefineObject() call.

> **See Also:** "Indicator Variables" on page 2-23

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_DEFINE_SOFT - Soft define mode. This mode increases the performance of the call. If this is the first define or some input parameter like dty or value_sz is changed from the previous define, this mode is ignored. Unexpected behavior results if a non-valid define handle is passed. An error is returned if the statement is not executed.
- OCI_DYNAMIC_FETCH - For applications requiring dynamically allocated data at the time of fetch, this mode must be used. 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.

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.

> **Note:** After using OCIEnvNlsCreate() to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes.

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 datatypes 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 datatype (that is, 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 datatypes.

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 datatype.

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 in OCI" on page 5-29.
- For more information about defines, see "Overview of Defining in OCI" on page 5-12.

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 next:

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 in OCI" on page 5-29.

Note: After using `OCIEnvNlsCreate()` to create the environment handle, the actual lengths and returned lengths of bind and define handles are always in number of bytes.

Related Functions

OCIDefineByPos()

See Also: *Oracle Database Application Developer's Guide - Fundamentals*

OCIDefineObject()

Purpose

Sets up additional attributes necessary for a named datatype 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 datatype 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 datatype instance(s). The application must call OCIObjectFree() to deallocate the named datatype 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 ub4s.

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 ub4s.

Comments

This function follows a call to `OCIDefineByPos()` to set initial define information. This call sets up additional attributes necessary for a Named Datatype 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 datatype being defined. The TDO can be retrieved with a call to `OCIDescribeAny()`.

> **See Also:** "OCIInitialize()" on page 15-22 for more information about initializing the OCI process environment.

Related Functions

OCIDefineByPos()

OCIDescribeAny()

Purpose

Describes existing schema and subschema 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 in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.
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 `hr.employees.employee_id@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 or subschema 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 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.

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. Will be in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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 the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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.

`OCI_NO_DATA` will be returned if the statement has no bind variables or if the starting bind position specified by the you in the invocation does not exist in the statement.

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()

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.

> **See Also:** For code examples, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".

This chapter contains these topics:

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 in OCI" on page 2-20

Conventions for OCI Functions

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 next.

Table 16–1 Mode of a Parameter

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.

Returns

This optional section describes the possible values that can be returned. It can be found either before or after the Comments section.

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".

Statement Functions

This section describes the statement functions.

Table 16–2 Statement Functions

Function	Purpose
OCIStmtExecute() on page 16-4	Sends statements to server for execution
OCIStmtFetch() on page 16-7	Fetches rows from a query (deprecated)
OCIStmtFetch2() on page 16-8	Fetches rows from a query
OCIStmtGetPieceInfo() on page 16-10	Gets piece information for piecewise operations
OCIStmtPrepare() on page 16-12	Prepares a SQL or PL/SQL statement for execution.
OCIStmtPrepare2() on page 16-14	Prepares a SQL or PL/SQL statement for execution.
OCIStmtRelease() on page 16-16	Releases the statement handle.
OCIStmtSetPieceInfo() on page 16-17	Sets 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 datatypes 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.

> **Note:** For array DML operations, set `iters` <= 32767 to get better performance.

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" on page 4-7, for information about this mode.
- OCI_COMMIT_ON_SUCCESS - When a statement is executed in this mode, the current transaction is committed after execution, if 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 additional 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 cannot be updated. See "Fetching Results" on page 4-12. 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 4-9

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-24345 A truncation or `NULL` fetch error occurred
- ORA-24347 Warning of a `NULL` column in an aggregate function

If you call `OCIStmtFetch()` with the `nrows` parameter set to 0, 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` - positions the result set on the previous row from the current row in the result set. You can fetch multiple rows using this mode, from the "previous row" also.
- `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 similarly 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.

If you call OCIStmtFetch2() with the nrows parameter set to 0, this cancels the cursor.

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 the 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 in OCI" on page 4-14 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 or fetch call returns `OCI_NEED_DATA` to get or return a dynamic bind or define value or piece, `OCIStmtGetPieceInfo()` returns the relevant information: bind or define handle, iteration, index number and which piece.

See Also: "Runtime Data Allocation and Piecewise Operations in OCI" on page 5-29 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 must be in the encoding specified by the `charset` parameter of a previous call to `OCIEnvNlsCreate()`.

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.

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 only possible value is:

- `OCI_DEFAULT` - default mode. The statement handle `stmtp` uses whatever is specified by its parent environment handle.

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:** "Preparing Statements" on page 4-3 for more information about using this call.

Related Functions

OCIAttrGet(), OCIStmtExecute()

OCIStmtPrepare2()

Purpose

This call prepares a SQL or PL/SQL statement for execution. The user has the option of using the statement cache, if it has been enabled.

Syntax

```
sword OCIStmtPrepare2 ( OCISvcCtx      *svchp,
                        OCIStmt        **stmthp,
                        OCIError       *errhp,
                        CONST OraText  *stmttext,
                        ub4            stmt_len,
                        CONST OraText  *key,
                        ub4            keylen,
                        ub4            language,
                        ub4            mode );
```

Parameters

svchp (IN)
The service context to be associated with the statement.

errhp (IN)
A pointer to the error handle for diagnostics.

stmthp (OUT)
Pointer to the statement handle returned.

stmttext (IN)
The statement text. The semantics of the `stmttext` are same as that of `OCIStmtPrepare()`, that is, the string must be `NULL`-terminated.

stmt_len (IN)
The statement text length.

key (IN)
For statement caching only. The key to the returned statement in the cache. This can be used for future calls to `OCIStmtPrepare2()`, in which case there is no need to pass in the statement text and related parameters. If the key is passed in, then the statement text and other parameters are ignored and the search is solely based on the key.

keylen (IN)
For statement caching only. The length of the key.

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)
This function can be used with and without statement caching. This is determined at the time of connection or session pool creation. If caching is enabled for a session, then all statements in the session will have caching enabled, and if caching is not enabled, then all statements will not be cached.

The valid modes are:

- OCI_DEFAULT - For non-caching, this is the only valid setting. If the statement is not found in the cache, it allocates a new statement handle and prepares the statement handle for execution. If it is not found and
  - only the text has been supplied: a new statement will be allocated and prepared and returned. The tag will be NULL. OCI_SUCCESS will be returned.
  - only the tag has been supplied: stmthp will be NULL. OCI_ERROR will be returned.
  - both text and key were supplied: a new statement will be allocated and prepared and returned. The tag will be NULL. OCI_SUCCESS_WITH_INFO will be returned, as the returned statement differs from the requested statement in that the tag is NULL.
- OCI_PREP2_CACHE_SEARCHONLY - In this case, if the statement is not found (a NULL statement handle is returned), you must take further action. If the statement is found, OCI_SUCCESS will be returned. Otherwise, OCI_ERROR will be returned.
- OCI_PREP2_GET_PLSQL_WARNINGS - If warnings are enabled in the session and the PL/SQL program is compiled with warnings, then OCI_SUCCESS_WITH_INFO will be the return status from the execution. Use OCIErrorGet() to find the new error number corresponding to the warnings.

Related Functions

OCIStmtRelease()

OCIStmtRelease()

Purpose

Releases the statement handle obtained by a call to `OCIStmtPrepare2()`.

Syntax

```
sword OCIStmtRelease ( OCIStmt        *stmthp,
                       OCIError       *errhp,
                       CONST OraText  *key,
                       ub4            keylen,
                       ub4            mode );
```

Parameters

stmthp (IN/OUT)
The statement handle returned by `OCIStmtPrepare2()`

errhp (IN)
The error handle used for diagnostics.

key (IN)
Only valid for statement caching. The key to be associated with the statement in the cache. This can be the key returned by `OCIStmtPrepare2()` or can be a new key. If a `NULL` key is passed in the statement will not be tagged.

keylen (IN)
Only valid for statement caching. The length of the key.

mode (IN)
The valid modes are

- `OCI_DEFAULT`
- `OCI_STRLS_CACHE_DELETE` - Only valid for statement caching. The statement will not be kept in the cache any more.

Related Functions

OCIStmtPrepare2()

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 datatypes or `REF`s, a pointer to the object or `REF` is returned.

alenp (IN/OUT)
The length of the piece or the value. Do not change this parameter between executions of the same SQL statement.

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 datatypes (SQLT_NTY) and `REF`s (SQLT_REF), that is, `*indp` is either an `sb2` or a `dvoid *` depending upon the datatype.

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 in OCI" on page 5-29

Related Functions

OCIAttrGet(), OCIAttrSet(), OCIStmtExecute(), OCIStmtFetch(), OCIStmtGetPieceInfo()

LOB Functions

This section describes the LOB functions which use the LOB locator. Use functions that end in "2" for all new applications.

Note: There is another way of accessing LOBs -- using the data interface for LOBs. You can bind or define character data for a `CLOB` column or `RAW` data for a `BLOB` column, as described in these sections:

See Also:

- "Binding LOB Data" on page 5-8 for usage and examples for both `INSERT` and `UPDATE statements`
- "Defining LOB Data" on page 5-15 for usage and examples of `SELECT` statements
- Chapter 7, "LOB and BFILE Operations"

Table 16–3 LOB Functions

Function	Purpose
OCIDurationBegin() on page 16-22	Start user duration for temporary LOB
OCIDurationEnd() on page 16-23	End user duration for temporary LOB
OCILobAppend() on page 16-24	Append one LOB to another
OCILobArrayRead() on page 16-26	Read LOB data for multiple locators.
OCILobArrayWrite() on page 16-30	Write LOB data for multiple locators.
OCILobAssign() on page 16-34	Assign one LOB locator to another
OCILobCharSetForm() on page 16-36	Get character set form from LOB locator
OCILobCharSetId() on page 16-37	Get character set ID from LOB locator
OCILobClose() on page 16-38	Close a previously opened LOB
OCILobCopy() on page 16-39	Copy all or part of one LOB to another
OCILobCopy2() on page 16-41	Copy all or part of one LOB to another.
OCILobCreateTemporary() on page 16-42	Create a temporary LOB
OCILobDisableBuffering() on page 16-44	Turn LOB buffering off
OCILobEnableBuffering() on page 16-45	Turn LOB buffering on
OCILobErase() on page 16-46	Erase a portion of a LOB
OCILobErase2() on page 16-48	Erase a portion of a LOB.
OCILobFileClose() on page 16-49	Close a previously opened `BFILE`
OCILobFileCloseAll() on page 16-50	Close all previously opened files
OCILobFileExists() on page 16-51	Check if a file exists on the server
OCILobFileGetName() on page 16-52	Get directory object and file name from the LOB locator
OCILobFileIsOpen() on page 16-54	Check if file on server is open using this locator

Table 16–3 (Cont.) LOB Functions

Function	Purpose
OCILobFileOpen() on page 16-55	Open a BFILE
OCILobFileSetName() on page 16-56	Set directory object and file name in the LOB locator
OCILobFlushBuffer() on page 16-57	Flush the LOB buffer
OCILobFreeTemporary() on page 16-58	Free a temporary LOB
OCILobGetChunkSize() on page 16-59	Get the chunk size of a LOB
OCILobGetLength() on page 16-60	Get length of a LOB
OCILobGetLength2() on page 16-61	Get length of a LOB.
OCILobGetStorageLimit() on page 16-62	Get the maximum length of an internal LOB (BLOB, CLOB, or NCLOB) in bytes.
OCILobIsEqual() on page 16-63	Compare two LOB locators for Equality
OCILobIsOpen() on page 16-64	Check to see if a LOB is open
OCILobIsTemporary() on page 16-65	Determine if a given LOB is temporary
OCILobLoadFromFile2() on page 16-68	Load a LOB from a BFILE.
OCILobLocatorAssign() on page 16-69	Assign one LOB locator to another
OCILobLocatorIsInit() on page 16-71	Check to see if a LOB locator is initialized
OCILobOpen() on page 16-72	Open a LOB
OCILobRead() on page 16-74	Read a portion of a LOB
OCILobRead2() on page 16-78	Read a portion of a LOB.
OCILobTrim() on page 16-82	Truncate a LOB
OCILobTrim2() on page 16-83	Truncate a LOB.
OCILobWrite() on page 16-84	Write into a LOB
OCILobWrite2() on page 16-88	Wrie into a LOB.
OCILobWriteAppend() on page 16-92	Write data beginning at the end of a LOB
OCILobWriteAppend2() on page 16-95	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 a user duration. In release 8.1 or later, user durations can be used when creating temporary LOBs. A 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 7-15

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 );
```

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)
OCI service context. This should be passed as `NULL` for cartridge services.

duration (IN)
A number to identify the user duration.

Comments

This function terminates a user duration. Temporary LOBs that are allocated for the user duration are freed.

See Also: "Temporary LOB Durations" on page 7-15

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 (that is, 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 `BFILE` locator as the source or the destination.

It is not mandatory that you wrap this LOB operation inside the Open or 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 or 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 or close statements.

Related Functions

OCILobTrim(), OCILobWrite(), OCILobCopy(), OCIErrorGet(), OCILobWriteAppend()

OCILobArrayRead()

Purpose

Reads LOB data for multiple locators in one round trip. This function can be used for LOBs of size greater than or less than 4 GB.

Syntax

```
sword OCILobArrayRead ( OCISvcCtx                *svchp,
                        OCIError                 *errhp,
                        ub4                      *array_iter,
                        OCILobLocator            **locp_arr,
                        oraub8                   *byte_amt_arr,
                        oraub8                   *char_amt_arr,
                        oraub8                   *offset_arr,
                        dvoid                    **bufp_arr,
                        oraub8                   bufl_arr,
                        ub1                      piece,
                        dvoid                    *ctxp,
                        OCICallbackLobArrayRead  (cbfp)
                                                 (
                                                  dvoid         *ctxp,
                                                  ub4           array_iter,
                                                  CONST dvoid   *bufp,
                                                  oraub8        lenp,
                                                  ub1           piecep
                                                  dvoid         **changed_bufpp,
                                                  oraub8        *changed_lenp
                                                 )
                        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.

array_iter (IN/OUT)
IN - This parameter indicates the size of the LOB locator array. For polling this is relevant only for the first call and is ignored in subsequent calls.

OUT - While in polling mode, it indicates the array index of the element read from.

locp_arr (IN)
An array of LOB or `BFILE` locators.

byte_amt_arr (IN/OUT)
An array of `oraub8` variables. The array size must be the same as the locator array size. The entries correspond to the amount in bytes for the locators.

IN - The number of bytes to read from the database. Used for `BLOB` and `BFILE` always. For CLOB and NCLOB, it is used only when the corresponding value in `char_amt_arr` is zero.

OUT - The number of bytes read into the user buffer.

char_amt_arr (IN/OUT)
An array of oraub8 variables. The array size must be the same as the locator array size. The entries correspond to the amount in characters for the locators.

IN - The maximum number of characters to read into the user buffer. Ignored for BLOB and BFILE.

OUT - The number of characters read into the user buffer. Undefined for BLOB and BFILE.

offset_arr (IN)
An array of oraub8 variables. The array size must be the same as the locator array size. For character LOBs (CLOBs, NCLOBs) it is the number of characters from the beginning of the LOB; for binary LOBs or BFILEs it is the number of bytes. The first position is 1.

bufp _arr (IN/OUT)
An array of pointers to buffers into which the piece will be read. The array size must be the same as the locator array size.

bufl_arr (IN)
An array of oraub8 variables indicating the buffer lengths for the buffer array. The array size must be the same as the locator array size

piece (IN)
OCI_ONE_PIECE - The call never assumes polling. If the amount indicated is more than the buffer length then the buffer is filled as much as possible.

For polling, pass OCI_FIRST_PIECE the first time and OCI_NEXT_PIECE in subsequent calls. OCI_FIRST_PIECE should be passed while using the callback.

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 terminated.

ctxp (IN)
The context for the callback function. Can be NULL.

array_iter (IN)
The index of the element read from.

bufp (IN/OUT)
A buffer pointer for the piece.

lenp (IN)
The length in bytes of the current piece in bufp.

piecep (IN)
Which piece: OCI_FIRST_PIECE, OCI_NEXT_PIECE, or OCI_LAST_PIECE.

changed_bufpp (OUT)
The callback function can put the address of a new buffer if it prefers to use a new buffer for the next piece to read. The default old buffer `bufp` is used if this parameter is set to `NULL`.

changed_lenp (OUT)
Length of the new buffer, if provided.

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 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 nonzero 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

It is an error to try to read from a `NULL` LOB or `BFILE`.

> **Note:** When reading or writing LOBs, the character set form (`csfrm`) specified should match the form of the locator itself.

For `BFILE`s, the operating system file must already exist on the server, and it must have been opened by `OCILobFileOpen()` or `OCILobOpen()` using the input locator. The database 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 `OCILobArrayRead()`, the first call needs to specify values for `offset_arr` and `amt_arr`, but on subsequent polling calls to `OCILobArrayRead()`, the user need not specify these values.

If the LOB is a `BLOB`, the `csid` and `csfrm` parameters are ignored.

> **Note:** To terminate an `OCILobArrayRead()` operation and free the statement handle, use the `OCIBreak()` call.

The following points apply to reading LOB data in streaming mode:

- When using polling mode, be sure to specify the `char_amt_arr` and `byte_amt_arr` and `offset_arr` parameters only in the first call to `OCILobArrayRead()`. On subsequent polling calls these parameters are ignored. If both `byte_amt_arr` and `char_amt_arr` are set to point to zero and `OCI_FIRST_PIECE` is passed, then polling mode is assumed and data is read to the end of the LOB. On output, `byte_amt_arr` gives the number of bytes read in

the current piece. For CLOBs and NCLOBs, char_amt_arr gives the number of characters read in the current piece.

- When using callbacks, the lenp parameter, which is input to the callback, indicates how many bytes are filled in the buffer. Check the lenp parameter during your callback processing because the entire buffer may not be filled with data.
- When using polling, look at the byte_amt_arr parameter to see how much the buffer is filled for the current piece. For CLOBs and NCLOBs, char_amt_arr returns the number of characters read in the buffer as well.

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 in OCI" on page 5-28
- For more information about BFILEs, refer to the description of BFILEs in the *Oracle Database Application Developer's Guide - Large Objects*
- 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 in OCI" on page 5-29

Related Functions

OCIErrorGet(), OCILobWrite2(), OCILobFileSetName(), OCILobWriteAppend2(), OCILobArrayWrite()

OCILobArrayWrite()

Purpose

Writes LOB data for multiple locators in one round trip. This function can be used for LOBs of size greater than or less than 4 GB.

Syntax

```
sword OCILobArrayWrite ( OCISvcCtx             *svchp,
                         OCIError              *errhp,
                         ub4                   *array_iter,
                         OCILobLocator         **locp_arr,
                         oraub8                *byte_amt_arr,
                         oraub8                *char_amt_arr,
                         oraub8                *offset_arr,
                         dvoid                 **bufp_arr,
                         oraub8                *bufl_arr,
                         ub1                   piece,
                         dvoid                 *ctxp,
                         OCICallbackLobArrayWrite (cbfp)
                                               (
                                                 dvoid    *ctxp,
                                                 ub4      array_iter,
                                                 dvoid    *bufp,
                                                 oraub8   *lenp,
                                                 ub1      *piecep
                                                 dvoid    **changed_bufpp,
                                                 oraub8   *changed_lenp
                                               )
                         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.

array_iter (IN/OUT)
IN - This parameter indicates the size of the LOB locator array. For polling this is relevant only for the first call and is ignored in subsequent calls.

OUT - While in polling mode it indicates the array index of the element just written to.

locp_arr (IN/OUT)
An array of LOB locators.

byte_amt_arr (IN/OUT)
An array of pointers to `oraub8` variables. The array size must be the same as the locator array size. The entries correspond to the amount in bytes for the locators.

IN - The number of bytes to write to the database. Always used for `BLOB`. For `CLOB` and `NCLOB` it is used only when `char_amt_arr` is zero.

OUT - The number of bytes written to the database.

char_amt (IN/OUT)
An array of pointers to `oraub8` variables. The array size must be the same as the locator array size. The entries correspond to the amount in characters for the locators.

IN - The maximum number of characters to write to the database. Ignored for `BLOB`.

OUT - The number of characters written to the database. Undefined for `BLOB`.

offset_arr (IN)
An array of pointers to `oraub8` variables. The array size must be the same as the locator array size. Each entry in the array 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 `BLOB`s, it is the number of bytes. The first position is 1.

bufp_arr (IN)/OUT
An array of pointers to buffers into which the pieces for the locators will be written. The array size must be the same as the locator array size.

bufl_arr (IN)
An array of `oraub8` variables indicating the buffer lengths for the buffer array. The array size must be the same as the locator array size.

Note: This parameter assumes an 8-bit byte. If your operating system uses a longer byte, you must adjust the value of `bufl_arr` 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 If this is `NULL`, then `OCI_NEED_DATA` will be returned for each piece. The callback function must return `OCI_CONTINUE` for the write to continue. If any other error code is returned, the LOB write is terminated.

The callback takes the following parameters:

ctxp (IN)
The context for the callback function. Can be `NULL`.

array_iter (IN)
The index of the element written to.

bufp (IN/OUT)
A buffer pointer for the piece. This is the same as the `bufp` passed as an input to the `OCILobArrayWrite()` 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`.

changed_bufpp (OUT)
The callback function can put the address of a new buffer if it prefers to use a new buffer for next piece to read. The default old buffer `bufp` is used if this parameter is set to `NULL`.

changed_lenp (OUT)
Length of the new buffer, if provided.

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 nonzero values:

- SQLCS_IMPLICIT - database character set ID
- SQLCS_NCHAR - NCHAR character set ID

  The default value is SQLCS_IMPLICIT.

Comments

If LOB data already exists it is overwritten with the data stored in the buffer. The buffers can be written to the LOBs 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.

The parameters `piece`, `csid`, and `csform` are the same for all locators of the array.

When using the polling mode for `OCILobArrayWrite()`, the first call needs to specify values for `offset_arr`, `byte_amt_arr`, and `char_amt_arr`, but on subsequent polling calls to `OCILobArrayWrite()`, 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_arr`. If no callback function is defined, then `OCILobArrayWrite()` returns the `OCI_NEED_DATA` error code. The application must call `OCILobArrayWrite()` again to write more pieces of the LOBs. 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 the database (through either input mechanism) is less than the amount specified by the byte_amt_arr or the char_amt_arr parameter, an ORA-22993 error is returned.

This function is valid for internal LOBs only. BFILEs are not valid, since they are read-only. If the LOB is a BLOB, the csid and csfrm parameters are ignored.

If both byte_amt_arr and char_amt_arr are set to point to zero amount and OCI_FIRST_PIECE is given as input, then polling mode is assumed and data is written until you specify OCI_LAST_PIECE. For CLOBs and NCLOBs, byte_amt_arr and char_amt_arr return the data written by each piece in terms of number of bytes and number of characters respectively. For BLOBs byte_amt_arr returns the number of bytes written by each piece while char_amt_arr is undefined on output.

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 or 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 or 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 or close statements.

See Also:

- For additional information on Unicode format, see "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-28
- 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 in OCI" on page 5-29

Related Functions

OCIErrorGet(), OCILobRead2(), OCILobAppend(), OCILobCopy(), OCILobWriteAppend2(), OCILobArrayRead()

OCILobAssign()

Purpose

Assigns one LOB or BFILE 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 or BFILE locator to copy from.

dst_locpp (IN/OUT)
LOB or BFILE 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 BFILEs, 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 BFILE 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 for each 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 character set form of the LOB locator, 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 `BFILE`s. The caller must allocate space for the `csfrm` (a `ub1`).

`csfrm` has two possible nonzero values:

- SQLCS_IMPLICIT - database character set ID, the default
- SQLCS_NCHAR - NCHAR character set ID

Comments

Returns the character set form of the input `CLOB` or `NCLOB` locator in the `csfrm` output parameter.

Related Functions

OCIErrorGet(), OCILobCharSetId(), OCILobLocatorIsInit()

OCILobCharSetId()

Purpose

Gets the LOB locator's database character set ID of the LOB locator, 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 `CLOB` or `NCLOB` locator in the `csid` output parameter.

Related Functions

OCIErrorGet(), OCILobCharSetForm(), OCILobLocatorIsInit()

OCILobClose()

Purpose

Closes a previously opened LOB or `BFILE`.

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 (`BFILEs`), close actually closes the server-side operating system file.

It is not mandatory that you wrap all LOB operations inside the open or 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 function-based 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 or 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 7-10

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` or `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 (that is, 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 (that is, they must both be `BLOBs` or both be `CLOBs`). LOB buffering must not be enabled for either locator.

This function does not accept a `BFILE` locator as the source or the destination.

It is not mandatory that you wrap this LOB operation inside the open or 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 or 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 or close statements.

Note: You can call `OCILobGetLength()` to determine the length of the source LOB.

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy2(), OCILobWrite(), OCILobWriteAppend()

OCILobCopy2()

Purpose

Copies all or a portion of a LOB value into another LOB value. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobCopy2 ( OCISvcCtx        *svchp,
                    OCIError         *errhp,
                    OCILobLocator    *dst_locp,
                    OCILobLocator    *src_locp,
                    oraub8           amount,
                    oraub8           dst_offset,
                    oraub8           src_offset );
```

Parameters

See Also: "OCILobCopy()" on page 16-39

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. `csfrm` has two possible nonzero values:

- `SQLCS_IMPLICIT` - database character set ID, to create a `CLOB`. `OCI_DEFAULT` can also be used to implicitly create a `CLOB`.
- `SQLCS_NCHAR` - `NCHAR` character set ID, to create an `NCLOB`.

The default value is `SQLCS_IMPLICIT`.

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` or `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 7-14.

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 `BFILE` 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 `BFILE` 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 `CLOB`s or`NCLOB`s, or bytes for `BLOB`s, 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 `CLOB`s or `NCLOB`s, or bytes for `BLOB`s, from the beginning of the LOB value from which to start erasing data. Starts at 1.

Comments

The actual number of characters or bytes erased is returned. For `BLOB`s, erasing means that zero-byte fillers overwrite the existing LOB value. For `CLOB`s, erasing means that spaces overwrite the existing LOB value.

This function is valid only for internal LOBs; `BFILE`s are not allowed.

It is not mandatory that you wrap this LOB operation inside the open or 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 or 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 or close statements.

Related Functions

OCIErrorGet(),OCILobErase2(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobWrite(), OCILobWriteAppend()

OCILobErase2()

Purpose

Erases a specified portion of the internal LOB data starting at a specified offset. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobErase2 ( OCISvcCtx       *svchp,
                     OCIError        *errhp,
                     OCILobLocator   *locp,
                     oraub8          *amount,
                     oraub8          offset );
```

Parameters

See Also: "OCILobErase()" on page 16-46

OCILobFileClose()

Purpose

Closes a previously opened BFILE.

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 BFILE locator that refers to the BFILE to be closed.

Comments

Closes a previously opened BFILE. It is an error if this function is called for an internal LOB. No error is returned if the BFILE exists but is not opened.

This function is only meaningful the first time it is called for a particular BFILE locator. Subsequent calls to this function using the same BFILE locator have no effect.

> **See Also:** For more information about BFILEs, refer to the description of BFILEs in the *Oracle Database Application Developer's Guide - Large Objects*.

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobOpen(), OCILobIsOpen()

OCILobFileCloseAll()

Purpose

Closes all open BFILEs 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 BFILEs on a given service context.

> **See Also:** For more information about BFILEs, refer to the description of BBFILEs in the *Oracle Database Application Developer's Guide - Large Objects*.

Related Functions

OCILobFileClose(), OCIErrorGet(), OCILobFileExists(), OCILobFileIsOpen()

OCILobFileExists()

Purpose

Tests to see if the BFILE 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 BFILE locator that refers to the file.

flag (OUT)
Returns TRUE if the BFILE exists on the server; FALSE if it does not.

Comments

Checks to see if the BFILE 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 BFILEs, refer to the description of BFILEs in the *Oracle Database Application Developer's Guide - Large Objects*.

Related Functions

OCIErrorGet(), OCILobFileClose(), OCILobFileCloseAll(), OCILobFileIsOpen(), OCILobOpen(), OCILobIsOpen()

OCILobFileGetName()

Purpose

Gets the BFILE locator's directory object 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)
BFILE locator for which to get the directory object and file name.

dir_alias (OUT)
Buffer into which the directory object name is placed. This can be in UTF-16. You must allocate enough space for the directory object name. The maximum length for the directory object 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 object and file name associated with this BFILE 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 BFILEs, refer to the description of BFILEs in the *Oracle Database Application Developer's Guide - Large Objects*

Related Functions

OCILobFileSetName(), OCIErrorGet()

OCILobFileIsOpen()

Purpose

Tests to see if the BFILE 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 BFILE locator being examined.

flag (OUT)
Returns TRUE if the BFILE 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 BFILE locator. It is an error to call this function for an internal LOB.

If the input BFILE 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 BFILEs, refer to the description of BFILEs in the *Oracle Database Application Developer's Guide - Large Objects*

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileOpen(), OCILobOpen(), OCILobIsOpen()

OCILobFileOpen()

Purpose

Opens a BFILE 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 BFILE to open. It is an error if the locator does not refer to a BFILE.

mode (IN)
Mode in which to open the file. The only valid mode is OCI_FILE_READONLY.

Comments

Opens a BFILE on the file system of the server. The BFILE can be opened for read-only access. BFILEs 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 BFILE locator. Subsequent calls to this function using the same BFILE locator have no effect.

> **See Also:** For more information about BFILEs, refer to the description of BFILEs in the *Oracle Database Application Developer's Guide - Large Objects*

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileIsOpen(), OCILobOpen(), OCILobIsOpen()

OCILobFileSetName()

Purpose

Sets the directory object and file name in the BFILE 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 BFILE locator for which to set the directory object and file name.

dir_alias (IN)
Buffer that contains the directory object name (must be in the encoding specified by the charset parameter of a previous call to OCIEnvNlsCreate()) to set in the BFILE locator.

d_length (IN)
Length of the input dir_alias parameter. In bytes.

filename (IN)
Buffer that contains the file name (must be in the encoding specified by the charset parameter of a previous call to OCIEnvNlsCreate()) to set in the BFILE 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 BFILEs, refer to the description of BFILEs in tthe *Oracle Database Application Developer's Guide - Large Objects*

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 section.

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 or 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 and 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 or 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 or 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: "Improving LOB Read/Write Performance" on page 7-8

Related Functions

OCIErrorGet(), OCILobGetStorageLimit(), 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 `BFILE`s, 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 `BFILE`s 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 `BFILE` 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 `CLOB`s and `NCLOB`s, and in bytes for `BLOB`s and `BFILE`s.

Note: Any zero-byte or space fillers in the LOB written by previous calls to `OCILobErase()` or `OCILobWrite2()` are also included in the length count.

Related Functions

OCIErrorGet(), OCILobFileSetName(),OCILobGetLength2(), OCILobRead(), OCILobWrite(), OCILobCopy(), OCILobAppend(), OCILobLoadFromFile(), OCILobWriteAppend()

OCILobGetLength2()

Purpose

Gets the length of a LOB. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobGetLength2 ( OCISvcCtx      *svchp,
                         OCIError       *errhp,
                         OCILobLocator  *locp,
                         oraub8         *lenp );
```

Parameters

See Also: OCILobGetLength() on page 16-60

OCILobGetStorageLimit()

Purpose

Gets the maximum length of an internal LOB (BLOB, CLOB, or NCLOB) in bytes.

Syntax

```
sword OCILobGetStorageLimit ( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCILobLocator  *locp,
                              oraub8         *limitp );
```

Parameters

svchp (IN)
The service context handle.

errhp (IN/OUT)
An error handle that you can pass to `OCIError()` for diagnostic information in the event iof an error.

locp (IN)
A LOB locator that uniquely references the LOB. The locator must be one that was obtained from the server specified by `svchp`.

limitp (OUT)
The maximum length of the LOB (in bytes) that can be stored in the database.

Comments

Because block size ranges from 2KB to 32KB, the maximum LOB size ranges from 8 terabytes to 128 terabytes.

See Also: "Using LOBs of Size Greater than 4 GB" on page 7-4

Related Functions

OCILobGetChunkSize()

OCILobIsEqual()

Purpose

Compares two LOB or BFILE 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 or BFILE locators for equality. Two LOB or BFILE locators are equal if and only if they both refer to the same LOB or BFILE value.

Two NULL locators are considered *not* equal by this function.

Related Functions

OCILobAssign(), OCILobLocatorIsInit()

OCILobIsOpen()

Purpose

Tests whether a LOB or BFILE 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 7-10

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 and 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 `BFILE`.

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 `BFILE`. It is the number of bytes from the beginning of the `BFILE`. The offset starts at 1.

Comments

Loads and copies a portion or all of a `BFILE` value into an internal LOB as specified. The data is copied from the source `BFILE` to the destination internal LOB (`BLOB` or `CLOB`). No character set conversions are performed when copying the `BFILE` data to a `CLOB` or `NCLOB`. Also, when binary data is loaded into a `BLOB`, no character set conversions are performed. Therefore, the `BFILE` 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 `BFILE`.

It is not mandatory that you wrap this LOB operation inside the open or 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 or 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 or close statements.

Related Functions

OCIErrorGet(), OCILobAppend(), OCILobWrite(), OCILobTrim(), OCILobCopy(), OCILobGetLength(), OCILobLoadFromFile2(), OCILobWriteAppend()

OCILobLoadFromFile2()

Purpose

Load and copy all or a portion of the file into an internal LOB. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobLoadFromFile2 ( OCISvcCtx        *svchp,
                            OCIError         *errhp,
                            OCILobLocator    *dst_locp,
                            OCILobLocator    *src_locp,
                            oraub8           amount,
                            oraub8           dst_offset,
                            oraub8           src_offset );
```

Parameters

See Also: OCILobLoadFromFile() on page 16-66

OCILobLocatorAssign()

Purpose

Assigns one LOB or BFILE 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/BFILE locator to copy from.

dst_locpp (IN/OUT)
The LOB/BFILE 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 BFILEs 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 destination will be made into a temporary LOB too. The source and the destination will conceptually be different temporary LOBs. In the OCI_DEFAULT mode, the source temporary LOB is deep copied and a destination locator is created to refer to the new deep copy of the temporary LOB. Hence OCILobIsEqual() will return FALSE after the

`OCILobLocatorAssign()` call. However, in the `OCI_OBJECT` mode, an optimization is made to minimize the number of deep copies, so the source and destination locators will point to the same LOB until any modification is made through either LOB locator. Hence `OCILobIsEqual()` will return TRUE right after `OCILobLocatorAssign()` until the first modification. In both these cases, after the `OCILobLocatorAssign()`, any changes to the source or the destination will not reflect in the other (that is, destination or source) LOB. If you want the source and the destination to point to the same LOB and want your changes to reflect in the other, then you must 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 or BFILE 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 or BFILE locator being tested

is_initialized (OUT)
Returns TRUE if the given LOB or BFILE locator is initialized; FALSE if it is not.

Comments

Tests to see if a given LOB or BFILE 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-33

BFILE locators can be initialized by one of the following methods:

- selecting a non-NULL BFILE into the locator
- pinning an object that contains a non-NULL BFILE 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 or `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 or `BFILE` was opened in read-only mode and the user tries to write to the LOB or `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 function-based 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 for each write operation.

It is not mandatory that you wrap all LOB operations inside the Open and 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 function-based 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 or 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 or close statements.

> **See Also:** "Functions for Opening and Closing LOBs" on page 7-10

Related Functions

OCIErrorGet(), OCILobClose(), OCILobFileCloseAll(), OCILobFileExists(), OCILobFileClose(), OCILobFileIsOpen(), OCILobFileOpen(), OCILobIsOpen()

OCILobRead()

Purpose

Reads a portion of a LOB or BFILE, 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 or BFILE locator that uniquely references the LOB or BFILE. 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:

Table 16–4 Characters or Bytes in amtp

LOB or BFILE	Input	Output with fixed-width client-side character set	Output with varying-width client-side character set
BLOBs and BFILEs	bytes	bytes	bytes
CLOBs and NCLOBs	characters	characters	bytes (1)

(1) The input amount refers to the number of characters to be read from the server-side CLOB or 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 (CLOBs, NCLOBs) it is the number of characters from the beginning of the LOB, for binary LOBs or BFILEs 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 terminated.

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 nonzero 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 or BFILE as specified by the call into a buffer. It is an error to try to read from a NULL LOB or BFILE.

> **Note:** When reading or writing LOBs, the character set form (csfrm) specified should match the form of the locator itself.

For BFILEs, 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 terminate 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 in OCI" on page 5-28
- For more information about BFILEs, refer to the description of BFILEs in the *Oracle Database Application Developer's Guide - Large Objects*
- 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 in OCI" on page 5-29

Related Functions

OCIErrorGet(),OCILobRead2(), OCILobWrite(), OCILobFileSetName(), OCILobWriteAppend()

OCILobRead2()

Purpose

Reads a portion of a LOB or BFILE, as specified by the call, into a buffer. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobRead2 ( OCISvcCtx          *svchp,
                    OCIError           *errhp,
                    OCILobLocator      *locp,
                    oraub8             *byte_amtp,
                    oraub8             *char_amtp,
                    oraub8             offset,
                    dvoid              *bufp,
                    oraub8             bufl,
                    ub1                piece,
                    dvoid              *ctxp,
                    OCICallbackLobRead2 (cbfp)
                                        (
                                         dvoid         *ctxp,
                                         CONST dvoid   *bufp,
                                         oraub8        lenp,
                                         ub1           piecep
                                         dvoid         **changed_bufpp,
                                         oraub8        *changed_lenp
                                        )
                    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 or BFILE locator that uniquely references the LOB or BFILE. This locator must be a locator that was obtained from the server specified by svchp.

byte_amtp (IN/OUT)
IN - The number of bytes to read from the database. Used for BLOB and BFILE always. For CLOB and NCLOB, it is used only when char_amtp is zero.

OUT - The number of bytes read into the user buffer.

char_amtp (IN/OUT)
IN - The maximum number of characters to read into the user buffer. Ignored for BLOB and BFILE.

OUT - The number of characters read into the user buffer. Undefined for BLOB and BFILE.

offset (IN)
On input, this is the absolute offset from the beginning of the LOB value. For character LOBs (CLOBs, NCLOBs) it is the number of characters from the beginning of the LOB, for binary LOBs or BFILEs 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.

piece (IN)
OCI_ONE_PIECE - The call never assumes polling. If the amount indicated is more than the buffer length then the buffer is filled as much as possible.

For polling, pass OCI_FIRST_PIECE the first time and OCI_NEXT_PIECE in subsequent calls. OCI_FIRST_PIECE should be passed while using the callback.

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 terminated.

ctxp (IN)
The context for the callback function. Can be NULL.

bufp (IN/OUT)
A buffer pointer for the piece.

lenp (IN)
The length in bytes of the current piece in bufp.

piecep (IN)
Which piece: OCI_FIRST_PIECE, OCI_NEXT_PIECE, or OCI_LAST_PIECE.

changed_bufpp (OUT)
The callback function can put the address of a new buffer if it prefers to use a new buffer for the next piece to read. The default old buffer bufp is used if this parameter is set to NULL.

changed_lenp (OUT)
Length of the new buffer, if provided.

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 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 nonzero 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 or BFILE as specified by the call into a buffer. It is an error to try to read from a NULL LOB or BFILE.

> **Note:** When reading or writing LOBs, the character set form (csfrm) specified should match the form of the locator itself.

For BFILEs, 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 OCILobRead2(), the first call needs to specify values for offset and amtp, but on subsequent polling calls to OCILobRead2(), the user need not specify these values.

If the LOB is a BLOB, the csid and csfrm parameters are ignored.

> **Note:** To terminate an OCILobRead2() operation and free the statement handle, use the OCIBreak() call.

The following points applly to reading LOB data in streaming mode:

- When using polling mode, be sure to specify the char_amtp and byte_amtp and offset parameters only in the first call to OCILobRead2(). On subsequent polling calls these parameters are ignored. If both byte_amtp and char_amtp are set to point to zero and OCI_FIRST_PIECE is passed then polling mode is assumed and data is read till the end of the LOB. On output, byte_amtp gives the number of bytes read in the current piece. For CLOBs and NCLOBs, char_amtp gives the number of characters read in the current piece.
- 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 because the entire buffer may not be filled with data.
- When using polling, look at the byte_amtp parameter to see how much the buffer is filled for the current piece. For CLOBs and NCLOBs, char_amtp returns the number of characters read in the buffer as well.

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 in OCI" on page 5-28
- For more information about `BFILEs`, refer to the description of `BFILEs` in the *Oracle Database Application Developer's Guide - Large Objects*
- 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 in OCI" on page 5-29

Related Functions

OCIErrorGet(), OCILobWrite2(), OCILobFileSetName(), OCILobWriteAppend2()

OCILobTrim()

Purpose

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 `BFILE`s 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. `BFILE`s are not allowed.

It is not mandatory that you wrap this LOB operation inside the open or 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 or 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 or close statements.

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobErase(),OCILobTrim2(), OCILobWrite(), OCILobWriteAppend()

OCILobTrim2()

Purpose

Truncates the LOB value to a shorter length. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobTrim2 ( OCISvcCtx       *svchp,
                    OCIError        *errhp,
                    OCILobLocator   *locp,
                    oraub8          newlen );
```

Parameters

See Also: OCILobTrim() on page 16-82

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:

Table 16–5 Characters or Bytes in amtp

LOB or BFILE	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` or `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 operating system 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 terminated. 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 nonzero 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. BFILEs 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 or 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 or 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 or 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 or close statements.

See Also:

- For additional information on Unicode format, see "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-28
- 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 in OCI" on page 5-29

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobWriteAppend(), OCILobWrite2()

OCILobWrite2()

Purpose

Writes a buffer into a LOB. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobWrite2 ( OCISvcCtx       *svchp,
                     OCIError        *errhp,
                     OCILobLocator   *locp,
                     oraub8          *byte_amtp,
                     oraub8          *char_amtp,
                     oraub8          offset,
                     dvoid           *bufp,
                     oraub8          buflen,
                     ub1             piece,
                     dvoid           *ctxp,
                     OCICallbackLobWrite2 (cbfp)
                                     (
                                       dvoid    *ctxp,
                                       dvoid    *bufp,
                                       oraub8   *lenp,
                                       ub1      *piecep
                                       dvoid    **changed_bufpp,
                                       oraub8   *changed_lenp
                                     )
                     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`.

byte_amtp (IN/OUT)
IN - The number of bytes to write to the database. Always used for `BLOB`. For `CLOB` and `NCLOB` it is used only when `char_amtp` is zero.

OUT - The number of bytes written to the database. In polling mode, it is the length of the piece, in bytes, just written.

char_amtp (IN/OUT)
IN - The maximum number of characters to write to the database. Ignored for `BLOB`.

OUT - The number of characters written to the database. Undefined for `BLOB`. In polling mode, it is the length of the piece, in characters, just written.

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 `char_amtp` value for `CLOB`s and `NCLOB`s if the amount is specified in terms of characters using the `char_amtp` parameter, while the `buflen` parameter is specified in terms of bytes.

Note: This parameter assumes an 8-bit byte. If your operating system 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 terminated. 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`.

changed_bufpp (OUT)
The callback function can put the address of a new buffer if it prefers to use a new buffer for next piece to read. The default old buffer `bufp` is used if this parameter is set to `NULL`.

changed_lenp (OUT)
Length of the new buffer, if provided.

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 nonzero 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 `OCILobWrite2()`, the first call needs to specify values for `offset`, `byte_amtp`, and `char_amtp`, but on subsequent polling calls to `OCILobWrite2()`, 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 `OCILobWrite2()` returns the `OCI_NEED_DATA` error code. The application must call `OCILobWrite2()` 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 the database (through either input mechanism) is less than the amount specified by the `byte_amtp` or the `char_amtp` parameter, an ORA-22993 error is returned.

This function is valid for internal LOBs only. `BFILE`s are not allowed, since they are read-only. If the LOB is a `BLOB`, the `csid` and `csfrm` parameters are ignored.

If both `byte_amtp` and `char_amtp` are set to point to zero amount and `OCI_FIRST_PIECE` is given as input, then polling mode is assumed and data is

written until you specify OCI_LAST_PIECE. For CLOBs and NCLOBs, byte_amtp and char_amtp return the data written by each piece in terms of number of bytes and number of characters respectively. For BLOBs byte_amtp returns the number of bytes written by each piece while char_amtp is undefined on output.

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 or 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 or 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 or close statements.

See Also:

- For additional information on Unicode format, see "PL/SQL REF CURSORs and Nested Tables in OCI" on page 5-28
- 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 in OCI" on page 5-29

Related Functions

OCIErrorGet(), OCILobRead2(), OCILobAppend(), OCILobCopy(), OCILobWriteAppend2()

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:

Table 16–6 Characters or Bytes in amtp

LOB or BFILE	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` or `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 or 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 operating system 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 terminated. 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 nonzero 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 `CLOB`s or `NCLOB`s.

It is not mandatory that you wrap this LOB operation inside the open or 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 or 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 or close statements.

See Also: "Improving LOB Read/Write Performance" on page 7-8

Related Functions

OCIErrorGet(), OCILobRead(), OCILobAppend(), OCILobCopy(), OCILobWrite(), OCILobWriteAppend2()

OCILobWriteAppend2()

Purpose

Writes data starting at the end of a LOB. This function must be used for LOBs of size greater than 4 GB. You can also use this function for LOBs smaller than 4 GB.

Syntax

```
sword OCILobWriteAppend2 ( OCISvcCtx           *svchp,
                           OCIError            *errhp,
                           OCILobLocator       *locp,
                           oraub8              *byte_amtp,
                           oraub8              *char_amtp,
                           dvoid               *bufp,
                           oraub8              buflen,
                           ub1                 piece,
                           dvoid               *ctxp,
                           OCICallbackLobWrite2 (cbfp)
                                        (
                                          dvoid    *ctxp,
                                          dvoid    *bufp,
                                          oraub8   *lenp,
                                          ub1      *piecep
                                          dvoid    **changed_bufpp,
                                          oraub8   *changed_lenp
                                         )
                           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.

byte_amtp (IN/OUT)
IN - The number of bytes to write to the database. Used for `BLOB`. For `CLOB` and `NCLOB` it is used only when `char_amtp` is zero.

OUT - The number of bytes written to the database.

char_amtp (IN/OUT)
IN - The maximum number of characters to write to the database. Ignored for `BLOB`.

OUT - The number of characters written to the database. Undefined for `BLOB`.

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 operating system 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 terminated. 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`.

changed_bufpp (OUT)
The callback function can put the address of a new buffer if it prefers to use a new buffer for next piece to be written. The default old buffer `bufp` is used if this parameter is set to `NULL`.

changed_lenp (OUT)
Length of the new buffer, if provided.

csid (IN)
The character set ID of the buffer data.

csfrm (IN)
The character set form of the buffer data.

`csfrm` has two possible nonzero 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 OCILobWriteAppend2() returns the OCI_NEED_DATA error code.

The application must call OCILobWriteAppend2() 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.

OCILobWriteAppend2() is not supported if LOB buffering is enabled.

If the LOB is a BLOB, the csid and csfrm parameters are ignored.

If both byte_amtp and char_amtp are set to point to zero amount and OCI_FIRST_PIECE is given as input, then polling mode is assumed and data is written until you specify OCI_LAST_PIECE. For CLOBs and NCLOBs, byte_amtp and char_amtp return the data written by each piece in terms of number of bytes and number of characters respectively. For BLOBs byte_amtp returns the number of bytes written by each piece while char_amtp is undefined on output.

It is not mandatory that you wrap this LOB operation inside the open or 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 or 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 or close statements.

See Also: "Improving LOB Read/Write Performance" on page 7-8

Related Functions

OCIErrorGet(), OCILobRead2(), OCILobAppend(), OCILobCopy(), OCILobWrite2()

Streams Advanced Queuing and Publish-Subscribe Functions

This section describes the Streams Advanced Queuing and publish-subscribe functions.

> **See Also:** " OCI Demonstration Programs" on page B-1 for Streams Advanced Queueing programs

Table 16–7 Advanced Queuing and Publish-Subscribe Functions

Function	Purpose
OCIAQDeq() on page 16-99	Advanced Queuing dequeue
OCIAQDeqArray() on page 16-101	Dequeue an array of messages
OCIAQEnq() on page 16-103	Advanced Queuing enqueue
OCIAQEnqArray() on page 16-105	Enqueue an array of messages
OCIAQListen() on page 16-107	Listen on one or more queues on behalf of a list of agents
OCIAQListen2() on page 16-108	Listen on one or more queues on behalf of a list of agents. Supports buffered messaging and persistent queues.
OCISubscriptionEnable() on page 16-111	Enable notifications on a subscription
OCISubscriptionPost() on page 16-112	Post to a subscription to receive notifications
OCISubscriptionRegister() on page 16-114	Register a subscription
OCISubscriptionUnRegister() on page 16-116	Unregister a subscription

OCIAQDeq()

Purpose

This call is used for a Streams 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, with OCI type constant `OCI_DTYPE_AQDEQ_OPTIONS`.

`OCI_DTYPE_AQDEQ_OPTIONS` has the additional attribute `OCI_ATTR_MSG_DELIVERY_MODE` (introduced in Oracle Database 10*g* Release 2) with values:

- `OCI_MSG_PERSISTENT` (default)
- `OCI_MSG_BUFFERED`
- `OCI_MSG_PERSISTENT_OR_BUFFERED`

message_properties (OUT)
The message properties for the message; stored in an `OCIAQMsgProperties` descriptor, with OCI type constant `OCI_DTYPE_AQMSG_PROPERTIES` which can have values:

- `OCI_AQ_PERSISTENT` (default)
- `OCI_AQ_BUFFERED`

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 10-9.

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`,.

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 Streams Advanced Queuing" on page 9-39
- For additional information about Advanced Queuing, refer to *Oracle Streams Advanced Queuing User's Guide and Reference*

Examples

For code examples, refer to the description of OCIAQEnq() on page 16-103.

Related Functions

OCIAQEnq(), OCIAQListen2(), OCIInitialize()

OCIAQDeqArray()

Purpose

This call dequeues an array of messages from a queue. The array of messages is all dequeued with the same option and has the same queue table payload column TDO.

Syntax

```
sword OCIAQDeqArray ( OCISvcCtx           *svchp,
                      OCIError            *errhp,
                      OraText             *queue_name,
                      OCIAQDeqOptions     *deqopt,
                      ub4                 *iters,
                      OCIAQMsgProperties  **msgprop,
                      OCIType             *payload_tdo,
                      dvoid               **payload,
                      dvoid               **payload_ind,
                      OCIRaw              **msgid,
                      dvoid               *ctxp,
                      OCICallbackAQDeq    (cbfp)
                                          (
                                          dvoid        *ctxp,
                                          dvoid        **payload,
                                          dvoid        **payload_ind
                                          ),
                      ub4                 flags );
```

Parameters

svchp (IN)
OCI service context (unchanged from `OCIAQDeq()`).

errhp (IN)
An error handle you can pass to `OCIErrorGet()` for diagnostic information in the event of an error (unchanged from `OCIAQDeq()`).

queue_name (IN)
The name of the queue from which messages are dequeued (unchanged from `OCIAQDeq()`).

deqopt (IN)
A pointer to an `OCIAQDeqOptions` descriptor (unchanged from `OCIAQDeq()`).

`OCI_DTYPE_AQDEQ_OPTIONS` OCI type constant has the additional attribute: `OCI_ATTR_MSG_DELIVERY_MODE` (introduced in Oracle Database 10*g* Release 2) with values:

- `OCI_MSG_PERSISTENT` (default)
- `OCI_MSG_BUFFERED`
- OCI_MSG_PERSISTENT_OR_BUFFERED

iters (IN/OUT)
On input, the number of messages to dequeue. On output, the number of messages successfully dequeued.

msgprop (OUT)
An array of pointers to `OCIAQMsgProperties` descriptors, of OCI type constant `OCI_DTYPE_AQMSG_PROPERTIES`, which can have values:

- `OCI_AQ_PERSISTENT` (default)
- `OCI_AQ_BUFFERED`

payload_tdo (OUT)
A pointer to the TDO of the queue table's payload column.

payload (OUT)
An array of pointers to dequeued messages.

payload_ind (OUT)
An array of pointers to indicators.

msgid (OUT)
An array of pointers to the message ID of the dequeued messages.

ctxp (IN)
The context that will be passed to the callback function.

cbfp (IN)
The callback that may be registered to provide a buffer pointer into which the dequeued message will be placed. If `NULL`, then messages will be dequeued into buffers pointed to by `payload`.

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.

A nonzero wait time, as specified in the `OCIAQDeqOptions`, is recognized only when there are no messages in the queue. If the queue contains messages that are eligible for dequeue, then the `OCIAQDeqArray()` function will dequeue up to `iters` messages and return immediately.

This function is not supported in nonblocking mode.

See Also:

- For more information about OCI and Advanced Queuing, refer to "OCI and Streams Advanced Queuing" on page 9-39
- For additional information about Advanced Queuing, refer to *Oracle Streams Advanced Queuing User's Guide and Reference*

Related Functions

OCIAQDeq(), OCIAQEnqArray(), OCIAQListen2(), OCIInitialize()

OCIAQEnq()

Purpose

This call is used for a Streams 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, of OCI type constant `OCI_DTYPE_AQMSG_PROPERTIES`, introduced in Oracle Database 10*g* Release 2.

Descriptor `OCI_DTYPE_AQMSG_PROPERTIES` has the attribute, `OCI_ATTR_MSG_DELIVERY_MODE`, which has values:

- `OCI_MSG_PERSISTENT` (default)
- `OCI_MSG_BUFFERED`

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 10-9

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 Streams Advanced Queuing" on page 9-39
- For more information about Advanced Queuing, refer to *Oracle Streams Advanced Queuing User's Guide and Reference*

To obtain a TDO for the payload, use OCITypeByName(), or OCITypeByRef().

Related Functions

OCIAQDeq(), OCIAQListen2(), OCIInitialize()

OCIAQEnqArray()

Purpose

This call enqueues an array of messages to a queue. The array of messages is enqueued with the same options and has the same payload column TDO.

Syntax

```
sword OCIAQEnqArray ( OCISvcCtx             *svchp,
                      OCIError              *errhp,
                      OraText               *queue_name,
                      OCIAQEnqOptions       *enqopt,
                      ub4                   *iters,
                      OCIAQMsgProperties    **msgprop,
                      OCIType               *payload_tdo,
                      dvoid                 **payload,
                      dvoid                 **payload_ind,
                      OCIRaw                **msgid,
                      dvoid                 *ctxp,
                      OCICallbackAQEnq      (cbfp)
                                            (
                                            dvoid       *ctxp,
                                            dvoid       **payload,
                                            dvoid       **payload_ind
                                            ),
                      ub4                   flags );
```

Parameters

svchp (IN)
The service context (unchanged from `OCIAQEnq()`).

errhp (IN/OUT)
The error handle (unchanged from `OCIAQEnq()`).

queue_name (IN)
The name of the queue in which messages are enqueued (unchanged from `OCIAQEnq()`).

enqopt (IN)
A pointer to an `OCIAQEnqOptions` descriptor (unchanged from `OCIAQEnq()`).

iters (IN/OUT)
On input, the number of messages to enqueue. On output, the number of messages successfully enqueued.

msgprop (IN)
An array of pointers to `OCIAQMsgProperties` descriptors, of OCI type constant `OCI_DTYPE_AQMSG_PROPERTIES`, introduced in Oracle Database 10*g* Release 2.

`OCI_DTYPE_AQMSG_PROPERTIES` has the attribute, `OCI_ATTR_MSG_DELIVERY_MODE`, which has values:

- `OCI_MSG_PERSISTENT` (default)
- `OCI_MSG_BUFFERED`

payload_tdo (IN)
A pointer to the TDO of the queue table's payload column.

payload (IN)
An array of pointers to messages to be enqueued.

payload_ind (IN)
An array of pointers to indicators, or a `NULL` pointer if indicator variables are not used.

msgid (OUT)
An array of pointers to the message ID of the enqueued messages or a `NULL` pointer if no message IDs are returned.

ctxp (IN)
The context that will be passed to the registered callback function.

cbfp (IN)
A callback that may be registered to provide messages dynamically. If `NULL`, then all messages must be materialized prior to calling `OCIAQEnqArray()`.

flags (IN)
Not currently used; pass as `OCI_DEFAULT`.

Comments

This function is not supported in nonblocking mode.

Related Functions

OCIAQEnq(), OCIAQDeqArray(), OCIAQListen2(), 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)
Timeout interval 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(), OCIAQListen2(), OCISvcCtxToLda(), OCISubscriptionEnable(), OCISubscriptionPost(), OCISubscriptionRegister(),OCISubscriptionUnRegister()

OCIAQListen2()

Purpose

Listens on one or more queues on behalf of a list of agents. Supports buffered messaging as well as persistent queues. Introduced in Oracle Database 10*g* Release 2.

Syntax

```
sword OCIAQListen2 (OCISvcCtx         *svchp,
                    OCIError          *errhp,
                    OCIAQAgent        **agent_list,
                    ub4               num_agents,
                    OCIAQListenOpts   *lopts,
                    OCIAQAgent        **agent,
                    OCIAQLisMsgProps  *lmops,
                    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.

lopts (IN)
Type constant `OCI_DTYPE_AQLIS_OPTIONS` has the following attributes:

- `OCI_AQ_WAIT` - maximum wait time, in seconds, for the listen call
- `OCI_MSG_DELIVERY_MODE` - has one of the following values:
  - `OCI_MSG_PERSISTENT`
  - `OCI_MSG_BUFFERED`
  - `OCI_MSG_PERSISTENT_OR_BUFFERED`

agent (OUT)
Agent for which there is a message. `OCIAgent` is an OCI descriptor.

lmops (OUT)
`OCI_DTYPE_AQLIS_MSG_PROPERTIES` (listen message properties) has one attribute, `OCI_MSG_DELIVERY_MODE`, which has the following values:

- `OCI_MSG_PERSISTENT`
- `OCI_MSG_BUFFERED`

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: Subscription Handle Attributes on page A-44

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

OCIAQListen2(), 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-44

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

OCIAQListen2(), 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-44

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 goes to registered clients at most once.

This call is primarily used for nonpersistent 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

OCIAQListen2(), 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-44

`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 `SCHEMA.QUEUE:CONSUMER_NAME` if the registration is for a multi-consumer queue. The string should be in upper case.

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

OCIAQListen2(), 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-44

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

OCIAQListen2(), OCISvcCtxToLda(), OCISubscriptionEnable(), OCISubscriptionPost(), OCISubscriptionRegister().

Direct Path Loading Functions

This section describes the direct path loading functions.

Table 16–8 Direct Path Loading Functions

Function	Purpose
OCIDirPathAbort() on page 16-118	Terminates a direct path operation
OCIDirPathColArrayEntryGet() on page 16-119	Gets a specified entry in a column array
OCIDirPathColArrayEntrySet() on page 16-120	Sets a specified entry in a column array to a specific value
OCIDirPathColArrayRowGet() on page 16-122	Gets the base row pointers for a specified row number
OCIDirPathColArrayReset() on page 16-123	Resets the row array state
OCIDirPathColArrayToStream() on page 16-124	Converts from a column array to a direct path stream format
OCIDirPathDataSave() on page 16-126	Does a data savepoint, or commits the loaded data and finishes the load operation
OCIDirPathFinish() on page 16-127	Finishes and commits the loaded data
OCIDirPathFlushRow() on page 16-128	Deprecated.
OCIDirPathLoadStream() on page 16-129	Loads the data converted to direct path stream format
OCIDirPathPrepare() on page 16-130	Prepares direct path interface to convert or load rows
OCIDirPathStreamReset() on page 16-131	Resets the direct path stream state

OCIDirPathAbort()

Purpose

Terminates 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 a termination. For a direct path load, the data loaded prior to the terminate 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(), 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. Do not reset the column array if the last `OCIDirPathColArrayReset()` call returned `OCI_NEED_DATA` or `OCI_CONTINUE`. That is, you are in the middle of a row conversion. Use OCI_DIRPATH_COL_ERROR to purge the current row for `OCI_NEED_DATA`.

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 successfully converted in the last call. The attribute OCI_ATTR_COL_COUNT contains the column index into the column array for the column which caused the error. A stream must always be loaded after column array to stream conversion returns OCI_ERROR. It cannot be reset or converted to until it is loaded.
- 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 successfully converted in the last call.
- 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 successfully converted in the last call. 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(), 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. Finish is not allowed until all streams have been loaded, and there is not a partially loaded row.

A return value of `OCI_SUCCESS` indicates that the back-end has properly terminated the load.

Related Functions

OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathPrepare(), OCIDirPathStreamReset()

OCIDirPathFlushRow()

Purpose

Flushes a partially loaded row from server. This function is deprecated.

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, function-based index evaluation error, or an out of space condition, such as cannot allocate extent. `OCI_ATTR_ROW_COUNT` is the number of rows successfully loaded in the last call.
- `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
- successfully loaded in the last call.
- `OCI_NO_DATA` - Attempt to load an empty stream, or a stream which has been completely processed.

A stream must be repeatedly loaded until `OCI_SUCCESS`, `OCI_NEED_DATA`, or `OCI_NO_DATA` is returned. For example, a stream cannot be reset if `OCI_ERROR` is returned from `OCIDirPathLoadStream()`.

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 nonzero 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(), 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. A stream cannot be reset until it is successfully loaded (the loading returned `OCI_SUCCESS`, `OCI_NEED_DATA`, or `OCI_NO_DATA`).

Related Functions

OCIDirPathAbort(), OCIDirPathDataSave(), OCIDirPathFinish(), OCIDirPathPrepare()

Thread Management Functions

This section describes the thread management functions.

Table 16–9 Thread Management functions

Function	Purpose
OCIThreadClose() on page 16-133	Closes a thread handle
OCIThreadCreate() on page 16-134	Creates a new thread
OCIThreadHandleGet() on page 16-135	Retrieves the OCIThreadHandle of the thread in which it is called
OCIThreadHndDestroy() on page 16-136	Destroys and deallocates the thread handle
OCIThreadHndInit() on page 16-137	Allocates and initializes the thread handle
OCIThreadIdDestroy() on page 16-138	Destroys and deallocates a thread id
OCIThreadIdGet() on page 16-139	Retrieves the `OCIThreadId` of the thread in which it is called
OCIThreadIdInit() on page 16-140	Allocates and initializes the thread id
OCIThreadIdNull() on page 16-141	Determines whether or not a given `OCIThreadId` is the NULL thread ID
OCIThreadIdSame() on page 16-142	Determines whether or not two `OCIThreadId`s represent the same thread
OCIThreadIdSet() on page 16-143	Sets one `OCIThreadId` to another
OCIThreadIdSetNull() on page 16-144	Sets the NULL thread ID to a given `OCIThreadId`
OCIThreadInit() on page 16-145	Initializes OCIThread context
OCIThreadIsMulti() on page 16-146	Tells the caller whether the application is running in a multithreaded environment or a single-threaded environment
OCIThreadJoin() on page 16-147	Allows the calling thread to join with another thread
OCIThreadKeyDestroy() on page 16-148	Destroy and deallocate the key pointed to by `key`
OCIThreadKeyGet() on page 16-149	Gets the calling threads current value for a key
OCIThreadKeyInit() on page 16-150	Creates a key
OCIThreadKeySet() on page 16-151	Sets the calling threads value for a key
OCIThreadMutexAcquire() on page 16-152	Acquires a mutex for the thread in which it is called
OCIThreadMutexDestroy() on page 16-153	Destroys and deallocate a mutex
OCIThreadMutexInit() on page 16-154	Allocates and initializes a mutex
OCIThreadMutexRelease() on page 16-155	Releases a mutex
OCIThreadProcessInit() on page 16-156	Performs OCIThread process initialization
OCIThreadTerm() on page 16-157	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 OCIThreadIds 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 multithreaded environment or a single-threaded environment.

Syntax

```
boolean OCIThreadIsMulti ( );
```

Returns

`TRUE` if the environment is multithreaded;

`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 multithreaded application, this function must be called. The first call to it must occur strictly before any other OCIThread calls; that is, 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 16–10 Transaction Functions

Function	Purpose
OCITransCommit() on page 16-159	Commit a transaction on a service context
OCITransDetach() on page 16-162	Detach a transaction from a service context
OCITransForget() on page 16-163	Forget a prepared global transaction
OCITransMultiPrepare() on page 16-164	Prepare a transaction with multiple branches in a single cell
OCITransPrepare() on page 16-165	Prepare a global transaction for commit
OCITransRollback() on page 16-166	Roll back a transaction
OCITransStart() on page 16-167	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.

`OCI_DEFAULT` - If the transaction is non-distributed, the flags parameter is ignored, and `OCI_DEFAULT` can be passed as its value.

`OCI_TRANS_TWOPHASE` - OCI applications managing global transactions should pass this value to the flags parameter for a two-phase commit. The default is one-phase commit.

`OCI_TRANS_WRITEIMMED` - I/O is initiated by LGWR (Log Writer Process in the background) to write the (in-memory) redo buffers to the online redo logs. `IMMEDIATE` means that the redo buffers of the transaction are written out immediately by sending a message to the LGWR, which processes the message immediately.

`OCI_TRANS_WRITEBATCH` - No I/O is issued by LGWR to write the in-memory redo buffers of the transaction to the online redo logs. `BATCH` means that the LGWR batches the redo buffers before initiating I/O for the entire batch. An error occurs when you specify both `BATCH` and `IMMEDIATE`. `IMMEDIATE` is the default.

`OCI_TRANS_WRITEWAIT` - LGWR is requested to write the redo for the commit to the online redo logs and the commit waits for the redo buffers to be written to the online redo logs. `WAIT` means that the commit does not return until the in-memory redo buffers corresponding to the transaction are written in the (persistent) online redo logs.

`OCI_TRANS_WRITENOWAIT` - LGWR is requested to write the redo for the commit to the online redo logs, but the commit returns without waiting for the buffers to be written to the online redo logs. `NOWAIT` means that the commit returns to the user before the in-memory redo buffers are flushed to the online redo logs. An error occurs when you specify both `WAIT` and `NOWAIT`. `WAIT` is the default.

These last four options only affect the commit of top-level non-distributed transactions and are ignored for externally coordinated distributed transactions. They can be combined using the `OR` operator, subject to the stated restrictions.

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 terminated. 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 8-2.

```
int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISvcCtx *svchp;
  OCIStmt *stmthp;
  dvoid     *tmp;
  text sqlstmt[128];

  OCIEnvCreate(&envhp, OCI_DEFAULT, (dvoid *)0, 0, 0, 0,
        (size_t)0, (dvoid *)0);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                (size_t)0, (dvoid **) 0);
  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                (size_t)0, (dvoid **) 0);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                (size_t)0, (dvoid **) 0);

  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, (text *)"HR", strlen("HR"),
                (text *)"HR", strlen("HR"), 0, 0);

  /* update hr.employees employee_id=7902, increment salary */
  sprintf((char *)sqlstmt, "UPDATE EMPLOYEES SET SALARY = SALARY + 1 \
                                          WHERE  EMPLOYEE_ID = 7902");
```

```
  OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen((char *)sqlstmt),
                 OCI_NTV_SYNTAX, 0);
  OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* update hr.employees employee_id=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, if 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 terminated 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 8-4.

```
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];

  OCIEnvCreate(&envhp, OCI_DEFAULT, (dvoid *)0, 0, 0, 0,
        (size_t)0, (dvoid *)0);

  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 *)"HR",
```

```
            (ub4)strlen("HR"), OCI_ATTR_USERNAME, errhp);
  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"HR",
            (ub4)strlen("HR"),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 hr.employees employee_id=7902, increment salary */
  sprintf((char *)sqlstmt, "UPDATE EMPLOYEES SET SALARY = SALARY + 1 \
                                       WHERE EMPLOYEE_ID = 7902");
  OCIStmtPrepare(stmthp1, errhp, sqlstmt, strlen((char *)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 hr.employees employee_id=7934, increment salary */
  sprintf((char *)sqlstmt, "UPDATE EMPLOYEES SET SALARY = SALARY + 1 \
                                         WHERE EMPLOYEE_ID = 7934");
```

```
  OCIStmtPrepare(stmthp2, errhp, sqlstmt, strlen((char *)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];

  OCIEnvCreate(&envhp, OCI_DEFAULT, (dvoid *)0, 0, 0, 0,
        (size_t)0, (dvoid *)0);

  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 *)"HR",
           (ub4)strlen("HR"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"HR",
           (ub4)strlen("HR"),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 hr.employees employee_id=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMPLOYEES SET SALARY = SALARY + 1 \
                                        WHERE EMPLOYEE_ID = 7902");
OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen((char *)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 hr.employees employee_id=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 16–11 Miscellaneous Functions

Function	Purpose
OCIBreak() on page 16-174	Performs an immediate asynchronous break
OCIClientVersion() on page 16-175	Returns the client library version.
OCIErrorGet() on page 16-176	Returns error message and Oracle error
OCILdaToSvcCtx() on page 16-178	Toggles `Lda_Def` to service context handle
OCIPasswordChange() on page 16-179	Changes password
OCIPing() on page 16-181	Confirms that the connection and the server are active.
OCIReset() on page 16-182	Call after `OCIBreak()` to reset asynchronous operation and protocol
OCIRowidToChar() on page 16-183	Converts a Universal ROWID to character extended (base 64) representation.
OCIServerVersion() on page 16-184	Gets the Oracle version string
OCISvcCtxToLda() on page 16-185	Toggles service context handle to `Lda_Def`
OCIUserCallbackGet() on page 16-186	Identifies the callback that is registered for handle
OCIUserCallbackRegister() on page 16-188	Registers a user-created callback function

OCIBreak()

Purpose

This call performs an immediate (asynchronous) termination 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) termination 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. It can be called by a user thread in multithreaded applications, or by a user signal handler on UNIX systems. `OCIBreak()` is the obly OCI call allowed in a user signal handler.

Note: `OCIBreak()` works on on Windows systems, including Windows 2000, and Windows XP.

This call can take either the service context handle or the server context handle as a parameter to identify the function to be terminated.

See Also:

- Server Handle Attributes on page A-10.
- "Nonblocking Mode in OCI" on page 2-26
- "Canceling Calls" on page 2-24

Related Functions

OCIReset()

OCIClientVersion()

Purpose

Returns the five-digit Oracle version number of the client library at runtime.

Syntax

```
sword OCIClientVersion ( sword        *major_version,
                         sword        *minor_version,
                         sword        *update_num,
                         sword        *patch_num,
                         sword        *port_update_num );
```

Parameters

major_version (OUT)
The major version.

minor_version (OUT)
The minor version.

update_num (OUT)
The update number.

patch_num (OUT)
The patch number that was applied to the library.

port_update_num (OUT)
The port update number is the port-specific patch applied to the library.

Comments

`OCIClientVersion()` returns the version of OCI client the application is running with. This is useful for the application to know at runtime what version of OCI is being used. An application or a test program can determine the version and the patch set of a particular OCI client installation by calling this function. This is also useful if the application wants to have different codepaths depending upon the client patchset level.

In addition to `OCIClientVersion()` which is useful to find out the client version at runtime, there are two macros, `OCI_MAJOR_VERSION` and `OCI_MINOR_VERSION`, defined. These are useful for writing a generic application which can be built and run with different versions of OCI client. For example:

```
....
#if (OCI_MAJOR_VERSION > 9)
...
#endif
....
```

Related Functions

OCIServerVersion()

OCIErrorGet()

Purpose

Returns an error message in the buffer provided and an Oracle error code.

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 provided for the error message, in number of bytes. If the error message length is more than `bufsiz`, a truncated error message text is returned in `bufp`.

If `type` is set to OCI_HTYPE_ERROR, then the return code during truncation for `OCIErrorGet()` is `OCI_ERROR`. The client can then specify a bigger buffer and call `OCIErrorGet()` again.

If `bufsiz` is sufficient to hold the entire message text and the message could be successfully copied into `bufp`, the return code for `OCIErrorGet()` is `OCI_SUCCESS`.

type (IN)
The type of the handle (`OCI_HTYPE_ERROR` or `OCI_HTYPE_ENV`).

Comments

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.

Note that `OCIErrorGet()` must not be called when the return code is `OCI_SUCCESS`. Otherwise, an error message from a previously executed statement will be found by `OCIErrorGet()`.

The error handle is originally allocated with a call to `OCIHandleAlloc()`.

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.

See Also: "Error Handling in OCI" on page 2-20

Example

Here is a simplified example of a function for error checking using `OCIErrorGet()`:

```
static void checkerr(OCIError *errhp, sword status)
{
  text errbuf[512];
  ub4 buflen;
  sb4 errcode;

  if (status == OCI_SUCCESS) return;

  switch (status)
  {
  case OCI_SUCCESS_WITH_INFO:
    printf("Error - OCI_SUCCESS_WITH_INFO\n");
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
            errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
    break;
  case OCI_NEED_DATA:
    printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    printf("Error - OCI_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("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    printf("Error - OCI_STILL_EXECUTING\n");
    break;
  case OCI_CONTINUE:
    printf("Error - OCI_CONTINUE\n");
    break;
  default:
    printf("Error - %d\n", status);
    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 nonzero.

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 nonzero.

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)
The length in bytes of the new password string specified in `npasswd`. For a valid password string, `npasswd_len` must be nonzero.

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()

OCIPing()

Purpose

Makes a round trip call to the server to confirm that the connection and the server are active.

Syntax

```
sword OCIPing ( OCISvcCtx     *svchp,
                OCIError      *errhp,
                ub4           mode );
```

Parameters

svchp (IN)
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.

mode (IN)
The mode for the call. Use `OCI_DEFAULT`.

Comments

`OCIPing()` simply makes a dummy round trip call to the server, that is, a dummy packet is sent to the server for response. `OCIPing()` returns after the round trip is completed. No server operation is performed for this call itself.

`OCIPing()` can be used to make a lightweight call to the server. A successful return of the call indicates the connection and server are active. If the call blocks, the connection may be in use by other threads. If it fails, there may be some problem with the connection or the server, and the error can be retrieved from the error handle. Since `OCIPing()` is a round trip call, it also can be used to flush all the pending OCI client-side calls to the server, if any exist. For example, calling `OCIPing()` after `OCIHandleFree()` can force the execution of the pending call to close back-end cursors. It's useful when the application requires the back-end cursors to be closed immediately.

Related Functions

OCIHandleFree()

OCIReset()

Purpose

Resets the interrupted asynchronous operation and protocol. Must be called if an `OCIBreak()` call had been issued while a nonblocking 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 nonblocking mode only. Resets the interrupted asynchronous operation and protocol. Must be called if an `OCIBreak()` call had been issued while a nonblocking 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 representation 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 is returned in `bufp` 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(), OCIClientVersion()

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 (for example, `obindps()`, `ofen()`).

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 16–12, " OCI Function Codes" on page 16-191.

when (IN)
Defines when the callback is invoked. Valid modes are:

- `OCI_UCBTYPE_ENTRY` - the callback is invoked on entry into the OCI function.
- `OCI_UCBTYPE_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 16-188.

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 16-188

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 9-28.

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 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 16–12, " OCI Function Codes" on page 16-191.

when (IN)
Defines when the callback is invoked. Valid modes are:

- OCI_UCBTYPE_ENTRY - the callback is invoked on entry into the OCI function.
- OCI_UCBTYPE_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.

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,
            sb4     *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 16–12, " OCI Function Codes". Please note that callbacks can be registered for only the OCI calls listed in Table 16–7, " 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"

Table 16–12 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
10	OCISessionBegin	42	OCILobCopy, OCILobCopy2	74	OCILobFileSetName
11	OCISessionEnd	43	OCILobAppend	75	OCILobFileGetName
12	OCIPasswordChange	44	OCILobErase, OCILobErase2	76	OCILogon
13	OCIStmtPrepare	45	OCILobGetLength, OCILobGetLength2	77	OCILogoff
14	(not used)	46	OCILobTrim, OCILobTrim2	78	OCILobDisableBuffering
15	(not used)	47	,OCILobRead OCILobRead2	79	OCILobFlushBuffer
16	(not used)	48	OCILobWrite, OCILobWrite2	80	OCILobLoadFromFile, OCILobLoadFromFile2
17	OCIBindDynamic	49	(not used)	81	OCILobOpen
18	OCIBindObject	50	OCIBreak	82	OCILobClose

Table 16–12 (Cont.) OCI Function Codes

#	OCI Routine	#	OCI Routine	#	OCI Routine
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).

> **See Also:** For code examples, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".

This chapter contains these topics:

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 Advanced Topics in OCI".

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 17-2. 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 a 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.

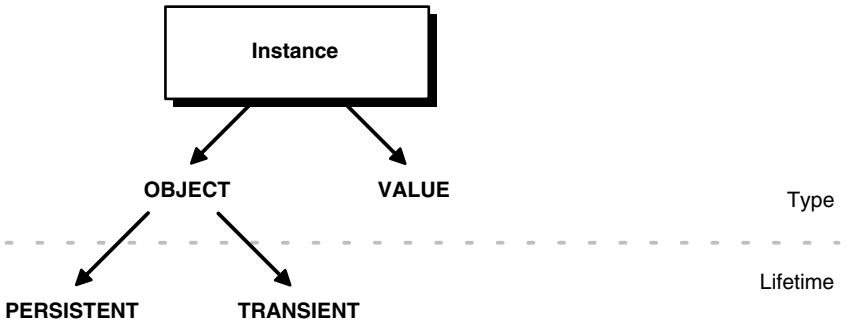
A value can be standalone or embedded. A standalone value is usually obtained by issuing a select statement. OCI also allows the client program to select a row of object table into a value by issuing a SQL statement. A referenceable object in the database can be represented as a value which cannot be 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:

Table 17–1 Type and Lifetime of Instances

Characteristic	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 10-3.

Conventions for OCI Functions

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 next:

Table 17–2 Mode of a Parameter

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.

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 18–2, " 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:

Table 17–3 Return Values of Navigational Functions

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 OCI handle passed to the function is invalid.

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, "Error Handling in OCI" on page 2-20.

Server Round Trips for Cache and Object Functions

For a table showing the number of server round trips required for individual OCI cache and object functions, refer to Appendix C, "OCI Function Server Round Trips".

Navigational Function Error Codes

Table 17–4 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 17–4 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

Table 17–4 (Cont.) OCI Navigational Functions Error Codes

Function	Possible ORA Errors
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
OCIObjectUnmarkByRef()	24350, 21560
OCIObjectUnpin()	24350, 21560, 21710
OCIOjectGetObjectRef()	24350, 21560, 21710

The ORA errors in Table 17–4 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 17–5 Flush or Refresh Functions

Function/Page	Purpose
OCICacheFlush() on page 17-7	Flush modified persistent objects in cache to server
OCICacheRefresh() on page 17-9	Refresh pinned persistent objects
OCIObjectFlush() on page 17-11	Flush a modified persistent object to the server
OCIObjectRefresh() on page 17-12	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 a 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` (for example, 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 17-11

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 a 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 13-9.

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 16-19.

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 unmarked objects are refreshed by this function.

The various meta-attribute flags and durations of an object are modified after being refreshed:

Table 17–6 Object Status After Refresh

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` indicator 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 17–7 Mark or Unmark Object and Cache Functions

Function/Page	Purpose
OCICacheUnmark() on page 17-15	Unmarks objects in the cache
OCIObjectMarkDelete() on page 17-16	Mark an object deleted / delete a value instance
OCIObjectMarkDeleteByRef() on page 17-17	Mark an object deleted given a ref
OCIObjectMarkUpdate() on page 17-18	Mark an object as updated/dirty
OCIObjectUnmark() on page 17-19	Unmarks an object
OCIObjectUnmarkByRef() on page 17-20	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 17-19 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 10-10.

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 17–8 Get Object Status functions

Function/Page	Purpose
OCIObjectExists() on page 17-22	Get the existent status of an instance
OCIObjectGetProperty() on page 17-23	Get the status of a particular object property
OCIObjectIsDirty() on page 17-26	Get the dirtied status of an instance
OCIObjectIsLocked() on page 17-27	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 10-12.

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 next.

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 13-11.

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 13-11.

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: "Locking with the NOWAIT Option" on page 13-10.

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 17–9 Miscellaneous Object functions

Function/Page	Purpose
OCIObjectCopy() on page 17-29	Copy one instance to another
OCIObjectGetAttr() on page 17-31	Gets an object attribute
OCIObjectGetInd() on page 17-33	Get `NULL` structure of an instance
OCIObjectGetObjectRef() on page 17-34	Return reference to a given object
OCIObjectGetTypeRef() on page 17-35	Get a reference to a TDO of an instance
OCIObjectLock() on page 17-36	Lock a persistent object
OCIObjectLockNoWait() on page 17-37	Lock a persistent object but do not wait for the lock
OCIObjectPin() on page 17-50	Create a new instance
OCIObjectSetAttr() on page 17-41	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 17-50

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 later)
- 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, in bytes.

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 indicator 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 indicator structure for the instance.

> **See Also:** "NULL Indicator Structure" on page 10-22 for a discussion of the NULL indicator structure and examples of its use.

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 10-12.

Related Functions

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()

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 13-10.

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 13-10.

`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 (for example, 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 3-24

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 3-24

Based on the parameters `typecode` (or `tdo`), `value` and `table`, different instances are created:

Table 17–10 Instances Created

Type Created	Not NULL	NULL
object type (*value*=`TRUE`)	value	value
object type (*value*=`FALSE`)	persistent object	transient object
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 10-26.

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 10-24

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 object and filename. Once the filename is set, the user can start reading from the FILE.

Note: Oracle now supports only binary FILEs (BFILEs).

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, in bytes.

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 17–11 Pin, Unpin, and Free Functions

Function/Page	Purpose
OCICacheFree() on page 17-44	Free objects in the cache
OCICacheUnpin() on page 17-45	Unpin persistent objects in cache or connection
OCIObjectArrayPin() on page 17-46	Pin an array of references
OCIObjectFree() on page 17-48	Free a previously allocated object
OCIObjectPin() on page 17-50	Pin an object
OCIObjectPinCountReset() on page 17-52	Unpin an object to zero pin count
OCIObjectPinTable() on page 17-53	Pin a table object with a given duration
OCIObjectUnpin() on page 17-55	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 17-48 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 10-8, and "Pin Count and Unpinning" on page 10-21.

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 17-50

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 round trip. 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 (for example, 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 13-10.

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 17-55.

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 the object is already pinned in the cache and marked dirty, then a pointer to that object is returned. The object will not be refreshed from the database.
- 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: "Pin Count and Unpinning" on page 10-21

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`, in bytes.

object_name (IN)
The name of the table.

o_n_length (IN)
The length of the table name specified in `object_name`, in bytes.

not_used (IN/OUT)
This parameter is not currently used. Pass as `NULL`.

pin_duration (IN)
The pin duration.

See Also: "OCIObjectPin()" on page 17-50.

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.

Note: The TDO (array of TDOs or table definition) obtained by this function will belong to the logical partition of the cache corresponding to the service handle (connection) passed in. If TDOs or tables are used across logical partitions, then the behavior is not known and may change between releases.

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 (that is, 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 17–12 Type Information Accessor Functions

Function/Page	Purpose
OCITypeArrayByName() on page 17-58	Get an array of TDOs given an array of object names
OCITypeArrayByRef() on page 17-60	Get an array of TDOs given an array of object references
OCITypeByName() on page 17-62	Get a TDO given an object name
OCITypeByRef() on page 17-64	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 (for example, 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 for each round trip.

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()`.

Note: The TDO (array of TDOs or table definition) obtained by this function will belong to the logical partition of the cache corresponding to the service handle (connection) passed in. If TDOs or tables are used across logical partitions, then the behavior is not known and may change between releases.

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 (for example,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()`.

Note: The TDO (array of TDOs or table definition) obtained by this function will belong to the logical partition of the cache corresponding to the service handle (connection) passed in. If TDOs or tables are used across logical partitions, then the behavior is not known and may change between releases.

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, in bytes.

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, 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.

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 13-11

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().

> **Note:** The TDO (array of TDOs or table definition) obtained by this function will belong to the logical partition of the cache corresponding to the service handle (connection) passed in. If TDOs or tables are used across logical partitions, then the behavior is not known and may change between releases.

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`.

Note: The TDO (array of TDOs or table definition) obtained by this function will belong to the logical partition of the cache corresponding to the service handle (connection) passed in. If TDOs or tables are used across logical partitions, then the behavior is not known and may change between releases.

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.

> **See Also:** For code examples, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".

This chapter contains these topics:

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 in OCI"

Conventions for OCI Functions

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 next:

Table 18–1 Mode of a Parameter

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.

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 18–2, " Function Return Values" on page 18-2.

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 18–2 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 OCI handle passed to the function is invalid.

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 in OCI" on page 2-20

Functions Returning Other Values

Some functions return values other than those listed in Table 18–2. 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 Round Trips for Datatype Mapping and Manipulation Functions

For a table showing the number of server round trips 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 in OCI".

OCI Collection and Iterator Functions

This section describes the Collection and Iterator functions.

Table 18–3 Collection and Iterator Functions

Function/Page	Purpose
OCICollAppend() on page 18-5	Collection appends element
OCICollAssign() on page 18-6	Assigns collection
OCICollAssignElem() on page 18-7	Collection assigns element
OCICollGetElem() on page 18-8	Gets pointer to an element
OCICollGetElemArray() on page 18-10	Gets an array of elements from a collection.
OCICollIsLocator() on page 18-11	Indicates whether a collection is locator-based or not
OCICollMax() on page 18-12	Returns maximum number of elements in collection
OCICollSize() on page 18-13	Gets current size of collection (in number of elements)
OCICollTrim() on page 18-15	Trims elements from the collection
OCIIterCreate() on page 18-16	Creates iterator to scan the varray elements
OCIIterDelete() on page 18-17	Deletes iterator
OCIIterGetCurrent() on page 18-18	Gets current collection element
OCIIterInit() on page 18-19	Initializes iterator to scan the given collection
OCIIterNext() on page 18-20	Gets next collection element
OCIIterPrev() on page 18-21	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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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()`.

Table 18–4 Element Pointers

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 cannot 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 by means of 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 in the target parameter.

Similarly, if the collection element was of type string (OCIString*), OCICollGetElem() returns pointer to string handle (that is, OCIString**). If a new string is assigned, through 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()

OCICollGetElemArray()

Purpose

Gets an array of elements from a collection given a starting index.

Syntax

```
sword OCICollGetElemArray ( OCIEnv              *env,
                            OCIError            *err,
                            CONST OCIColl       *coll,
                            sb4                 index,
                            boolean             *exists,
                            dvoid               **elem,
                            dvoid               **elemind,
                            uword               *nelems );
```

Parameters

env (IN/OUT)
The OCI environment handle initialized in object mode.

> **See Also:** "OCIEnvCreate()" on page 15-13 and "OCIInitialize()" on page 15-22

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)
Pointers to the elements in this collection to be returned.

index (IN)
Starting index of the elements.

exists (OUT)
Is set to `FALSE` if the element at the specified index does not exist, else it is set to `TRUE`.

elem (OUT)
Address of the desired elements to be returned.

elemind (OUT)
[optional] Address of the `NULL` indicators information to be returned. If (`elemind == NULL`) then the `NULL` indicator information will *not* be returned.

nelems (IN)
Maximum number of pointers to both `elem` and `elemind`.

Comments

Gets the address of the elements from the given position. Optionally, this function also returns the address of the element's `NULL` indicator information.

Related Functions

OCIErrorGet(), OCICollGetElem()

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 earlier example:

```
OCITableSize(...)
// 'size' returned is equal to 4
```

A trim operation (`OCICollTrim()`) decrements the count by the number of trimmed elements. Continuing the earlier 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 15-13 and "OCIInitialize()" on page 15-22.

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 (that is, 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 15-13 and "OCIInitialize()" on page 15-22

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 (that is, 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 18–5 Date Functions

Function/Page	Purpose
OCIDateAddDays() on page 18-24	Adds or subtracts days
OCIDateAddMonths() on page 18-25	Adds or subtracts months
OCIDateAssign() on page 18-26	Assigns date
OCIDateCheck() on page 18-27	Checks if the given date is valid
OCIDateCompare() on page 18-29	Compares dates
OCIDateDaysBetween() on page 18-30	Gets number of days between two dates
OCIDateFromText() on page 18-31	Converts string to date
OCIDateGetDate() on page 18-32	Gets the date portion of a date
OCIDateGetTime() on page 18-33	Gets the time portion of a date
OCIDateLastDay() on page 18-34	Gets date of last day of month
OCIDateNextDay() on page 18-35	Gets date of next day
OCIDateSetDate() on page 18-36	Sets the date portion of a date
OCIDateSetTime() on page 18-37	Sets the time portion of a date
OCIDateSysDate() on page 18-38	Gets current system date and time
OCIDateToText() on page 18-39	Converts date to string
OCIDateTimeAssign() on page 18-41	Performs datetime assignment
OCIDateTimeCheck() on page 18-42	Checks if the given date is valid
OCIDateTimeCompare() on page 18-44	Compares two datetime values
OCIDateTimeConstruct() on page 18-45	Constructs a datetime descriptor
OCIDateTimeConvert() on page 18-47	Converts one datetime type to another
OCIDateTimeFromArray() on page 18-48	Converts an array of size `OCI_DT_ARRAYLEN` to an `OCIDateTime` descriptor
OCIDateTimeFromText() on page 18-49	Converts the given string to Oracle datetime type in the `OCIDateTime` descriptor, according to the specified format
OCIDateTimeGetDate() on page 18-51	Gets the date (year, month, day) portion of a datetime value
OCIDateTimeGetTime() on page 18-52	Gets the time (hour, min, second, fractional second) out of a datetime value
OCIDateTimeGetTimeZoneName() on page 18-53	Gets the time zone name portion of a datetime value
OCIDateTimeGetTimeZoneOffset() on page 18-54	Gets the time zone (hour, minute) portion of a datetime value
OCIDateTimeIntervalAdd() on page 18-55	Adds an interval to a datetime to produce a resulting datetime
OCIDateTimeIntervalSub() on page 18-56	Subtracts an interval from a datetime and stores the result in a datetime

Table 18–5 (Cont.) Date Functions

Function/Page	Purpose
OCIDateTimeSubtract() on page 18-57	Takes two datetimes as input and stores their difference in an interval
OCIDateTimeSysTimeStamp() on page 18-58	Gets the system current date and time as a timestamp with time zone
OCIDateTimeToArray() on page 18-59	Converts a `OCIDateTime` descriptor to an array
OCIDateTimeToText() on page 18-60	Converts the given date to a string according to the specified format
OCIDateZoneToZone() on page 18-62	Converts date from one time zone to another zone
OCIIntervalAdd() on page 18-63	Adds two intervals to produce a resulting interval
OCIIntervalAssign() on page 18-64	Copies one interval to another
OCIIntervalCheck() on page 18-65	Checks the validity of an interval
OCIIntervalCompare() on page 18-67	Compares two intervals
OCIIntervalDivide() on page 18-68	Divides an interval by an Oracle NUMBER to produce an interval
OCIIntervalFromNumber() on page 18-69	Converts an Oracle NUMBER to an interval
OCIIntervalFromText() on page 18-70	Given an interval string, returns the interval represented by the string
OCIIntervalFromTZ() on page 18-72	Returns an `OCI_DTYPE_INTERVAL_DS`.
OCIIntervalGetDaySecond() on page 18-73	Gets values of day, hour, minute, and second from an interval
OCIIntervalGetYearMonth() on page 18-74	Gets year and month from an interval
OCIIntervalMultiply() on page 18-75	Multiplies an interval by an Oracle NUMBER to produce an interval
OCIIntervalSetDaySecond() on page 18-76	Sets day, hour, minute, and second in an interval
OCIIntervalSetYearMonth() on page 18-77	Sets year and month in an interval
OCIIntervalSubtract() on page 18-78	Subtracts two intervals and stores the result in an interval
OCIIntervalToNumber() on page 18-79	Converts an interval to an Oracle NUMBER
OCIIntervalToText() on page 18-80	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:

Table 18–6 Error Bits

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)
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 is:

`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:

Table 18–7 Comparison Results

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 the *Oracle Database 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).

```
OCIDate one_day, next_day;
/* Add code here to set one_day to be '18-APR-96' */
OCIDateNextDay(err, &one_day, "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,
                      CONST 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 the *Oracle Database 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 next:

Table 18–8 Error Bits

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)
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

Table 18–8 (Cont.) Error Bits

Macro Name	Bit Number	Error
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 is:

```
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:

Table 18–9 Comparison Results

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. A string representation of the time zone offset from GMT in the format "[+|-][HH:MM]". For example, "-08:00".

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 containing a date 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, 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 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 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 the *Oracle Database 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 is greater than `SB4MAXVAL`,
- or, the resulting year is less than `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:

Table 18–10 Error Bits

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)
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:

Table 18–11 Comparison Results

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 is 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, hour, minute, 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 is true:

- the resulting year is greater than SB4MAXVAL
- the resulting year is less than SB4MINVAL

Related Functions

OCIIntervalDivide()

OCIIntervalSetDaySecond()

Purpose

Sets day, hour, minute, 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,

- the resulting year is greater than `SB4MAXVAL`
- the resulting year is less than `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 is greater than `SB4MAXVAL`
- the resulting year is less than `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.

See Also: "OCINumber Examples" on page 11-10

Table 18–12 NUMBER Functions

Function/Page	Purpose
OCINumberAbs() on page 18-84	Computes the absolute value
OCINumberAdd() on page 18-85	Adds NUMBERs
OCINumberArcCos() on page 18-86	Computes the arc cosine
OCINumberArcSin() on page 18-87	Computes the arc sine
OCINumberArcTan() on page 18-88	Computes the arc tangent
OCINumberArcTan2() on page 18-89	Computes the arc tangent of two NUMBERs
OCINumberAssign() on page 18-90	Assigns one NUMBER to another
OCINumberCeil() on page 18-91	Computes the ceiling of NUMBER
OCINumberCmp() on page 18-92	Compares NUMBERs
OCINumberCos() on page 18-93	Computes the cosine
OCINumberDec() on page 18-94	Decrements a NUMBER
OCINumberDiv() on page 18-95	Divides two NUMBERs
OCINumberExp() on page 18-96	Raises *e* to the specified Oracle NUMBER power
OCINumberFloor() on page 18-97	Computes the floor of a NUMBER
OCINumberFromInt() on page 18-98	Converts an integer to an Oracle NUMBER
OCINumberFromReal() on page 18-99	Convert a real to an Oracle NUMBER
OCINumberFromText() on page 18-100	Convert a string to an Oracle NUMBER
OCINumberHypCos() on page 18-101	Computes the hyperbolic cosine
OCINumberHypSin() on page 18-102	Computes the hyperbolic sine
OCINumberHypTan() on page 18-103	Computes the hyperbolic tangent
OCINumberInc() on page 18-104	Increments an Oracle NUMBER
OCINumberIntPower() on page 18-105	Raises a given base to an integer power
OCINumberIsInt() on page 18-106	Tests if a NUMBER is an integer
OCINumberIsZero() on page 18-107	Tests if a NUMBER is zero
OCINumberLn() on page 18-108	Computes the natural logarithm
OCINumberLog() on page 18-109	Computes the logarithm to an arbitrary base
OCINumberMod() on page 18-110	Modulo division
OCINumberMul() on page 18-111	Multiplies NUMBERs
OCINumberNeg() on page 18-112	Negates a NUMBER
OCINumberPower() on page 18-113	Exponentiation to base *e*
OCINumberPrec() on page 18-114	Rounds a NUMBER to a specified number of decimal places

Table 18–12 (Cont.) NUMBER Functions

Function/Page	Purpose
OCINumberRound() on page 18-115	Rounds an Oracle NUMBER to a specified decimal place
OCINumberSetPi() on page 18-116	Initializes a NUMBER to Pi
OCINumberSetZero() on page 18-117	Initializes a NUMBER to zero
OCINumberShift() on page 18-118	Multiplies by 10, shifting specified number of decimal places
OCINumberSign() on page 18-119	Obtains the sign of an Oracle NUMBER
OCINumberSin() on page 18-120	Computes the sine
OCINumberSqrt() on page 18-121	Computes the square root of a NUMBER
OCINumberSub() on page 18-122	Subtracts NUMBERs
OCINumberTan() on page 18-123	Computes the tangent
OCINumberToInt() on page 18-124	Converts an Oracle NUMBER to an integer
OCINumberToReal() on page 18-125	Converts an Oracle NUMBER to a real
OCINumberToRealArray() on page 18-126	Converts an array of NUMBER to a real array.
OCINumberToText() on page 18-127	Converts an Oracle NUMBER to a string
OCINumberTrunc() on page 18-129	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(), on page 18-119

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)
`NUMBER`s to compare.

result (OUT)
Comparison result:

Table 18–13 Comparison Results

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

```
sword OCINumberDec ( 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()

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 *Oracle Database 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 either 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 either of the NUMBER arguments is NULL.

Caution: An Oracle number overflow causes an unpredictable result value.

Related Functions

OCIErrorGet(), OCINumberHypSin(), 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 either 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

```
sword 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 either 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 the NUMBER argument 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 either 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 either 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)
A valid OCI error handle. This function does not check for errors because the function will never produce an error.

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:

Table 18–14 Values of result

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 either 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. These constants are defined in float.h.

You must pass a valid OCINumber to this function. Otherwise, the result is undefined.

Related Functions

OCIErrorGet(), OCINumberFromReal()

OCINumberToRealArray()

Purpose

Converts an array of NUMBER to an array of real.

Syntax

```
sword OCINumberToRealArray ( OCIError         *err,
                             CONST OCINumber  **number,
                             uword            elems,
                             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)
Pointer to array of NUMBER to be converted.

elems (IN)
Maximum number of NUMBER pointers.

rsl_length (IN)
The size of the desired result, that is, sizeof({ float | double | long double }).

rsl (OUT)
Pointer to array of space for storing the result.

Comments

Native type conversion function that 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 constants are defined in the float.h header file.

You must pass a valid OCINumber to this function. Otherwise, the result is undefined.

Returns

- OCI_SUCCESS - the function completes successfully.
- OCI_INVALID_HANDLE - if err is NULL.
- OCI_ERROR - if number or rsl is NULL, or rsl_length is 0.

Related Functions

OCIErrorGet(), OCINumberToReal()

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 *Oracle Database 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 18–15 Raw Functions

Function/Page	Purpose
OCIRawAllocSize() on page 18-131	Get allocated size of raw memory in bytes
OCIRawAssignBytes() on page 18-132	Assign raw bytes to raw
OCIRawAssignRaw() on page 18-133	Assign raw to raw
OCIRawPtr() on page 18-134	Get raw data Pointer
OCIRawResize() on page 18-135	Resize memory of variable-length raw
OCIRawSize() on page 18-136	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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 15-13 and "OCIInitialize()" on page 15-22

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 18–16 Ref Functions

Function/Page	Purpose
OCIRefAssign() on page 18-138	Assign one REF to another
OCIRefClear() on page 18-139	Clear or nullify a REF
OCIRefFromHex() on page 18-140	Convert hexadecimal string to REF
OCIRefHexSize() on page 18-141	Return size of hexadecimal representation of REF
OCIRefIsEqual() on page 18-142	Compare two REFs for equality
OCIRefIsNull() on page 18-143	Test if a REF is `NULL`
OCIRefToHex() on page 18-144	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 15-13 and OCIInitialize() on page 15-22

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 (*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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 (that is, 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 18–17 String Functions

Function/Page	Purpose
OCIStringAllocSize() on page 18-146	Get allocated size of string memory in bytes
OCIStringAssign() on page 18-147	Assign string to string
OCIStringAssignText() on page 18-148	Assign text string to string
OCIStringPtr() on page 18-149	Get string pointer
OCIStringResize() on page 18-150	Resize string memory
OCIStringSize() on page 18-151	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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 bytes.

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()

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 18–18 Table Functions

Function/Page	Purpose
OCITableDelete() on page 18-153	Delete element
OCITableExists() on page 18-154	Test whether element exists
OCITableFirst() on page 18-155	Return first index of table
OCITableLast() on page 18-156	Return last index of table
OCITableNext() on page 18-157	Return next available index of table
OCITablePrev() on page 18-158	Return previous available index of table
OCITableSize() on page 18-159	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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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 15-13 and OCIInitialize() on page 15-22

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

This chapter presents the cartridge functions.

> **See Also:** For code examples, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".

This chapter contains these topics:

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 procedures in *Oracle Database Application Developer's Guide - Fundamentals*

Then the cartridge services functions are described.

> **See Also:** For more information about using these functions, see *Oracle Database Data Cartridge Developer's Guide*

Conventions for OCI Functions

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 next:

Table 19–1 Mode of a Parameter

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.

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 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 19–2 External Procedures Functions

Function/Page	Purpose
OCIExtProcAllocCallMemory() on page 19-4	Allocates memory for the duration of the External Procedure
OCIExtProcRaiseExcp() on page 19-5	Raises an Exception to PL/SQL
OCIExtProcRaiseExcpWithMsg() on page 19-6	Raises an exception with a message
OCIExtProcGetEnv() on page 19-7	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 19-2

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 19-2

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 19-2

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 19-2.

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

This section describes the memory services functions.

Table 19–3 Memory Services Functions

Function/Page	Purpose
OCIDurationBegin() on page 19-9	Starts a user duration.
OCIDurationEnd() on page 19-10	Terminates a user duration.
OCIMemoryAlloc() on page 19-11	Allocates memory of a given size from a given duration.
OCIMemoryResize() on page 19-12	Resizes a memory chunk.
OCIMemoryFree() on page 19-13	Frees a memory chunk.

See Also: For more information about using these functions, see *Oracle Database 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, that is, 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

This section describes the maintaining context functions.

Table 19–4 Maintaining Context Functions

Function/Page	Purpose
OCIContextSetValue() on page 19-15	Save a value (or address) for a particular duration.
OCIContextGetValue() on page 19-16	Return the value stored in the context.
OCIContextClearValue() on page 19-17	Remove the value stored in the context.
OCIContextGenerateKey() on page 19-18	Returns a unique 4-byte value each time it is called.

See Also: For more information about using these functions, see Oracle Database 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

This section describes the parameter manager interface functions.

Table 19–5 Parameter Manager Interface Functions

Function/Page	Purpose
OCIExtractFromFile() on page 19-26	The keys and their values in the given file are processed.
OCIExtractFromList() on page 19-33	Generates a list of values for the parameter denoted by `index` in the parameter list.
OCIExtractFromStr() on page 19-27	The keys and the their values in the given string are processed.
OCIExtractInit() on page 19-20	Initializes the parameter manager.
OCIExtractReset() on page 19-22	Re-initializes memory.
OCIExtractSetKey() on page 19-24	Registers information about a key with the parameter manager.
OCIExtractSetNumKeys() on page 19-23	Informs the parameter manager of the number of keys that will be registered.
OCIExtractTerm() on page 19-21	Releases all dynamically allocated storage.
OCIExtractToBool() on page 19-29	Gets the boolean value for the specified key.
OCIExtractToInt() on page 19-28	Gets the integer value for the specified key.
OCIExtractToList() on page 19-32	Generates a list of parameters from the parameter structures that are stored in memory.
OCIExtractToOCINum() on page 19-31	Gets the number value for the specified key.
OCIExtractToStr() on page 19-30	Gets the string value for the specified key.

See Also: For more information about using these functions, see Oracle Database 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 Globalization Support information is stored inside the parameter manager context and used in subsequent calls to `OCIExtract` functions.

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  *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()` functions 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`; for diagnostic information call `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`.

input (IN)
A `NULL`-terminated input string.

Comments

`OCIExtractSetNumKeys()` and `OCIExtractSetKey()` functions 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

This section describes the file I/O interface functions.

Table 19–6 File I/O Interface Functions

Function/Page	Purpose
OCIFileClose() on page 19-39	Closes a previously opened file.
OCIFileExists() on page 19-44	Tests to see if the file exists.
OCIFileFlush() on page 19-46	Writes buffered data to a file.
OCIFileGetLength() on page 19-45	Gets the length of a file.
OCIFileInit() on page 19-35	Initializes the OCIFile package.
OCIFileOpen() on page 19-37	Opens a file.
OCIFileRead() on page 19-40	Reads from a file into a buffer.
OCIFileSeek() on page 19-42	Changes the current position in a file.
OCIFileTerm() on page 19-36	Terminates the OCIFile package.
OCIFileWrite() on page 19-41	Writes *buflen* bytes into the file.

See Also: For more information about using these functions, see Oracle Database 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

This section describes the string formatting functions.

Table 19–7 String Formatting Functions

Function/Page	Purpose
OCIFormatInit() on page 19-48	Initializes the OCIFormat package.
OCIFormatString() on page 19-50	Writes a text string into the supplied text buffer.
OCIFormatTerm() on page 19-49	Terminates the OCIFormat package.

See Also: For more information about using these functions, see Oracle Database 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 '%', that is, 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 next. 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.

Table 19–8 Format Modifier Flags

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.

Table 19–9 Codes

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 for example, 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'`	operating system-specific pointer printout

Table 19–9 (Cont.) Codes

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;
sbig_ora  returnLen;

/* 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), &returnLen
                (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), &returnLen
                (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.

> **See Also:** For code examples, see the demonstration programs included with your Oracle installation. For additional information, refer to Appendix B, "OCI Demonstration Programs".

This chapter contains these topics:

- 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 Any Type and Data functions in detail.

> **See Also:** "AnyType, AnyData and AnyDataSet Interfaces" on page 11-20

Conventions for OCI Functions

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 parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described next:

Table 20–1 Mode of a Parameter

Mode	Description
IN	A parameter that passes data to the OCI

Table 20–1 (Cont.) Mode of a Parameter

Mode	Description
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

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 20–2 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 OCI handle passed to the function is invalid.

See Also: For more information about return codes and error handling, see the section "Error Handling in OCI" on page 2-20

OCI Type Interface Functions

This section describes the Type Interface functions.

Table 20–3 Type Interface Functions

Function/Page	Purpose
OCITypeAddAttr() on page 20-4	Adds an attribute to an object type that was constructed earlier with typecode `OCI_TYPECODE_OBJECT`.
OCITypeBeginCreate() on page 20-5	Begins the construction process for a transient type. The type will be anonymous (no name).
OCITypeEndCreate() on page 20-6	Finishes construction of a type description. Subsequently, only access will be allowed.
OCITypeSetBuiltin() on page 20-7	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`, and so on).
OCITypeSetCollection() on page 20-8	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, in bytes.

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 3-24. Additional information on built-in types (precision, scale for numbers, character set information for `VARCHAR2`s, and so on) 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, and so on).

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, and so on). 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 20–4 Any Data Functions

Function/Page	Purpose
OCIAnyDataAccess() on page 20-10	Retrieves the data value of an `OCIAnyData`.
OCIAnyDataAttrGet() on page 20-12	Gets the value of the attribute at the current position in the `OCIAnyData`.
OCIAnyDataAttrSet() on page 20-15	Sets the attribute at the current position with a given value.
OCIAnyDataBeginCreate() on page 20-17	Allocates an `OCIAnyData` for the given duration and initializes it with the type information.
OCIAnyDataCollAddElem() on page 20-19	Adds the next collection element to the collection attribute of the `OCIAnyData` at the current attribute position.
OCIAnyDataCollGetElem() on page 20-21	Accesses sequentially the elements in the collection attribute at the current position in the `OCIAnyData`.
OCIAnyDataConvert() on page 20-23	Constructs an `OCIAnyData` with the given data value which will be of the given type.
OCIAnyDataDestroy() on page 20-25	Frees an AnyData .
OCIAnyDataEndCreate() on page 20-26	Marks the end of `OCIAnyData` creation.
OCIAnyDataGetCurrAttrNum() on page 20-27	Returns the current attribute number of the `OCIAnyData`.
OCIAnyDataGetType() on page 20-28	Gets the type corresponding to an AnyData value.
OCIAnyDataIsNull() on page 20-29	Checks if `OCIAnyData` is `NULL`.
OCIAnyDataTypeCodeToSqlt() on page 20-30	Converts the `OCITypeCode` for an AnyData value to the SQLT code that corresponds to the representation of the value as returned by the OCIAnyData API.

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

```
sword 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, in bytes, 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, in bytes, 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:

Table 20–5 Datatypes and Attribute Values

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 20–5 (Cont.) Datatypes and Attribute Values

Datatypes	attr_value
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:

Table 20–6 Datatypes and Attribute Values

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)

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, and so on), this parameter can be NULL. It should be non-NULL for user defined types (OCI_TYPECODE_OBJECT, OCI_TYPECODE_REF, collection types, and so on).

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()" on page 20-15 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()" on page 20-12) 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

```
sword 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, and so on), 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 AnyData.

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 AnyData value. 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 the contents of the type within the 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 checked.

isNull (IN/OUT)
TRUE if NULL, else FALSE.

OCIAnyDataTypeCodeToSqlt()

Purpose

Converts the `OCITypeCode` for an AnyData value to the SQLT code that corresponds to the representation of the value as returned by the OCIAnyData API.

Syntax

```
sword  OCIAnyDataTypeCodeToSqlt ( OCIError       *errhp,
                                  OCITypeCode    tc,
                                  ub1            *sqltcode,
                                  ub1            *csfrm) ;
```

Parameters

errhp (IN/OUT)
The OCI error handle. If there is an error, it is recorded in `errhp` and this function returns `OCI_ERROR`. Diagnostic information can be obtained by calling `OCIErrorGet()`.

tc (IN)
`OCITypeCode` corresponding to the AnyData value.

sqltcode (OUT)
SQLT code corresponding to the user format of the typecode.

csfrm (OUT)
Charset form corresponding to the user format of the typecode. Meaningful only for character types. Returns SQLCS_IMPLICIT or SQLCS_NCHAR (for NCHAR types).

Comments

This function converts `OCI_TYPECODE_CHAR` as well as `OCI_TYPECODE_VARCHAR2` to SQLT_VST (which corresponds to the OCIString mapping) with a charset form of SQLCS_IMPLICIT. `OCI_TYPECODE_NVARCHAR2` will also return SQLT_VST (OCIString mapping is used by the OCIAnyData API) with a charset form of SQLCS_NCHAR.

> **See Also:** For more information see "NCHAR Typecodes for OCIAnyData Functions" on page 11-24

OCI Any Data Set Interface Functions

This section describes the Any Data Set Interface functions.

Table 20–7 Any Data Set Functions

Function/Page	Purpose
OCIAnyDataSetAddInstance() on page 20-32	Adds a new skeleton instance to the `OCIAnyDataSet` and all the attributes of the instance are set to `NULL`.
OCIAnyDataSetBeginCreate() on page 20-33	Allocates an `OCIAnyDataSet` for the given duration and initializes it with the type information.
OCIAnyDataSetDestroy() on page 20-34	Frees the `OCIAnyDataSet`.
OCIAnyDataSetEndCreate() on page 20-35	Marks the end of `OCIAnyDataSet` creation.
OCIAnyDataSetGetCount() on page 20-36	Gets the number of instances in the `OCIAnyDataSet`
OCIAnyDataSetGetInstance() on page 20-37	Returns the `OCIAnyData` corresponding to an instance at the current position and updates the current position.
OCIAnyDataSetGetType() on page 20-38	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, and so on), this parameter can be NULL. It should be non-NULL for user defined types (OCI_TYPECODE_OBJECT, OCI_TYPECODE_REF, collection types, and so on).

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.

21

OCI Globalization Support Functions

This chapter describes the OCI Globalization support functions.

This chapter contains these topics:

Introduction to Globalization Support in OCI

This chapter describes the globalization support functions in detail.

See Also: *Oracle Database Globalization Support Guide*

Conventions for OCI Functions

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:

Table 21–1 Mode of a Parameter

Mode	Description
IN	A parameter that passes data to the OCI
OUT	A parameter that receives data from the OCI on this call

Table 21–1 (Cont.) Mode of a Parameter

Mode	Description
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.

Returns

The values returned. The standard return values have the following meanings:

Table 21–2 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 OCI handle passed to the function is invalid.

See Also: For more information about return codes and error handling, see the section "Error Handling in OCI" on page 2-20

Related Functions

A list of related function calls. An optional section.

OCI Locale Functions

This section describes the OCI locale functions.

An Oracle locale consists of language, territory, and character set definitions. The locale determines conventions such as day and month names, as well as date, time, number, and currency formats. A globalized application obeys a user's locale setting and cultural conventions. For example, when the locale is set to German, users expect to see day and month names in German.

Table 21–3 OCI Locale Functions

Function/Page	Purpose
OCINlsCharSetIdToName() on page 21-4	Returns the Oracle character set name from the specified character set ID.
OCINlsCharSetNameToId() on page 21-5	Returns the Oracle character set ID for the specified Oracle character set name.
OCINlsEnvironmentVariableGet() on page 21-6	Returns the character set ID from `NLS_LANG` or the national character set id from `NLS_NCHAR`
OCINlsGetInfo() on page 21-8	Copies locale information from an OCI environment or user session handle into an array pointed to by the destination buffer within a specified size.
OCINlsNumericInfoGet() on page 21-11	Copies numeric language information from the OCI environment handle into an output number variable.

OCINlsCharSetIdToName()

Purpose

Returns the Oracle character set name from the specified character set ID.

Syntax

```
sword OCINlsCharSetIdToName ( dvoid     *hndl,
                              oratext   *buf,
                              size_t    buflen
                              ub2       id );
```

Parameters

hndl (IN/OUT)
OCI environment or session handle. If the handle is invalid, then the function returns `OCI_INVALID_HANDLE`.

buf (OUT)
Points to the destination buffer. If the function returns `OCI_SUCCESS`, then the parameter contains a `NULL`-terminated string for the character set name.

buflen (IN)
The size of the destination buffer. The recommended size is `OCI_NLS_MAXBUFSZ` to guarantee storage for an Oracle character set name. If the size of the destination buffer is smaller than the length of the character set name, then the function returns `OCI_ERROR`.

id (IN)
Oracle character set ID

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`

OCINlsCharSetNameToId()

Purpose

Returns the Oracle character set ID for the specified Oracle character set name.

Syntax

```
ub2 OCINlsCharSetNameToId ( dvoid           *hndl,
                            CONST oratext   *name );
```

Parameters

hndl (IN/OUT)
OCI environment or session handle. If the handle is invalid, then the function returns zero.

name (IN)
Pointer to a `NULL`-terminated Oracle character set name. If the character set name is invalid, then the function returns zero.

Returns

Character set ID if the specified character set name and the OCI handle are valid. Otherwise, it returns `0`.

OCINlsEnvironmentVariableGet()

Purpose

Returns the character set ID from NLS_LANG or the national character set id from NLS_NCHAR.

Syntax

```
sword OCINlsEnvironmentVariableGet ( dvoid      *val,
                                     size_t     size,
                                     ub2        item,
                                     ub2        charset,
                                     size_t     *rsize );
```

Parameters

val (IN/OUT)
Returns a value of a globalization support environment variable such as the NLS_LANG character set ID or the NLS_NCHAR character set ID.

size (IN)
Specifies the size of the given output value, which is applicable only to string data. The maximum length for each piece of information is OCI_NLS_MAXBUFSZ bytes. In the case of numeric data, this argument is ignored.

item (IN)
Specifies one of the following values to get from the globalization support environment variable:

- OCI_NLS_CHARSET_ID: NLS_LANG character set ID in ub2 datatype.
- OCI_NLS_NCHARSET_ID: NLS_NCHAR character set ID in ub2 datatype.

charset (IN)
Specifies the character set ID for retrieved string data. If it is 0, then the NLS_LANG value is used. OCI_UTF16ID is a valid value for this argument. In the case of numeric data, this argument is ignored.

rsize (OUT)
The length of the return value in bytes.

Comments

Following globalization support convention, the national character set ID is the same as the character set ID if NLS_NCHAR is not set. If NLS_LANG is not set, then the default character set ID is returned.

To allow for future enhancements of this function (to retrieve other values from environment variables) the datatype of the output val is a pointer to dvoid. String data is not terminated by NULL.

Note that the function does not take an environment handle, so the character set ID and the national character set ID that it returns are the values specified in NLS_LANG and NLS_NCHAR, instead of the values saved in the OCI environment handle. To get the character set IDs used by the OCI environment handle, call OCIAttrGet() for OCI_ATTR_ENV_CHARSET and OCI_ATTR_ENV_NCHARSET, respectively.

Returns

`OCI_SUCCESS` or `OCI_ERROR`

Related Functions

OCIEnvNlsCreate()

OCINlsGetInfo()

Purpose

Obtains locale information from an OCI environment or user session handle into an array pointed to by the destination buffer within a specified size.

Syntax

```
sword OCINlsGetInfo ( dvoid       *hndl,
                      OCIError    *errhp,
                      OraText     *buf,
                      size_t      buflen,
                      ub2         item );
```

Parameters

hndl (IN/OUT)
The OCI environment or user session handle initialized in object mode.

errhp (IN/OUT)
The OCI error handle. If there is an error, then it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

buf (OUT)
Pointer to the destination buffer. Returned strings are terminated by a `NULL` character.

buflen (IN)
The size of the destination buffer. The maximum length for each piece of information is `OCI_NLS_MAXBUFSZ` bytes.

`OCI_NLS_MAXBUFSIZE`: When calling `OCINlsGetInfo`(), you need to allocate the buffer to store the returned information. The buffer size depends on which item you are querying and what encoding you are using to store the information. Developers should not need to know how many bytes it takes to store `January` in Japanese using JA16SJIS encoding. The `OCI_NLS_MAXBUFSZ` attribute guarantees that the buffer is big enough to hold the largest item returned by `OCINlsGetInfo`().

item (IN)
Specifies which item in the OCI environment handle to return. It can be one of the following values:

`OCI_NLS_DAYNAME1`: Native name for Monday
`OCI_NLS_DAYNAME2`: Native name for Tuesday
`OCI_NLS_DAYNAME3`: Native name for Wednesday
`OCI_NLS_DAYNAME4`: Native name for Thursday
`OCI_NLS_DAYNAME5`: Native name for Friday
`OCI_NLS_DAYNAME6`: Native name for Saturday
`OCI_NLS_DAYNAME7`: Native name for Sunday
`OCI_NLS_ABDAYNAME1`: Native abbreviated name for Monday
`OCI_NLS_ABDAYNAME2`: Native abbreviated name for Tuesday
`OCI_NLS_ABDAYNAME3`: Native abbreviated name for Wednesday
`OCI_NLS_ABDAYNAME4`: Native abbreviated name for Thursday
`OCI_NLS_ABDAYNAME5`: Native abbreviated name for Friday
`OCI_NLS_ABDAYNAME6`: Native abbreviated name for Saturday
`OCI_NLS_ABDAYNAME7`: Native abbreviated name for Sunday

OCI_NLS_MONTHNAME1: Native name for January
OCI_NLS_MONTHNAME2: Native name for February
OCI_NLS_MONTHNAME3: Native name for March
OCI_NLS_MONTHNAME4: Native name for April
OCI_NLS_MONTHNAME5: Native name for May
OCI_NLS_MONTHNAME6: Native name for June
OCI_NLS_MONTHNAME7: Native name for July
OCI_NLS_MONTHNAME8: Native name for August
OCI_NLS_MONTHNAME9: Native name for September
OCI_NLS_MONTHNAME10: Native name for October
OCI_NLS_MONTHNAME11: Native name for November
OCI_NLS_MONTHNAME12: Native name for December
OCI_NLS_ABMONTHNAME1: Native abbreviated name for January
OCI_NLS_ABMONTHNAME2: Native abbreviated name for February
OCI_NLS_ABMONTHNAME3: Native abbreviated name for March
OCI_NLS_ABMONTHNAME4: Native abbreviated name for April
OCI_NLS_ABMONTHNAME5: Native abbreviated name for May
OCI_NLS_ABMONTHNAME6: Native abbreviated name for June
OCI_NLS_ABMONTHNAME7: Native abbreviated name for July
OCI_NLS_ABMONTHNAME8: Native abbreviated name for August
OCI_NLS_ABMONTHNAME9: Native abbreviated name for September
OCI_NLS_ABMONTHNAME10: Native abbreviated name for October
OCI_NLS_ABMONTHNAME11: Native abbreviated name for November
OCI_NLS_ABMONTHNAME12: Native abbreviated name for December
OCI_NLS_YES: Native string for affirmative response
OCI_NLS_NO: Native negative response
OCI_NLS_AM: Native equivalent string of AM
OCI_NLS_PM: Native equivalent string of PM
OCI_NLS_AD: Native equivalent string of AD
OCI_NLS_BC: Native equivalent string of BC
OCI_NLS_DECIMAL: Decimal character
OCI_NLS_GROUP: Group separator
OCI_NLS_DEBIT: Native symbol of debit
OCI_NLS_CREDIT: Native symbol of credit
OCI_NLS_DATEFORMAT: Oracle date format
OCI_NLS_INT_CURRENCY: International currency symbol
OCI_NLS_DUAL_CURRENCY: Dual currency symbol
OCI_NLS_LOC_CURRENCY: Locale currency symbol
OCI_NLS_LANGUAGE: Language name
OCI_NLS_ABLANGUAGE: Abbreviation for language name
OCI_NLS_TERRITORY: Territory name
OCI_NLS_CHARACTER_SET: Character set name
OCI_NLS_LINGUISTIC_NAME: Linguistic sort name
OCI_NLS_CALENDAR: Calendar name
OCI_NLS_WRITING_DIR: Language writing direction
OCI_NLS_ABTERRITORY: Territory abbreviation
OCI_NLS_DDATEFORMAT: Oracle default date format
OCI_NLS_DTIMEFORMAT: Oracle default time format
OCI_NLS_SFDATEFORMAT: Local date format
OCI_NLS_SFTIMEFORMAT: Local time format
OCI_NLS_NUMGROUPING: Number grouping fields
OCI_NLS_LISTSEP: List separator
OCI_NLS_MONDECIMAL: Monetary decimal character
OCI_NLS_MONGROUP: Monetary group separator
OCI_NLS_MONGROUPING: Monetary grouping fields

OCI_NLS_INT_CURRENCYSEP: International currency separator

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR

OCINlsNumericInfoGet()

Purpose

Obtains numeric language information from the OCI environment handle and puts it into an output number variable.

Syntax

```
sword OCINlsNumericInfoGet ( dvoid       *hndl,
                             OCIError    *errhp,
                             sb4         *val,
                             ub2         item );
```

Parameters

errhp (IN/OUT)
The OCI error handle. If there is an error, then it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

val (OUT)
Pointer to the output number variable. If the function returns `OCI_SUCCESS`, then the parameter contains the requested globalization support numeric information.

item (IN)
It specifies which item to get from the OCI environment handle and can be one of following values:

- `OCI_NLS_CHARSET_MAXBYTESZ`: Maximum character byte size for OCI environment or session handle character set
- `OCI_NLS_CHARSET_FIXEDWIDTH`: Character byte size for fixed-width character set; `0` for variable-width character set

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`

OCI Locale-Mapping Function

This section contains an OCI locale-mapping function.

Table 21–4 OCI Locale-Mapping Function

Function/Page	Purpose
OCINlsNameMap() on page 21-13	Maps Oracle character set names, language names, and territory names to and from Internet Assigned Numbers Authority (IANA) and International Organization for Standardization (ISO) names.

OCINlsNameMap()

Purpose

Maps Oracle character set names, language names, and territory names to and from Internet Assigned Numbers Authority (IANA) and International Organization for Standardization (ISO) names.

Syntax

```
sword OCINlsNameMap ( dvoid            *hndl,
                      oratext          *buf,
                      size_t           buflen,
                      CONST oratext    *srcbuf,
                      uword            flag );
```

Parameters

hndl (IN/OUT)
OCI environment or session handle. If the handle is invalid, then the function returns OCI_INVALID_HANDLE.

buf (OUT)
Points to the destination buffer. If the function returns OCI_SUCCESS, then the parameter contains a NULL-terminated string for the requested name.

buflen (IN)
The size of the destination buffer. The recommended size is OCI_NLS_MAXBUFSZ to guarantee storage of a globalization support name. If the size of the destination buffer is smaller than the length of the name, then the function returns OCI_ERROR.

srcbuf (IN)
Pointer to a NULL-terminated globalization support name. If it is not a valid name, then the function returns OCI_ERROR.

flag (IN)
It specifies the direction of the name mapping and can take the following values:

OCI_NLS_CS_IANA_TO_ORA: Map character set name from IANA to Oracle
OCI_NLS_CS_ORA_TO_IANA: Map character set name from Oracle to IANA.
OCI_NLS_LANG_ISO_TO_ORA: Map language name from ISO to Oracle
OCI_NLS_LANG_ORA_TO_ISO: Map language name from Oracle to ISO
OCI_NLS_TERR_ISO_TO_ORA: Map territory name from ISO to Oracle
OCI_NLS_TERR_ORA_TO_ISO: Map territory name from Oracle to ISO
OCI_NLS_TERR_ISO3_TO_ORA: Map territory name from 3-letter ISO abbreviation to Oracle
OCI_NLS_TERR_ORA_TO_ISO3: Map territory name from Oracle to 3-letter ISO abbreviation

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR.

OCI String Manipulation Functions

Two types of data structures are supported for string manipulation:

- multibyte strings
- wide-character strings

Multibyte strings are encoded in native Oracle character sets. Functions that operate on multibyte strings take the string as a whole unit with the length of the string calculated in bytes. wide-character (`wchar`) string functions provide more flexibility in string manipulation. They support character-based and string-based operations with the length the string calculated in characters.

The wide-character datatype is Oracle-specific and should not be confused with the `wchar_t` datatype defined by the ANSI/ISO C standard. The Oracle wide-character datatype is always 4 bytes in all operating systems, while the size of `wchar_t` depends on the implementation and the operating system. The Oracle wide-character datatype normalizes multibyte characters so that they have a fixed width for easy processing. This guarantees no data loss for round trip conversion between the Oracle wide-character set and the native character set.

String manipulation can be classified into the following categories:

- Conversion of strings between multibyte and wide character
- Character classifications
- Case conversion
- Calculations of display length
- General string manipulation, such as comparison, concatenation, and searching

Table 21–5 summarizes the OCI string manipulation functions.

Table 21–5 OCI String Manipulation Functions

Function/Page	Purpose
OCIMultiByteInSizeToWideChar() on page 21-16	Converts part of a multibyte string into the wide-character string.
OCIMultiByteStrCaseConversion() on page 21-17	Converts a multibyte string into the specified case and copies the result into the destination array.
OCIMultiByteStrcat() on page 21-18	Appends a multibyte string to the destination string.
OCIMultiByteStrcmp() on page 21-19	Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods.
OCIMultiByteStrcpy() on page 21-20	Copies a multibyte string into the destination array. It returns the number of bytes copied.
OCIMultiByteStrlen() on page 21-21	Returns the number of bytes in a multibyte string.
OCIMultiByteStrncat() on page 21-22	Appends at most n bytes from a multibyte string to the destination string.
OCIMultiByteStrncmp() on page 21-23	Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. Each string is in the specified length.
OCIMultiByteStrncpy() on page 21-25	Copies a specified number of bytes of a multibyte string into the destination array.

Table 21–5 (Cont.) OCI String Manipulation Functions

Function/Page	Purpose
OCIMultiByteStrnDisplayLength() on page 21-26	Returns the number of display positions occupied by the multibyte string within the range of n bytes.
OCIMultiByteToWideChar() on page 21-27	Converts a NULL-terminated multibyte string into wide-character format.
OCIWideCharInSizeToMultiByte() on page 21-28	Converts part of a wide-character string to the multibyte string.
OCIWideCharMultiByteLength() on page 21-29	Determines the number of bytes required for a wide-character in multibyte encoding.
OCIWideCharStrCaseConversion() on page 21-30	Converts a wide character string into the specified case and copies the result into the destination array.
OCIWideCharStrcat() on page 21-31	Appends a wide-character string to the destination string.
OCIWideCharStrchr() on page 21-32	Searches for the first occurrence of a wide character in a string. It returns a point to the wide character if the search is successful.
OCIWideCharStrcmp() on page 21-33	Compares two wide-character strings by binary, linguistic, or case-insensitive comparison methods.
OCIWideCharStrcpy() on page 21-34	Copies a wide-character string into a destination array. It returns the number of characters copied.
OCIWideCharStrlen() on page 21-35	Returns the number of characters in a wide-character string.
OCIWideCharStrncat() on page 21-36	Appends at most n characters from a wide-character string to the destination string.
OCIWideCharStrncmp() on page 21-37	Compares two wide-character strings by binary, linguistic, or case-insensitive methods. Each string is a specified length.
OCIWideCharStrncpy() on page 21-39	Copies at most n characters of a wide-character string into the destination array.
OCIWideCharStrrchr() on page 21-40	Searches for the last occurrence of a character in a wide-character string.
OCIWideCharToLower() on page 21-41	Converts a specified wide character into the corresponding lowercase character.
OCIWideCharToMultiByte() on page 21-42	Converts a NULL-terminated wide-character string into a multibyte string.
OCIWideCharToUpper() on page 21-43	Converts a specified wide character into the corresponding uppercase character.

OCIMultiByteInSizeToWideChar()

Purpose

Converts part of a multibyte string into the wide-character string.

Syntax

```
sword OCIMultiByteInSizeToWideChar ( dvoid         *hndl,
                                     OCIWchar      *dst,
                                     size_t        dstsz,
                                     CONST OraText *src,
                                     size_t        srcsz,
                                     size_t        *rsize );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set of the string.

dst (OUT)
Pointer to a destination buffer for `wchar`. It can be `NULL` pointer when `dstsz` is zero.

dstsz (IN)
Destination buffer size in number of characters. If it is zero, then this function returns the number of characters needed for the conversion.

src (IN)
Source string to be converted.

srcsz (IN)
Length of source string in bytes.

rsize (OUT)
Number of characters written into destination buffer, or number of characters for converted string if `dstsz` is zero. If it is a `NULL` pointer, then nothing is returned.

Comments

This routine converts part of a multibyte string into the wide-character string. It converts as many complete characters as it can until it reaches the output buffer size limit or input buffer size limit or it reaches a `NULL` terminator in a source string. The output buffer is `NULL`-terminated if space permits. If `dstsz` is zero, then this function returns only the number of characters not including the ending `NULL` terminator needed for a converted string. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`..

Related Functions

OCIMultiByteToWideChar()

OCIMultiByteStrCaseConversion()

Purpose

Converts the multibyte string pointed to by `srcstr` into uppercase or lowercase as specified by the flag and copies the result into the array pointed to by `dststr`.

Syntax

```
size_t OCIMultiByteStrCaseConversion ( dvoid          *hndl,
                                       OraText        *dststr,
                                       CONST OraText  *srcstr,
                                       ub4            flag );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle.

dststr (OUT)
Pointer to destination array. The result string is `NULL`-terminated.

srcstr (IN)
Pointer to source string.

flag (IN)
Specify the case to which to convert:

- `OCI_NLS_UPPERCASE`: Convert to uppercase
- `OCI_NLS_LOWERCASE`: Convert to lowercase

This flag can be used with `OCI_NLS_LINGUISTIC` to specify that the linguistic setting in the locale is used for case conversion.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes for result string, not including the `NULL` terminator.

OCIMultiByteStrcat()

Purpose

Appends a copy of the multibyte string pointed to by `srcstr` to the end of the string pointed to by `dststr`.

Syntax

```
size_t OCIMultiByteStrcat ( dvoid           *hndl,
                            OraText         *dststr,
                            CONST OraText   *srcstr );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

dststr (IN/OUT)
Pointer to the destination multibyte string for appending. The output buffer is `NULL`-terminated.

srcstr (IN)
Pointer to the source string to append.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes in the result string, not including the `NULL` terminator.

Related Functions

OCIMultiByteStrncat()

OCIMultiByteStrcmp()

Purpose

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods.

Syntax

```
int OCIMultiByteStrcmp ( dvoid          *hndl,
                         CONST OraText  *str1,
                         CONST OraText  *str2,
                         int            flag );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle.

str1 (IN)
Pointer to a `NULL`-terminated string.

str2 (IN)
Pointer to a `NULL`-terminated string.

flag (IN)
It is used to decide the comparison method. It can take one of the following values:

- `OCI_NLS_BINARY`: Binary comparison This is the default value.
- `OCI_NLS_LINGUISTIC`: Linguistic comparison specified in the locale

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

- 0, if `str1 = str2`
- Positive, if `str1 > str2`
- Negative, if `str1 < str2`

Related Functions

OCIMultiByteStrncmp()

OCIMultiByteStrcpy()

Purpose

Copies the multibyte string pointed to by `srcstr` into the array pointed to by `dststr`.

Syntax

```
size_t OCIMultiByteStrcpy ( dvoid          *hndl,
                            OraText        *dststr,
                            CONST OraText  *srcstr );
```

Parameters

hndl (IN/OUT)
Pointer to the OCI environment or user session handle.

dststr (OUT)
Pointer to the destination buffer.The output buffer is `NULL`-terminated.

srcstr (IN)
Pointer to the source multibyte string.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes copied, not including the `NULL` terminator.

Related Functions

OCIMultiByteStrncpy()

OCIMultiByteStrlen()

Purpose

Returns the number of bytes in the multibyte string pointed to by `str`, not including the `NULL` terminator.

Syntax

```
size_t OCIMultiByteStrlen ( dvoid          *hndl,
                            CONST OraText  *str );
```

Parameters

hndl (IN/OUT)
Pointer to the OCI environment or user session handle.

str (IN)
Pointer to the source multibyte string.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes, not including the `NULL` terminator.

Related Functions

OCIWideCharStrlen()

OCIMultiByteStrncat()

Purpose

Appends a specified number of bytes from a multibyte string to a destination string.

Syntax

```
size_t OCIMultiByteStrncat ( dvoid           *hndl,
                             OraText         *dststr,
                             CONST OraText   *srcstr,
                             size_t          n );
```

Parameters

hndl (IN/OUT)
Pointer to OCI environment or user session handle.

dststr (IN/OUT)
Pointer to the destination multibyte string for appending.

srcstr (IN)
Pointer to the source multibyte string to append.

n (IN)
The number of bytes from `srcstr` to append.

Comments

This function is similar to `OCIMultiByteStrcat`(). At most `n` bytes from `srcstr` are appended to `dststr`. Note that the `NULL` terminator in `srcstr` stops appending and the function appends as many character as possible within `n` bytes. `dststr` is `NULL`-terminated. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes in the result string, not including the `NULL` terminator

Related Functions

OCIMultiByteStrcat()

OCIMultiByteStrncmp()

Purpose

Compares two multibyte strings by binary, linguistic, or case-insensitive comparison methods. A length is specified for each string.

Syntax

```
int OCIMultiByteStrncmp ( dvoid          *hndl,
                          CONST OraText  *str1,
                          size_t         len1,
                          OraText        *str2,
                          size_t         len2,
                          int            flag );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle.

str1 (IN)
Pointer to the first string.

len1 (IN)
The length of the first string to compare.

str2 (IN)
Pointer to the second string.

len2 (IN)
The length of the second string to compare.

flag(IN)
It is used to decide the comparison method. It can take one of the following values:

- `OCI_NLS_BINARY`: Binary comparison. This is the default value.
- `OCI_NLS_LINGUISTIC`: Linguistic comparison specified in the locale

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

Comments

This function is similar to `OCIMultiByteStrcmp`(), except that at most `len1` bytes from `str1` and `len2` bytes from `str2` are compared. The `NULL` terminator is used in the comparison. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

- 0, if `str1` = `str2`
- Positive, if `str1` > `str2`
- Negative, if `str1` < `str2`

Related Functions

OCIMultiByteStrcpy()

OCIMultiByteStrncpy()

Purpose

Copies a multibyte string into an array.

Syntax

```
size_t OCIMultiByteStrncpy ( dvoid           *hndl,
                             OraText         *dststr,
                             CONST OraText   *srcstr,
                             size_t           n );
```

Parameters

hndl (IN/OUT)
Pointer to OCI environment or user session handle.

srcstr (OUT)
Pointer to the destination buffer.

dststr (IN)
Pointer to the source multibyte string.

n (IN)
The number of bytes from `srcstr` to copy.

Comments

This function is similar to `OCIMultiByteStrcpy`(). At most `n` bytes are copied from the array pointed to by `srcstr` to the array pointed to by `dststr`. Note that the `NULL` terminator in `srcstr` stops copying and the function copies as many characters as possible within `n` bytes. The result string is `NULL`-terminated. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes in the resulting string, not including the NULL terminator.

Related Functions

OCIMultiByteStrcpy()

OCIMultiByteStrnDisplayLength()

Purpose

Returns the number of display positions occupied by the multibyte string within the range of `n` bytes.

Syntax

```
size_t OCIMultiByteStrnDisplayLength ( dvoid           *hndl,
                                       CONST OraText   *str1,
                                       size_t          n );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle.

str (IN)
Pointer to a multibyte string.

n (IN)
The number of bytes to examine.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of display positions.

OCIMultiByteToWideChar()

Purpose

Converts an entire NULL-terminated string into the wide-character string.

Syntax

```
sword OCIMultiByteToWideChar ( dvoid           *hndl,
                               OCIWchar        *dst,
                               CONST OraText   *src,
                               size_t          *rsize );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set of string

dst (OUT)
Destination buffer for wchar.

src (IN)
Source string to be converted.

rsize (OUT)
Number of characters converted including NULL terminator. If it is a NULL pointer, then nothing is returned.

Comments

The wchar output buffer are NULL-terminated. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate() function, then this function produces an error.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR.

Related Functions

OCIWideCharToMultiByte()

OCIWideCharInSizeToMultiByte()

Purpose

Converts part of a wide-character string to multibyte format.

Syntax

```
sword OCIWideCharInSizeToMultiByte ( dvoid           *hndl,
                                     OraText         *dst,
                                     size_t          dstsz,
                                     CONST OCIWchar  *src,
                                     size_t          srcsz,
                                     size_t          *rsize );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set of string.dst (OUT)

Destination buffer for multibyte. It can be a NULL pointer if dstsz is zero.

dstsz (IN)
Destination buffer size in bytes. If it is zero,then the function returns the size in bytes need for converted string.

src (IN)
Source wchar string to be converted.

srcsz (IN)
Length of source string in characters.

rsize (OUT)
Number of bytes written into destination buffer, or number of bytes need to store the converted string if dstsz is zero. If it is a NULL pointer, then nothing is returned.

Comments

Converts part of a wide-character string into the multibyte format. It converts as many complete characters as it can until it reaches the output buffer size or the input buffer size or until it reaches a NULL terminator in the source string. The output buffer is NULL-terminated if space permits. If dstsz is zero, then the function returns the size in bytes not including the NULL terminator needed to store the converted string. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate() function, then this function produces an error.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR.

OCIWideCharMultiByteLength()

Purpose

Determines the number of bytes required for a wide character in multibyte encoding.

Syntax

```
size_t OCIWideCharMultiByteLength ( dvoid     *hndl,
                                    OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` character.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes required for the wide character.

OCIWideCharStrCaseConversion()

Purpose

Converts a wide-character string into a specified case and copies the result into the destination array.

Syntax

```
size_t OCIMultiByteStrCaseConversion ( dvoid             *hndl,
                                       OraText           *dststr,
                                       CONST OraText     *srcstr,
                                       ub4               flag );
```

Parameters

ndl (IN/OUT)
OCI environment or user session handle.

dststr (OUT)
Pointer to destination array.

srcstr (IN)
Pointer to source string.

flag (IN)
Specify the case to which to convert:

- `OCI_NLS_UPPERCASE`: Convert to uppercase.
- `OCI_NLS_LOWERCASE`: Convert to lowercase.

This flag can be used with `OCI_NLS_LINGUISTIC` to specify that the linguistic setting in the locale is used for case conversion.

Comments

Converts the wide-character string pointed to by `srcstr` into uppercase or lowercase as specified by the flag and copies the result into the array pointed to by `dststr`. The result string is `NULL`-terminated. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of bytes for result string, not including the `NULL` terminator.

OCIWideCharStrcat()

Purpose

Appends the wide-character string pointed to by `wsrcstr` to the wide-character string pointed to by `wdststr`.

Syntax

```
size_t OCIWideCharStrcat ( dvoid            *hndl,
                           OCIWchar         *wdststr,
                           CONST OCIWchar   *wsrcstr );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wdststr (IN/OUT)
Pointer to the destination `wchar` string. The output buffer is `NULL`-terminated.

wsrcstr (IN)
Pointer to the source wide-character string to append.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of characters in the result string, not including the `NULL` terminator.

Related Functions

OCIWideCharStrncat()

OCIWideCharStrchr()

Purpose

Searches for the first occurrence of a specified character in a wide-character string.

Syntax

```
OCIWchar *OCIWideCharStrchr ( dvoid            *hndl,
                              CONST OCIWchar   *wstr,
                              OCIWchar         wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wstr (IN)
Pointer to the `wchar` string to search.

wc (IN)
`wchar` to search for.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

A `wchar` pointer if successful, otherwise a `NULL` pointer.

Related Functions

OCIWideCharStrrchr()

OCIWideCharStrcmp()

Purpose

Compares two wide-character strings by binary (based on `wchar` encoding value), linguistic, or case-insensitive comparison methods.

Syntax

```
int OCIWideCharStrcmp ( dvoid            *hndl,
                        CONST OCIWchar   *wstr1,
                        CONST OCIWchar   *wstr2,
                        int              flag );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wstr1 (IN)
Pointer to a `NULL`-terminated `wchar` string.

wstr2 (IN)
Pointer to a `NULL`-terminated `wchar` string.

flag (IN)
Used to decide the comparison method. It can take one of the following values:

- `OCI_NLS_BINARY`: Binary comparison. This is the default value.
- `OCI_NLS_LINGUISTIC`: Linguistic comparison specified in the locale definition.

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

The UNICODE_BINARY sort method cannot be used with `OCIWideCharStrcmp()` to perform a linguistic comparison of supplied wide character arguments.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

- 0, if `wstr1 = wstr2`
- Positive, if `wstr1 > wstr2`
- Negative, if `wstr1 < wstr2`

Related Functions

OCIWideCharStrncmp()

OCIWideCharStrcpy()

Purpose

Copies a wide-character string into an array.

Syntax

```
size_t OCIWideCharStrcpy ( dvoid            *hndl,
                           OCIWchar         *wdststr,
                           CONST OCIWchar   *wsrcstr );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wdststr (OUT)
Pointer to the destination `wchar` buffer.The result string is `NULL`-terminated.

wsrcstr (IN)
Pointer to the source `wchar` string.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of characters copied not including the `NULL` terminator.

Related Functions

OCIWideCharStrncpy()

OCIWideCharStrlen()

Purpose

Returns the number of characters in a wide-character string.

Syntax

```
size_t OCIWideCharStrlen ( dvoid            *hndl,
                           CONST OCIWchar   *wstr );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wstr (IN)
Pointer to the source `wchar` string.

Comments

Returns the number of characters in the `wchar` string pointed to by `wstr`, not including the `NULL` terminator. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

The number of characters not including the `NULL` terminator.

OCIWideCharStrncat()

Purpose

Appends at most n characters from a wide-character string to the destination.

Syntax

```
size_t OCIWideCharStrncat ( dvoid            *hndl,
                            OCIWchar         *wdststr,
                            CONST OCIWchar   *wsrcstr,
                            size_t           n );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wdststr (IN/OUT)
Pointer to the destination wchar string.

wsrcstr (IN)
Pointer to the source wchar string.

n (IN)
Number of characters from wsrcstr to append.

Comments

This function is similar to OCIWideCharStrcat(). At most n characters from wsrcstr are appended to wdststr. Note that the NULL terminator in wsrcstr stops appending. wdststr is NULL-terminated. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate() function, then this function produces an error.

Returns

The number of characters in the result string, not including the NULL terminator.

Related Functions

OCIWideCharStrcat()

OCIWideCharStrncmp()

Purpose

Compares two wide-character strings by binary, linguistic, or case-sensitive methods. Each string has a specified length.

Syntax

```
int OCIWideCharStrncmp ( dvoid              *hndl,
                         CONST OCIWchar     *wstr1,
                         size_t             len1,
                         CONST OCIWchar     *wstr2,
                         size_t             len2,
                         int                flag );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wstr1 (IN)
Pointer to the first `wchar` string.

len1 (IN)
The length from the first string for comparison.

wstr2 (IN)
Pointer to the second `wchar` string.

len2 (IN)
The length from the second string for comparison.

flag (IN)
It is used to decide the comparison method. It can take one of the following values:

- `OCI_NLS_BINARY`: For the binary comparison, this is default value.
- `OCI_NLS_LINGUISTIC`: For the linguistic comparison specified in the locale.

This flag can be used with `OCI_NLS_CASE_INSENSITIVE` for case-insensitive comparison. For example, use `OCI_NLS_LINGUISTIC|OCI_NLS_CASE_INSENSITIVE` to compare strings linguistically without regard to case.

Comments

This function is similar to `OCIWideCharStrcmp`(). It compares two wide-character strings by binary, linguistic, or case-insensitive comparison methods. At most `len1` bytes from `wstr1` and `len2` bytes from `wstr2` are compared. The `NULL` terminator is used in the comparison. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

The UNICODE_BINARY sort method cannot be used with `OCIWideCharStrncmp()` to perform a linguistic comparison of supplied wide character arguments.

Returns

- 0, if `wstr1 = wstr2`

- Positive, if `wstr1 > wstr2`
- Negative, if `wstr1 < wstr2`

Related Functions

OCIWideCharStrcmp()

OCIWideCharStrncpy()

Purpose

Copies at most a n characters from a wide-character string into a destination.

Syntax

```
size_t OCIWideCharStrncpy ( dvoid            *hndl,
                            OCIWchar         *wdststr,
                            CONST OCIWchar   *wsrcstr,
                            size_t           n );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wdststr (OUT)
Pointer to the destination wchar buffer.

wsrcstr (IN)
Pointer to the source wchar string.

n (IN)
Number of characters from wsrcstr to copy.

Comments

This function is similar to OCIWideCharStrcpy(), except that at most n characters are copied from the array pointed to by wsrcstr to the array pointed to by wdststr. Note that the NULL terminator in wdststr stops copying and the result string is NULL-terminated. If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate() function, then this function produces an error.

Returns

The number of characters copied not including the NULL terminator.

Related Functions

OCIWideCharStrcpy()

OCIWideCharStrrchr()

Purpose

Searches for the last occurrence of a character in a wide-character string.

Syntax

```
OCIWchar *OCIWideCharStrrchr ( dvoid             *hndl,
                               CONST OCIWchar    *wstr,
                               OCIWchar          wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wstr (IN)
Pointer to the `wchar` string to search in.

wc (IN)
`wchar` to search for.

Comments

Searches for the last occurrence of `wc` in the `wchar` string pointed to by `wstr`. If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

`wchar` pointer if successful, otherwise a `NULL` pointer.

Related Functions

OCIWideCharStrchr()

OCIWideCharToLower()

Purpose

Converts the wide-character string specified by `wc` into the corresponding lowercase character, if it exists, in the specified locale. If no corresponding lowercase character exists, then it returns `wc` itself.

Syntax

```
OCIWchar OCIWideCharToLower ( dvoid       *hndl,
                              OCIWchar    wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for lowercase conversion.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

A `wchar`.

Related Functions

OCIWideCharToUpper()

OCIWideCharToMultiByte()

Purpose

Converts an entire NULL-terminated wide-character string into a multibyte string.

Syntax

```
sword OCIWideCharToMultiByte ( dvoid            *hndl,
                               OraText          *dst,
                               CONST OCIWchar   *src,
                               size_t           *rsize );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set of string.

dst (OUT)
Destination buffer for multibyte string. The output buffer is NULL-terminated.

src (IN)
Source wchar string to be converted.

srcsz (IN)
Length of source string in characters.

rsize (OUT)
Number of bytes written into destination buffer. If it is a NULL pointer, then nothing is returned.

Comments

If OCI_UTF16ID is specified for SQL CHAR data in the OCIEnvNlsCreate() function, then this function produces an error.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR.

Related Functions

OCIMultiByteToWideChar()

OCIWideCharToUpper()

Purpose

Converts the wide-character string specified by `wc` into the corresponding uppercase character, if it exists, in the specified locale. If no corresponding uppercase character exists, then it returns `wc` itself.

Syntax

```
OCIWchar OCIWideCharToUpper ( dvoid     *hndl,
                              OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for uppercase conversion.

Comments

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

A `wchar`.

Related Functions

OCIWideCharToLower()

OCI Character Classification Functions

Table 21–6 lists the OCI character classification functions.

Table 21–6 OCI Character Classification Functions

Function/Page	Purpose
OCIWideCharIsAlnum() on page 21-45	Tests whether the wide character is a letter or a decimal digit.
OCIWideCharIsAlpha() on page 21-46	Tests whether the wide character is an alphabetic letter.
OCIWideCharIsCntrl() on page 21-47	Tests whether the wide character is a control character.
OCIWideCharIsDigit() on page 21-48	Tests whether the wide character is a decimal digital character.
OCIWideCharIsGraph() on page 21-49	Tests whether the wide character is a graph character.
OCIWideCharIsLower() on page 21-50	Tests whether the wide character is a lowercase character.
OCIWideCharIsPrint() on page 21-51	Tests whether the wide character is a printable character.
OCIWideCharIsPunct() on page 21-52	Tests whether the wide character is a punctuation character.
OCIWideCharIsSingleByte() on page 21-53	Tests whether the wide character is a single-byte character when converted to multibyte.
OCIWideCharIsSpace() on page 21-54	Tests whether the wide character is a space character.
OCIWideCharIsUpper() on page 21-55	Tests whether the wide character is an uppercase character.
OCIWideCharIsXdigit() on page 21-56	Tests whether the wide character is a hexadecimal digit.

OCIWideCharIsAlnum()

Purpose

Tests whether a wide character is a letter or decimal digit.

Syntax

```
boolean OCIWideCharIsAlnum ( dvoid      *hndl,
                             OCIWchar   wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsAlpha()

Purpose

Tests whether a wide character is an alphabetic letter.

Syntax

```
boolean OCIWideCharIsAlpha ( dvoid     *hndl,
                             OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsCntrl()

Purpose

Tests whether a wide character is a control character.

Syntax

```
boolean OCIWideCharIsCntrl ( dvoid       *hndl,
                             OCIWchar    wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsDigit()

Purpose

Tests whether a wide character is a decimal digit character.

Syntax

```
boolean OCIWideCharIsDigit ( dvoid      *hndl,
                             OCIWchar   wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsGraph()

Purpose

Tests whether a wide character is a graph character. A graph character is a character with a visible representation and normally includes alphabetic letters, decimal digits, and punctuation.

Syntax

```
boolean OCIWideCharIsGraph ( dvoid     *hndl,
                             OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsLower()

Purpose

Tests whether a wide character is a lowercase letter.

Syntax

```
boolean OCIWideCharIsLower ( dvoid      *hndl,
                             OCIWchar    wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsPrint()

Purpose

Tests whether a wide character is a printable character.

Syntax

```
boolean OCIWideCharIsPrint ( dvoid     *hndl,
                             OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsPunct()

Purpose

Tests whether a wide character is a punctuation character.

Syntax

```
boolean OCIWideCharIsPunct ( dvoid      *hndl,
                             OCIWchar   wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsSingleByte()

Purpose

Tests whether a wide character is a single-byte character when converted into multibyte.

Syntax

```
boolean OCIWideCharIsSingleByte ( dvoid     *hndl,
                                  OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsSpace()

Purpose

Tests whether a wide character is a space character. A space character causes white space only in displayed text (for example, space, tab, carriage return, new line, vertical tab or form feed).

Syntax

```
boolean OCIWideCharIsSpace ( dvoid    *hndl,
                             OCIWchar wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsUpper()

Purpose

Tests whether a wide character is an uppercase letter

Syntax

```
boolean OCIWideCharIsUpper ( dvoid     *hndl,
                             OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCIWideCharIsXdigit()

Purpose

Tests whether a wide character is a hexadecimal digit (0-9, A-F, a-f)

Syntax

```
boolean OCIWideCharIsXdigit ( dvoid     *hndl,
                              OCIWchar  wc );
```

Parameters

hndl (IN/OUT)
OCI environment or user session handle to determine the character set.

wc (IN)
`wchar` for testing.

Returns

`TRUE` or `FALSE`.

OCI Character Set Conversion Functions

Conversion between Oracle character sets and Unicode (16-bit, fixed-width Unicode encoding) is supported. Replacement characters are used if a character has no mapping from Unicode to the Oracle character set. Therefore, conversion back to the original character set is not always possible without data loss.

Table 21–7 lists the OCI character set conversion functions.

Table 21–7 OCI Character Set Conversion Functions

Function/Page	Purpose
OCICharSetConversionIsReplacementUsed() on page 21-58	Indicates whether replacement characters were used for characters that could not be converted in the last invocation of `OCINlsCharSetConvert()` or `OCICharSetToUnicode()`.
OCICharSetToUnicode() on page 21-59	Converts a multibyte string to Unicode.
OCINlsCharSetConvert() on page 21-60	Converts a string from one character set to another.
OCIUnicodeToCharSet() on page 21-62	Converts a Unicode string into multibyte.

OCICharSetConversionIsReplacementUsed()

Purpose

Indicates whether the replacement character was used for characters that could not be converted during the last invocation of `OCICharSetToUnicode`() or `OCICharSetConvert()`.

Syntax

```
boolean OCICharSetConversionIsReplacementUsed ( dvoid *hndl );
```

Parameters

hndl (IN/OUT)
Pointer to an OCI environment or user session handle.

Comments

Conversion between the Oracle character set and Unicode (16-bit, fixed-width Unicode encoding) is supported. Replacement characters are used if there is no mapping for a character from Unicode to the Oracle character set. Thus, not every character can make a round trip conversion to the original character. Data loss occurs with certain characters.

Returns

The function returns `TRUE` if the replacement character was used when `OCICharSetConvert`() or `OCICharSetToUnicode()` was last invoked. Otherwise the function returns `FALSE`.

OCICharSetToUnicode()

Purpose

Converts a multibyte string pointed to by `src` to Unicode into the array pointed to by `dst`.

Syntax

```
sword OCICharSetToUnicode ( dvoid            *hndl,
                            ub2              *dst,
                            size_t           dstlen,
                            CONST OraText    *src,
                            size_t           srclen,
                            size_t           *rsize );
```

Parameters

hndl (IN/OUT)
Pointer to an OCI environment or user session handle.

dst (OUT)
Pointer to a destination buffer.

dstlen (IN)
The size of the destination buffer in characters.

src (IN)
Pointer to a multibyte source string.

srclen (IN)
The size of the source string in bytes.

rsize (OUT)
The number of characters converted. If it is a `NULL` pointer, then nothing is returned.

Comments

The conversion stops when it reaches the source limitation or destination limitation. The function returns the number of characters converted into a Unicode string. If `dstlen` is `0`, then the function scans the string, counts the number of characters, and returns the number of characters into `rsize`, but does not convert the string.

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`.

OCINlsCharSetConvert()

Purpose

Converts a string pointed to by `src` in the character set specified by `srcid` to the array pointed to by `dst` in the character set specified by `dstid`. The conversion stops when it reaches the data size limitation of either the source or the destination. The function returns the number of bytes converted into the destination buffer.

Syntax

```
sword OCINlsCharSetConvert ( dvoid         *envhp,
                             OCIError      *errhp,
                             ub2           dstid,
                             dvoid         *dstp,
                             size_t        dstlen,
                             ub2           srcid,
                             CONST dvoid   *srcp,
                             size_t        srclen,
                             size_t        *rsize );
```

Parameters

errhp (IN/OUT)
OCI error handle. If there is an error, then it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

dstid (IN)
Character set ID for the destination buffer.

dstp (OUT)
Pointer to the destination buffer.

dstlen (IN)
The maximum size in bytes of the destination buffer.

srcid (IN)
Character set ID for the source buffer.

srcp (IN)
Pointer to the source buffer.

srclen (IN)
The length in bytes of the source buffer.

rsize (OUT)
The number of characters converted. If the pointer is `NULL`, then nothing is returned.

Comments

Although either the source or the destination character set ID can be specified as `OCI_UTF16ID`, the length of the original and the converted data is represented in bytes, rather than number of characters. Note that the conversion does not stop when it encounters null data. To get the character set ID from the character set name, use `OCINlsCharSetNameToId()`. To check if derived data in the destination buffer contains replacement characters, use

OCICharSetConversionIsReplacementUsed(). The buffers should be aligned with the byte boundaries appropriate for the character sets. For example, the ub2 datatype should be used to hold strings in UTF-16.

Returns

OCI_SUCCESS or OCI_ERROR; number of bytes converted

OCIUnicodeToCharSet()

Purpose

Converts a Unicode string to a multibyte string into an array.

Syntax

```
sword OCIUnicodeToCharSet ( dvoid       *hndl,
                            OraText     *dst,
                            size_t      dstlen,
                            CONST ub2   *src,
                            size_t      srclen,
                            size_t      *rsize );
```

Parameters

hndl (IN/OUT)
Pointer to an OCI environment or user session handle.

dst (OUT)
Pointer to a destination buffer.

dstlen (IN)
The size of destination buffer in bytes.

src (IN)
Pointer to a Unicode string.

srclen (IN)
The size of the source string in characters.

rsize (OUT)
The number of bytes converted. If it is a `NULL` pointer, then nothing is returned.

Comments

The conversion stops when it reaches the source limitation or destination limitation. The function returns the number of bytes converted into a multibyte string. If `dstlen` is zero, then the function returns the number of bytes into `rsize` without conversion.

If a Unicode character is not convertible for the character set specified in OCI environment or user session handle, then a replacement character is used. In this case, `OCICharsetConversionIsReplacementUsed`() returns `TRUE`.

If `OCI_UTF16ID` is specified for SQL `CHAR` data in the `OCIEnvNlsCreate()` function, then this function produces an error.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`. or `OCI_ERROR`.

OCI Messaging Functions

The user message API provides a simple interface for cartridge developers to retrieve their own messages as well as Oracle messages.

See Also: *Oracle Database Data Cartridge Developer's Guide*

Table 21–8 lists the OCI messaging functions.

Table 21–8 OCI Messaging Functions

Function/Page	Purpose
OCIMessageClose() on page 21-64	Closes a message handle and frees any memory asosciated with the handle.
OCIMessageGet() on page 21-65	Retrieves a message. If the buffer is not zero, then the function copies the message into the buffer.
OCIMessageOpen() on page 21-66	Opens a message handle in a specified language.

OCIMessageClose()

Purpose

Closes a message handle and frees any memory associated with this handle.

Syntax

```
sword OCIMessageClose ( dvoid     *hndl,
                        OCIError  *errhp,
                        OCIMsg    *msgh );
```

Parameters

hndl (IN/OUT)
Pointer to an OCI environment or user session handle for message language.

errhp (IN/OUT)
The OCI error handle. If there is an error, then it is recorded in `errhp` and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

msgh (IN/OUT)
A pointer to a message handle that was previously opened by `OCIMessageOpen()`.

Returns

`OCI_SUCCESS`, `OCI_INVALID_HANDLE`, or `OCI_ERROR`.

OCIMessageGet()

Purpose

Gets a message with the given message number.

Syntax

```
OraText *OCIMessageGet ( OCIMsg      *msgh,
                         ub4         msgno,
                         OraText     *msgbuf,
                         size_t      buflen );
```

Parameters

msgh (IN/OUT)
Pointer to a message handle which was previously opened by `OCIMessageOpen()`.

msgno (IN)
The message number

msgbuf (OUT)
Pointer to a destination buffer for the retrieved message. If `buflen` is zero, then it can be a `NULL` pointer.

buflen (IN)
The size of the destination buffer.

Comments

If `buflen` is not zero, then the function copies the message into the buffer pointed to by `msgbuf`. If `buflen` is zero, then the message is copied into a message buffer inside the message handle pointed to by `msgh`.

Returns

It returns the pointer to the `NULL`-terminated message string. If the translated message cannot be found, then it tries to return the equivalent English message. If the equivalent English message cannot be found, then it returns a `NULL` pointer.

OCIMessageOpen()

Purpose

Opens a message-handling facility in a specified language.

Syntax

```
sword OCIMessageOpen ( dvoid           *hndl,
                       OCIError        *errhp,
                       OCIMsg          *msghp,
                       CONST OraText   *product,
                       CONST OraText   *facility,
                       OCIDuration     dur );
```

Parameters

hndl (IN/OUT)
Pointer to an OCI environment or user session handle for message language.

errhp (IN/OUT)
The OCI error handle. If there is an error, then it is recorded in `errhp`, and the function returns a `NULL` pointer. Diagnostic information can be obtained by calling `OCIErrorGet()`.

msghp (OUT)
A message handle for return.

product (IN)
A pointer to a product name. The product name is used to locate the directory for messages. Its location depends on the operating system. For example, in Solaris, the directory of message files for the `rdbms` product is `$ORACLE_HOME/rdbms`.

facility (IN)
A pointer to a facility name in the product. It is used to construct a message file name. A message file name follows the conversion with `facility` as prefix. For example, the message file name for the `img` facility in the American language is `imgus.msb`, where `us` is the abbreviation for the American language and `msb` is the message binary file extension.

dur (IN)
The duration for memory allocation for the return message handle. It can have the following values:

- `OCI_DURATION_PROCESS`
- `OCI_DURATION_SESSION`
- `OCI_DURATION_STATEMENT`

Comments

It first tries to open the message file corresponding to `hndl`. If it succeeds, then it uses that file to initialize a message handle. If it cannot find the message file that corresponds to the language, then it looks for a primary language file as a fallback. For example, if the Latin American Spanish file is not found, then it tries to open the Spanish file. If the fallback fails, then it uses the default message file, whose language

is AMERICAN. The function returns a pointer to a message handle into the msghp parameter.

Returns

OCI_SUCCESS, OCI_INVALID_HANDLE, or OCI_ERROR.

22

OCI XML DB Functions

This chapter describes the OCI XML DB functions.

This chapter contains these topics:

Introduction to XML DB Support in OCI

This chapter describes the XML DB functions in detail.

See Also: "OCI Support for XML" on page 13-17

Conventions for OCI Functions

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:

Table 22–1 Mode of a Parameter

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

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

Unless otherwise stated, the function returns:

Table 22–2 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 OCI handle passed to the function is invalid.

See Also: For more information about return codes and error handling, see the section "Error Handling in OCI" on page 2-20

OCI XML DB Functions

This section describes the XML DB functions.

Table 22–3 XML DB Functions

Function/Page	Purpose
OCIXmlDbFreeXmlCtx() on page 22-4	Free an XML context.
OCIXmlDbInitXmlCtx() on page 22-5	Initialize an XML context for XML data from the database.

OCIXmlDbFreeXmlCtx()

Purpose

Free any allocations made by the call to `OCIXmlDbInitXmlCtx()`.

Syntax

```
void OCIXmlDbFreeXmlCtx ( xmlct   *xctx );
```

Parameters

xctx (IN)
The XML context to terminate.

Comments

None.

Related Functions

OCIXmlDbInitXmlCtx()

OCIXmlDbInitXmlCtx()

Purpose

To initialize an XML context for XML data from the database.

Syntax

```
xmlctx *OCIXmlDbInitXmlCtx (   OCIEnv               *envhp,
                               OCISvcCtx            *svchp,
                               OCIError             *errhp,
                               ocixmldbparam        *params,
                               ub4                  num_params );
```

Parameters

envhp (IN)
The OCI environment handle.

svchp (IN)
The OCI service handle.

errhp (IN)
The OCI error handle.

params (IN)
The optional possible values in this array are pointers to:

The OCI duration. Default value is `OCI_DURATION_SESSION`.

An error handler which is a user-registered callback of prototype:

```
void (*err_handler) (sword errcode, (CONST OraText *) errmsg);
```

The two parameters of `err_handler` are:

errcode (OUT)
A numerical error value.

errmsg (OUT)
The error message text.

num_params (IN)
Number of parameters to be read from `params`. If the value of `num_params` exceeds the size of array `params`, unexpected behavior results.

Comments

See Also: "OCI Support for XML" on page 13-17 for a usage example.

Returns

Returns either:

- A pointer to structure `xmlctx`, with error handler and callbacks populated with appropriate values. This is later used for all OCI calls.
- NULL, if no database connection is available.

Related Functions

OCIXmlDbFreeXmlCtx()

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()`.

This appendix contains these topics:

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_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 */OCIDuration`

OCI_ATTR_BIND_DN

Mode
READ/WRITE

Description
The login name (DN) to use when connecting to the LDAP server.

Attribute Datatype
`OraText **/OraText *`

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 13-4 and "Flushing Changes to Server" on page 13-8

Attribute Datatype
`boolean */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. Usually you can set the value at 10%, the default, of the optimal size, `OCI_ATTR_CACHE_OPT_SIZE`. Setting this attribute to 0 results in a value of 10 being used. The object cache uses the maximum and optimal values for freeing unused memory in the object cache.

> **See Also:** "Object Cache Parameters" on page 13-4

Attribute Datatype
`ub4 */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. Setting this attribute to 0 results in a value of 8M bytes being used.

> **See Also:** "Object Cache Parameters" on page 13-4

Attribute Datatype
`ub4 */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_EVTCBK

Mode
WRITE

Description
This attribute registers an event callback function.

See Also: "HA Event Notification" on page 9-36

Attribute Datatype
`OCIEventCallback`

OCI_ATTR_EVTCTX

Mode
WRITE

Description
This attribute registers a context passed to an event callback.

See Also: "HA Event Notification" on page 9-36

Attribute Datatype
dvoid *

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_LDAP_AUTH

Mode
READ/WRITE

Description
The authentication mode. The following are the valid values:

0x0: No authentication; anonymous bind.

0x1: Simple authentication; user name and 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 */ub2`

OCI_ATTR_LDAP_CRED

Mode
READ/WRITE

Description
If the authentication method is "simple authentication" (user name and password authentication), then this attribute holds the password to use when connecting to the LDAP server.

Attribute Datatype
`OraText **/OraText *`

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
`OraText **/OraText *`

OCI_ATTR_LDAP_HOST

Mode
READ/WRITE

Description
The name of the host on which the LDAP server runs.

Attribute Datatype
`OraText **/OraText *`

OCI_ATTR_LDAP_PORT

Mode
READ/WRITE

Description
The port on which the LDAP server is listening.

Attribute Datatype
`ub2 */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 */OCIPinOpt`

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 10-24

Attribute Datatype
`boolean */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 13-11

Attribute Datatype
`boolean */boolean`

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 */OCIDuration`

OCI_ATTR_SHARED_HEAPALLOC

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
`OraText **/OraText *`

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
This attribute 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 (for example, 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 attribute is 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 ** / OCIServer *`

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 **/ OCISession *`

OCI_ATTR_STMTCACHESIZE

Mode
READ/WRITE

Description
The default value of the statement cache size is 20 statements, for a statement cache enabled session. The user can increase or decrease this value, by setting this attribute on the service context handle. This attribute can also be used to enable or disable statement caching for the session, pooled or non-pooled. Statement caching can be enabled by setting the attribute to a nonzero size and disabled by setting it to zero.

Attribute Datatype
`ub4 */ ub4`

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 **` / `OCITrans *`

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
`OraText **` (READ) / `OraText *` (WRITE)

OCI_ATTR_FOCBK

Mode
READ/WRITE

Description

> **See Also:** "Transparent Application Failover Callbacks in OCI" on page 9-30

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
`OraText **` (READ) / `OraText *` (WRITE)

OCI_ATTR_IN_V8_MODE

Mode
READ

Description
Allows you to determine whether an application has switched to Oracle release 7 mode (for example, 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 nonblocking mode. When set, it toggles the nonblocking mode attribute. You must set this attribute only after `OCISessionBegin()` or `OCILogon2()` has been called. Otherwise, an error will be returned.

See Also: "Nonblocking Mode in OCI" on page 2-26

Attribute Datatype
ub1 */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 8-7

Attribute Datatype
`OraText **/OraText *`

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 is 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");
```

OCI_ATTR_TAF_ENABLED

Mode
READ

Description
Set to TRUE if the server handle is TAF-enabled and FALSE if not.

See Also: "Custom Pooling: Tagged Server Handles" on page 9-38

Attribute Datatype
boolean *

OCI_ATTR_USER_MEMORY

Mode
READ

Description
If the handle was allocated with extra memory, this attribute will return a pointer to the user memory. A NULL pointer will be returned for those handles not allocated with extra memory.

See Also: "Custom Pooling: Tagged Server Handles" on page 9-38

Attribute Datatype
dvoid *

Authentication Information Handle

These attributes also apply to the user session handle.

See Also: "User Session Handle Attributes" on page A-12

User Session Handle Attributes

These attributes also apply to the authentication information handle.

OCI_ATTR_ACTION

Mode
WRITE

Description
The name of the current action within the current module. Can be set to `NULL`. When the current action terminates, set this attribute again with the name of the next action or `NULL`, if there is no next action. Can be up to 32 bytes long.

Attribute Datatype
`OraText *`

Example
```
OCIAttrSet(session, OCI_HTYPE_SESSION,(dvoid *)"insert into employees",
          (ub4)strlen("insert into employees"), OCI_ATTR_ACTION, error_handle);
```

OCI_ATTR_APPCTX_ATTR

Mode
WRITE

Description
Specifies an attribute name of the externally initialized context.

Attribute Datatype
`OraText *`

> **See Also:** "Session Handle Attributes Used to Set an Externally Initialized Context" on page 8-17

OCI_ATTR_APPCTX_LIST

Mode
READ

Description
Gets the application context list descriptor for the session.

Attribute Datatype
`OCIParam **`

OCI_ATTR_APPCTX_NAME

Mode
WRITE

> **See Also:** "Session Handle Attributes Used to Set an Externally Initialized Context" on page 8-17

Description
Specifies the namespace of the externally initialized context.

Attribute Datatype
`OraText *`

OCI_ATTR_APPCTX_SIZE

Mode
WRITE

Description
Initializes the externally initialized context array size with the number of attributes.

Attribute Datatype
`ub4`

OCI_ATTR_APPCTX_VALUE

Mode
WRITE

Description
Specifies a value of the externally initialized context.

Attribute Datatype
`OraText *`

> **See Also:** "Session Handle Attributes Used to Set an Externally Initialized Context" on page 8-17

OCI_ATTR_CALL_TIME

Mode
READ

Description
Returns the server-side time for the preceding call in milliseconds.

Attribute Datatype
`ub8 *`

OCI_ATTR_CERTIFICATE

Mode
WRITE

Description
Specifies the certificate of the client for use in proxy authentication. Certificate-based proxy authentication using `OCI_ATTR_CERTIFICATE` will not be supported in future Oracle Database releases. Use `OCI_ATTR_DISTINGUISHED_NAME` or `OCI_ATTR_USERNAME` attribute instead.

Attribute Datatype
`ub1 *`

OCI_ATTR_CLIENT_IDENTIFIER

Mode
WRITE

Description
Specifies the user identifier in the session handle. Can be up to 64 bytes long. It can contain the user name, but you are asked not to include the password for security reasons. The first character of the identifier should not be ':'. If it is, the behavior is unspecified.

Attribute Datatype
`OraText *`

Example
```
OCIAttrSet(session, OCI_HTYPE_SESSION,(dvoid *)"janedoe",
           (ub4)strlen("janedoe"), OCI_ATTR_CLIENT_IDENTIFIER,
           error_handle);
```

OCI_ATTR_CLIENT_INFO

Mode
WRITE

Description
Client application additional information. Can also be set by the DBMS_APPLICATION_INFO package. It is stored in the V$SESSION view. Up to 64 bytes long.

Attribute Datatype
`OraText *`

OCI_ATTR_COLLECT_CALL_TIME

Mode
READ/WRITE

Description
When set to `TRUE`, causes the server to measure call time, in milliseconds, for each subsequent OCI call.

Attribute Datatype
boolean */boolean

OCI_ATTR_CURRENT_SCHEMA

Mode
READ/WRITE

Description
Calling `OCIAttrSet()` with this attribute has the same effect as the SQL command `ALTER SESSION SET CURRENT_SCHEMA`, if the schema name and the session exist. The schema is altered on the next OCI call that does a round trip to the server, avoiding an extra round trip. If the new schema name does not exist, the same error is returned as the error returned from `ALTER SESSION SET CURRENT_SCHEMA`. The new schema name is placed before database objects in DML or DDL commands you then enter.

When a client using this attribute communicates with a server that has a software release earlier than 10*g* Release 2, the `OCIAttrSet()` call will be ignored. This attribute is also readable by `OCIAttrGet()`.

Attribute Datatype
ub4/ub4

Example
```
text schema[] = "hr";
err = OCIAttrSet( (dvoid ) mysessp, OCI_HTYPE_SESSION, (dvoid*)schema,
      (ub4)strlen( (char *)schema), OCI_ATTR_CURRENT_SCHEMA, (OCIError *)myerrhp);
```

OCI_ATTR_DISTINGUISHED_NAME

Mode
WRITE

Description
Specifies distinguished name of the client for use in proxy authentication.

Attribute Datatype
OraText *

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
OraText **

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 8-7

Attribute Datatype
ub1 *

Example
The following code sample shows how this attribute is used:

```
OCIAttrSet ((dvoid *) authp, (ub4)OCI_HTYPE_SESSION, (dvoid *) mig_session,
            (ub4) sz, (ub4)OCI_ATTR_MIGSESSION, errhp);
```

OCI_ATTR_MODULE

Mode
WRITE

Description
The name of the current module running in the client application. When the current module terminates, call with the name of the new module, or NULL if there is no new module. Can be up to 48 bytes long.

Attribute Datatype
OraText *

Example
```
OCIAttrSet(session, OCI_HTYPE_SESSION,(dvoid *)"add_employee",
           (ub4)strlen("add_employee"), OCI_ATTR_MODULE, error_handle);
```

OCI_ATTR_PASSWORD

Mode
WRITE

Description
Specifies a password to use for authentication.

Attribute Datatype
OraText *

OCI_ATTR_PROXY_CLIENT

Mode
WRITE

Description
Specifies the target user name for access through a proxy.

Attribute Datatype
OraText *

OCI_ATTR_PROXY_CREDENTIALS

Mode
WRITE

Description
Specifies that the credentials of the application server are to be used for proxy authentication.

Attribute Datatype
OCISession

OCI_ATTR_USERNAME

Mode
READ/WRITE

Description
Specifies a user name to use for authentication.

Attribute Datatype
OraText **/OraText *

Administration Handle Attributes

OCI_ATTR_ADMIN_PFILE

Mode
READ/WRITE

Description
Set this attribute before a call to `OCIDBStartup()` to specify the location of the client-side parameter file that is used to start up the database. If this attribute is not set then the server-side parameter file is used. If the server-side parameter file does not exist, an error is returned.

Attribute Datatype
`OraText */OraText *`

Connection Pool Handle Attributes

OCI_ATTR_CONN_TIMEOUT

> **Note:** Shrinkage of the pool only occurs when there is a network round trip. If there are no operations, then the connections stay alive.

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 *`

Session Pool Handle Attributes

The attributes used for session pooling are:

OCI_ATTR_SPOOL_BUSY_COUNT

Mode
READ

Description
Returns the number of busy sessions.

Attribute Datatype
ub4 *

OCI_ATTR_SPOOL_GETMODE

Mode
READ/WRITE

Description
This attribute determines the behavior of the session pool when all sessions in the pool are found to be busy and the number of sessions has already reached the maximum. Values are:

- `OCI_SPOOL_ATTRVAL_WAIT` - the thread waits and blocks until a session is freed. This is the default value.
- `OCI_SPOOL_ATTRVAL_NOWAIT` - an error is returned.
- `OCI_SPOOL_ATTRVAL_FORCEGET` - a new session is created even though all the sessions are busy and the maximum number of sessions has been reached. `OCISessionGet()` returns a warning. In this case, if new sessions are created that have exceeded the maximum, `OCISessionGet()` returns a warning.

  Note that if this value is set, it is possible that there can be an attempt to create more sessions than can be supported by the instance of the Oracle database server. In this case, the server will return the following error:

  ```
  ORA 00018 - Maximum number of sessions exceeded
  ```

  In this case, the error will be propagated to the session pool user.

  When read, the appropriate attribute value is returned.

Attribute Datatype
ub1 */ ub1

OCI_ATTR_SPOOL_INCR

Mode
READ

Description
Returns the session increment parameter.

Attribute Datatype
ub4 *

OCI_ATTR_SPOOL_MAX

Mode
READ

Description
Returns the number of maximum sessions.

Attribute Datatype
ub4 *

OCI_ATTR_SPOOL_MIN

Mode
READ

Description
Returns the number of minimum sessions.

Attribute Datatype
ub4 *

OCI_ATTR_SPOOL_OPEN_COUNT

Mode
READ

Description
Returns the number of open sessions.

Attribute Datatype
ub4 *

OCI_ATTR_SPOOL_TIMEOUT

Mode
READ/WRITE

Description
The sessions idle for more than this time (in seconds) are terminated periodically, to maintain an optimum number of open sessions.This attribute can be set dynamically. If this attribute is not set, the least recently used sessions may be timed out if and when space in the pool is required.

Attribute Datatype
ub4 */ ub4

OCI_ATTR_SPOOL_STMTCACHESIZE

Mode
READ/WRITE

Description
Sets the default statement cache size for each of the sessions in a session pool, to this value. The statement cache size for a particular session in the pool can, at any time, be overridden by using `OCI_ATTR_STMTCACHESIZE` on that session.

See Also: "Statement Caching in OCI" on page 9-20

Attribute Datatype
`ub4 */ 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 OraText string can be up to 64 bytes long.

Attribute Datatype
`OraText **` (READ) / `OraText *` (WRITE)

OCI_ATTR_TRANS_TIMEOUT

Mode
READ/WRITE

Description
Can set or read a timeout interval 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_BIND_COUNT

Mode
READ

Description
Returns the number of bind positions on the statement handle.

Attribute Datatype
ub4 *

Example

```
OCIHandleAlloc(env,(void **) &pStatement, OCI_HTYPE_STMT, (size_t)0, (dvoid **)0);
OCIStmtPrepare (pStatement, err, pszQuery, (ub4)strlen(pszQuery),
                (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCIAttrGet(pStatement, OCI_HTYPE_STMT, &iNbParameters, NULL, OCI_ATTR_BIND_COUNT,
           err);
```

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

```
...
int i = 0;
ub4 parmcnt = 0;
```

```
ub2 type = 0;
OCIParam *colhd = (OCIParam *) 0;   /* column handle */

/* Describe of a select-list */
OraText *sqlstmt = (OraText *)"SELECT * FROM employees WHERE employee_id = 100";

checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                    (ub4)strlen((char *)sqlstmt),
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, 1, 0,
        (OCISnapshot *)0, (OCISnapshot *)0, OCI_DESCRIBE_ONLY));

/* Get the number of columns in the select list */
checkerr(errhp, OCIAttrGet((dvoid *)stmthp, OCI_HTYPE_STMT, (dvoid *)&parmcnt,
                     (ub4 *)0, OCI_ATTR_PARAM_COUNT, errhp));

/* go through the column list and retrieve the datatype of each column. We
   start from pos = 1 */
for (i = 1; i <= parmcnt; i++)
{
  /* get parameter for column i */
  checkerr(errhp, OCIParamGet((dvoid *)stmthp, OCI_HTYPE_STMT, errhp,
           (dvoid **)&colhd, i));

  /* get data-type of column i */
  type = 0;
  checkerr(errhp, OCIAttrGet((dvoid *)colhd, OCI_DTYPE_PARAM,
          (dvoid *)&type, (ub4 *)0, OCI_ATTR_DATA_TYPE, errhp));

}
...
```

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 after SELECT statements. For INSERT, UPDATE, and DELETE statements, it is the number of rows processed by the most recent statement. 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 2-25 and "DATE" on page 3-11

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.

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

Table A–1 (Cont.) SQL Command Codes

Code	SQL Function	Code	SQL Function	Code	SQL Function
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
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	173	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
`OraText *`

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 OCI Binding" on page 5-25

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_UC2SID, 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 preceding OCIBind call has to be stated in bytes.

If OCI_ATTR_CHARSET_FORM is set, then OCI_ATTR_CHARSET_ID should be set only afterward. Setting OCI_ATTR_CHARSET_ID prior to setting OCI_ATTR_CHARSET_FORM will cause unexpected results.

> **See Also:** "Character Conversion in OCI Binding and Defining" on page 5-22

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 5-24

Attribute Datatype
sb4 *

OCI_ATTR_MAXDATA_SIZE

Mode
READ/WRITE

Description

> **See Also:** "Using OCI_ATTR_MAXDATA_SIZE Attribute" on page 5-24

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 preceding OCIDefine call has to be stated in bytes.

If OCI_ATTR_CHARSET_FORM is set, then OCI_ATTR_CHARSET_ID should be set only afterward. Setting OCI_ATTR_CHARSET_ID prior to setting OCI_ATTR_CHARSET_FORM will cause unexpected results.

See Also: "Character Conversion in OCI Binding and Defining" on page 5-22

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 5-24

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 (that is, `BLOB`, `CLOB`, `NCLOB`).

Applications should pass address of a `ub4` which has a value of 0; for example, declare:

```
ub4 lobEmpty = 0
```

then pass address `&lobEmpty`.

Attribute Datatype
`ub4 *`

Complex Object Attributes

See Also: "Complex Object Retrieval" on page 10-15

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 `REF`s of type `OCI_ATTR_COMPLEXOBJECTCOMP_TYPE`.

Attribute Datatype
`ub4 *`

Streams Advanced Queuing Descriptor Attributes

See Also: *Oracle Streams Advanced Queuing User's Guide and Reference*

OCIAQEnqOptions Descriptor Attributes

The following attributes are properties of the OCIAQEnqOptions descriptor:

OCI_ATTR_MSG_DELIVERY_MODE

Mode
WRITE

Description
The enqueue call can enqueue a persistent or a buffered message into a queue, by setting the OCI_MSG_DELIVERY_MODE attribute in the OCIAQEnqOptions descriptor to OCI_MSG_PERSISTENT or OCI_MSG_BUFFERED, respectively. The default value for this attribute is OCI_MSG_PERSISTENT.

Attribute Datatype
ub2

OCI_ATTR_RELATIVE_MSGID

Mode
READ/WRITE

Description
This feature is deprecated and may be removed in a future release.

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
This feature is deprecatedfor new applications, but it is retained for compatibility.

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
`OraText *`

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
`OraText *`

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_REMOVE_NODATA - 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_MSG_DELIVERY_MODE

Mode
WRITE

Description
You can specify the dequeue call to dequeue persistent, buffered, or both kinds of messages from a queue, by setting the OCI_MSG_DELIVERY_MODE attribute in the OCIAQDeqOptions descriptor to OCI_MSG_PERSISTENT, OCI_MSG_BUFFERED, or OCI_MSG_PERSISTENT_OR_BUFFERED, respectively. The default value for this attribute is OCI_MSG_PERSISTENT.

Attribute Datatype
ub2

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_DEQ_FIRST_MSG_ONE_GROUP - indicates that a call to OCIAQDeqArray() will reset the position to the beginning of the queue and dequeue messages from a single transaction group that are available and match the search criteria. If the number of messages in the single transaction group exceeds iters, then you must make a subsequent call to OCIAQDeqArray() using the OCI_DEQ_NEXT_MSG_ONE_GROUP navigation.
- OCI_DEQ_NEXT_MSG_ONE_GROUP - indicates that a call to OCIAQDeqArray() will dequeue the next set of messages (up to iters) that are available, match the search criteria and belong to the message group.
- OCI_DEQ_FIRST_MSG_MULTI_GROUP - indicates that a call to OCIAQDeqArray() will reset the position to the beginning of the queue and dequeue messages (possibly across different transaction groups) that are available and match the search criteria, until reaching the iters limit. To distinguish between transaction groups, a new message property, OCI_ATTR_TRANSACTION_NO, will be defined. All messages belonging to the same transaction group will have the same value for this message property.
- OCI_DEQ_NEXT_MSG_MULTI_GROUP - indicates that a call to OCIAQDeqArray() will dequeue the next set of messages (possibly across different transaction groups) that are available and match the search criteria, until reaching the iters limit. To distinguish between transaction groups, a new message property, OCI_ATTR_TRANSACTION_NO, will be defined. All messages belonging to the same transaction group will have the same value for this message property.

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.

Note: If the OCI_DEQ_NO_WAIT option is used to poll a queue, then messages are not dequeued after polling an empty queue. Use the OCI_DEQ_FIRST_MSG option instead of the default OCI_DEQ_NEXT_MSG setting of OCI_ATTR_NAVIGATION. You can also use a nonzero wait setting (1 is suggested) of OCI_ATTR_WAIT for the dequeue.

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
`OraText *`

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
`OraText *`

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_DELIVERY_MODE

Mode
READ

Description
After a dequeue call, the OCI client can read the `OCI_MSG_DELIVERY_MODE` attribute in the `OCIAQMsgProperties` descriptor to determine whether a persistent or buffered message was dequeued. The value of the attribute is `OCI_MSG_PERSISTENT` for persistent messages and `OCI_MSG_BUFFERED` for buffered messages.

Attribute Datatype
`ub2`

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_TRANSACTION_NO

Mode
READ

Description
For transaction-grouped queues, this identifies the transaction group of the message. This attribute is populated after a successful `OCIAQDeqArray()` call. All messages in a group have the same value for this attribute. This attribute cannot be used by the `OCIAQEnqArray()` call to set the transaction group for an enqueued message.

Attribute Datatype
`OraText *`

OCI_ATTR_ORIGINAL_MSGID

Mode
READ/WRITE

Description
The ID of the message in 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 ID of the message in the last queue that generated this message, 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
`OraText *`

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
`OraText *`

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
`OraText **/OraText *`

Subscription Handle Attributes

See Also: "Publish-Subscribe Notification in OCI" on page 9-47

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
`ub4 (dvoid *, OCISubscription *, dvoid *, ub4, dvoid *, ub4)`

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 `OCI Subscription Register()`.

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
`OraText *`

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`, `OCI_SUBSCR_NAMESPACE_DBCHANGE`, 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_PORTNO

Mode
READ/WRITE

Description
Port number on the server, set on the environment handle. The port number is sent to clients by `OCISessionBegin()`.

Attribute Datatype
`ub4 *`

OCI_ATTR_SUBSCR_QOSFLAGS

Mode
READ/WRITE

Description
Quality of service levels of the server. The possible settings are:

1. `OCI_SUBSCR_QOS_RELIABLE` - Reliable. If database crashes, do not lose notification. Not supported for nonpersistent queues or buffered messaging.
2. `OCI_SUBSCR_QOS_PURGE_ON_NTFN` - Once received, purge notification and remove subscription.

Attribute Datatype
`ub4 *`

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
OraText *

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 *

OCI_ATTR_SUBSCR_TIMEOUT

Mode
READ/WRITE

Description
Registration timeout interval in seconds.

Attribute Datatype
ub4 *

Change Notification Attributes

OCI_ATTR_CHNF_CHANGELAG

Mode
WRITE

Description
The number of transactions that the client is to lag in change notifications.

Attribute Datatype
ub4 *

OCI_ATTR_CHNF_OPERATIONS

Mode
WRITE

Description
Used to filter notifications based on operation type.

Attribute Datatype
`ub4 *`

> **See Also:** "Database Change Notification" on page 9-61 for details about the flag values

OCI_ATTR_CHNF_ROWIDS

Mode
WRITE

Description
If `TRUE`, the change notification message includes row level details such as operation type and `ROWID`. The default is `FALSE`.

Attribute Datatype
`boolean *`

OCI_ATTR_CHNF_TABLENAMES

Mode
READ

Description
Attributes provided to retrieve the list of table names that were registered. These attributes are available from the subscription handle, after the query is executed.

Attribute Datatype
`OCIColl **`

Change Notification Descriptor Attributes

OCI_ATTR_CHDES_DBNAME

Mode
READ

Description
Name of the database.

Attribute Datatype
`OraText **`

OCI_ATTR_CHDES_NFTYPE

Mode
READ

Description
Flags describing the notification type.

Attribute Datatype
`ub4 *`

> **See Also:** "Change Notification Descriptor" on page 9-63 for the flag values

OCI_ATTR_CHDES_ROW_OPFLAGS

Mode
READ

Description
Operation type: `INSERT`, `UPDATE`, `DELETE`, or `OTHER`.

Attribute Datatype
`ub4 *`

OCI_ATTR_CHDES_ROW_ROWID

Mode
READ

Description
String representation of a ROWID.

Attribute Datatype
`OraText **`

OCI_ATTR_CHDES_TABLE_CHANGES

Mode
READ

Description
A collection type describing operations on tables. Each element of the collection is a table change descriptor (`OCITableChangeDesc *`) of type `OCI_DTYPE_TABLE_CHDES` which has the attributes that begin with `OCI_ATTR_CHDES_TABLE`. See the following entries.

Attribute Datatype
`OCIColl **`

OCI_ATTR_CHDES_TABLE_NAME

Mode
READ

Description
Schema and tablename. `HR.EMPLOYEES`, for example.

Attribute Datatype
`OraText **`

OCI_ATTR_CHDES_TABLE_OPFLAGS

Mode
READ

Description
Flags describing the operations on the table.

Attribute Datatype
`ub4 *`

See Also: "Change Notification Descriptor" on page 9-63 for the flag values

OCI_ATTR_CHDES_TABLE_ROW_CHANGES

Mode
READ

Description
An embedded collection describing the changes to the rows of the table. Each element of the collection is a row change descriptor (`OCIRowChangeDesc *`) of type `OCI_DTYPE_ROW_CHDES` which has the attributes `OCI_ATTR_CHDES_ROW_OPFLAGS` and `OCI_ATTR_CHDES_ROW_ROWID`.

Attribute Datatype
`OCIColl **`

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 12-1 and"Direct Path Loading of Object Types" on page 12-12

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
`OraText **/OraText *`

OCI_ATTR_DIRPATH_DCACHE_DISABLE

Mode
READ/WRITE

Description
Setting this attribute to 1 indicates that the date cache will be disabled if exceeded. The default value is 0, which means that lookups in the cache will continue on cache overflow.

> **See Also:** "Using a Date Cache in Direct Path Loading of Dates in OCI" on page 12-11 for a complete description of this attribute and of the four following attributes.

Attribute Datatype
`ub1 */ub1 *`

OCI_ATTR_DIRPATH_DCACHE_HITS

Mode
READ

Description
Queries the number of date cache hits.

Attribute Datatype
`ub4 *`

OCI_ATTR_DIRPATH_DCACHE_MISSES

Mode
READ

Description
Queries the current number of date cache misses.

Attribute Datatype
`ub4 *`

OCI_ATTR_DIRPATH_DCACHE_NUM

Mode
READ

Description
Queries the current number of entries in a date cache.

Attribute Datatype
ub4 *

OCI_ATTR_DIRPATH_DCACHE_SIZE

Mode
READ/WRITE

Description
Sets the date cache size (in elements) for a table. To disable the date cache, set to 0, which is the default value.

Attribute Datatype
ub4 */ub4 *

OCI_ATTR_DIRPATH_INDEX_MAINT_METHOD

Mode
READ/WRITE

Description
Performs index row insertion on a per-row basis.

Valid value is:

OCI_DIRPATH_INDEX_MAINT_SINGLE_ROW

Attribute Datatype
ub1 */ub1 *

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:

```
OraText *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
OraText **/OraText *

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_DIRPATH_SKIPINDEX_METHOD

Mode
READ/WRITE

Description
Indicates how the handling of unusable indexes will be performed.

Valid values are:

- OCI_DIRPATH_INDEX_MAINT_SKIP_UNUSABLE (skip unusable indexes)
- OCI_DIRPATH_INDEX_MAINT_DONT_SKIP_UNUSABLE (do not skip unusable indexes)
- OCI_DIRPATH_INDEX_MAINT_SKIP_ALL (skip all index maintenance)

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-59

Attribute Datatype
`OCIParam* *`

OCI_ATTR_NAME

Mode
READ/WRITE

Description
Name of the table to be loaded into.

Attribute Datatype
`OraText**/OraText *`

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
`OraText **/OraText *`

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
`OraText **/OraText *`

Direct Path Function Context Handle (OCIDirPathFuncCtx) Attributes

For further explanations of these attributes:

> **See Also:** "Direct Path Function Context and Attributes" on page 12-26

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-59

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
`OraText **/OraText *`

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 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 successfully converted in the last call to `OCIDirPathColArrayToStream()`.

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 successfully loaded by the last `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 12-7

```
...
  /* 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
`ub4 */ub4 *`

OCI_ATTR_DATA_TYPE

Mode
READ/WRITE

Description
Returns or sets the external datatype of the column. Valid datatypes are:

- SQLT_CHR
- SQLT_DATE
- SQLT_TIMESTAMP
- SQLT_TIMESTAMP_TZ
- SQLT_TIMESTAMP_LTZ
- SQLT_INTERVAL_YM
- SQLT_INTERVAL_DS
- SQLT_INT

- SQLT_UIN
- SQLT_FLT
- SQLT_PDN
- SQLT_BIN
- SQLT_NUM
- SQLT_NTY
- SQLT_REF
- SQLT_VST
- SQLT_VNU

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
OraText **/OraText *

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. Intialize both the column name and the column name length to 0 before calling `OCIAttrGet()`.

Attribute Datatype
`OraText **/OraText *`

OCI_ATTR_PRECISION

Mode
READ/WRITE

Description
Returns or sets the precision.

Attribute Datatype
`ub1 */ub1 *` for explicit describes

`sb2 */sb2 *` for implicit describes

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 datatypes.

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: "OCI_ATTR_SHARED_HEAPALLOC" on page A-7

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
`OraText *`

OCI_ATTR_MEMPOOL_HOMENAME

Mode
READ/WRITE

Description
Directory name where the executables that use the same shared subsystem instance are located.

Attribute Datatype
`OraText *`

OCI_ATTR_MEMPOOL_INSTNAME

Mode
READ/WRITE

Description
Any user-defined name to identify an instance of the shared subsystem.

Attribute Datatype
`OraText *`

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 if 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 *`

Event Handle Attributes

The `OCIEvent` handle encapsulates the attributes from the event payload. This handle is implicitly allocated prior to calling the event callback.

See Also: "HA Event Notification" on page 9-36

The event callback obtains the attributes of an event using `OCIAttrGet()` with the following attributes:

OCI_ATTR_DBDOMAIN

Mode
READ

Description
When called with this attribute, `OCIAttrGet()` retrieves the name of the database domain that has been affected by this event.

Attribute Datatype
`OraText **`

OCI_ATTR_DBNAME

Mode
READ

Description
When called with this attribute, `OCIAttrGet()` retrieves the name of the database that has been affected by this event.

Attribute Datatype
`OraText **`

OCI_ATTR_EVENTTYPE

Mode
READ

Description
The type of event that occurred, OCI_EVENTTYPE_HA.

Attribute Datatype
ub4 *

OCI_ATTR_HA_SOURCE

Mode
READ

Description
If the event type is OCI_EVENTTYPE_HA, get the source of the event with this attribute. Valid values are OCI_HA_SOURCE_DATABASE, OCI_HA_SOURCE_NODE, OCI_HA_SOURCE_INSTANCE, OCI_HA_SOURCE_SERVICE, OCI_HA_SOURCE_SERVICE_MEMBER, OCI_HA_SOURCE_ASM_INSTANCE, OCI_HA_SOURCE_SERVICE_PRECONNECT.

Attribute Datatype
ub4 *

OCI_ATTR_HA_SRVFIRST

Mode
READ

Description
When called with this attribute, OCIAttrGet() retrieves the first server handle in the list of server handles affected by a Real Application Clusters (RAC) HA DOWN event.

Attribute Datatype
OCIServer **

OCI_ATTR_HA_SRVNEXT

Mode
READ

Description
When called with this attribute OCIAttrGet() retrieves the next server handle in the list of server handles affected by a Real Application Clusters (RAC) HA DOWN event.

Attribute Datatype
OCIServer **

OCI_ATTR_HA_STATUS

Mode
READ

Description
Valid value is OCI_HA_STATUS_DOWN. Only DOWN events are suscribed to currently.

Attribute Datatype
ub4 *

OCI_ATTR_HA_TIMESTAMP

Mode
READ

Description
The time that the HA event occurred.

Attribute Datatype
`OCIDateTime **`

OCI_ATTR_HOSTNAME

Mode
READ

Description
When called with this attribute, `OCIAttrGet()` retrieves the name of the host that has been affected by this event.

Attribute Datatype
`OraText **`

OCI_ATTR_INSTNAME

Mode
READ

Description
When called with this attribute, `OCIAttrGet()` retrieves the name of the instance that has been affected by this event.

Attribute Datatype
`OraText **`

OCI_ATTR_SERVICENAME

Mode
READ

Description
When called with this attribute, `OCIAttrGet()` retrieves the name of the service that has been affected by this event.

Attribute Datatype
`OraText **`

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 operating systems.

The demonstration programs are available with your Oracle installation. The location, names, and availability of the programs may vary on different operating systems. On a UNIX workstation, the programs are installed in the `$ORACLE_HOME/rdbms/demo` directory.

OCI header files that are required for OCI client application development on UNIX platforms are in the `$ORACLE_HOME/rdbms/public` directory. The `demo_rdbms.mk` file is in the `$ORACLE_HOME/rdbms/demo` directory and serves as an example makefile, starting with the 10.2 release. On Windows systems, `make.bat` is the analogous file in the samples directory.

Unless you significantly modify the `demo_rdbms.mk` file, you are not affected. This is because the `demo_rdbms.mk` file already includes the `$ORACLE_HOME/rdbms/public` directory. Ensure that your highly customized makefiles have the `$ORACLE_HOME/rdbms/public` directory in the `INCLUDE` path.

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.

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 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 functionality.
`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 `REF`s.
`cdemodsa.c`	Describing information about a table.
`cdemodsc.c`	Describing information about an object type.

Table B–1 (Cont.) OCI Demonstration Programs

Program Name	Features Illustrated
cdemofo.c	Registering and operating application failover callbacks.
cdemolb.c	Create and insert LOB data and then read, write, copy, append and trim the data.
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.
cdemocoll.c	Insert and select of nested table and varray.
cdemorid.c	Using INSERT, UPDATE, DELETE statements and fetches to get multiple rowids in one round trip.
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 *Oracle Streams Advanced Queuing User's Guide and Reference*).
ociaqdemo00.c	Streams Advanced Queuing. Enqueues 100 messages.
ociaqdemo01.c	Dequeues messages by blocking.
ociaqdemo02.c	Listens for multiple agents.
ociaqarrayenq.c	Array enqueue of 10 messages.
ociaqarraydeq.c	Array dequeue of 10 messages.
cdemodp.c, cdemodp_lip.c	Loading data with the direct path load functions.
cdemdpco.c	Loading a column object with the direct path load functions.
cdemdpno.c	Loading a nested column object with the direct path load functions.
cdemdpin.c	Loading derived type (inheritance) - direct path.
cdemdpit.c	Loading an object table with inheritance - direct path.
cdemdpro.c	Loading a reference with the direct path load functions.
cdemdpss.c	Loading SQL strings with the direct path load functions.
cdemoucb.c, cdemoucbl.c	Using static and dynamic user callbacks.
cdemoupk.c, cdemoup1.c, cdemoup2.c	Using dynamic user callbacks with multiple packages.
cdemodt.c	Datetime and interval example. Demonstrates IN and OUT binds with PL/SQL procedure or function.
cdemosc.c	Scrollable cursor.
cdemol2l.c	Accesses LOBs using the LONG API (Data Interface).
cdemoin1.c	Inheritance demo which modifies an inherited type in a table and displays a record from the table.
cdemoin2.c	Inheritance demo to do attribute substitutability.
cdemoin3.c	Inheritance demo that describes an object, inherited types, object tables, and a sub-table.

Table B–1 (Cont.) OCI Demonstration Programs

Program Name	Features Illustrated
`cdemoanydata1.c`	Anydata demo. Inserts and selects rows to and from anydata table.
`cdemoanydata2.c`	Anydata demo. Creates a type piecewise using `OCITypeBeginCreate()` and then describes the new type created.
`cdemosp.c`	Session pooling.
`cdemocp.c`	Connection pooling.
`cdemocpproxy.c`	Connection pooling with proxy functionality.
`cdemostc.c`	Statement caching.
`cdemouni.c`	Program for OCI UTF16 API.
`nchdemo1.c`	Shows nchar implicit conversion feature and codepoint feature.

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.

This appendix contains these topics:

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	Number of Server Round Trips
`OCIBreak()`	1
`OCIDBShutdown()`	1
`OCIDBStartup()`	1
`OCIEnvCreate()`	0
`OCIEnvInit()`	0
`OCIErrorGet()`	0
`OCIInitialize()`	0

Table C–1 (Cont.) Server Round Trips for Relational Operations

Function	Number of Server Round Trips
`OCILdaToSvcCtx()`	0
`OCILogoff()`	1
`OCILogon()`	1
`OCILogon2()`	connection pool/session pool: same as OCISessionGet(). Normal: 2 (depends on authentication and TAF situation.
`OCIPasswordChange()`	1
`OCIPing()`	1
`OCIReset()`	0
`OCIServerAttach()`	1
`OCIServerDetach()`	1
`OCIServerVersion()`	1
`OCISessionBegin()`	1
`OCISessionEnd()`	1
`OCISessionGet()`	session pool: 0 - increment of logins. connection pool: 1 to (1+ (increment * logins)). Depends on cache hit: one for the user session, optional increment for primary sessions. Normal: 1 login
`OCISessionPoolCreate()`	sessMin * logins
`OCISessionPoolDestroy()`	sessions in cache * logoffs
`OCISessionRelease()`	session pooling: 0, except when explicit session destroys flag set. Normal: 1 login
`OCISvcCtxToLda()`	0
`OCIStmtExecute()`	1
`OCIStmtFetch()`	0 or 1
`OCIStmtFetch2()`	0 in pre-fetch, otherwise 1
`OCIStmtGetPieceInfo()`	1
`OCIStmtPrepare()`	0
`OCIStmtSetPieceInfo()`	0
`OCITerminate()`	1
`OCITransCommit()`	1
`OCITransDetach()`	1
`OCITransForget()`	1
`OCITransPrepare()`	1
`OCItransRollback()`	1
`OCITransStart()`	1

Table C–1 (Cont.) Server Round Trips for Relational Operations

Function	Number of Server Round Trips
OCIUserCallbackGet()	0
OCIUserCallbackregister()	0

LOB Function Round Trips

Table C–2 lists the server round trips incurred by the OCILob calls.

Note: To minimize the number of round trips, you can use the data interface for LOBs. You can bind or define character data for a CLOB column or RAW data for a BLOB column.

See Also:

- "Binding LOB Data" on page 5-8 for usage and examples for both INSERT and UPDATE statements
- "Defining LOB Data" on page 5-15 for usage and examples of SELECT statements

For calls whose number of round trips is "0 or 1", if LOB buffering is on, and the request can be satisfied in the client, no round trips are incurred.

Table C–2 Server Round Trips for OCILob Calls

Function	Number of Server Round Trips
OCILobAppend()	1
OCILobArrayRead()	1
OCILobArraywrite()	1
OCILobAssign()	0
OCILobCharSetForm()	0
OCILobCharSetId()	0
OCILobClose()	1
OCILobCopy()	1
OCILobCopy2()	1
OCILobCreateTemporary()	1
OCILobDisableBuffering()	0
OCILobEnableBuffering()	0
OCILobErase()	1
OCILobErase2()	1
OCILobFileClose()	1
OCILobFileCloseAll()	1
OCILobFileExists()	1
OCILobFileGetName()	0
OCILobFileIsOpen()	1

Table C–2 (Cont.) Server Round Trips for OCILob Calls

Function	Number of Server Round Trips
OCILobFileOpen()	1
OCILobFileSetName()	0
OCILobFlushBuffer()	1 for each modified page in the buffer for this LOB
OCILobFreeTemporary()	1
OCILobGetChunkSize()	1
OCILobGetLength()	1
OCILobGetLength2()	1
OCILobGetStorageLimit()	1
OCILobIsEqual()	0
OCILobIsOpen()	1
OCILobIsTemporary()	0
OCILobLoadFromFile()	1
OCILobLoadFromFile2()	1
OCILobLocatorAssign()	1 round trip if either the source or the destination locator refers to a temporary LOB
OCILobLocatorIsInit()	0
OCILobOpen()	1
OCILobRead()	0 or 1
OCILobRead2()	0 or 1
OCILobTrim()	1
OCILobTrim2()	1
OCILobWrite()	0 or 1
OCILobWrite2()	0 or 1
OCILobWriteAppend()	0 or 1
OCILobWriteAppend2()	0 or 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	Number of Server Round Trips
OCIObjectNew()	0
OCIObjectPin()	1; 0 if the desired object is already in cache
OCIObjectUnpin()	0
OCIObjectPinCountReset()	0
OCIObjectLock()	1
OCIObjectMarkUpdate()	0

Table C–3 (Cont.) Server Round Trips for Object and Cache Functions

Function	Number of Server Round Trips
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	Number 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	Number 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
`OCIIter*()`	0; 1 if the collection is not loaded in the cache

Any Type and Data Function Round Trips

The number of server round trips required by Any Type and Data functions are listed in Table C–6. The functions not listed do not generate any round trips.

Table C–6 Server Round Trips for Any Type and Data Functions

Function	Number of Server Round Trips
`OCIAnyDataAttrGet()`	0; 1 if the type information is not loaded in the cache
`OCIAnyDataAttrSet()`	0; 1 if the type information is not loaded in the cache
`OCIAnyDataCollGetElem()`	0; 1 if the type information is not loaded in the cache

Other Local Functions

The functions listed in Table C–7 are local and do not require a server round trip:

Table C–7 Locally Processed Functions

Local Function Name	Notes
`OCIAttrGet()`	When describing an object type, this call does make one round trip to fetch the type descriptor object.
`OCIAttrSet()`	
`OCIBindByName()`	
`OCIBindByPos()`	
`OCIBindDynamic()`	
`OCIBindObject()`	
`OCIBindArrayOfStruct()`	
`OCIDefineByPos()`	

Table C–7 (Cont.) Locally Processed Functions

Local Function Name	Notes
OCIDefineDynamic()	
OCIDefineArrayOfStruct()	
OCIDefineObject()	
OCIDescriptorAlloc()	
OCIDescriptorFree()	
OCIEnvInit()	
OCIEnvCreate()	
OCIErrorGet()	
OCIHandleAlloc()	
OCIHandleFree()	
OCILdaToSvcCtx()	
OCISvcCtxToLda()	
OCIStmtGetBindInfo()	
OCIStmtPrepare()	
OCIStmtRelease()	0
OCIStmtGetBindInfo()	
OCIStmtPrepare2()	0

D

Getting Started with OCI for Windows

This appendix describes only the features of OCI that apply to the Windows 2003, Windows 2000, and Windows XP operating systems. Windows NT is no longer supported.

This chapter contains these topics:

What Is Included in the OCI Package for Windows?

The Oracle Call Interface for Windows package includes:

- Oracle Call Interface (OCI)
- Required Support Files (RSFs)
- Oracle Universal Installer
- Header files for compiling OCI applications
- Library files for linking OCI applications
- Sample programs for demonstrating how to build OCI applications

The OCI for Windows package includes the additional libraries required for linking your OCI programs.

> **See Also:** "OCI Instant Client" on page 1-16 for a simplified OCI installation option.

Oracle Directory Structure for Windows

When you install OCI, Oracle Universal Installer creates an *ORACLE_BASE*`\`*ORACLE_HOME* directory on the hard drive of your computer. The default Oracle home directory is `C:\oracle\ora10`.

The OCI files are located in the *ORACLE_BASE**ORACLE_HOME* directory, as are the library files needed to link and run OCI applications, and link with other Oracle for Windows products, such as Oracle Forms.

The *ORACLE_BASE**ORACLE_HOME* directory contains the following directories that are relevant to OCI:

Table D–1 ORACLE_HOME Directories and Contents

Directory Name	Contents
\bin	Executable and help files
\oci	Oracle Call Interface directory for Windows files
\oci\include	Header files, such as oci.h and ociap.h
\oci\samples	Sample programs
\precomp\admin\ottcfg.cfg	Object Type Translator utility and default configuration file

Sample OCI Programs for Windows

When OCI is installed, a set of sample programs and their corresponding project files are copied to the *ORACLE_BASE**ORACLE_HOME*\oci\samples subdirectory. Oracle recommends that you build and run these sample programs to verify that OCI has been successfully installed and to familiarize yourself with the steps involved in developing OCI applications.

To build a sample, run a batch file (make.bat) at the command prompt. For example, to build the cdemo1.c sample, enter the following command in the directory samples:

```
C:> make cdemo1
```

After you finish using these sample programs, you can delete them if you choose.

ociucb.c should be compiled using ociucb.bat. This batch file creates a DLL and places it in the *ORACLE_BASE**ORACLE_HOME*\bin directory. To load user callback functions, set the environment registry variable ORA_OCI_UCBPKG to OCIUCB.

Compiling OCI Applications for Windows

When you compile an OCI application, you must include the appropriate OCI header files. The header files are located in the *ORACLE_BASE**ORACLE_HOME*\oci\include directory.

For Microsoft Visual C++, specify *ORACLE_BASE**ORACLE_HOME**oci**lib**msvc* in the libraries section of the Option dialog box. For the Borland compiler, specify *ORACLE_BASE**ORACLE_HOME**oci**lib**bc*.

For example, if you are using Microsoft Visual C++ 7.0, you would need to put in the appropriate path, \oracle\ora10\oci\include, in the Directories page of the Options dialog in the Tools menu.

Note: The only Microsoft Visual C++ releases supported for the current OCI release are 7.0 or higher.

See Also: Your compiler's documentation for specific information about compiling your application and special compiler options

Linking OCI Applications for Windows

The OCI calls are implemented in dynamic link libraries (DLLs) that Oracle provides. The DLLs are located in the *ORACLE_BASE**ORACLE_HOME*\bin directory and are part of the Required Support Files (RSFs).

Oracle only provides the `oci.lib` import library for use with the Microsoft Compiler. Borland compiler is also supported by Oracle for use with OCI. Oracle recommends that applications must always link with `oci.lib` to avoid re-linking or compilation with every release.

When using `oci.lib` with the Microsoft Compiler, you do not have to indicate any special link options.

oci.lib

Oracle recommends that applications be linked with `oci.lib`, which takes care of loading the correct versions of the Oracle DLLs.

Client DLL Loading When Using Load Library()

The following directories are searched in this order by `LoadLibrary()`:

- Directory from which the application is loaded or the directory where `oci.dll` is located
- Current directory
- Windows:
  - 32-bit Windows system directory `(system32)`. Use the `GetWindowsDirectory()` function to obtain the path of this directory.
  - 16-bit Windows directory `(system)`. There is no Win32 function that obtains the path of this directory, but it is searched.
- Directories that are listed in the PATH environment variable

Running OCI Applications for Windows

To run an OCI application, ensure that the entire corresponding set of Required Support Files (RSFs) is installed on the computer that is running your OCI application.

The Oracle XA Library

The XA Application Program Interface (API) is typically used to enable an Oracle database to interact with a transaction processing (TP) monitor, such as:

- BEA Tuxedo
- IBM Transarc Encina
- IBM CICS

You can also use TP monitor statements in your client programs. The use of the XA API is supported from OCI.

The Oracle XA Library is automatically installed as part of Enterprise Edition. Table D–2 lists the components created in your Oracle home directory. `oci.lib` now contains the XA exports:

Table D–2 Oracle XA Components

Component	Location
`xa.h`	*ORACLE_BASE*`\`*ORACLE_HOME*`\oci\include`

Compiling and Linking an OCI Program with the Oracle XA Library

To compile and link an OCI program:

1. Compile *program*`.c` by using Microsoft Visual C++ or the Borland compiler, making sure to include *ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\xa` in your path.

Table D–3 Link Libraries

Library	Location
`oraxa10.lib`	*ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\xa`
`oci.lib`	`ORACLE_BASE\ORACLE_HOME\oci\lib\msvc` or, *ORACLE_BASE*`\`*ORACLE_HOME*`\oci\lib\bc`

2. Link *program*`.obj` with the following libraries shown in Table D–3:
3. Run *program*`.exe`.

Using XA Dynamic Registration

The database supports the use of XA dynamic registration. XA dynamic registration improves the performance of applications interfacing with XA-compliant TP monitors. For TP Monitors to use XA dynamic registration with an Oracle database on Windows, you must add either an environmental variable or a registry variable to the Windows computer on which your TP monitor is running. See either of the following sections for instructions:

- Adding an Environmental Variable for the Current Session
- Adding a Registry Variable for All Sessions

Adding an Environmental Variable for the Current Session

Adding an environmental variable at the command prompt affects only the current session.

Adding an Environmental Variable:

From the computer where your TP monitor is installed, enter the following at the command prompt:

```
C:\> set ORA_XA_REG_DLL = vendor.dll
```

where *vendor*`.dll` is the TP monitor DLL provided by your vendor.

Adding a Registry Variable for All Sessions

Adding a registry variable affects all sessions on your Windows computer. This is useful for computers where only one TP monitor is running.

Adding a Registry Variable:

1. Go to the computer where your TP monitor is installed.
2. Enter the following at the command prompt:

   ```
   C:\> regedt32
   ```

   The Registry Editor window appears.
3. Go to `HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\HOME`*`ID`*.
4. Choose the Add Value option in the Edit menu. The Add Value dialog box appears.
5. Enter `ORA_XA_REG_DLL` in the Value Name text box.
6. Select `REG_EXPAND_SZ` from the Datatype list box.
7. Choose OK. The *String Editor* dialog box appears.
8. Type *`vendor`*`.dll` in the String field, where *`vendor`*`.dll` is the TP monitor DLL provided by your vendor.
9. Choose OK. The Registry Editor adds the parameter.
10. Choose Exit from the Registry menu.

    The registry exits.

XA and TP Monitor Information

Refer to the following general information about XA and TP monitors:

- *Distributed TP: The XA Specification* (C193) published by the Open Group. The Open Group, 1010 El Camino Real, Suite 380, Menlo Park, CA 94025, U.S.A.
- See the Web site at:

  `http://www.opengroup.org/publications/catalog/tp.htm`
- Your specific TP monitor documentation

> **See Also:** *Oracle Database Application Developer's Guide - Fundamentals* "Developing Applications with Oracle XA", for more information about the Oracle XA Library and using XA dynamic registration

Using the Object Type Translator for Windows

The Object Type Translator (OTT) is used to create C struct representations of objects that have been created and stored in a database.

To take advantage of objects run OTT against the database, and a header file is generated that includes the C structs. For example, if a `PERSON` type has been created in the database, OTT can generate a C struct with elements corresponding to the attributes of `PERSON`. In addition, a null indicator struct is created that represents null information for an instance of the C struct.

The INTYPE file tells the OTT which object types should be translated. This file also controls the naming of the generated structs.

Note: The INTYPE File Assistant is not available, starting with Oracle Database 10g Release 1.

Note that the CASE specification inside the INTYPE files, such as CASE=LOWER, applies only to C identifiers that are not specifically listed, either through a TYPE or TRANSLATE statement in the INTYPE file. It is important to provide the type name with the appropriate cases, such as TYPE Person and Type PeRsOn, in the INTYPE file.

OTT on Windows can be invoked from the command line. Additionally, a configuration file may be named on the command line. For Windows, the configuration file is ottcfg.cfg, located in *ORACLE_BASE**ORACLE_HOME*\precomp\admin.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X